Developing Time-Oriented Database Applications in SQL

THE MORGAN KAUFMANN SERIES IN DATA MANAGEMENT SYSTEMS

Series Editor, Jim Gray

Developing Time-Oriented Database Applications in SQL

RICHARD T. SNODGRASS

Chapter opener art based on illustrations from the following sources: Chapters 1, 12, and 13 from Brackin, A. J., *Clocks: Chronicling Time, Encyclopedia of Discovery and Invention*, Lucent Books, 1991; Chapters 2, 3, 4, 5, and 6 from Woodward, P., *My Own Right Time: An Exploration of Clockwork Design*, Oxford University Press, 1995; Chapters 7, 8, and 9 from Britton, F. J., *The Escapements: Their Action, Construction, and Proportion*, Geo. K. Hazlitt & Co., reprinted by Arlington Book Co, 1984; and Chapters 10 and 11 from Headrick, M. V., *Clock and Watch Escapement Mechanics*, self-published, 1997.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances where Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Á Merrie
Ce jour, et toujours

Mach' es wie die Sonnenuhr
Zähl' die heiteren Stunden nur

Do like the sundial:
Count only the bright hours

— *German proverb*

Foreword

by Jim Gray
Microsoft Research

Precise clocks were developed so that seafarers could find their longitude. Precise temporal data techniques were recently developed to help database designers record and reason about temporal information. It is paradoxical that we are only now coming to understand how to think about time and represent it in formal systems. After all, time is the fourth dimension; it is at the core of existence. Yet, it is only recently that we have come to understand the fundamental concepts of instants, intervals, periods, sequenced changes, valid time, transaction time, and a bitemporal view of information.

Richard Snodgrass and his colleagues have explored temporal data concepts over the last two decades. They now have a fairly complete solution to the problems. Indeed the concepts are now being added to the SQL language standard. This book summarizes their work and presents it in a very accessible and useful way.

Temporal databases, viewed from this modern perspective, are surprisingly simple and powerful. The book gives examples of 85-line SQL programs that collapse to 3-line programs when the new concepts are applied. It introduces the concepts using concrete examples and conventional SQL. I found this mix of theory and practice very instructive and very easy to follow.

The book explains that temporal databases can be designed in two steps. First, the static database can be designed. Then, in a second pass, each table and constraint is given its temporal attributes. This makes design much more tractable. This approach is made all the more attractive by the fact that the temporal SQL language extensions are just modifiers to standard queries and updates—this very elegant approach makes temporal issues orthogonal to the other language issues.

I highly recommend this book to anyone interested in temporal data—either designing and building databases that record information over time, or just understanding the concepts that underlie representing temporal information. This book does an excellent job of organizing and summarizing this important area.

Foreword

by Jim Melton
Oracle Corporation

It's about time—time that a book like this was written and time that the SQL community got the benefits of the careful analysis and thought put into the subject.

Rick Snodgrass is one of the relatively few researchers in the field of temporal databases and has proved himself to be one of the more important of those few, in part because he insists on applying the theoretical knowledge gained from his research to practical applications and to real products.

Snodgrass proposed in 1992 that temporal extensions to SQL be developed by the temporal database community. In response to this proposal, a virtual committee was formed to design extensions to the 1992 edition of the SQL standard (ANSI X3.135.-1992 and ISO/IEC 9075:1992); those extensions, known as TSQL2, were developed during 1993 by this committee meeting only via email. In late 1993, Snodgrass first presented this work to the group responsible for the American National Standard for Database Language SQL, ANSI Technical Committee X3H2 (now known as NCITS H2).

In response to Snodgrass's presentation, X3H2 proposed to the International Organization for Standardization (ISO) that the project to extend the standard for SQL be enhanced by adding a subproject for temporal extensions to the language. This proposal was accepted in 1994, and an initial document for ISO/IEC 9075-7, known as SQL/Temporal, was started. Over the next two years, a series of proposals from Snodgrass and others were considered by the ISO group responsible for SQL (ISO/IEC JTC1/SC21/WG3, later ISO/IEC JTC1/SC32/WG3), but progress was slowed considerably by the need to focus on what has recently been published as SQL:1999. Work will undoubtedly resume on progressing SQL/Temporal in 1999 for publication early in the next millennium, and Snodgrass will no doubt play a significant role in its standardization.

The book you hold has been a long time in the making, not only because the subject matter can seem overwhelmingly complex if not presented carefully, but also because of the great number of examples that Snodgrass has taken from real application systems and translated into standard SQL and its proposed extensions. (Of course, not all of the examples can be used in all SQL products today; some of them are directed toward specific vendors' systems, while others depend on future extensions to the language.) The result of that care and extensive use of examples is great clarity and focus, yielding ready comprehension to readers willing to give the book the attention it deserves. I recommend this book very highly to all SQL practitioners, especially those with an interest in the temporal semantics of data.

Contents

Preface

This is how it goes.

We develop a database application, and initially the project proceeds smoothly enough. There are alternatives to weigh during the schema design, problems to contend with while writing the SQL code, and constant reconfiguration and interaction with other programs and legacy data, but all in all the project is under control. Then we decide that one of the tables needs another DATE column, recording when the row was valid. (After all, we added a birth date column a few weeks ago, with no surprises.) So we rework the part of the application that maintains that table, noticing that the code is getting more complicated. During testing, we discover that the primary key no longer is sufficient. We add the DATE column to the primary key, acknowledging that this is only a stopgap measure, and hope that the input data will be well formed, as there isn't time to write code that checks those constraints properly. In the back of our mind is the lingering doubt that perhaps referential integrity checking isn't working quite right either.

We soon realize that we need another DATE column to record when the row was no longer valid. In doing so we encounter a raft of off-by-one bugs, in which some less-than comparisons should have been '<=', and other places where we need to add "+ '1' DAY". We think we've found all the code locations that need to be changed, but we're not sure. And we now know for a fact that the primary and foreign keys are wrong, but we don't know how to even approach that mess.

The code to modify the database is becoming increasingly convoluted. Each modification has to at least consider changing the DATE columns, but it isn't at all clear how to approach such changes in a systematic fashion. And even the most trivial queries, such as "Who was Aaron's manager when he worked on the Capital account?", which before we could code in our sleep, now become painful to even contemplate writing in SQL.

Around this time, users start complaining that reports aren't consistent, that copies of the end-of-the-year summary have different numbers in them. Looking

into this anomaly, we finally figure out that the reports were run at different times, and the data had been changed in the meantime. We then realize that there is no way to correlate the end-of-the-year report with the cash flow report, unless they are run at the same time. Users are adopting an irreverent view of these reports: if you wait a few days, maybe the numbers will fix themselves.

To address the inconsistencies in the reports, someone suggests a quick fix: add another DATE column. The development group responds with astonishment and chagrin. How can we possibly get the code working with *another* DATE column, when we all know how much work resulted from adding the previous column? In fact, some in the group despair of ever getting the code as is, with just two DATE columns, working correctly—there are just too many arbitrary decisions, each layered on other equally ill-motivated quick fixes.

Looking back on the history of the development process, everyone has a vague idea that the problems started when that pesky DATE column was first added. How could one column flummox the whole system? And why do some columns, such as the birth date column, slide in smoothly, and other DATE columns cause no end of problems?

A PARADIGM SHIFT

Thomas Kuhn, in his insightful and highly influential book, *The Structure of Scientific Revolutions* [64], argued that science does not proceed in a linear, monotonic accumulation of knowledge, but rather exhibits intellectually jarring discontinuities, as radical ideas become the established world view, replacing the now-discredited prior conceptual foundation.

Two decades of research into temporal databases have unequivocally shown that a *time-varying table*, containing certain kinds of DATE columns, is a completely different animal than its cousin, the table without such columns. Effectively designing, querying, and modifying time-varying tables requires a different set of approaches and techniques than the traditional ones taught in database courses and training seminars. Developers are naturally unaware of these research results (and researchers are often clueless as to the realities of real-world applications development). As such, developers often reinvent concepts and techniques with little knowledge of the elegant conceptual framework that has evolved and recently consolidated, and researchers continue to conceal this framework with overly formal prose, never bothering to make the connection with existing tools at hand.

This book is an attempt to recast the insights from some 1600 papers in the research literature into terms usable by those brave SQL application coders working in the trenches. These concepts are integrated with the state-of-the-art approaches utilized by forward-thinking developers, as showcased in the case studies that form the bulk of the book. The result is, to use Kuhn's phrase, a *paradigm shift* in how

we think about time-varying data. This shift impacts how such tables are specified, how they are maintained, and how they are queried.

PREREQUISITES

I assume you are comfortable with the SQL query language. This book is not a primer on that language, though I do cover the temporal data types and temporal constructs of SQL-92 in depth. There are many excellent books that serve as introductions to SQL.

It helps if you have implemented an application involving time-varying data, if only to realize firsthand how difficult and confusing such a project can be, and thus to appreciate the degree to which the approach presented here helps clear out the undergrowth and achieve an elegant and unfettered design. One chapter assumes familiarity with the entity-relationship model; the rest of the book focuses solely on the relational model.

The conceptual tools introduced here are in a specific and fundamental way extensions of existing strategies, so everything you've learned until now (well, almost everything) will be useful in this brave new world. The hardest part, for which I'll provide careful guidance, is to jettison the notion that this DATE column "is just another column." Operating under the old assumptions unhappily doesn't work, as project after project after project has shown. Paradigm shifts are always scary, but the benefits are there for those willing to make the jump.

WHAT TO READ

The best way to understand the principles of time-varying applications and their expression in SQL is to work through a series of tangible examples. By examining the design issues that arise and the kinds of constraints, queries, and modifications that we wish to express in implementing these specific applications, you will gain an appreciation of the abstract principles at play. For this reason, the bulk of this book is comprised of case studies.

Each case study sets the stage with a discussion of the application domain, which includes oil field records, cattle location information, and cadastral data. The relevant tables are introduced, followed by a discussion of the design, querying, and modification of these (time-varying) tables. While the applications and the people mentioned in the case studies all exist, the specific SQL examples have been tailored to bring out the issues under discussion.

The case studies were easy to locate. It seems that most database applications involve time-varying data. Indeed, applications that are inherently *not* temporal are about as prevalent as the proverbial hen's teeth. In fact, the only places you

encounter nontemporal examples are in books and seminars, a phenomenon that unintentionally emphasizes the inherent complexity of time-varying applications.

To understand the fundamental concepts, you are encouraged to read *all* the chapters, even if you aren't an oil field engineer or a veterinarian. Each case study brings out a new category of temporal data, with its unique characteristics and demands. In fact, by studying other fields, you are relieved of the minutiae of your current environment. By studying a foreign language or culture, a deeper understanding of your own language or culture often follows as an additional, or even sometimes primary, benefit. After you have read the book, a productive approach to address a new set of requirements is to ask, To which case study is the application under development most closely related? Then the relevant code fragments can be customized to the problem at hand.

A few sections are marked with an asterisk to indicate advanced material. Feel free to skip these sections on a first, or even second, reading.

CASE STUDIES

Befitting the book's categorization as nonfiction, the people and their situations are as described herein. The specifics of their solutions to the problems presented by time-varying data have been adapted to better illustrate general approaches that I wish to emphasize. Most of the SQL code was written by use for the book, but it is reminiscent of that appearing in the actual applications. In the discussion, I have attempted to not oversimplify. Much of the complexity inherent in these applications is cleverly hidden in the details, and any realistic solution must ultimately confront the enterprise in all its glory and intricacy.

CD-ROM

The included CD-ROM contains the code fragments implemented in a variety of commercial systems, including IBM DB2 Universal Database (UDB), Ingres, Informix–Universal Server, Microsoft Access, Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL. While these code fragments have been tested, the author and the publisher make no claims as to the suitability or correctness of these code fragments.

Also included are versions of some of the systems discussed in Chapter 12.

ERRORS

I would appreciate hearing about any errors that you find in the book, as well as receiving any other constructive suggestions you may have. (I'd especially like to hear

of better ways to write individual code fragments.) Please email your comments to the author at *snodgrass@mkp.com*.

ACKNOWLEDGMENTS

The specific content of this book, as well as its overall composition, came about through collaboration with my dear friend and colleague Christian S. Jensen, who has assembled and directs the world's strongest temporal database research group at the University of Aalborg in Denmark. He is present on every single page, looking over my shoulder, probing, clarifying, generalizing, exemplifying. While the words in this book are mine (well, except for "exemplifying," which is definitely one of *his* words!), the ideas are jointly ours, comingled with the contributions of the many authors discussed in the "Readings" sections. Research is primarily an excuse to play with ideas, and concepts provide opportunities to interact on a deep level with others. Much of the joy I have experienced in this crazy business has directly or indirectly involved Christian, whether in the heady probing of trying to get at core truths, or in appreciating the shimmering simplicity of the resulting insights that, once uncovered, are in retrospect obvious.

Many others helped with the case studies, tested the code fragments, read drafts of this book, and offered nontechnical sustenance.

Cheryl Bach, Jim Barnett, Nigel Corbin, Brad De Groot, Jens Gadgaard, and Chris Janton were generous with their time in explaining the details of their applications, which provided the impetus for the case studies.

I thank Anindya Datta, Igor Viguier, and the Department of Information Systems at the University of Arizona for access to the Sybase DBMS. Cliff Leung from IBM, Paul Brown from Informix, Nick Kline and Goetz Graefe from Microsoft, Rafi Ahmed from Oracle, and Jack Reid from UniSQL Customer Support Services provided help with their respective DBMS products.

I appreciate comments on the manuscript from Rafi Ahmed, John Bair, Paul Brown, Jim Gray, Brad De Groot, Jeff Gross, Robert Brett Gulledge, Christian S. Jensen, and Dennis Shasha. Jim Melton went above and beyond the call of duty in providing *two* detailed reviews of the entire manuscript.

Amad Arsalan, Scott Calvert, Wen-Ke Chen, Brad De Groot, Alvin Gendrano, Bruce Huang, Vijaykumar Immanuel, Jie Li, Wei Li, Giedrius Slivinskas, Helen Thomas, Brad Traweek, and Inés F. Vega-López provided corrections to the code fragments. Alan Brucks provided information on the year 2000 problem.

I thank the following people who created the content of the CD-ROM: HTML templates (Christopher Cooper, Jose Alvin Gendrano, Rachana Shah, and Jian Yang), data types: IBM DB2 UDB (Brad Traweek and Giedrius Slivinskas), Informix–Universal Server (Jason Cox and Anand Wagle), Ingres (Wen-Ke Chen), Microsoft Access (Inés F. Vega-López, Lincoln Turner, and Ze-Yuan Zou), Microsoft SQL Server

My editor, Diane Cerra, was wonderful throughout the involved process of writing and producing this book. I especially appreciate her constant quest for quality. I've written for several editors and publishers, and Diane and Morgan Kaufmann are by far the most author- and book-friendly.

The design of this book is considerably more complex than that of my other books. Edward Wade worked closely with me on this design. Although I would not have thought it possible, Edward cared as much as I did about getting the design just so, putting his heart into the project. The design feels *right*, and Edward deserves most of the credit. Edward, it was truly a joy to work with you.

Finally, I thank my wife, Merrie, and my two children, Eric and Melanie, for continually emphasizing by their example that life is so much more than book writing. They provided delightful distractions—how could I resist rambling through the park, or reading a poem by Shel Silverstein, or helping out on a pinewood derby car?

One of the oft-unexpected benefits of taking photographs, whether as a hobby or as a profession, is that you *see* more vividly. The veins within a fallen leaf become all-absorbing, and shadows on a building are suddenly profound and evocative. My experience in writing this book has been similar, in that the words I have been fortunate to encounter in my reading and listening have particularly

resonated with me. During the course of the last two years, I have enjoyed the unwitting companionship of many fine writers, of books and of music, including Beth Nielsen Chapman, Mary Chapin Carpenter, Iris DeMent, Louise Erdrich, Nanci Griffith, Stephen Jay Gould, Lucy Kaplansky, Barbara Kingsolver (a fellow Tucsonan!), William Langewiesche, Alison Moore (another Tucsonan), Ron Querry (Tucson is blessed with accomplished writers), Anna Quindlen, Dava Sobel, Dar Williams, and Lucinda Williams. Their prose and poetry has enriched and sustained me on this at times arduous yet ultimately gratifying journey. My hope is that our paths continue to cross.

Introduction

I t was "as if you fired a 15-inch naval shell at a piece of tissue paper and the shell came right back and hit you." Thus Ernest Rutherford described his astonishment at the result of his undergraduate student's experiment in 1911. The experiment was a simple one: expose thin foils of gold to α particles and watch for appreciable scattering. The then-current model of matter was that it was a "bunch of electrons and some nondescript smeared out jelly of positive charge." The α particle weighs some 8000 times more than an electron, yet was unexpectedly deflected by the jelly. This observation led to a radical change in our conception of matter, resulting in the Rutherford atom, a tight nucleus surrounded by orbiting electrons.

The scattering of particles and waves such as X rays provides much information on the inner structure of matter. Diffraction patterns are stunningly beautiful in their regularity, reflecting in a highly indirect fashion the ordering of atoms of the crystalline solids (such as $TaSe_2$, shown in Figure 1.1) exposed to the beam. These patterns can be analyzed to understand this geometric structure and other properties. Indeed, much of what is known about the structure of solids is due to analysis of diffraction patterns. Such a study was critical, for example, in understanding the spiral staircase of DNA's double helix.

The diffraction patterns are emphatically *not* magnifications of the crystalline structure; rather, the various distances and angles of the blips can be translated back via sophisticated calculations to the unseen lining up of atoms stuck in a three-dimensional gridlock. Only by understanding the phenomenon of diffraction of wave motion, and the impact of a periodic array of barriers on the impinging wave, can physicists accurately orient the atoms and piece together the underlying pattern.

An SQL table containing dates and times also exhibits a pleasing regularity, with dates in one column recurring in other rows in another column, and the dates in many rows marching forward almost in lockstep. This regularity is indeed sugges-
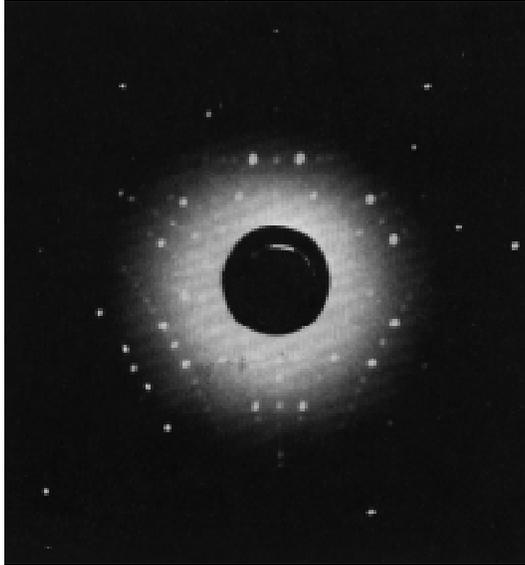
Figure 1.1 Electron diffraction pattern of TaSe₂. (Image reprinted, by permission, from _Structural and Chemical Analysis of Materials_, Figure 11.5(b), by J.P. Eberhart. © 1991 John Wiley & Sons, Ltd.)

tive of an inner structure, which SQL so effectively masks. Only by understanding the ways in which time-varying behavior can be modeled, and by studying the mapping of this information into tabular form, can an SQL table in a time-varying application be effectively designed, queried, and maintained.

1.1   A TRIAD OF TRIPLES

It is human nature to differentiate, to tease apart, to contrast. Identifying dichotomies, partitioning into two mutually exclusive groups, seems to be a fundamental strategy for contending with diversity. We favor black and white, this and that, us and them, over shades of grey, a spectrum of possibilities, a global community. We stereotype ourselves and others and all things by membership in identified groups: plant or animal, minority or majority, right or wrong. Logic and libraries emphasize the distinction between true (nonfiction) and false (fiction). Much of the prevalence of computers in today's society derives from the clarifying simplicity of strings of just two values, 0 and 1, encoding everything from names to the relative strength of a chess configuration.

As prevalent as this binary structure is, a collection of three items of similar import seems to resonate even deeper. While a division into two parts is appealing in its reductionism and simplicity, a trichotomy is attractive precisely because it is

not either-or. A triad cannot be reduced to black and white, but is forever resigned to contain ambiguity and complexity. Three-level logics embrace the value of "unknown." The Greeks viewed the world as comprising the earth, the sea, and the sky (heaven). Christians rejoice in the Trinity; they also speak of earth, heaven, and hell. In Buddhism, there are the three roots of evil: lust, hatred, and delusion. Many religions differentiate the mind, the body, and the soul. Pythagoras celebrated the triangle, the simplest geometric figure. We perceive three spatial dimensions. Rainbows are combinations of three primary colors. Many governments are partitioned into three branches, for a similar reason that a stool has three legs. The harmonic basis of Western music is a chord of three tones consisting of a root with its third and fifth. A literary trilogy carries with it a satisfying completeness.

In this book we examine how to implement a time-varying application in the SQL structured query language. We focus on three sets of orthogonal concepts:

- Temporal data types
- Kinds of time
- Temporal statements

These concepts are encountered in every time-varying application. If SQL adequately supported these concepts, our task, and yours in actually developing the application, would have been much easier: just use SQL in the appropriate fashion to bring forth the desired behavior. Despite the near universality of time and the time-varying nature of the enterprise being modeled—a static and unmalleable configuration is rare and uninteresting—SQL quite frankly does a lousy job in capturing those aspects that are changing in time, or in providing constructs to effectively model, query, or modify such information. Instead, you, the application developer, are saddled with the task of transforming these concepts into something that SQL can deal with. This book will emphasize the best way to think about time-varying data and will show with many examples how to map these concepts into a temporally unfriendly SQL.

Each of these concepts itself consists of three orthogonal elements. There are three fundamental temporal data types:

- *Instant:* something happened at an instant of time (e.g., "now," which happens to be June 29, 1998, 4:06:39 P.M., when I am writing this, or sometime, perhaps much later, when you are reading this)
- *Interval:* a length of time (e.g., three months)
- *Period:* an anchored duration of time (e.g., the fall semester, August 24 through December 18, 1998)

SQL-92 includes support for instants and intervals, though in places it confounds the two. Most DBMS products, though, only support instants, with intervals being

simulated with integers or floating-point numerics. Periods are always left to the application developer to simulate using supplied data types.

There are three fundamental kinds of time. We'll define more precisely and illustrate these terms shortly, in the next chapter.

- *User-defined time:* an uninterpreted time value
- *Valid time:* when a fact was true in the modeled reality
- *Transaction time:* when a fact was stored in the database

These kinds of time are orthogonal: a table can be associated with none, one, two, or even all three kinds of time. Understanding each kind of time and determining which is present in an application is absolutely critical. We will characterize each in detail. SQL-92 has rudimentary support only for user-defined time; the language provides no help whatsoever with the other two types of time. That is left for you to manage, manually, in your application. We'll see exactly how to do so.

There are three basic kinds of time-oriented statements:

- *Current:* now
- *Sequenced:* at each instant of time
- *Nonsequenced:* ignoring time

The trichotomy applies equally well to queries, modification statements, views, and integrity constraints. The most useful is sequenced, for which SQL-92 provides absolutely no help. In fact, getting SQL to express a sequenced statement is often quite painful, yet that is usually what is required by the application. We will show exactly how to write all three kinds of statements. SQL-92 provides some support for nonsequenced statements; current statements are again entirely up to the programmer.

So, the several hundred pages of this book attempt to convey three sets of three orthogonal concepts, most of which SQL is woefully ignorant. Those notions foreign to SQL must be transformed from their clean, crisp manifestation into an often baroque expression in SQL. To add to the challenge, no DBMS supports the SQL standard in its entirety; instead, there are infuriating inconsistencies and substitutions each vendor has chosen to impose on its users. We help you navigate these treacherous waters, and avoid the ever-present shoals.

1.2   THE SQL STANDARD

SQL is actually a series of standards. SQL-86 included no temporal data types, even though some commercial implementations in the late 1980s did support such data types. SQL-89 added support for referential integrity; no temporal data types were

added. Several temporal data types were introduced in SQL-92: DATE, TIME, TIME-STAMP, and INTERVAL. SQL3 is currently in draft form (and has been so for several years; the path from draft to final accepted standard is a ponderous one). Portions of SQL3 are expected to be approved as a standard in late 1999. Part 7, SQL/Temporal, introduced a new constructor, PERIOD. In this book, "SQL" refers to SQL-92, and "SQL/Temporal" refers to this draft part of SQL3. We emphasize facilities currently available in database products, but also mention features on the horizon.

1.3   CONVENTIONS

"At least one" queries can be easily stated using an additional correlation name in the FROM clause.

In the case studies, for each kind of query (modification, assertion, constraint, view), the general principle behind the query is discussed. For some complex queries, pseudocode may be provided. Following the pseudocode, a particular query is given as a code fragment. Notable features of the query are then examined. Code fragments are often referenced later in the discussion. The references are abbreviated as, for example, CF-1.1. Important points are emphasized as pull quotes (those pieces of information set off from text and extending into the page margins).

As an illustration of the stylistic conventions used throughout this book, consider (conventional) queries of the form "... at least one ..."—for example, "Which manager makes less than *at least one* of their subordinates?" This query must find a suitable subordinate for each manager listed. Such queries can be written with an EXISTS or IN subquery.

Code Fragment 1.1   Which managers make less than at least one of their subordinates?

```
SELECT DISTINCT EID
FROM Employee AS M
WHERE EXISTS ( SELECT *
               FROM Employee AS E
               WHERE E.Mgr = M.EID AND E.Salary > M.Salary )
```

While this query in some ways parallels the English version, the effect can be more easily obtained by augmenting the FROM clause and the WHERE clause.

"At least one" queries

Mention the table providing the sought-after entity in the FROM clause.
Reference that table in the WHERE clause to locate the entity.

Here, the sought-after entity is a subordinate (in the `Employee` table).

Code Fragment 1.2   **Which managers make less than at least one of their subordinates (without using EXISTS)?**

```
SELECT DISTINCT M.EID
FROM Employee AS M, Employee AS E
WHERE E.Mgr = M.EID AND E.Salary > M.Salary
```

The nested correlation name has been moved up to the main FROM clause, resulting in a more succinct query, a process termed "decorrelation."

A stylized image of an *escapement* (the critical component of a mechanical clock) is displayed on the opening page of each chapter in this book. Such clocks have an *escape wheel* connected to a weight (such as in a grandfather clock) or a coiled spring (such as in a wristwatch). The escape wheel, which is connected by gears to the hands of the clock, would spin continuously if not retarded by the escapement, which periodically stops and releases the escape wheel.

Over the last 500 years, horologists have devised all manner of ingenious escapements. Examples include Arnold's chronometer escapement, Bond's gravity escapement, Brocot's pin pallet escapement, Congreve's extreme detached escapement, the Debaufre escapement, Froment's electrical escapement, Graham's dead-beat escapement, Grimthorpe's gravity escapement (used in Big Ben), Harrison's grasshopper escapement, and the very early verge and foliot escapement, some of which are illustrated in the chapter openers.

Interspersed throughout the case studies will be brief sidebars on a multitude of calendars and on the fascinating alchemy of art, science, and engineering that characterizes the development of increasingly accurate clocks through the ages. The clock descriptions are accompanied by a stylized "sun" icon, the calendar descriptions by a "moon" icon.

Finally, each chapter ends with a section on implementation considerations, identifying ways in which commercial systems deviate from the standard and providing adaptations of the chapter's code fragments that will run on these systems. A final section, Readings, provides pointers that elaborate on the material in that chapter and in the clock and calendar sidebars, indicating the correspondence to the sidebars with the sun and moon icons, respectively.

Calendars

Calendars are mankind's way of contending with years and lunar months composed of a nonintegral number of days. As Stephen Jay Gould so eloquently writes "If God were Pythagoras in Galileo's universe, calendrics would never have become an intellectual subject at all. The relevant cycles for natural timekeeping would all be nice, crisp, easy multiples of each other....But God, thank goodness, includes both Loki and Odin, the comedian and the scholar; the jester and the saint. God did not fashion a very regular universe after all. And we poor sods of his image are therefore condemned to struggle with calendrical questions till the cows come home." [35, p. 134]

Gnomonics

Man's first subdiurnal clock was most likely his shadow: when it started getting longer, the day was half over. The next advance was to substitute a *gnomon*, or staff of known length whose shadow can be measured (*gnomon* is a Greek word for "pointer"). Obelisks at town centers, which provide a more accurate designation of noon, by virtue of their height, were used in Egypt by around 1450 B.C.E. for the measurement of time and the construction of calendars, thereby signaling the beginning of the science of sundials, or *gnomonics*.

In the third century B.C.E., a Chaldean priest by the name of Berossos hollowed out a half-sphere in a rectangular block of stone, positioned a gnomon in the center, and inscribed lines dividing the arc of the shadow into 12 hours, in accordance with the 12 constellations crossed successively by the sun, the zodiac (Figure 1.2). This *hemispherium* was the first sundial to measure hours. Berossos then realized that the bottom half of the sphere was never used, so he removed this useless portion, resulting in the lighter and thus more portable *hemicyclium* (Figure 1.3).
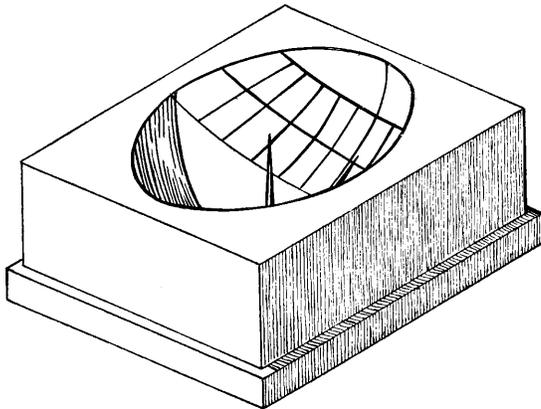
Figure 1.2 Berossos's hemispherium. (From Rohr, R. R. J., *Sundials: History, Theory, and Practice*. Dover Publications, NY, 1996.)
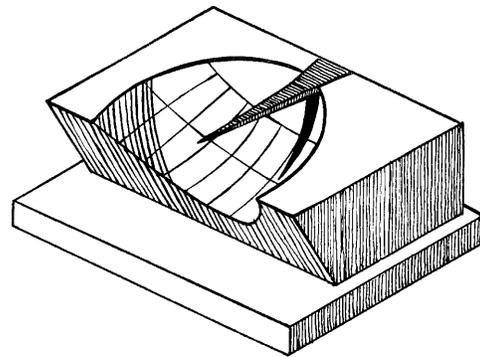
Figure 1.3 Berossos's hemicyclium. (From Rohr, R. R. J., *Sundials: History, Theory, and Practice*. Dover Publications, NY, 1996.)

1.4 IMPLEMENTATION CONSIDERATIONS

The SQL code for the fragments is almost entirely in standard SQL-92 (any exceptions will be clearly noted). Unfortunately, due to the noncompliance of all existing DBMSs, a few of these fragments run on no existing platform. Hence, each case

study concludes with a discussion of how the general approach can be applied to various specific DBMSs, including IBM DB2 Universal Database (UDB), Informix–Universal Server, Microsoft Access and Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL. Each of these products supports a different variant of SQL, introducing limitations that must be worked around and extensions that can be exploited. Each also implements the various constructs in SQL differently, so an approach that is impractical on one product may be the preferred one on another product.

The included CD-ROM contains the code fragments implemented on one or more DBMSs. The specific versions on which the fragments were tested were IBM DB2 UDB; Informix–Universal Server 9.12; Microsoft Access 95, Access 97, and Access 2000; Microsoft SQL Server 6.5 and 7.0; Sybase SQLServer 10; Oracle8 Server; and UniSQL. However, because vendors work very hard to ensure their products are upward compatible, these fragments can be expected to continue to apply in future versions of these systems.

The descriptions of the specific DBMSs parallel each other, so each can be read independently of the rest. Indeed, it is expected that you will be using only one DBMS; the material on the other products may be safely skipped.

1.5   READINGS

The official designation of SQL-86 is American National Standards Institute (ANSI) X3.135-1986 and International Organization for Standardization (ISO) 9075-1987, "Database Language SQL." This standard, at 110 pages, is relatively brief. SQL-89 is ANSI X3.135-1989 and ISO/IEC 9075:1989; this language added referential integrity. In addition, ANSI published X3.168-1989, "Database languages—Embedded SQL," which made specifications for embedding SQL in conventional programming languages normative (required); ISO chose not to publish an analogous standard. SQL-92, which does have normative embedding, is ANSI X3.135-1992 and ISO/IEC 9075:1992, "Database languages SQL" [44]. Melton and Simon provide a comprehensive, readable presentation of SQL-92, including a thorough explanation of the SQL standardization process (Jim Melton is the editor of the SQL standards) [71]. The standard itself is a precise, though soporific 580 pages.

In 1995 ISO standardized ISO/IEC 9075-3:1995, "Database languages—SQL–Part 3: Call-Level Interface"; the next year, ISO/IEC 9075-4:1996, "Database languages—SQL–Part 4: Persistent Stored Modules," appeared. These were originally considered parts of the draft SQL3 specification, but were standardized before the core part of that language. There is also a 150-page "Database language SQL—Technical Corrigendum 3" that provides (mostly minor) corrections and disambiguations to SQL-92, Part 3, and Part 4 [19].

SQL3 is an evolving document, with 10 parts, two of which have achieved standardization, as just mentioned. The core portions of the language, SQL/Framework (Part 1), SQL/Foundation (Part 2), SQL/CLI (Call Level Interface: Part 3), SQL/PSM (Persistent Stored Modules: Part 4), and SQL/Bindings (Host Level Bindings: Part 5) are nearing the FDIS ballot stage, when the SQL committees of the member countries will vote on the question of whether the specification is ready to be an international standard. SQL/Temporal (Part 7) will not go into balloting until the next millennium. SQL/Foundation by itself is 850 pages; together all 10 parts of this specification exceed a back-straining 2000 pages.

A page maintained by Keith Hare *(www.jcc.com/sql_stnd.htm)* is a central source of information about the SQL standard. The information available there includes the current status of the standard, information about the standards committees and the standards process, and pointers to other standards pages. Eisenberg and Melton provide a crisply written overview of database standards [29].

Ernest Rutherford, whose model of the atom constituted an essential step toward our current understanding of matter, did not receive the Nobel Prize for that contribution, only because he had already received this ultimate recognition some three years earlier in 1908, at the ripe old age of 37, for his work with radioactive elements and X rays. The context of these experiments is ably described by Abraham Pais (who provides the second quote in the first paragraph of this chapter) in his superb biography of the Danish physicist Neils Bohr [77, p. 123]. Bohr refined and extended Rutherford's model to arrive at our current understanding (which is often called the "Bohr atom"), attaining the Nobel Prize in 1922, also at the age of 37. Rutherford's undergraduate student, Henry G. J. Moseley, subsequently used Rutherford's model, Bohr's theory, and his own X-ray diffraction studies to understand the periodic table of the elements in terms of atomic numbers.

Stephen Jay Gould has written a delightful and highly recommended monograph entitled *Questioning the Millennium: A Rationalist's Guide to a Precisely Arbitrary Countdown*. "If we regard millennial passion in particular, and calendrical fascination in general, as driven by the pleasure of ordering and the joy of understanding, then this strange little subject…becomes a wonderful microcosm for everything that makes human beings so distinctive, so potentially noble, and often so actually funny" [35, pp. 157–158].

René Rohr's *Sundials: History, Theory, and Practice* provides just that: a fascinating 2500-year chronology, a readable explication of the mathematics behind sundial design, and sage advice on positioning the gnomon and drawing the arcs [80]. Along the way, 51 photographs and over 100 figures illustrate the myriad forms sundials have taken over this period.
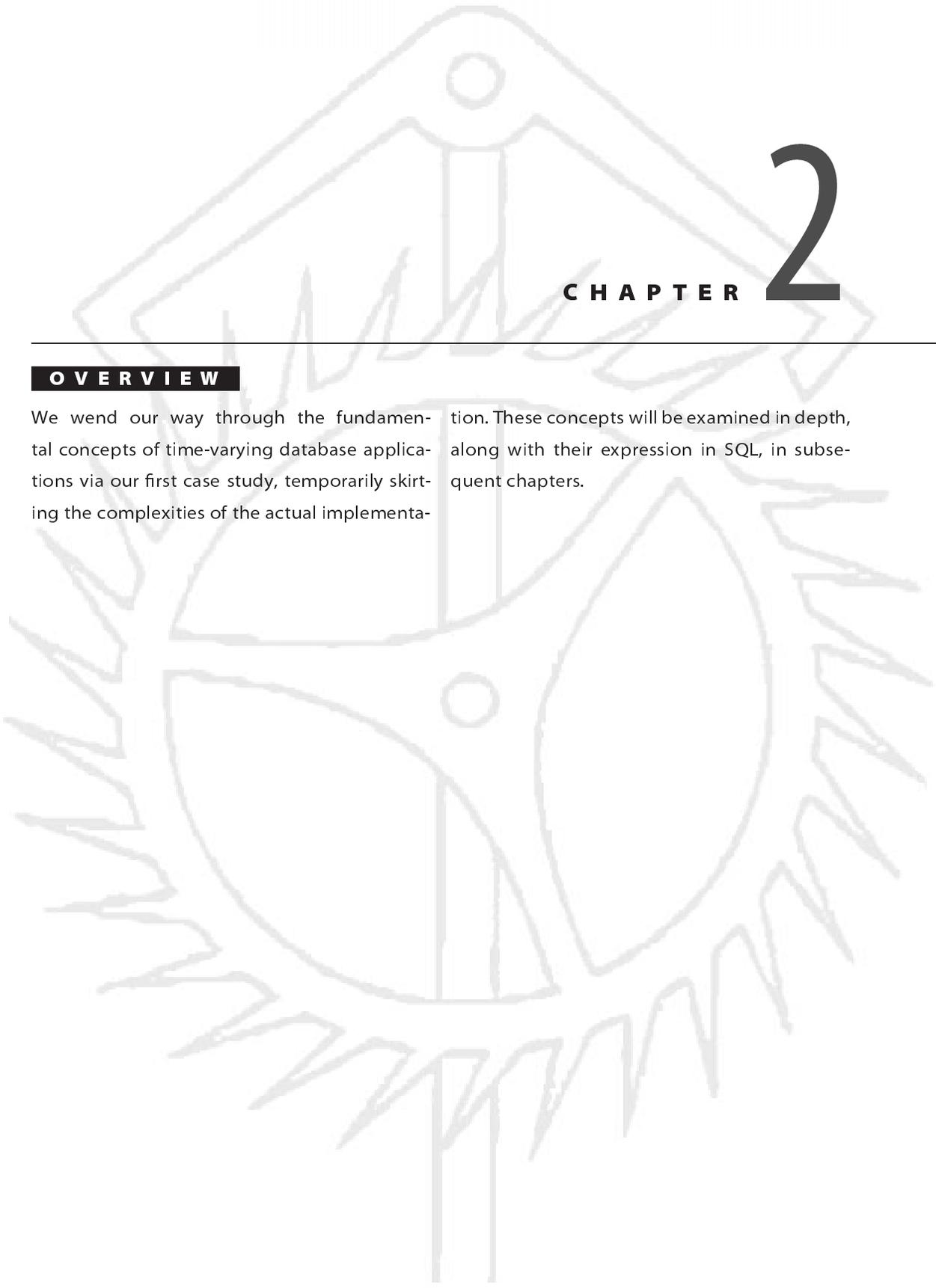
O V E R V I E W

We wend our way through the fundamental concepts of time-varying database applications via our first case study, temporarily skirting the complexities of the actual implementation. These concepts will be examined in depth, along with their expression in SQL, in subsequent chapters.

Fundamental Concepts

They started getting sick in early June of 1997. Some just had a bad stomachache; others had severe cramping and were passing blood. They suffered from a potentially lethal strain of the bacterium *Escherichia coli* (O157:H7). By mid-August some dozen-odd cases, all in Colorado, were 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 the outbreak.

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

Put simply, the lack of an adequate time-varying database cost Hudson Foods $25 million.

Dr. Brad De Groot is a veterinarian working in Clay Center, Nebraska, about 60 miles southeast of Columbus. He is 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 healthy cow later succumbs to a disease). These relationships can lead to an understanding of how disease is transferred to and among cattle, and ultimately to better detection and prevention regimes. As input to this epidemiologic study, Brad is collecting data from commercial feed yard record-keeping systems to extract the movement of some 55,000 head of cattle through the myriad pens of large feedlots in Nebraska.

VALID-TIME STATE TABLES

> It's present everywhere, but occupies no space.
> We can measure it, but we can't see it, touch it,
> get rid of it, or put it in a container.
> Everyone knows what it is and uses it every day,
> but no one has been able to define it.
> We can spend it, save it, waste it, or kill it,
> but we can't destroy it or even change it,
> and there's never any more or less of it.
>
> —Jespersen and Fitz-Randolph, *From Sundials to Atomic Clocks*

In a feed yard, 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 reside in each pen of each feed yard. The full schema for this table has nine columns; we'll just consider a few of them.

Brad wishes to capture the history of which cattle were coresident, to study how disease moves from cow to cow. He adds two columns, `FROM_DATE` and `TO_DATE`, to this table:

`LOT_LOC(LOT_ID_NUM, PEN_ID, HD_CNT, FROM_DATE, TO_DATE)`

These two columns will enable many interesting queries to be expressed (some of considerable intricacy), while enormously complicating previously innocuous constructs such as primary and foreign keys. These columns render the table a "valid-time state table": it records information valid at some time in the modeled reality, and it records states, that is, facts that are true over a period of time. The `FROM_DATE` and `TO_DATE` columns delimit the "period of validity" of the information in the row. The "temporal granularity" of this table is a day.

The first three columns are integer columns. The last two columns are of type DATE. SQL supports three kinds of instants, DATE, TIME, and TIMESTAMP, which differ in their range of values (e.g., DATEs range over 9999 years, whereas TIMEs range over only 24 hours) and their temporal granularity (a day for DATE and a second for default TIMEs). Chapter 3 covers these and the INTERVAL data types in all their glory (and grubbiness).

The last two columns denote the starting instant (actually, the starting day) of the period of validity of the row and the terminating instant of the period of validity. Unfortunately, SQL-92 does not support periods, so the period of validity must be implemented with two delimiting instants. Chapter 4 lists the various ways periods can be implemented with the data types that SQL does provide, and the operations (predicates and constructors) that may be applied to periods.

Table 2.1 records the movement of three lots of cattle in the feed yard. In this table we see that 17 head of cattle were in pen 1 for 11 days, moving inauspiciously

Table 2.1 The `LOT_LOC` **table.**

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

off the feed yard on February 18 (SQL-92 DATE literals are expressed as year-month-day). Also, 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.

Without the timestamp columns (`FROM_DATE` and `TO_DATE`), the primary key of `LOT_LOC` is the pair (`LOT_NUM_ID`, `PEN_ID`), which can be informally expressed as "the (lot identifier, pen identifier) pair is unique to a single row." With the timestamp columns, this can be generalized to "at any point in time, the (lot identifier, pen identifier) pair is unique to a single row." It is unfortunate that SQL's PRIMARY KEY construct is inadequate for valid-time state tables; expressing this manifest constraint in SQL-92 requires a complex assertion, as will be shown in Chapter 5, which covers all manner of definitional requirements of valid-time state tables.

2.1.1   Queries

Queries over conventional tables ask, "What is?" Queries over time-varying tables can be placed in three broad classes. For each conventional (nontemporal) query over a table without these two DATE columns, there exist "current" ("What is now?"), "sequenced" ("What was, and when?"), and "nonsequenced" ("What was, at any time?") variants over the corresponding valid-time state table.

Consider the nontemporal query "How many head of cattle from lot 219 in feed yard 1 are in each pen?" The current analog over the `LOT_LOC` valid-time state table is "How many head of cattle from lot 219 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 requesting such rows. 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 were in each pen." The result (Table 2.2) provides the requested history. We see that lot 219 moved around a bit.

The nonsequenced variant is "How many head of cattle from lot 219 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. Table 2.3 shows the result. Nonsequenced queries are often awkward to express in English, but can sometimes be useful.

As another example, consider the nontemporal query "Which lots are coresident 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, which turn out to be notoriously difficult to express in SQL-92.

The current version, "Which lots are currently coresident in a pen?", will return the empty table when evaluated on Table 2.1, as none of the lots are currently coresident (lots 219 and 374 are currently in the feed yard, but in different pens).

The nonsequenced variant is "Which lots were in the same pen, perhaps at different times?" The result is Table 2.4: all three lots had once been in pen 1.

Note however that at no time were any cattle from lot 137 coresident with either of the other two lots. To determine coresidency, the sequenced variant is used: "Give the history of lots being coresident in a pen." This requires the cattle to actually be in the pen together, at the same time. The result of this query on Table 2.1 is the following:

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

As we will see in Chapter 6, current and nonsequenced queries are relatively easy to express in SQL, but sequenced queries, which are prevalent, are surprisingly arduous. That chapter provides many examples to illustrate how such queries are phrased in SQL.

2.1.2 Modifications

Modifications (that is, insertions, deletions, and updates) comprise the bulk of many applications and are challenging when applied to time-varying data. We'll illustrate modifications on the LOT table, which captures the gender of the cattle in each lot. Surprisingly (especially to the cattle!), the gender attribute is time-varying. As 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.

Table 2.2 The history of lot 219.

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

Table 2.3 Result of a nonsequenced query.

PEN_ID	HD_CNT
1	43
1	20
2	23
2	43

Table 2.4 Result of another nonsequenced query.

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

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. 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 2.1 illustrates the transitions in gender that are possible, all of which are irreversible. (And you thought that this book was going to be about databases!)

Capturing the (time-varying) gender of cattle is important in epidemiological studies, for the gender can affect disease transfer to and between cattle. Hence, in Brad's feed yard database schema, LOT is a valid-time state table.

Bull calf ─────────────→ Bull
                            │
                            │
                            ▼
                          Steer

Heifer calf ───────────→ Cow
                            │
                            │
                            ▼
                      Neutered cow
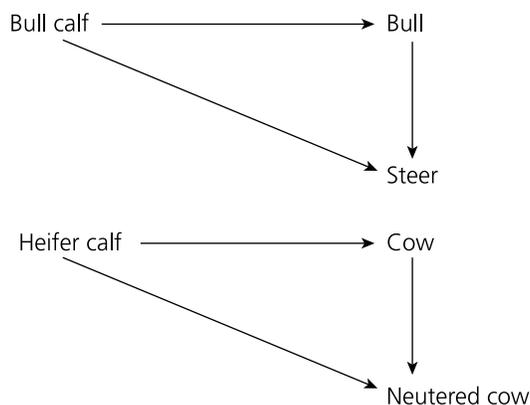
Figure 2.1 Gender transitions.

A slice of the LOT table is shown in Table 2.5 (in this excerpt, we've omitted several columns not relevant to this discussion). The GNDR_CODE is an integer code. For expository purposes, we will use single letters, with c indicating the lot consists of bull calves, h indicating the lot are heifers, and s indicating the lot are steers. 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.

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. As with queries, there are three general classes of modifications: current, sequenced, and nonsequenced.

"Lot 433 arrives today" is a current insertion. "Lot 101 leaves the feed yard today" is a current deletion. The two modifications in concert result in Table 2.6. All information on lot 234 after today has been deleted. (As this is being written on January 13, 1999, "today" is shown in SQL as 1999-01-13, exposing the nonlinear fashion in which this book evolved.)

"The cattle in lot 799 are being steered today" is a current update, with the result shown in Table 2.7.

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."

"Lot 426, a collection of heifers, was on the feed yard from March 26 to April 14" is a sequenced insertion. "Lot 234 will be absent from the feed yard for the first three weeks of October, when the steering will take place" is a sequenced deletion.

Table 2.5 The LOT table.

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	c	1998-03-12	9999-12-31

Table 2.6 Result of a current insertion and deletion.

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
433	c	1999-01-13	9999-12-31
234	c	1998-02-17	1999-01-13
799	c	1998-03-13	9999-12-31

Table 2.7 Lot 799 was steered today.

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
433	c	1999-01-13	9999-12-31
234	c	1998-02-17	1999-01-13
799	c	1998-03-12	1999-01-13
799	s	1999-01-13	9999-12-31

A sequenced update is the temporal analog of a nontemporal update, with a specified period of applicability. "Lot 799 was steered only for the month of March" is a sequenced update. (Something magical happened on April 1. The idea here is to show how to implement sequenced updates in general, and not just on cattle.)

As with queries, a nonsequenced modification treats the timestamps identically to the other columns and often 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."

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.

Chapter 7 shows that current and nonsequenced modifications are not that hard to express in SQL, but sequenced modifications, which are often the most useful, are doggedly obstinate, almost to the point of intractability. In that chapter we provide abundant guidance on the care and feeding of modifications of time-varying tables.

2.2   TRANSACTION-TIME STATE TABLES

The `LOT_LOC` and `LOT` tables capture the history of reality. The first row of Table 2.6 says that had we checked the cattle in lot 101 anytime during the first three months of 1998, we would have seen that they were calves.

Brad's database also includes the `LOT_CONTAINS` table, with the following schema (again, we omit mention of some of the columns):

```
LOT_CONTAINS (LOT_ID_NUM, BKP_ID, A_NAME)
```

The primary key of this table is `LOT_ID_NUM`, so at any time, this value uniquely identifies one row, which records the backup identifier and application name for that lot.

Brad copies the files from the feed yard system, then later processes the information. The `LOT_CONTAINS` table stores for each lot the backup file from which the current information on that pen was extracted. Because this data tends to be dirty, containing inconsistencies and omissions, Brad would like to track the information in the `LOT_CONTAINS` table. In particular, he would like to reconstruct its state at any date in the past. He adds two columns, a `START_DATE` indicating when that row was first inserted into the table, and a `STOP_DATE` indicating when that row was updated or deleted. We should emphasize that rows are *logically* deleted, because physically deleting old rows would prevent past states from being reconstructed. A table that can be reconstructed as of a previous date is termed a "transaction-time state table," as it captures the transactions applied to the table.

As Table 2.8 shows, a row concerning lot 101 was first inserted on January 1, 1998. The `BKP_ID` was incorrect, and so was changed on February 5 from 17 to 18. A row concerning lot 433 was inserted on January 13 and is still considered to be correct (as signaled by a `STOP_DATE` of "forever").

The Tropical Year

The earth orbits around the sun, requiring a *tropical year* to return to the same point in space, which has been measured to take 365.2422 days. The fact that the tropical year is not an exact multiple of days, or even a simple fractional multiple of days, such as $365\frac{1}{4}$, has caused all manner of difficulty in designing calendars. Calendars are expected to be synchronized with both the rising of the sun and the seasons, and sometimes with the waxing and waning of the moon.

Table 2.8 The `LOT_CONTAINS` **table.**

LOT_ID_NUM	BKP_ID	A_NAME	START_DATE	STOP_DATE
101	17	'ADE'	1998-01-01	1998-02-05
101	18	'ADE'	1998-02-05	9999-12-31
433	23	'SMP'	1998-01-19	9999-12-31

While Tables 2.5 and 2.8 both have two DATE columns, the interpretation of these columns is dramatically divergent. Valid-time tables capture a history of reality, while transaction-time tables capture a history of the changing state of a table. We cannot ask the `LOT` table what its state was three days ago, but we *can* ask the `LOT_CONTAINS` table that question. Similarly, we cannot ask the `LOT_CONTAINS` table what was true in reality three days ago, but we can ask the `LOT` table that question. While any row of the `LOT` table may change, as we correct mistakes about the captured history, the `LOT_CONTAINS` table grows monotonically, with old rows remaining unchanged in perpetuity.

The most relevant query on a transaction-time state table is to reconstruct a past state. "Provide the state of the `LOT_CONTAINS` table on January 12, 1998" yields the following result:

LOT_ID_NUM	BKP_ID	A_NAME
101	17	'ADE'

Note that the `BKP_ID` for lot 101 was (erroneously) thought to be 17 on that Monday, and lot 433 hadn't yet arrived.

Now we ask, "Provide the state of the `LOT_CONTAINS` table on *February* 12, 1998," with the following result:

LOT_ID_NUM	BKP_ID	A_NAME
101	18	'ADE'
433	23	'SMP'

The `BKP_ID` for lot 101 is now the correct value of 18.

Only current modifications are permitted on transaction-time state tables, as past states cannot be changed. Modifications must permit subsequent reconstructions. The modification "Correct the backup identifier for lot 433 to 37" produces the result shown in Table 2.9.

Chapters 8 and 9 discuss transaction-time tables in detail, emphasizing various representations, some requiring only one timestamp column.

Table 2.9 The corrected backup identifier.

LOT_ID_NUM	BKP_ID	A_NAME	START_DATE	STOP_DATE
101	17	'ADE'	1998-01-01	1998-02-05
101	18	'ADE'	1998-02-05	9999-12-31
433	23	'SMP'	1998-01-19	1999-01-13
433	37	'SMP'	1999-01-13	9999-12-31

Table 2.10 The LOT bitemporal table.

LOT_ID_NUM	GNDR_CODE	FROM_DATE	TO_DATE	START_DATE	STOP_DATE
101	c	1998-01-01	9999-12-31	1998-01-03	1998-03-19
234	c	1998-02-17	9999-12-31	1998-02-17	9999-12-31
799	c	1998-03-12	9999-12-31	1998-03-12	9999-12-31
101	c	1998-01-01	1998-03-23	1998-03-19	9999-12-31
101	s	1998-03-23	9999-12-31	1998-03-19	9999-12-31

2.3   BITEMPORAL TABLES

Valid time, capturing the history of a changing reality, and transaction time, capturing the sequence of states of a changing table, are orthogonal, and can thus be separately utilized or applied in concert. A table supporting both is termed a "bitemporal table."

The LOT table is critical to Brad's epidemiological analysis, so he also tracks the changes made to this table. This table already supports valid time; he adds two columns, START_DATE and STOP_DATE, to capture transaction time.

Table 2.10 has four timestamps, befitting its bitemporal nature. There is a wealth of information in such tables, if care is taken in reading them. Let's examine this table row by row.

- *Row 1:* On January 3 (the START_DATE), the fact that lot 101, a group of calves, arrived in the feed yard two days previously, on January 1 (the FROM_DATE), is recorded. The valid time for this fact is January 1 to forever (the TO_DATE), indicating that they are expected to remain calves. We'll return to the STOP_DATE when we discuss the fourth row.
- *Row 2:* On February 17 (the START_DATE), the fact that lot 234, also a group of calves, arrived in the feed yard that day (the FROM_DATE) is recorded. A STOP_DATE of forever indicates that the fact is still considered to be correct.
- *Row 3:* On March 12, the fact that lot 799, a group of calves, arrived in the feed yard that day is recorded.

Table 2.11 The history as known on March 15.

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

Table 2.12 The history as known on April 1.

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

- *Row 4:* On Thursday, March 19, unbeknownst to the cattle in lot 101, these cattle were scheduled to be steered early the next week, on Monday, March 23. So we logically update the first row by setting its STOP_DATE to the current date, insert row 4, indicating that lot 101 consisted of calves from January 1 to March 23, and insert row 5.
- *Row 5:* On Thursday, March 19, the fact that lot 101 is a collection of steers from March 23 to forever was recorded, and that fact is still considered correct.

Since this table supports transaction time, we can reconstruct its state in the past. "Provide the history of the LOT table as best known on March 15, 1998" (the Ides of March, beware!) would generate the result shown in Table 2.11. As of March 15, we hadn't yet scheduled lot 101's steering. "Provide the history as best known on April 1" yields a different result (Table 2.12).

Interactions between valid and transaction time are especially interesting, as in "When were steerings scheduled (as opposed to being recorded after the fact)?" which would identify one such steering:

LOT_ID_NUM	When_Scheduled	When_Recorded
101	1998-03-23	1998-03-19

As bitemporal tables include transaction time, all modifications are transaction-time current. However, we can still provide the period of applicability for modifications, as in "Lot 234 will be absent from the feed yard for the first three weeks of October."

Chapter 10 explores the glorious expressiveness of bitemporal tables, as well as the intricacies of expressing queries and modifications on such tables.

2.4   SUMMARY

This chapter has introduced *what* we want to do with time-varying tables and has provided a glimpse of *how* to do them: add one or more timestamp columns. Chapters 3 through 10 furnish the intellectual tools to code applications in SQL over temporal tables.

On the inside front cover, a *Concept Map* provides guideposts for our journey through the triad of triples. In this map, the sections that explicate each concept are listed in italics, following the concept.

The three temporal data types—instants, intervals, and periods—are covered first. Valid-time state tables are the focus of Chapters 5–7. Chapter 5 considers how such tables may be specified in the schema; integrity constraints are used heavily, as the existing SQL constructs of UNIQUE, PRIMARY KEY, and FOREIGN KEY are inadequate for time-varying tables.

The three kinds of queries—current, sequenced, and nonsequenced—are the topic of Chapter 6; the analogous kinds of modifications are examined in Chapter 7.

Chapters 8 and 9 consider transaction-time tables, emphasizing the critical distinction between valid time and transaction time (SQL unfortunately completely blurs this distinction). Chapter 8 considers instant-stamped tables, and Chapter 9 considers period-stamped tables.

Chapter 10 introduces bitemporal tables, supporting both valid and transaction time. Again, we delve into the intricacies of defining, querying, and modifying these tables.

We then return to Brad's feed yard application in Chapter 11, as a thorough review of these strategies. Finally, Chapter 12 indicates where future versions of SQL are headed vis-à-vis temporal support and shows that constructs proposed for SQL3 offer a dramatic reduction in both the number of lines of SQL code and the mental gymnastics required, thereby ending this exploration on an optimistic note.

We now have sufficient background to understand the metaphor of the cover illustration. The sphere on the cover is machined in such a way that it projects shadows of three different clock faces. The sphere represents a fact in a bitemporal table, say, an employee table. The shadow

Hours

A day is demarcated by a physical change, the peaking of the sun in the sky, indicating noon. Not so for an hour; it is a purely arbitrary division. The ancients studied the stars closely and knew that the sun crossed 12 constellations: Aquarius, Pisces, Aries, Taurus, Gemini, Cancer, Leo, Virgo, Libra, Scorpio, Sagittarius, and Capricorn. This sequence is called the zodiac, from the Greek word *zodios*, meaning "figure of an animal." The Chaldeans thus divided the day (that is, the time between sunrise and sunset) into 12 portions, or hours. However, because the daylight is shorter in winter than in summer, these were considered *horae temporariae*, or "temporary hours."

on the left shows the time 10:25 A.M., indicating that the fact became true in reality (the valid FROM_TIME) in midmorning. The shadow on the right shows the time 12:35 P.M., indicating that the fact was stored in the database (the transaction START_TIME) about a half hour after noon. Part of the fact is the time of birth (a user-defined time) of 6:05 P.M. We see that a particular fact in the database may include a user-defined time and may be associated with both a valid time and a transaction time.

2.5 READINGS

Other terms have been applied to the valid-time, transaction-time, and bitemporal tables introduced in this chapter. They have been called *temporal tables*. The term *time-varying* has been used, but this is a misnomer, as *all* tables in practice vary over time, as rows are added, removed, and changed. The term *time-oriented tables* is also not quite precise; just what does it mean to be "time-oriented"? (For that reason, the title of this book is unfortunate. To be honest, I originally preferred *Developing Temporal Database Applications in SQL,* but felt that title might confuse people who did not know the technical definition of "temporal," which no longer characterizes you, gentle reader.) Such tables have also been called *historical tables*, but this implies that they record information only about the past. Valid-time tables often store information about the future, for example, in planning or scheduling applications. The accepted terminology then is to refer to such tables as temporal tables, or more specifically as, say, a valid-time table.

The official definition of a temporal database is "a database that supports some aspect of time, not counting user-defined time" [49]. The intuition here is that adding a user-defined time column such as birth date to an employee table does not render it temporal, especially since the birth date 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 information managed by the enterprise.

It is perhaps surprising that the discipline of 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; several hundred now appear each year. Many are included in the most recent bibliography, which has pointers to six prior bibliographies over the past 17 years [105]. Three brief surveys [56, 76, 103] provide entry into this research field. The most complete exposition, albeit somewhat dated by now, is Tansel et al. [102]; a more recent text provides an updated summary [107].

The cover illustration was inspired in part by the cover of Hofstadter's *Gödel, Escher, Bach* [40], which showed two pieces of wood carved to project shadows of the letters *G*, *E*, and *B* on the three planes.
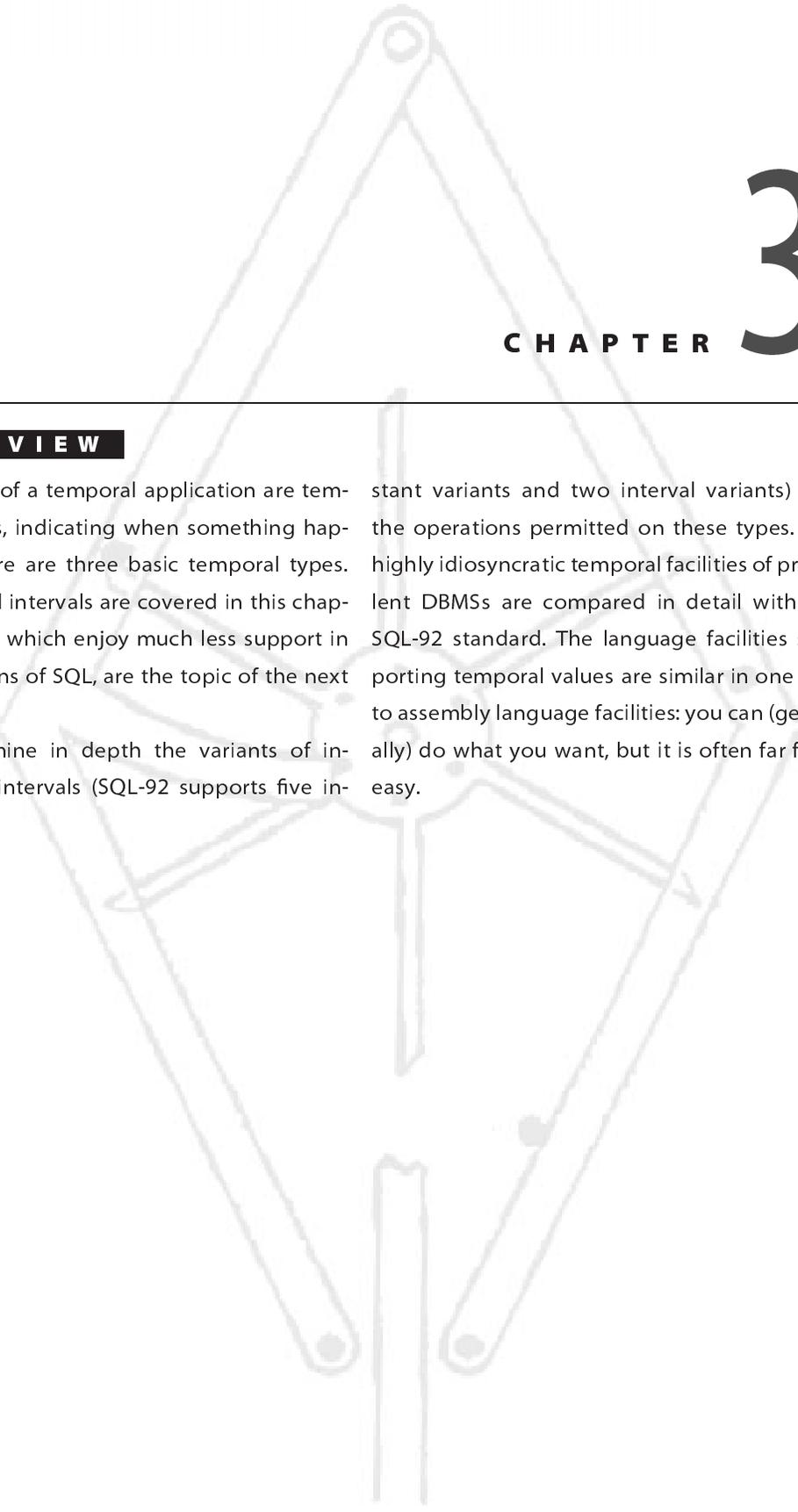
OVERVIEW

At the core of a temporal application are temporal values, indicating when something happened. There are three basic temporal types. Instants and intervals are covered in this chapter; periods, which enjoy much less support in most versions of SQL, are the topic of the next chapter.

We examine in depth the variants of instants and intervals (SQL-92 supports five instant variants and two interval variants) and the operations permitted on these types. The highly idiosyncratic temporal facilities of prevalent DBMSs are compared in detail with the SQL-92 standard. The language facilities supporting temporal values are similar in one way to assembly language facilities: you can (generally) do what you want, but it is often far from easy.

Instants and Intervals

Jim Barnett is the quintessential Texan: barrel-chested, sporting a thick mustache, a graduate of the University of Texas, an oil man. His speech has an easy cadence, peppered with humor. He is an engineer for GeoQuest, a data management company owned by Schlumberger, a Paris-based instrumentation company. (This name is of Germanic origin, but is pronounced as a French word.)

The oil and gas business is a dynamic, worldwide industry, with its practitioners transferred far and wide, and often. September of 1995 found Jim working in Dubai, U.A.E., 8000 miles from his home base in Houston, Texas, working with the Arabian Oil Company (AOC) to systematize its records on wells and oil and gas production and distribution. Several of his clients are in Al Khafji, a company town immediately south of the Saudi border with Kuwait. Al Khafji saw action in the Gulf War, with the AOC workers leaving scant hours before the town was invaded.

Jim helped design the database underlying GeoQuest's FINDER product. FINDER implements the industrial standard Public Petroleum Data Model (PPDM) via some 300 tables on the Oracle DBMS.

The enterprise (here, wells, production, land, lithography, and seismic data) must be modeled using the available data types, such as numerics, character strings, and dates. Character strings record the names of things, numerics record the values that have been measured or noted, and dates record the when of things.

As this book considers the time-varying nature of data, we focus here on the date columns. FINDER utilizes all manner of dates. Many tables have `Start_Date` and `End_Date` columns, denoting a period of time; we will examine this usage in detail in the next chapter. Other tables have `Start_Date` and `Next_Event_Date` columns, recording a succession of events.

FINDER also uses other approaches to capture time-varying data. The `Fac_Daily_Prod` table tracks daily production of a facility. Each row of this table records a month's worth of production. The `PRODUCTION_YEAR` column (of type `NUMBER(4,0)`) and `MONTH` column (of type `VARCHAR(3)`) denote the particular month, and 31 columns, `DAY1` through `DAY31`, of type `NUMBER(12,4)`, provide the daily production.

Duration data is also present in the FINDER schema. In the `Well_Core_Hdr` table, the `Dry_Time` column (of type `NUMBER(7,2)`) and the `Dry_Time_Unit` (of type `VAR-CHAR(12)`) in concert capture the drying time of the chemical analysis of a well core (a sample extracted from a known depth from the well).

Jim found that addressing the AOC requirements involved adding even more date columns. To most tables he added the following columns: `Created_By`, `Create_Date`, `Updated_By`, and `Last_Update`, to track more carefully who effected a change and when the change was made. Other temporal columns were needed by particular tables, such as the `Sample_Date`, `Dispatch_Date`, and `Return_Date` columns of the `Well_Core_Sample` table.

SQL defines several temporal types for use in columns. Any respectable DBMS provides similar data types, though few compliant with the standard. SQL also provides useful predicates, constructors, and functions for manipulating time values. Again, DBMSs include somewhat similar, though usually incompatible, operators. This chapter summarizes the temporal support that SQL and prevalent DBMSs provide, and shows how to use these facilities to perform common tasks.

Jim's task was made easier (or perhaps more difficult) by the fact that Oracle supports but one temporal data type, DATE, of a fixed granularity, to a second. This often shifted the decision from which temporal type was best to which other available type, for example, NUMBER, should be used.

As in all data modeling, the first question that must be asked is, What is the *semantics*, that is, the *meaning*, of the enterprise to be captured? In this sense, the `Dry_Time` column of the `Well_Core_Hdr` table is of a fundamentally different nature than the `Core_Date` column of that table, even if both are to a precision of seconds. And both are fundamentally different than the `Start_Date` and `End_Date` columns. How these columns are typed and correctly manipulated in SQL depends critically on determining their underlying semantics. In this and the next chapter, we examine the different temporal semantics that are available, and explore how Jim made these distinctions when specifying the AOC extensions to the FINDER data model.

3.1   INSTANTS

An instant is an anchored location on the time line. An SQL-92 datetime denotes an instant.

An *instant* is an anchored location on the time line. I am writing this on the instant of 2:38 P.M., January 14, 1997, two days after HAL's birth. (From *2001: A Space Odyssey:* "I am a HAL 9000 computer, production number 3. I became operational at the HAL plant in Urbana, Illinois, on January 12, 1997.") An instant occurs but once, and then is forever in the past.

This data type is most fundamental. Other types can be implemented by, or simulated to some degree with, instants; indeed, most DBMSs provide no other temporal data types.

SQL terms instants *datetimes* and provides three specific forms and two variations.

3.1.1   The DATE Type

An SQL-92 DATE stores the year, month, and day values of an instant. The year value must be in the range 0001 C.E. (Common Era, formerly called A.D.) through 9999 C.E. Note that a DATE value cannot denote B.C.E. (Before the Common Era, formerly called B.C.) dates. While the SQL designers point to some technical issues in justifying this design decision (such as there being no year 0 C.E., or year 0 B.C.E.), its true rationale may lie in initial use of relational products primarily in administrative data processing rather than in scientific applications. Both uses could have been accommodated much better by centering this 10,000-year range at, say, 1 C.E., rather than favoring the years 5000 C.E. through 9999 C.E. over B.C.E. dates.

The month value is limited to the values 1 through 12, denoting the 12 Gregorian months. The day value is limited to the values 1 through 31, although the month and year value can apply additional restrictions limiting the maximum to 28, 29, or 30. For example, February 29, 1996, is legal, as is February 29, 2000, but February 29, 1900, is not. None of these fields can be negative. This notation is derived from the ISO 8601 standard.

Date literals consist of the year as four digits, followed by a hyphen, followed by the month as two digits, followed by a hyphen, followed by the day as two digits, in descending granularity (thereby presumably allowing less-than comparisons to be implemented via lexicographic comparisons). HAL's birth date is then `DATE '1997-01-12'`. Note that this literal requires 10 characters. The length of a DATE is specified as 10 *positions*, which is defined as "the number of characters from the character set `SQL_TEXT` that it would take to represent any value" in the DATE type. SQL does not prescribe what internal format an implementation employs for such values.

A.D. versus B.C.

There is no 0 A.D.; 1 A.D. follows 1 B.C. The reason for this seeming anomaly is that when the bifurcation into B.C. and A.D. was proposed, by a sixth-century monk named Dionysius Exiguus, under instruction by Pope St. John I, the concept of zero had not been invented. That epic event would have to wait several centuries, first for Arabic mathematicians to devise the notion of zero as a placeholder and as a value unto itself, then for farsighted Pope Sylvester II, reigning over the last millennial transition from 999 to 1003, to advocate this concept in Western usage.

3.1.2   The TIMESTAMP Type

Should the user desire a finer precision than day, the TIMESTAMP data type is available in SQL. This variation stores the year, month, and day, as in DATE, along with the hour, minute, and second, and a number of fractional digits of the second. The default is six fractional digits, corresponding to microseconds. None of the fields may be negative.

The timestamp's *precision*, or number of fractional digits, can be specified in parentheses when this data type is used; the precision defaults to 6. A precision of 3 (i.e., `TIMESTAMP(3)`), indicates a granularity of milliseconds; a precision of 0, seconds; a precision of 15, femtoseconds. The maximum precision is defined by the implementation; a negative precision is not allowed. Twenty-four hour clock time is used, so the hour value ranges from 0 to 23. The minute value ranges from 0 to 59, and the second value from 0 to 61 (more on this shortly).

SQL uses Coordinated Universal Time (UTC), based on atomic clocks.

The time portion of a timestamp literal is denoted in descending granularity: hour, minute, second, each two digits and separated with colons, followed by a period and fractional digits, if the precision is greater than zero. Hence the present time, as near as I can tell from my watch, is `TIMESTAMP '1997-01-15 11:35:29.123456'`. The length of a TIMESTAMP value is 26 positions (the length includes the period character); the length of `TIMESTAMP(0)` is 19 positions.

3.1.3   The TIME Type

Will there really be a morning?
Is there such a thing as day?
.   .   .   .
Oh, some scholar! Oh, some sailor!
Oh, some wise man from the skies!
Please to tell a little pilgrim
Where the place called morning lies!

—Emily Dickinson, "Will there really be a morning?"

The SQL-92 datetime types DATE, TIME, and TIMESTAMP differ in the fields (year, month, day, hour, minute, and second) they contain.

The SQL TIME data type stores the hour, minute, and second, and a number of optional fractional digits of the second. The default is no fractional digits, corresponding to integral seconds; a nonzero precision is denoted, as with TIMESTAMP, in parentheses.

TIME literals are as one would expect: in descending granularity, separated with colons (e.g., `TIME '11:35:29'`). The length of a TIME value is eight positions; if the precision is nonzero, then it is nine positions (for the decimal point) plus the precision. Unlike DATE

and TIMESTAMP values, TIME values include a zero element: '00:00:00'.

As we will see in Section 3.7, and as hinted in Emily Dickinson's poem, TIME is not really an instant data type at all; it is a funny kind of interval (to be discussed below), representing a number (between 0 and 86,400) of seconds (along with optional fractional seconds).

3.1.4   Time Zone Variants

Greenwich Mean Time ensures that the sun, on those days it is visible, is directly overhead Greenwich, England, each noon. Locales distant from England, shift UTC by a certain number of hours and minutes so that the sun is approximately overhead locally at noon. Mountain Standard Time subtracts 7 hours from UTC. We would expect 24 time zones, each corresponding to 60 minutes of longitude, but as with all things political, there are many exceptions. Nepal's time zone is 15 minutes off from India's as an expression of independence. Many locales also change the offset, advancing their clocks by one hour in the summer and turning them back in the winter, at specified days. Arizona, unlike the other states in the Mountain Time Zone, does not adopt this adjustment, called daylight saving time, presumably as an expression of independence from the federal government. The Navajo Indian reservation, located within Arizona, does use daylight saving time, perhaps to be different than Arizona. And the Hopi Indian reservation, which is completely surrounded by the Navajo Indian reservation, does not adopt daylight saving time, perhaps to differentiate themselves from the Navajos. So you can drive a few hours in Arizona and go in and out of daylight saving time four times.

The time zone can be stored with SQL-92 TIME and TIMESTAMP values.

Each SQL session has an associated default offset from UTC that is used in that session. This offset can range from -12:59 to +13:00 (the reason for the additional hour on each side is daylight saving time). The offset is assumed for TIME and TIMESTAMP values manipulated within the SQL session. Hence, time literals denote the local time, whereas times are stored as UTC time (with no time zone, i.e., Greenwich Mean Time).

The TIME WITH TIME ZONE data type includes with the stored value an explicit offset from UTC. This is written as a sign (the hyphen character for a minus sign, or the plus character) followed by the offset hour as two digits, a colon, and the offset minute as two digits (e.g., `TIME '11:08:27-07:00'`). This added information requires an additional six positions: four digits, a hyphen, and a colon. Fractional seconds appear before the time zone (e.g., `TIME '11:08:27.123456-07:00'`).

The TIMESTAMP WITH TIME ZONE data type is also available. Without fractional digits, the length of this type is 25 positions, more with fractional digits. An example is `TIMESTAMP '1997-01-15 11:35:29.123456-07:00'`, requiring 32 positions.

3.2   INTERVALS

> …and an ocean tumbled by with a private boat for Max
> and he sailed off through night and day
> and in and out of weeks
> and almost over a year
> to where the wild things are.
>
> —Maurice Sendak, *Where the Wild Things Are*

> And which of you by being anxious can add one cubit to his span of life?
>
> —Matthew 6:27

An interval is an unanchored, directional duration of the time line.

An *interval* is an *unanchored* contiguous portion of the time line. An interval is relative; an instant is absolute. An interval can be added to an instant, yielding another instant. Intervals cannot be added to spatial points, nor spatial intervals (such as cubits) to temporal intervals, except as (often highly effective) literary devices, as the above quotes illustrate.

The distance between two instants is an interval. Unlike instants, intervals have direction. An interval can be positive or negative, denoting a shift to the future or to the past.

Intervals are less prominent in the FINDER schema than instants. Some are signaled with "duration" or "interval" in their name, examples being the `Period Durtn` column in `Well_Test_Period` and the `Sampling_Intvl` column of the `Seil Survey_Hdr` table, or by mentioning the time during which something was happening, as in the `Time_String_In_Hole` column of the `Ventilates_Seay-nr` table. Other interval columns are more obscure, such as the `Income Time` column of the `Stage Flowback` table (other columns having a name with that suffix denote instants, e.g., `Start_Time`). It appears that in the FINDER schema, all intervals are positive. As we'll see, while SQL has an interval type, relying on the designer to differentiate instants from intervals in other ways.

3.2.1   The INTERVAL Type

Intervals have a qualifier that specifies the leading field, an optional trailing field, and an optional precision for the leading and trailing fields.

The SQL-92 interval type is complex. Whereas the other SQL types require but a few lines to describe, intervals require over three *pages* just to specify the syntax. Even then, some details are left unstated (as will be discussed in Section 3.7.3).

SQL differentiates year-month intervals and day-time intervals. The first can be considered to be equivalent to an integral number of years or months; the latter considered equivalent to an integral number of days, hours, minutes, seconds, or fractions of a second. This distinction is due to varying month lengths in the Gregorian calendar. The individual units (months, hours, microseconds) are termed *granules*, so an interval value is a (signed) integer number of granules.

> Solid stone is just sand and water,
> Sand and water, and a million years gone by
>
> —Beth Nielsen Chapman, "Sand and Water"

Intervals are combinations of the fields year, month, day, hour, minute, and second, though not all combinations are

Sundials

A sundial, or more ostentatiously, a *heliochronometer*, in contrast to most other clocks, does not measure an interval of time; rather, it indicates a given instant of time. A sundial can be moved to another longitude and remain accurate; a mechanical watch must be reset when moved—hence the presence of multiple time zones on many modern watches.

Sundials, when adjusted for the correct latitude, are exceedingly accurate, measuring true solar time (see page 95), at least when the sky is not cloudy. There is no drift with a sundial, unlike mechanical clocks.

allowed, as we will see. Intervals have a qualifier associated with them that specifies the leading field, an optional trailing field, and an optional precision for the leading and trailing fields. If no trailing field is present, the interval contains only the leading field.

3.2.2 Year-Month Intervals

For year-month intervals, the only fields available are year and month. Such an interval can contain only years (INTERVAL YEAR), only months (INTERVAL MONTH), or both (`INTERVAL YEAR TO MONTH`). For the leading or only field, a precision, specifying the maximum number of digits, is permitted (`INTERVAL YEAR(`p`)`, `INTERVAL MONTH(`p`)`, `INTERVAL YEAR(`p`) to MONTH`); the precision defaults to two digits and must be positive. Nonleading fields can have up to two digits.

Year-month intervals contain a year, a month, or both fields.

Year-month literals are denoted with the year (e.g., `INTERVAL '3' YEAR`), the month (e.g., `INTERVAL '7' MONTH`), or the year followed by a hyphen followed by the month (e.g., `INTERVAL '3-7' YEAR TO MONTH`, for three years and seven months). Note that the fields must be specified in literals, but the precision need not be. If years and months are present, then the number of months must be between 1 and 12. The length, in positions, of a year-month interval is the precision of the year field if alone, the precision of the month field if alone, or the precision of the year field plus three, for the hyphen and two digits of the month, if both are present.

The hyphen in intervals is *not* a minus sign; it serves instead to separate field values. An interval literal can have a sign preceding the quoted portion. A positive interval literal is indicated by the absence of a sign or by a plus sign (e.g., `INTERVAL '3-4' YEAR TO MONTH = INTERVAL +'3-4' YEAR TO MONTH`). A negative interval literal is indicated with a minus sign (a hyphen) preceding the string portion of the literal (e.g., `INTERVAL -'3-7' YEAR TO MONTH`, for three years and seven months going back into the past). The one exception is the zero element, for which positive and negative literals denote the same value: `INTERVAL +'0-0' YEAR TO MONTH = INTERVAL '0-0' YEAR TO MONTH = INTERVAL -'0-0' YEAR TO MONTH`.

We note in passing that the Technical Corrigendum 3, currently in draft form, permits a sign to *also* appear within the quoted portion. In fact, two signs can be present, with the normal mathematical interpretation—for example, double negation results in a positive literal. Hence, `INTERVAL +'3-4' YEAR TO MONTH = INTERVAL '+3-4' YEAR TO MONTH = INTERVAL +'+3-4' YEAR TO MONTH = INTERVAL -'-3-4' YEAR TO MONTH = INTERVAL '3-4' YEAR TO MONTH`.

3.2.3   Day-Time Intervals

> The gods confound the man who first found out
> How to distinguish hours. Confound him, to,
> Who in this place set up a sundial,
> To cut and hack my days so wretchedly
> Into small pieces!

—Plautus (quoted by David S. Landes), *Boeotia*

Day-time intervals may contain up to four fields: day, hour, minute, and second, with optional fractional seconds. All fields between the leading and trailing fields are included. Hence, `INTERVAL DAY TO SECOND` contains four fields, while `INTERVAL DAY TO HOUR` contains two fields, and `INTERVAL DAY` (or, equivalently, `INTERVAL DAY TO DAY`) contains only one field. As with year-month intervals, we can specify a precision for the leading field, which defaults to two digits. Examples include `INTERVAL DAY(4) TO HOUR`, which can represent up to 9999 days, and up to 24 hours; `INTERVAL HOUR(3) TO SECOND`, which can represent up to approximately 40 days; and `INTERVAL MINUTE(4) TO SECOND`, which can represent almost a week (within a few minutes), to the granularity of seconds.

> Day-time intervals contain day, hour, minute, and second fields, in any contiguous sequence.

Day-time interval types and literals are even more complex when seconds are involved because the standard wished to accommodate fractional seconds (no other field can have a fractional value). If the leading (i.e., only) field is SECOND, then it can have a precision, which defaults to two digits (e.g., `INTERVAL SECOND(8)`, which can represent three years). (A bit of trivia: there are $\pi \times 10^7$ seconds in a year, to

an accuracy of greater than 1 in 100.) If the trailing (or only) field is SECOND, it can also have a fractional precision, which defaults to six (e.g., `INTERVAL DAY(3) TO SECOND(3)`, which represents a count of milliseconds). A single SECOND field can thus have two precisions, separated with a comma (e.g., `INTERVAL SECOND(5,3)`, which can represent milliseconds up to a little more than a day). Such intervals require nine positions, including the period.

Day-time literals are just what you might expect, making the correspondence with timestamp literals (e.g., `INTERVAL '1 23:45:12' DAY TO SECOND`). In all cases, the length in positions of an interval type is identical to the number of characters required by any literal of that type.

3.3 PREDICATES

For such a diverse set of types (DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE, and two variants of INTERVAL: year-month and day-time), SQL-92 supports only four classes of temporal predicates: equality, less-than, is null, and overlaps.

There are several variants of the equality predicate; these variants apply to all the data types. When applied to two expressions, '=' determines whether the values of these expressions are identical. When applied to a value and a set of values (of the same type), `=ANY` determines if the left-hand value is identical to at least one of the values in the right-hand set. `=SOME` and `IN` are nonorthogonal equivalents. `MATCH` also relies on equality testing. The queries in CF-3.1 are identical in meaning.

Code Fragment 3.1 **Seven ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE BirthDate = DATE '1970-01-01'
SELECT * FROM Employee
WHERE BirthDate =ANY (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE BirthDate =ALL (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE BirthDate =SOME (VALUES ((DATE '1970-01-01')))
```

```
SELECT * FROM Employee
WHERE BirthDate IN (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE NOT BirthDate NOT IN (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE BirthDate MATCH (VALUES ((DATE '1970-01-01')))
```

Here, VALUES constructs a table with one row consisting of one column. More variations are possible using the UNIQUE and PARTIAL reserved words available with MATCH. (We mention MATCH for completeness. This construct, particularly with its options, is intended for determining whether or not candidate rows would satisfy referential integrity constraints.)

In all of the above examples, the two values being compared are of a specific type (DATE). Two datetimes can be compared if they are *comparable*, which is defined as having the same fields. Intervals are compared by first converting to a common base granularity, then converting to integers, then doing the integer comparison. So `INTERVAL '3-7' YEAR TO MONTH` can be compared to `INTERVAL '43' MONTH` (and in fact these two intervals are equal), while neither of these intervals can be compared with `INTERVAL '23' DAY`, as the two intervals are incomparable.

Since every SQL-92 data type, including the temporal types, is ordered, less-than is defined on them all. The operators '<', '<=', '>', '>=', and '<>' comprise the available combinations. Each combination is a disjunction, OR-ing the two possibilities, so '<=' means "less than or equal to." The last, '<>', means "less than or greater than," or equivalently, "not equal to."

Code Fragment 3.2   **Four more ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE NOT BirthDate <> DATE '1970-01-01'

SELECT * FROM Employee
WHERE NOT BirthDate <>ANY (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE NOT BirthDate <>ALL (VALUES ((DATE '1970-01-01')))

SELECT * FROM Employee
WHERE NOT BirthDate <>SOME (VALUES ((DATE '1970-01-01')))
```

There are yet other ways to test for equality of temporal values in SQL; the following discussion will provide more than a dozen.

The BETWEEN construct is a useful form of inequality. The predicate

```
value₁ BETWEEN  value₂ AND  value₃
```

is equivalent to

```
value₂ <=  value₁ AND  value₁ <=  value₃
```

Note that the BETWEEN predicate is ordered, in that $value_2 \leq value_3$ is required.

Code Fragment 3.3 **Two more ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE BirthDate BETWEEN DATE '1970-01-01' AND DATE '1970-01-01'

SELECT * FROM Employee
WHERE NOT BirthDate NOT BETWEEN
        DATE '1970-01-01' AND DATE '1970-01-01'
```

These exploit the fact that equality is allowed on both sides of the BETWEEN.

As with other data types, the value of any temporal column can be NULL. And as with other data types, predicates on null temporal values have the value *unknown*, except for *value* IS NULL, which returns true when the *value* is null and false otherwise, and *value* IS NOT NULL, which naturally returns true if the *value* is not null.

The final temporal predicate, OVERLAPS, differs from the rest, in that it only applies to temporal values, and then only to values of particular temporal types. As we'll see in Chapter 4, OVERLAPS is a way to get periods in the back door.

The format of this predicate is

```
period information₁ OVERLAPS  period information₂
```

Either *period information* is constructed either via

```
( start time,  duration)
```

or

```
( start time,  end time)
```

where *start time* and *end time* are instants, that is, SQL datetimes, and *duration* is an interval that can be added to *start time* (we'll cover adding intervals to datetimes in more detail in the next section). These two forms can be mixed and matched at will.

The predicate returns true if *period information₁* overlaps *period information₂*, that is, if they share at least one instant, or, equivalently, if the start of *period information₁* is less than the end of *period information₂* and the start of *period information₂* is less than the end of *period information₁* (try it!). By

using a zero duration, or identical start and end instants, we can construct periods of one granule, as the following illustrate.

Code Fragment 3.4   **Yet four more ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE (BirthDate, INTERVAL '0' DAY)
    OVERLAPS (DATE '1970-01-01', INTERVAL '0' DAY)

SELECT * FROM Employee
WHERE (BirthDate, BirthDate)
    OVERLAPS (DATE '1970-01-01', INTERVAL '0' DAY)

SELECT * FROM Employee
WHERE (BirthDate, INTERVAL '0' DAY)
    OVERLAPS (DATE '1970-01-01', DATE '1970-01-01')

SELECT * FROM Employee
WHERE (BirthDate, BirthDate)
    OVERLAPS (DATE '1970-01-01', DATE '1970-01-01')
```

NULL can be used in either position within a *period information*; often the predicate will return true (or false) anyway.

Code Fragment 3.5   **Three more ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE (BirthDate, NULL)
    OVERLAPS (DATE '1970-01-01', INTERVAL '0' DAY)

SELECT * FROM Employee
WHERE (BirthDate, NULL)
    OVERLAPS (DATE '1970-01-01', NULL)

SELECT * FROM Employee
WHERE (BirthDate, NULL)
    OVERLAPS (NULL, DATE '1970-01-01')
```

3.4   CONSTRUCTORS

A *temporal constructor* is an expression that returns a temporal value. (Some might consider a predicate to be a boolean constructor, but we find it helpful to differentiate predicates and other constructors.)

The Gregorian Calendar

The Gregorian calendar was necessitated by the fact that a year is not an integral number of days. The tropical year is roughly 365.242191 days, or equivalently, 365 days, 5 hours, 48 minutes, and 45.96768 seconds. ("Nature, apparently, can make a gorgeous hexagon, but she cannot (or did not deign to) make a year with a nice even number of days or lunations" [35, p. 137].) The Julian calendar starts off with 12 months of various lengths that add up to 365 days. It then makes a correction of imposing a leap day every fourth year, as a sequence of 365, 365, 365, 366 days, averaging out at 365.25 days per year. This is pretty close: it makes the year about 11 minutes and 154 seconds longer than it actually is. But 11 minutes a year can add up, and the civil calendar got more and more out of step from the solar calendar. By 1581, the vernal equinox was on April 2, rather than the accepted March 21. So Pope Gregory XIII appointed a committee, with the Jesuit mathematician Christopher Clavius as chair. His committee came up with two solutions, both imposed by Pope Gregory in a papal bull issued on February 24, 1582. First, to get the civil and solar calendars back in sync, 10 days, October 5 through 14, 1582, were simply dropped—they never existed! Second, the definition of leap years (a year divisible by 4) was amended to not include a century year (multiple of 100), but to still include years divisible by 400. So, every 25th leap year was removed, but every 100th was restored. 1900 is not a leap year, but 2000 is, a fact still misunderstood by some software packages. This yields the Gregorian year to be 365.2425 days long, departing from the solar calendar by some 25.96 seconds: pretty darned close! At this rate, a discrepancy of one day accumulates every 2800 years or so.

3.4.1 Datetime Constructors

SQL provides seven constructors returning datetimes (DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, and TIMESTAMP WITH TIME ZONE). We discuss each in turn, after providing an example.

- `DATE '1996-02-24' + INTERVAL '7' DAY` This expression evaluates to `DATE '1996-03-02'`, as 1996 was a leap year. The instant is shifted forward (or back, for negative intervals) by the length of the interval. For expressions involving an interval and a datetime, the interval must contain only fields that are also contained in the datetime. `DATE '1996-02-24' + INTERVAL '12:30' HOUR TO MINUTE` is thus disallowed, as is `DATE '1996-02-24' + INTERVAL '2 12' DAY TO HOUR`.
- `INTERVAL '7' DAY + DATE '1996-02-24'` This expression also evaluates to `DATE '1996-03-02'`, as addition of intervals and datetimes is commutative.
- `DATE '1996-03-02' - INTERVAL '7' DAY` This expression evaluates to `DATE '1996-02-24'` and is not commutative.
- `TIMESTAMP '1996-02-24 12:34:56' AT LOCAL` This expression assumes that the value is expressed in terms of GMT and applies the local time zone offset

to get the local time. As Tucson, Arizona is always at Mountain Standard Time (MST), seven hours behind Greenwich, this evaluates to TIMESTAMP '1996-02-24 19:34:56'. As another example, TIMESTAMP '1996-02-24 12:34:56+02:00' AT LOCAL takes the Danish TIMESTAMP WITH TIME ZONE, specifically, at Mean European Time with daylight saving (MET DST), normalizes it to UTC (i.e., 10:34:56), then applies the Tucson, Arizona time zone offset, yielding TIMESTAMP '1996-02-24 03:34:56'. This construct may not be applied to DATE values. If we use a value without specifying an AT clause, AT LOCAL is assumed.

- TIMESTAMP '1996-02-24 12:34:56' AT TIME ZONE INTERVAL '-7:00' HOUR TO MINUTE   The expression allows the user to specify a particular time zone offset, which must be an hour to minute interval. It returns TIMESTAMP '1996-02-24 19:34:56'. As with the previous example, this construct may not be applied to DATE values.

- CURRENT_DATE returns the current date (the date of the current instant). CURRENT_TIME and CURRENT_TIMESTAMP function analogously. All such so-called datetime value functions within a statement are effectively performed simultaneously. Such functions appearing in two separate statements are allowed to return different results.

- CAST('1996-02-24' AS DATE)   The CAST function converts a value in a source data type (here, CHARACTER) to the specified target data type. When the target data type (here, DATE) is a temporal type, then the cast may be regarded as a temporal constructor. In this case, the function returns DATE '1996-02-24'. While only character strings may be cast to (and from) datetimes in SQL-92, products often extend this to integers and other types.

The following types can be converted to a datetime value.

- CHARACTER   A character string can be converted to a DATE, TIME, or TIMESTAMP value. The string must be identical to a literal of the datetime type. The example above converts a character string to a DATE. CAST('12:34:56' TO TIME) is another example.

- TIME   A time value may be converted to a TIME or TIMESTAMP value, the latter filling in the year, month, and day with the value of CURRENT_DATE. If the target type has a time zone, then these fields are set to the current time zone of the session. This is being written on Wednesday, July 23, 1997. CAST(TIME '12:34:56' AS TIMESTAMP WITH TIME ZONE) results in TIMESTAMP '1997-07-23 12:34:56-07:00'. If the target has a smaller precision than the source, the additional digits are discarded. If the target has a greater precision than the source, the needed digits are set to 0. Hence, CAST(TIME '12:34:56.123' AS TIME(6)) results in TIME '12:34:56.123000'; casting this value to TIME(1) results in TIME '12:34:56.1'.

- TIMESTAMP   A timestamp value may be converted to a DATE, TIME, or TIME-STAMP value, by extracting the requested fields and adjusting, if necessary, the precision. Hence, `CAST(TIMESTAMP '1997-07-23 12:34:56.123' AS TIME(6))` results in `TIME '12:34:56.123000'`; casting this value to DATE results in `DATE '1997-07-23'`.
- DATE   A date value may be converted into a DATE by simply copying the value or into a TIMESTAMP by setting the hour, minute, and second to 0. `CAST(DATE '1997-01-01' AS TIMESTAMP(4))` yields `'1997-01-01 00:00:00.0000'`.

Note that when a TIME value is cast to a TIMESTAMP, the current date provides the missing fields, but when a DATE value is cast to a TIMESTAMP, the missing fields are set to zero.

3.4.2   Interval Constructors

SQL provides a variety of constructors that return year-month or day-time intervals. We summarize all but the cast function, which warrants closer scrutiny.

- `INTERVAL '3' DAY + INTERVAL '4' DAY` evaluates to `INTERVAL '7' DAY`. The result is at a precision so that information is not lost and contains the fields of both arguments. Hence, `INTERVAL '3' DAY + INTERVAL '4' HOUR` yields `INTERVAL '3 4' DAY TO HOUR`, and `INTERVAL '3' DAY + INTERVAL '8 4' DAY TO HOUR` yields `INTERVAL '11 4' DAY TO HOUR`. The SQL-92 semantics treats `INTERVAL '3' DAY` as exactly 3 days (72 hours).
- `INTERVAL '3' DAY - INTERVAL '4' DAY` yields `INTERVAL -'1' DAY`. As with addition, subtraction results in the union of the fields, to the necessary precision. `INTERVAL '3' DAY - INTERVAL -'8 4' DAY TO HOUR` results in `INTERVAL '11 4' DAY TO HOUR`.
- `(DATE '1997-01-01' - DATE '1996-01-01') DAY` yields `INTERVAL '366' DAY`, as 1996 was a leap year. Note that both the parentheses and a qualifier must be specified; this provides the granularity of the result. The subtraction is done at the least significant field of the qualifier, then the interval is converted to an interval of that field as the end field, with a start field chosen to not lose any information. So `(DATE '1997-01-01' - DATE '1996-01-01') YEAR TO MONTH` will convert both to months (23,952 and 23,940, respectively, though it turns out the origin doesn't matter), then the difference is taken, resulting in `INTERVAL '12' MONTH`, then the result is converted to the requested qualifier, or `INTERVAL '1-0' YEAR TO MONTH`.
- `INTERVAL '4' DAY * 3` yields `INTERVAL '12' DAY`. Multiplication is symmetric; this result is also obtained from `3 * INTERVAL '4' DAY`. Multiple fields can be accommodated; the interval is first converted to a scalar at the smallest field, then converted back after the multiplication. `INTERVAL '12:30' HOUR TO`

MINUTE * 3 yields 750 minutes times 3, or 2250 minutes, or INTERVAL '37:30' HOUR TO MINUTE.

- INTERVAL '4' DAY / 2 yields INTERVAL '2' DAY. This is similar to multiplication, though it is not symmetric. Hence, 2 / INTERVAL '4' DAY is not permitted.
- - INTERVAL '4' DAY yields, naturally, INTERVAL -'4' DAY. Unary plus is also provided: + INTERVAL '4' DAY yields itself.

The final constructor is CAST. Datetimes cannot be cast to intervals, nor intervals to datetimes. In fact, year-month intervals cannot be cast to day-time intervals, nor vice versa. The only casts that result in year-month intervals are from three sources:

- CHARACTER   As with datetimes, a character string may also be cast to a year-month interval, assuming the character string would have been acceptable as a literal. CAST('2' AS INTERVAL MONTH) works, but CAST('3-7' AS INTERVAL MONTH) does not.
- *year-month interval*   The source interval is first converted to a scalar in units of the least significant field of the target type. For CAST(INTERVAL '8-7' YEAR TO MONTH AS INTERVAL MONTH(2)), the source value would be converted to 103 months. This value is then *normalized* (a term not defined in the standard) to conform to the target type. If the precision is not sufficient, as here, an exception is raised. As another example, CAST(INTERVAL '3' YEAR AS INTERVAL YEAR TO MONTH), the source value would be converted to 36 months, then normalized to 3 years and 0 months, resulting in INTERVAL '3-0' YEAR TO MONTH.
- *exact numeric*   Here, the target interval must contain a single field, YEAR or MONTH. The source value is interpreted as a number of such units. CAST(103 AS INTERVAL MONTH) would evaluate to INTERVAL '103' MONTH; CAST(103 AS INTERVAL MONTH(2)) would raise an overflow exception.

Similarly, the only casts that result in day-time intervals are as follows:

- CHARACTER   CAST('2 12:34' AS INTERVAL DAY TO MINUTE) works, but CAST('12:34' AS INTERVAL DAY TO MINUTE) does not.
- *day-time interval*   As before, the source interval is first converted to a scalar in units of the least significant field of the target type. For CAST('85 23:59:60' AS INTERVAL HOUR TO SECOND), the source value is converted to 7,434,060 seconds. This value is then normalized to conform to the target type, resulting in INTERVAL '2065:00:00' HOUR TO SECOND. Had a target type of INTERVAL HOUR(3) TO SECOND been specified, an overflow exception would have been raised.
- *exact numeric*   Here, the target interval must contain a single field, DAY, HOUR, MINUTE, or SECOND. The source value is interpreted as a number of such units. To convert an exact numeric to a multifield interval, two casts are required. CAST(CAST(7434060 AS INTERVAL SECOND) AS INTERVAL DAY TO SECOND) would evaluate to our original value INTERVAL '86 00:00:00' DAY TO SECOND.

3.4.3 Other Constructors

Temporal values can also participate in casts to other types. We list these here for completeness:

- `CAST(DATE '1997-01-01' AS CHARACTER)` returns the character string `'1997-01-01'`. All temporal types can be cast to fixed- or variable-length character strings.
- `CAST(INTERVAL '743060' SECOND AS INTEGER)` returns the value 743,060. The source interval must have a single field. Intervals with multiple fields can be converted to exact numerics via two casts; for example, `CAST(CAST(INTERVAL '2064:60:60' HOUR TO SECOND AS INTERVAL SECOND) AS INTEGER)` yields the same value.

Finally, individual fields can be extracted from datetimes and intervals:

- `EXTRACT(YEAR FROM DATE '1970-01-01')` returns the integer value 1970.
- `EXTRACT(MINUTE FROM INTERVAL '12:34:56' HOUR TO SECOND)` returns 34.
- `EXTRACT(TIMEZONE_HOUR FROM TIME '12:34:56-07:00')` returns -7.
- `EXTRACT(TIMEZONE_MINUTE FROM TIME '12:34:56-07:00')` returns 0.

The last two exemplify new reserved words that were required to obtain these additional fields from datetimes with time zones.

Code Fragment 3.6 **Yet another four ways to ask for information on those born on January 1, 1970.**

```
SELECT * FROM Employee
WHERE CAST(BirthDate AS CHAR) = '1970-01-01'

SELECT * FROM Employee
WHERE CAST(BirthDate AS CHAR) LIKE '1970-01-01'

SELECT * FROM Employee
WHERE CAST((DATE '1971-01-01' - BirthDate) DAY AS INT) = 365
  AND CAST((DATE '1971-01-01' - BirthDate) YEAR AS INT) = 1

SELECT * FROM Employee
WHERE EXTRACT(YEAR FROM BirthDate) = 1970
  AND EXTRACT(MONTH FROM BirthDate) = 1
  AND EXTRACT(DAY FROM BirthDate) = 1
```

The resulting data type (fields and precision) varies among the operators. Table 3.1 summarizes the cases. Here, d denotes a datetime value, i an interval value, and n a numeric (exact or approximate) value. Union ($\cup$) is shorthand for combining the fields of both operands. This table lists all the constructors involving temporal values.

Table 3.1 Result type of SQL-92 expressions involving temporal values.

Expression	Result type
$d + i$	type of d
$i + d$	type of d
$d - i$	type of d
d AT LOCAL	type of d
d AT TIME ZONE i	type of d
CAST($type_1$ AS $type_2$)	$type_2$
EXTRACT($field$ FROM d)	n (exact numeric)
CURRENT_DATE	DATE
CURRENT_TIME	TIME
CURRENT_TIMESTAMP	TIMESTAMP
$i_1 + i_2$	type of $i_1 \cup$ type of i_2
$i_1 - i_2$	type of $i_1 \cup$ type of i_2
$(d - d)$ $qual$	$qual$
$i * n$	type of i
$n * i$	type of i
i_1 / i_2	n (integer)
i / n	type of i
$+ i$	type of i
$- i$	type of i
EXTRACT($field$ FROM i)	n (exact numeric)

3.5   IMPLEMENTATION CONSIDERATIONS

Although temporal types have been in the SQL standard since 1992 and were defined in the mid-1980s, it is surprising, and unfortunate, that unlike other portions of SQL, the types and their predicates and constructors are not supported by most DBMSs. Instead, each vendor has defined an incompatible and idiosyncratic set of temporal types and operators, replete with inconsistencies and seemingly arbitrary design decisions. Temporal types are among the most variable features of commercial DBMSs. Coupled with this is the often poor documentation available from the vendors of temporal features of their products. Determining the operations supported on temporal type(s) can be a frustrating exercise, with the information, if present at all, spread across the documentation. The following is an attempt to gather in one place the information about temporal support in a few prominent DBMSs. We make no claim for comprehensiveness, but then, neither do most

No vendor supports SQL-92 at the Full SQL level of conformance. All products include idiosyncrasies in their temporal support that render porting to other DBMSs difficult.

vendors. Interestingly, only Informix–Universal Server supports a type that provides partial support for intervals; all of the other DBMSs require intervals to be simulated with integers, fixed-point, or floating-point numbers.

3.5.1   IBM DB2 Universal Database

We start with IBM DB2 Universal Database (UDB), as it is closest to the SQL-92 standard in its support of temporal data types.

IBM DB2 UDB supports the DATE, TIME, and TIMESTAMP instant types, with a few deviations from SQL-92. The TIME type has a fixed precision of 0, indicating a granularity of seconds. The TIMESTAMP type has a fixed precision of 6, indicating a granularity of microseconds. Time zone information is not included in DB2 instant values; however, the current time zone is available in the `CURRENT TIME-ZONE` register. Instant literals are specified as a conversion function of the name of the data type operating on a character string, for example, `DATE('1997-01-15')` or `CAST('1997-01-15' AS DATE)`, which is preferred, because `DATE( )` could be a user-defined function. Timestamp literals replace the space between the day and hour with a dash, for example, `TIMESTAMP('1997-01-15-11.35.29.123456')`.

There is no INTERVAL data type in DB2 UDB. Instead, DB2 UDB supports specialized versions of the DECIMAL data type, termed *durations*.

- A *date duration*, in the format *YYYYMMDD*, is a `DECIMAL(8,0)` number representing an interval of days, with a range of 10,000 years.
- A *time duration*, in the format *HHMMSS.*, is a `DECIMAL(6,0)` number representing an interval of seconds, with a range of one day. Note that the decimal point is required in a time duration.
- A *timestamp duration*, in the format *YYYYMMDD.HHMMSSZZZZZZ*, is a `DECIMAL(20,6)` number representing an interval of microseconds, with a range of 10,000 years.

These values can be stored in DECIMAL columns and represented by DECIMAL constants. Hence, "`DATE('1997-11-08') + 00010101.`" adds one year, one month, and one day to the indicated instant, resulting in the date 1998-12-09.

DB2 UDB also supports a kind of highly restricted interval literal, termed a *labeled duration*, which is a numeric expression followed by a time unit (singular or plural). Labeled durations can only be used in an addition or subtraction with an instant type. An example is `DATE('1997-11-08') + 1 MONTH`. The available units are YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MICROSECOND, and plural versions of these keywords.

The function `TIMESTAMPDIFF` takes two parameters, a code specifying the granularity (e.g., 256 denotes years, 16 denotes days), and a character string that is the result of subtracting two timestamps and converting the result to character form. Note that there are many datetime functions provided.

Table 3.2 shows how the facilities in SQL-92 can be simulated, to some degree, in IBM DB2 UDB. In the SQL-92 column, *d* denotes a datetime value, *i* an interval value, and *n* a numeric (exact or approximate) value. In the IBM DB2 UDB column, *d* denotes a datetime value, *i* denotes a DB2 timestamp duration, and `itype` is an integer denoting an interval type.

3.5.2   Informix–Universal Server

Informix–Universal Server supports two instant types, DATE and DATETIME, and an interval type, INTERVAL. Time zones are not supported. An Informix DATE is stored internally as an integer denoting the number of dates since December 31, 1899; for example, day 1 is January 1, 1900. DATEs occupy four bytes, so the maximum date is sometime after 5 million C.E. Informix DATE literals are inconsistent with SQL-92; in Informix, the month is first, followed by the day, followed by the year, all separated with dashes, with the entire string delimited with double quotes. However, an Informix DATE literal only supports two digits, for example, `DATE("10/01/98")`, with the year `00` being 1900. To denote the years after 1999, you have to add an INTERVAL explicitly. Hence, to designate January 1, 2000, you have to use something like `DATE("12/01/99") + INTERVAL(0-1) YEAR TO MONTH` or `DATE("01/01/99") + INTERVAL(1-0) YEAR TO MONTH`.

The Informix DATETIME type is equivalent to TIMESTAMP in SQL-92 and can have a user-specified precision, such as `YEAR TO MONTH` or `YEAR TO SECOND`. SQL-92's TIME type is identical to Informix's `DATETIME HOUR TO SECOND`. Fractional seconds are denoted with `FRACTION($n$)`. SQL-92's TIMESTAMP type is then equivalent to Informix's `DATETIME YEAR TO FRACTION(6)`. Interestingly, Informix DATETIME literals are consistent with SQL-92 (except that Informix DATETIME literals don't use quotes) but are inconsistent with Informix DATEs. Specifically, in an Informix DATETIME literal, the year comes first, as a four-digit number, followed by the month and day, then hour, minute, and second, without quotes (!). At this second, my watch reads `DATETIME(1998-04-08 12:13:52)`, about time for lunch; my calendar reads `DATE("04/08/98")`.

Informix–Universal Server supplies utilities such as DATE, MDY (month/day/year), YEAR, and WEEKDAY for formatting and converting dates. The current DATE is given by TODAY; the current DATETIME is given by CURRENT. The EXTEND function can be used to alter the precision of instants. This function extracts the year and month from the corresponding values of CURRENT; the minutes and the seconds are set to zero if not provided. Hence, `EXTEND(DATETIME(16 19) DAY TO HOUR, YEAR TO SECOND)` returns `DATETIME(1997-01-01 19:00:00)`. It can also be used to convert strings into instant types. The standard predicates are also available on instants.

As in SQL-92, an Informix INTERVAL must be either a year-month interval or a day-time interval. Note that intervals can be added to instants (yielding an instant),

Table 3.2 SQL-92 operations in IBM DB2 UDB.

SQL-92	IBM DB2 UDB Equivalent
Types:	
DATE	DATE
TIME	TIME (precision fixed at 0)
TIMESTAMP	TIMESTAMP (precision fixed at 6)
TIME WITH TIME ZONE	no equivalent
TIMESTAMP WITH TIME ZONE	no equivalent
INTERVAL YEAR TO MONTH	A date duration(DECIMAL(8, 0)) with a 0 DAY field
INTERVAL DAY TO SECOND	A timestamp duration with 0 YEAR, MONTH, and MICROSECOND fields, negative values not available
Literals:	
DATE '1997-01-01'	DATE('1997-01-01')
TIME '12:34:56'	TIME('12:34:56')
TIMESTAMP '1997-01-01 12:34:56'	TIMESTAMP('1997-01-01-12.34.56.000000')
INTERVAL '3-4' YEAR TO MONTH	40 MONTHS, 00030400 (only in an expression)
INTERVAL '1 23:45:12' DAY TO SECOND	00000001234512.000000, 171912 SECONDS (only in an expression)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 BETWEEN d_2 AND d_3	d_1 BETWEEN d_2 AND d_3
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 BETWEEN i_2 AND i_3	i_1 BETWEEN i_2 AND i_3
d IS NULL	d IS NULL
i IS NULL	i IS NULL
(d_1, i) OVERLAPS (d_2, d_3)	$d_1 < d_3$ AND $d_2 < (d_1 + i)$
Datetime Constructors:	
$d + i$	$d + i$
$i + d$	$i + d$
$d - i$	$d - i$
d AT i	$d + i$
d AT LOCAL	d + CURRENT TIMEZONE
CURRENT_DATE	CURRENT DATE
CURRENT_TIME	CURRENT TIME
CURRENT_TIMESTAMP	CURRENT TIMESTAMP

Table 3.2 (continued)

SQL-92	IBM DB2 UDB Equivalent
Interval Constructors:	
$i_1 + i_2$	not possible
$i_1 - i_2$	not possible
$(d_1 - d_2)$ `qual`	`TIMESTAMPDIFF(`*itype*`, CHAR(`d_1` - `d_2`))`
$(d_1 - d_2)$ `MONTH`	`TIMESTAMPDIFF(64, CHAR(`d_1` - `d_2`))`
`i * n`	not possible
`n * i`	not possible
i_1 `/` i_2	not possible
`i / n`	not possible
`+ i`	`i`
`- i`	not possible
Other Operators:	
`CAST(`*d* `AS DATE)`	`CAST(`*d* `AS DATE)`
`CAST(`*d* `AS TIME)`	`CAST(`*d* `AS TIME)`
`CAST(`*d* `AS TIMESTAMP)`	`CAST(`*d* `AS TIMESTAMP)`
`CAST(`*i* `AS INTERVAL YEAR` `TO MONTH)`	not possible
`CAST(`*i* `AS INTERVAL DAY` `TO SECOND)`	not possible
`CAST(`*d* `AS CHAR)`	`CHAR(`*d*`)`
`CAST(`*i* `AS CHAR)`	not possible
`CAST(`*i* `AS INTEGER)`	
i **is** `YEAR TO DAY`	`JULIAN_DAY(DATE('001-01-01') + `*i*`)` `- JULIAN_DAY(DATE('001-01-01-00))`
i **is** `DAY TO HOUR`	`24 * DAY(`*i*`) + HOUR(`*i*`)`
i **is** `DAY TO MINUTE`	`1440 * DAY(`*i*`) + 60 * HOUR(`*i*`)` `+ MINUTE(`*i*`)`
i **is** `DAY TO SECOND`	`86400 * DAY(`*i*`) + 3600 * HOUR(`*i*`)` `+ 60 * MINUTE(`*i*`) + SECOND(`*i*`)`
`EXTRACT(DAY FROM `*d*`)`	`DAY(`*d*`)`
`EXTRACT(DAY FROM `*i*`)`	`DAY(`*i*`)`
`EXTRACT(HOUR FROM `*i*`)`	`HOUR(`*i*`)`
Operators not in SQL-92:	
convert *d* to Julian day	`JULIAN_DAY(`*d*`)`

but instants can't be added to intervals, the reason being that the resulting type must be an interval.

Table 3.3 shows how the facilities in SQL-92 can be simulated, to some degree, in Informix–Universal Server.

Table 3.3 SQL-92 operations in Informix–Universal Server.

SQL-92	Informix–Universal Server Equivalent
Types:	
DATE	DATE
TIME	DATETIME HOUR TO SECOND
TIMESTAMP	DATETIME YEAR TO FRACTION(6)
TIME WITH	no equivalent
TIMESTAMP WITH TIME ZONE	no equivalent
INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH
INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND
Literals:	
DATE '1997-01-01'	DATE("01/01/97")
TIME '12:34:56'	DATETIME(12:34:56) HOUR TO SECOND
TIMESTAMP '1997-01-01 12:34:56'	DATETIME(1997-01-01-12:34:56) YEAR TO SECOND
INTERVAL '3-4' YEAR TO MONTH	INTERVAL(3-4) YEAR TO MONTH
INTERVAL '1 23:45:12' DAY TO SECOND	INTERVAL(1 23:45:12) DAY TO SECOND
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 BETWEEN d_2 AND d_3	d_1 BETWEEN d_2 AND d_3
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 BETWEEN i_2 AND i_3	i_1 BETWEEN i_2 AND i_3
d IS NULL	d IS NULL
i IS NULL	i IS NULL
(d_1, i) OVERLAPS (d_2, d_3)	$d_1 < d_3$ AND $d_2 < (d_1 + i)$
Datetime Constructors:	
$d + i$	$d + i$
$i + d$	$d + i$
$d - i$	$d - i$
d AT i	not supported
d AT LOCAL	not supported
CURRENT_DATE	TODAY
CURRENT_TIME	CURRENT HOUR TO SECOND
CURRENT_TIMESTAMP	CURRENT

Table 3.3 (continued)

SQL-92	Informix–Universal Server Equivalent
Interval Constructors:	
$i_1 + i_2$	$i_1 + i_2$
$i_1 - i_2$	$i_1 - i_2$
$(d_1 - d_2)$ `qual`	$d_1 - d_2$ (if both have the same precision)
$(d_1 - d_2)$ `MONTH`	not supported
$i * n$	$i * n$
$n * i$	$i * n$
i_1 / i_2	not possible
i / n	not possible
$+ i$	$+ i$
$- i$	$- i$
Other Operators:	
`CAST(`d` AS DATE)`	`DATE(`d`)`
`CAST(`d` AS TIME)`	`EXTEND(DATE(`d`), HOUR TO SECOND)`
`CAST(`d` AS TIMESTAMP)`	`EXTEND(`d`, YEAR TO SECOND)`
`CAST(`i` AS INTERVAL YEAR TO MONTH)`	`INTERVAL(`i`) YEAR TO MONTH`
`CAST(`i` AS INTERVAL DAY TO SECOND)`	`INTERVAL(`i`) DAY TO SECOND`
`CAST(`d` AS CHAR)`	not possible
`CAST(`i` AS CHAR)`	not possible
`CAST(`i` AS INTEGER)`	not supported
`EXTRACT(DAY FROM `d`)`	`DAY(`d`)` (returns an integer)
`EXTRACT(DAY FROM `i`)`	not possible
`EXTRACT(HOUR FROM `i`)`	not possible
Operators not in SQL-92:	
extract weekday from d (where d is `DATE`)	`WEEKDAY(`d`)`

3.5.3   Microsoft Access

While SQL-92 supplies six temporal types (DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE, and INTERVAL), Microsoft Access supplies just one, Date/Time, which is similar to SQL-92's TIMESTAMP type. Access Date/Time values are stored as an IEEE 8-byte floating-point number, with the integral portion denoting days since December 30, 1899, and the fractional portion denoting fractions of a day, to a precision of eight decimal places, or equivalently, a granularity of slightly less than one millisecond. The range is restricted to 1 C.E. to 9999 C.E.; dates before 1899 are represented with negative values.

Literals are delimited with '#', for example, #5/10/96#, which uses the "U.S. format" (month, day, year), even on international versions of Microsoft Windows. To have the format depend on the locale, use `DateValue`. For example,

The Hijri Calendar

The Hijri is an Islamic calendar based on lunar cycles, with one year consisting of 12 (purely lunar) months. It was first introduced in 638 C.E. by Umar ibn Al-Khattab. The first day of this calendar, Muharram 1 (New Year), 1 A.H. ("Anno Hegirae") corresponds to June 16, 622 C.E.

Since the Islamic calendar is purely lunar, as opposed to solar or lunar-solar, the Hijri year is shorter than the Gregorian year by about 11 days. Also contrary to most calendars, months in the Hijri year are not related to seasons, which are themselves tied to the solar cycle. Important Muslim festivals, which always fall in the same Hijri month, may occur in different seasons. For example, the Hajj and Ramadan can take place in the summer as well as the winter. It is only over a roughly 33-year cycle that lunar months resynchronize with the solar year.

Interestingly, the start of a Hijri month is defined not by an astronomical new moon, but rather by an actual sighting of the crescent moon at a particular locale. This implies that a month will start at different Gregorian times in different locales, and indeed the start is affected by weather conditions and various optical factors of the atmosphere.

`DateValue('5/10/96')` when evaluated in the U.S. will return the same date as `DateValue('10/5/96')` when evaluated in the U.K.

In an effort to address the year 2000 problem, Access 2000 has a special interpretation of two-digit years. `#1/1/00#` through `#12/31/29#` are interpreted as the dates January 1, 2000, through December 31, 2029. `#1/1/30#` through `#12/31/99#` are interpreted as the dates January 1, 1930, through December 31, 1999. Of course, this just moves the year 2000 problem ahead 30 years, as well as invalidating previous data from the first third of this century. More detail may be found in Section 3.6.5.

The format of a literal is specified in a format property setting. Custom formats may be specified using some thirty-odd multicharacter symbols, such as "`ww`," which specifies a number between 1 and 53 denoting the week of the year. The `format` routine allows a format to be used once, for example, `format("1/10/99", "dd/mm/yy")`. Predefined formats, set in the properties, may also be used within this function, for example, `format("1/10/99", "short date")`. The Windows 95 system settings (in "Regional Settings") dictate the initial values for these format properties. The `CDate( )` function takes a string and attempts to convert it into a date, using context to determine which fields are where in the string.

The equality and inequality predicates are available for Access Date/Times. OVERLAP is not available.

Extraction of fields is accomplished through a variety of functions, such as `Day( )` and `Second( )`. There is also an extraction function, `DatePart( )`, for example, `DatePart("yyyy", [OrderDate])` would return a four-digit year.

Intervals are simulated with two functions, `DateAdd` and `DateDiff`. `DateAdd` takes a string expression specifying the interval granularity (`"yyyy"` denotes year, `"q"` denotes quarter, `"m"` denotes month, `"y"` denotes day of year, `"d"` denotes day, `"w"` denotes weekday, `"ww"` denotes week, `"h"` denotes hour, `"m"` denotes minute, and `"s"` denotes second), a numeric expression (positive or negative) denoting the number of granules, and a date to be shifted. As an example, `DateAdd("d", 7, #2/24/96#)` yields the date March 2, 1996. Analogously, `DateDiff` takes the parameters of a string expression denoting the granularity, as well as two days, and returns a long integer specifying the number of time intervals between these dates. For example, `DateDiff("d", #3/2/96#, #2/26/96#)` evaluates to the value 7.

The weekday granularity depends on which day is considered the first day of the week. Several of the functions, including `DateDiff`, take an optional parameter specifying a particular day (1 = Sunday through 7 = Saturday, with Sunday being the default). The week granularity depends on which week is considered the first week of the year. Several of the functions take an optional parameter specifying this detail (1 = start with the week in which January 1 occurs, 2 = start with the first week that has at least four days in the year, 3 = start with the first full week of the year, with 1 being the default).

Table 3.4 shows how the facilities in SQL-92 can be simulated, to some degree, in Access. In the Access column, d denotes an Access Date/Time value and j denotes an Access FLOAT, indicating a (fractional) count of Julian days.

3.5.4   Microsoft SQL Server

Microsoft SQL Server supplies two temporal data types, DATETIME and SMALL-DATETIME, with precisions of 1/300 second and 1 minute, respectively. The range of these two types is January 1, 1753 to December 31, 9999 for the DATETIME type, and from January 1, 1900 to June 6, 2079 for the SMALLDATETIME type. A DATETIME requires eight bytes, four bytes for the number of days since the base date and four bytes for the time of day. A SMALLDATETIME requires only four bytes, two bytes for the number of days since the base date and two bytes for the number of minutes since midnight.

Intervals can be represented as SQL Server integers, of an integral number of granules in the required granularity.

Table 3.5 shows how the facilities in SQL-92 can be simulated, to some degree, in Microsoft SQL Server. In the SQL-92 column, d denotes a datetime value, 'i' an interval value, and n a numeric (exact or approximate) value. In the Microsoft column, d denotes an SQL Server DATETIME value, and i denotes an SQL Server integer representing an integral number of seconds.

Microsoft SQL Server automatically handles certain data type conversions; in such cases, the `convert` function is optional. For example, when a character expression is compared with a DATETIME expression, the character expression is implicitly converted to a DATETIME.

Table 3.4 SQL-92 operations in Microsoft Access 2000.

SQL-92	Microsoft Access 2000 Equivalent
Types:	
`DATE`	`Date/Time`, ignoring the hour, minute, and second fields
`TIME`	`Date/Time`, ignoring the century, year, month, and day fields
`TIMESTAMP`	`Date/Time`
`TIME WITH TIME ZONE`	no equivalent
`TIMESTAMP WITH TIME ZONE`	no equivalent
`INTERVAL YEAR TO MONTH`	`INTEGER` (months)
`INTERVAL DAY TO SECOND`	`INTEGER` (seconds) or `FLOAT` (days)
Literals:	
`DATE '1997-01-01'`	`DateValue('1997-01-01')` or `#1997-01-01#`
`TIME '12:34:56'`	`TimeValue('12:34:56', 'HH:MI:SS')` or `#12:34:56#`
`TIMESTAMP '1997-01-01 12:34:56'`	`Format("1997-01-01 12:34:56", "YYYY-MM-DD HH:NN:SS")` or `#1997-01-01 12:34:56#`
`INTERVAL '3-4' YEAR TO MONTH`	40 (months)
`INTERVAL '1 23:45:12' DAY TO SECOND`	171912 (seconds) or 1.9897222 (Julian days)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 `BETWEEN` d_2 `AND` d_3	d_1 `BETWEEN` d_2 `AND` d_3
$i_1 = i_2$	$j_1 = j_2$
$i_1 < i_2$	$j_1 < j_2$
$i_1 <> i_2$	$j_1 <> j_2$
i_1 `BETWEEN` i_2 `AND` i_3	j_1 `BETWEEN` j_2 `AND` j_3
d `IS NULL`	d `IS NULL`
i `IS NULL`	j `IS NULL`
(d_1, i) `OVERLAPS` (d_2, d_3)	$d_1 < d_3$ `AND` $d_2 <$ `DateAdd("m",` $j, d_1)$
Datetime Constructors:	
$d + i$	`DateAdd("d",`$d, j)$
$i + d$	`DateAdd("d",`$d, j)$
$d - i$	`DateAdd("d",`$d, -j)$
d `AT` i	time zones not supported
d `AT LOCAL`	time zones not supported
`CURRENT_DATE`	`Date()` The time part is set to 0
`CURRENT_TIME`	`Time()` The date part is set to 0 (i.e., December 31, 1899)
`CURRENT_TIMESTAMP`	`Now()`

Table 3.4 (continued)

SQL-92	Microsoft Access 2000 Equivalent
Interval Constructors:	
$i_1 + i_2$	$j_1 + j_2$
$i_1 - i_2$	$j_1 - j_2$
$(d - d)$ qual	$d_1 - d_2$ (result is a fractional number of days)
$d - d$ MONTH	DateDiff("m", d_2, d_1) (result is an integral number of months)
$i * n$	$j * n$
$n * i$	$n * j$
i_1 / i_2	Trunc(j_1 / j_2)
i / n	Trunc(j / n)
$+ i$	j
$- i$	$-j$
Other Operators:	
CAST(d AS DATE)	DateValue(d) or Format(d, "YYYY-MM-DD")
CAST(d AS TIME)	TimeValue(d) or Format(d, "HH:MI:SS")
CAST(d AS TIMESTAMP)	
d is a DATE	d
d is a TIME	not possible
CAST(i AS INTERVAL YEAR TO MONTH)	not possible
CAST(i AS INTERVAL DAY TO SECOND)	not possible
CAST(d AS CHAR)	Cstr(d)
CAST(i AS CHAR)	Cstr(j)
CAST(i AS INTEGER)	
i is DAY	CLng(j)
i is HOUR	CLng($j * 24$)
i is MINUTE	CLng($j * 1440$)
i is SECOND	CLng($j * 86400$)
EXTRACT(DAY FROM d)	Day(d)
EXTRACT(DAY FROM i)	Day(j)
EXTRACT(HOUR FROM i)	Hour(j)

Table 3.5 SQL-92 operations in Microsoft SQL Server.

SQL-92	Microsoft SQL Server Equivalent
Types:	
DATE	DATETIME or SMALLDATETIME, ignoring the hour, minute, and second fields
TIME	DATETIME or SMALLDATETIME, ignoring the century, year, month, and day fields
TIMESTAMP	DATETIME (to second granularity)
TIME WITH TIME ZONE	no equivalent
TIMESTAMP WITH TIME ZONE	no equivalent
INTERVAL YEAR TO MONTH	int (integral number of months)
INTERVAL DAY TO SECOND	int (integral number of seconds)
Literals:	
DATE '1997-01-01'	convert(datetime, "1997-01-01", 102)
TIME '12:34:56'	"12:34:56" or convert(datetime, "12:34:56")
TIMESTAMP '1997-01-01 12:34:56'	"1997-01-01 12:34:56" or convert(datetime, "1997-01-01 12:34:56") or convert(datetime, "1997-01-01 12:34:56", 102)
INTERVAL '3-4' YEAR TO MONTH	40 (months)
INTERVAL '1 23:45:12' DAY TO SECOND	171812 (seconds)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 BETWEEN d_2 AND d_3	d_1 BETWEEN d_2 AND d_3
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 BETWEEN i_2 AND i_3	i_1 BETWEEN i_2 AND i_3
d IS NULL	d IS NULL
i IS NULL	i IS NULL
(d_1, i) OVERLAPS (d_2, d_3)	$d_1 < d_3$ AND $d_2 <$ dateadd(second,i,d_1)
Datetime Constructors:	
$d + i$	dateadd(second, i, d)
$i + d$	dateadd(second, i, d)
$d - i$	dateadd(second, -i, d)
d AT i	not supported
d AT LOCAL	not supported

Table 3.5 (continued)

SQL-92	Microsoft SQL Server Equivalent
Datetime Constructors, cont.:	

`CURRENT_DATE`	`convert (datetime,`
	`(convert(char(3),`
	`datename(month,getdate())) + " "`
	`+ convert(char(2),`
	`datename(day,getdate())) + ","`
	`+ convert(char(4),`
	`datename(year,getdate())))))`

`CURRENT_TIME`	`convert (datetime,`
	`convert(char(2),`
	`datename(hour,getdate())) + ":"`
	`+ convert(char(2),`
	`datename(minute,getdate())) + ":"`
	`+ convert(char(2),`
	`datename(second,getdate())))`

`CURRENT_TIMESTAMP`	`getdate()`
Interval Constructors:	
$i_1 + i_2$	$i_1 + i_2$
$i_1 - i_2$	$i_1 - i_2$
$(d_1 - d_2)$ `qual`	`datediff(`*`qual`*`, `d_1`, `d_2`)` (result is an integral number at the indicated granularity)
$(d_1 - d_2)$ `MONTH`	`datediff(month, `d_1`, `d_2`)`
$i * n$	$i * n$
$n * i$	$n * i$
i_1 / i_2	`convert(int, `i_1 / i_2`)`
i / n	`convert(int, `i / n`)`
$+ i$	i
$- i$	$- i$
Other Operators:	

`CAST(`*d* ` AS DATE)`	`convert (datetime,`
	`(convert(char(3),`
	`datename(month,`*`d`*`)) + " "`
	`+ convert(char(2),`
	`datename(day,`*`d`*`)) + ","`
	`+ convert(char(4),`
	`datename(year,`*`d`*`))))`

`CAST(`*d* ` AS TIME)`	`convert (datetime,`
	`convert(char(2),`
	`datename(hour,`*`d`*`)) + ":"`
	`+ convert(char(2),`
	`datename(minute,`*`d`*`)) + ":"`
	`+ convert(char(2),`
	`datename(second,`*`d`*`)))`

Table 3.5 (continued)

SQL-92	Microsoft SQL Server Equivalent
Other Operators, cont.:	
`CAST(d AS TIMESTAMP)`	
d is a `DATE`	`d`

d is a `TIME`	`convert (datetime,`
	  `(convert(char(3),`
	  `datename(month,getdate())) + " "`
	  `+ convert(char(2),`
	  `datename(day,getdate())) + ","`
	  `+ convert(char(4),`
	  `datename(year,getdate())) + " "`
	  `+ convert(char(2),`
	  `datename(hour,d)) + ":"`
	  `+ convert(char(2),`
	  `datename(minute,d)) + ":"`
	  `+ convert(char(2),`
	  `datename(second,d))))`

`CAST(i AS INTERVAL YEAR TO MONTH)`	not possible
`CAST(i AS INTERVAL DAY TO SECOND)`	not possible
`CAST(d AS CHAR)`	`convert(char, d)`
`CAST(i AS CHAR)`	`convert(char, i)`
`CAST(i AS INTEGER)`	`i` already an integer
`EXTRACT(DAY FROM d)`	`datename(day,d)` (returns a string)
`EXTRACT(DAY FROM i)`	`convert(int, i/86400)`
`EXTRACT(HOUR FROM i)`	`convert(int, i/3600)`

3.5.5 Sybase SQLServer

Support for time in Sybase SQLServer is essentially identical to that of Microsoft SQL Server because they started from the same code base. For details, see the discussion on that system, in Section 3.5.4.

Table 3.6 shows how the facilities in SQL-92 can be simulated, to some degree, in Sybase SQLServer. In the SQL-92 column, *d* denotes a datetime value, *i* an interval value, and *n* a numeric (exact or approximate) value. In the Sybase SQLServer column, *d* denotes an SQLServer DATETIME value, and *i* denotes an SQLServer integer representing an integral number of seconds.

3.5.6 Oracle8 Server

As with Access, Oracle8 Server provides but one temporal type, here called DATE. An Oracle DATE comprises seven fields, century, year, month, day, hour, minute, and second, each as one byte. Oracle8 Server can store dates from January 1, 4712 B.C.E. (Julian Day 1) to December 31, 4712 C.E., while disallowing the nonexistent

Table 3.6 SQL-92 operations in Sybase SQLServer.

SQL-92	Sybase SQLServer Equivalent
Types:	
`DATE`	`DATETIME` or `SMALLDATETIME`, ignoring the hour, minute, and second fields
`TIME`	`DATETIME` or `SMALLDATETIME`, ignoring the century, year, month, and day fields
`TIMESTAMP`	`DATETIME` (to second granularity)
`TIME WITH TIME ZONE`	no equivalent
`TIMESTAMP WITH TIME ZONE`	no equivalent
`INTERVAL YEAR TO MONTH`	`int` (integral number of months)
`INTERVAL DAY TO SECOND`	`int` (integral number of seconds)
Literals:	
`DATE '1997-01-01'`	`convert(datetime, "1997-01-01", 105)`
`TIME '12:34:56'`	`convert(datetime, "12:34:56")`
`TIMESTAMP '1997-01-01 12:34:56'`	`convert(datetime, "1997-01-01 12:34:56",105)`
`INTERVAL '3-4' YEAR TO MONTH`	40 (months)
`INTERVAL '1 23:45:12' DAY TO SECOND`	171812 (seconds)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 `BETWEEN` d_2 `AND` d_3	$d_2 <= d_1$ `AND` $d_1 <= d_3$
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 `BETWEEN` i_2 `AND` i_3	$i_2 <= i_1$ `AND` $i_1 <= i_3$
d `IS NULL`	d `IS NULL`
i `IS NULL`	i `IS NULL`
(d_1, i) `OVERLAPS` (d_2, d_3)	$d_1 < d_3$ `AND` $d_2 <$ `dateadd(second,`i`,`d_1`)`
Datetime Constructors:	
$d + i$	`dateadd(second,`i`,`d`)`
$i + d$	`dateadd(second,`i`,`d`)`
$d - i$	`dateadd(second,-`i`,`d`)`
d `AT` i	not supported
d `AT LOCAL`	not supported
`CURRENT_DATE`	`convert (datetime, (convert(char(3), datename(month,getdate())) + " " + convert(char(2), datename(day,getdate())) + "," + convert(char(4), datename(year,getdate())))))`

Table 3.6 (continued)

SQL-92	Sybase SQLServer Equivalent
Datetime Constructors, cont.:	
`CURRENT_TIME`	`convert (datetime, convert(char(2), datename(hour,getdate()))) + ":" + convert(char(2), datename(minute,getdate()))) + ":" + convert(char(2), datename(second,getdate())))`
`CURRENT_TIMESTAMP`	`getdate()`
Interval Constructors:	
$i_1 + i_2$	$i_1 + i_2$
$i_1 - i_2$	$i_1 - i_2$
$(d_1 - d_2)$ `qual`	`datediff(`$qual$`, `d_1`, `d_2`)` (result is an integral number at the indicated granularity)
$(d_1 - d_2)$ `MONTH`	`datediff(month, `d_1`, `d_2`)`
$i * n$	$i * n$
$n * i$	$n * i$
i_1 / i_2	`convert(int, `i_1 / i_2`)`
i / n	`convert(int, `i / n`)`
$+ i$	i
$- i$	$- i$
Other Operators:	
`CAST(`d` AS DATE)`	`convert (datetime, (convert(char(3), datename(month,`d`)) + " " + convert(char(2), datename(day,`d`)) + "," + convert(char(4), datename(year,`d`))))`
`CAST(`d` AS TIME)`	`convert (datetime, convert(char(2), datename(hour,`d`))+":" + convert(char(2), datename(minute,`d`)) + ":" + convert(char(2), datename(second,`d`)))`
`CAST(`d` AS TIMESTAMP)`	
d is a `DATE`	d

Table 3.6 (continued)

SQL-92	Sybase SQLServer Equivalent
Other Operators, cont.:	
`CAST(`*d*` AS TIMESTAMP)`	

d is a `TIME`	`convert (datetime,`
	`(convert(char(3),`
	`datename(month,getdate()))) + " "`
	`+ convert(char(2),`
	`datename(day,getdate()))) + ","`
	`+ convert(char(4),`
	`datename(year,getdate()))) + " "`
	`+ convert(char(2),`
	`datename(hour,`*d*`)) + ":"`
	`+ convert(char(2),`
	`datename(minute,`*d*`)) + ":"`
	`+ convert(char(2),`
	`datename(second,`*d*`))))`

`CAST(`*i*` AS INTERVAL YEAR TO MONTH)`	not possible
`CAST(`*i*` AS INTERVAL DAY TO SECOND)`	not possible
`CAST(`*d*` AS CHAR)`	`convert(char, `*d*`)`
`CAST(`*i*` AS CHAR)`	`convert(char, `*i*`)`
`CAST(`*i*` AS INTEGER)`	*i* already an integer
`EXTRACT(DAY FROM `*d*`)`	`datename(day,`*d*`)` (returns a string)
`EXTRACT(DAY FROM `*i*`)`	`convert(int, `*i*`/86400)`
`EXTRACT(HOUR FROM `*i*`)`	`convert(int, `*i*`/3600)`

year 0000. It is thus superior to SQL's TIMESTAMP type in permitting B.C.E. dates, but is inferior in not permitting fractions of a second and in stopping about halfway to the year 9999.

SQL-92's INTERVAL DAY can be simulated using Oracle NUMBER, which provides a day count. Smaller granularities can be partially simulated with fractional days. `NUMBER(12,5)` is sufficient for seconds; `NUMBER(18,11)` will support microseconds. Both support the full range of 4700 years. It is not possible to simulate SQL-92 year-month intervals, though subtraction to the months granularity is possible via `MONTHS_BETWEEN`. `NEXT_DAY` gives the date of the next day of the week (specified as a character string such as `'Monday'`) after a specified date value.

The predicates that Oracle8 Server supports on DATEs are '=', '<', '<=', '>', '>=', '<>', and BETWEEN. These are based on the seven-byte internal representation of dates. For not equals, Oracle8 Server also allows '!=', '^ =', and '¬=' (on some systems).

Oracle dates can be converted to character strings via the `TO_CHAR` function, which takes as a second argument the format desired. SQL-92's `CAST(`value` AS CHARACTER)` may thus be simulated with `TO_CHAR(`value`, 'YYYY-MM-DD HH24:MI:SS')` (`HH24` requests military time, that is, a 24-hour clock). This format string is quite

flexible, with over 30 options available. As an example, TO_CHAR(value, 'DY Month DD, YYYY') will produce a string looking like 'THU July 24, 1997'. A third parameter to this function specifies other aspects of the output, such as the language, for example, TO_CHAR(BirthDate, 'Month DD, YYYY, HH12:MI A.M.', 'NLS_DATE_LANGUAGE = American').

The TO_CHAR function can also extract individual fields; for example, TO_CHAR(BirthDate, 'MM') returns the month. Finally, the Julian day number can be computed via TO_CHAR(BirthDate, 'J'), returning an integer (2,440,588, for January 1, 1970).

The TO_DATE function produces a date value from a character string, via a format string, for example, TO_DATE('January 15, 1989, 11:00 A.M.', 'Month DD, YYYY, HH12:MI A.M.'). This function can also convert an integer (containing the Julian number) to a date, for example, TO_DATE(2440588, 'J'). The Julian number identifies a particular day, so the hour, minute, and second fields are set to 0. There seems to be no way to extract fractional days from an Oracle DATE, nor convert fractional days to a DATE value. However, an Oracle (fractional) NUMBER, representing a day-time interval, *can* be added (or subtracted) from a DATE, yielding a DATE.

Oracle8 Server provides a variety of other date functions. ADD_MONTHS adds an integer number of months to a DATE value. By using a negative integer, months can be subtracted. GREATEST picks the latest date from a list of dates; LEAST is analogous. LAST_DAY returns the date of the last day of the month that the provided date is in.

The NEW_TIME function allows you to shift a DATE from one specified time zone to another. The source and target time zones are three-character strings; only a few time zones are supported. There seems to be no way within Oracle8 Server to find out your own time zone.

The TRUNC function removes the hour, minute, and second fields from a DATE value, resulting in a day starting at midnight. TRUNC also accepts an optional string parameter, specifying a field below which truncation should occur; for example, TRUNC(BirthDate, 'HH24') would zero out the minute and second fields. An analogous ROUND function is also available for DATEs.

Finally, the current date and time is returned by SYSDATE.

Table 3.7 shows how the facilities in SQL-92 can be simulated, to some degree, in Oracle8 Server. In the SQL-92 column, d denotes a datetime value, i an interval value, and n a numeric (exact or approximate) value. In the Oracle8 Server column, d denotes an Oracle DATE value and j denotes an Oracle NUMBER representing Julian days.

So, how did Jim Barnett represent instants and intervals in the FINDER schema? Well, for the most part he used Oracle DATEs. In fact, in the resulting schema one of every five columns is a DATE column. The fundamental distinction between instants and intervals is hidden in the Oracle schema; column names and comments

Table 3.7 SQL-92 operations in Oracle8 Server.

SQL-92	Oracle8 Server Equivalent
Types:	
`DATE`	`DATE`, ignoring the hour, minute, and second fields
`TIME`	`DATE`, ignoring the century, year, month, and day fields
`TIMESTAMP`	`DATE` (to second granularity)
`TIME WITH TIME ZONE`	no equivalent
`TIMESTAMP WITH TIME ZONE`	no equivalent
`INTERVAL YEAR TO MONTH`	no equivalent
`INTERVAL DAY TO SECOND`	`NUMBER(12, 5)`
Literals:	
`DATE '1997-01-01'`	`TO_DATE('1997-01-01', 'YYYY-MM-DD')`
`TIME '12:34:56'`	`TO_DATE('12:34:56', 'HH24:MI:SS')`
`TIMESTAMP '1997-01-01 12:34:56'`	`TO_DATE('1997-01-01 12:34:56', 'YYYY-MM-DD HH24:MI:SS')`
`INTERVAL '3-4' YEAR TO MONTH`	not possible
`INTERVAL '1 23:45:12' DAY TO SECOND`	`TO_NUMBER(SUBSTR('1 23:45:12', 1,LENGTH('1 23:45:12')-9))` `+ TO_NUMBER(SUBSTR('1 23:45:12', LENGTH('1 23:45:12')-7,2))/24` `+ TO_NUMBER(SUBSTR('1 23:45:12', LENGTH('1 23:45:12')-4,2))/1440` `+ TO_NUMBER(SUBSTR('1 23:45:12', LENGTH('1 23:45:12')-1,2))/86400` (result is a fractional Julian day)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 `BETWEEN` d_2 `AND` d_3	d_1 `BETWEEN` d_2 `AND` d_3
$i_1 = i_2$	$j_1 = j_2$
$i_1 < i_2$	$j_1 < j_2$
$i_1 <> i_2$	$j_1 <> j_2$
i_1 `BETWEEN` i_2 `AND` i_3	j_1 `BETWEEN` j_2 `AND` j_3
d `IS NULL`	d `IS NULL`
i `IS NULL`	j `IS NULL`
(d_1, i) `OVERLAPS` (d_3, d_4)	$d_1 < d_4$ `AND` $d_3 < (d_1 + j)$
Datetime Constructors:	
$d + i$	$d + j$, `ADD_MONTHS(`d`, `j`)`
$i + d$	$j + d$, `ADD_MONTHS(`d`, `j`)`
$d - i$	$d - j$, `ADD_MONTHS(`d`, -`j`)`
d `AT` i	not supported
d `AT LOCAL`	not supported

Table 3.7 (continued)

SQL-92	Oracle8 Server Equivalent
Datetime Constructors, cont.:	
`CURRENT_DATE`	`TRUNC(SYSDATE)`

`CURRENT_TIME`	`TO_DATE(`
	`  TO_CHAR(SYSDATE, 'HH24:MI:SS'),`
	`  'HH24:MI:SS')`

`CURRENT_TIMESTAMP`	`SYSDATE`
Interval Constructors:	
$i_1 + i_2$	$j_1 + j_2$
$i_1 - i_2$	$j_1 - j_2$
$(d_1 - d_2)$ `qual`	$d_1 - d_2$ (result is a (fractional) Julian number)
$(d_1 - d_2)$ `MONTH`	`MONTHS_BETWEEN(`d_1`, `d_2`)` (result is a (fractional) number of months)
$i * n$	$j * n$
$n * i$	$n * j$
i_1 / i_2	j_1 / j_2
i / n	j / n
$+ i$	$+ j$
$- i$	$- j$
Other Operators:	
`CAST(`d` AS DATE)`	`TRUNC(`d`)`
`CAST(`d` AS TIME)`	`TO_DATE(TO_CHAR(`d`, 'HH24:MI:SS'),` `  'HH24:MI:SS')`
`CAST(`d` AS TIMESTAMP)`	
d **is a** `DATE`	`TRUNC(`d`)`
d **is a** `TIME`	`TRUNC(SYSDATE)+(`d`-TRUNC(`d`))`
`CAST(`i` AS INTERVAL YEAR TO MONTH)`	**not possible**
`CAST(`i` AS INTERVAL DAY TO SECOND)`	j
`CAST(`d` AS CHAR)`	`TO_CHAR(`d`,` `  'YYYY-MM-DD' || ' HH24:MI:SS')`
`CAST(`i` AS CHAR)`	`TRUNC(`j`, 0) || TO_CHAR(`j`` `  + TO_DATE(1, 'J'), ' HH24:MI:SS')`
`CAST(`i` AS INTEGER)`	
i **is** `DAY`	`TRUNC(`j`, 0)`
i **is** `HOUR`	`TRUNC(`j` * 24, 0)`
i **is** `MINUTE`	`TRUNC(`j` * 1440, 0)`
i **is** `SECOND`	`TRUNC(`j` * 86400, 0)`
`EXTRACT(DAY FROM `d`)`	`TRUNC(`d`, 'DD') - TRUNC(`d`,'MM') + 1`
`EXTRACT(DAY FROM `i`)`	`TRUNC(`j`,0)`
`EXTRACT(HOUR FROM `i`)`	`TRUNC(`j`*24, 0) - (TRUNC(`j`,0)*24)`

Table 3.7 (continued)

SQL-92	Oracle8 Server Equivalent
Operators not in SQL-92:	
convert d to Julian day	TO_CHAR(d, 'J')
convert Julian day n to DATE	n + TO_DATE(1, 'J')
pick the earliest date	LEAST(d_1, ..., d_n)
pick the latest date	GREATEST(d_1, ..., d_n)
pick the last day of the month	LAST_DAY(d)
get the next day of the week	NEXT_DAY(d, 'Monday')

are necessary to highlight these differences. So a Start_Time is an instant, but an Incrmnt_Time is an interval (the comment states that the "Increment Time is taken in reference to the start of the period").

Intervals were represented with numerics. The Time_String_In_Hole column of the Well_Log_Service table is of type NUMBER(8,2); the granularity must be inferred from other information. For example, the comment for the Sample_Interval column of the Seis_Trace_Hdr table specifies "sampling interval in milliseconds."

When he needed more control over the granularity, Jim found that Oracle was of little help. Most of these values were expressed as a pair of columns. Although the Start_Time is a DATE, thereby utilizing a possible granularity of seconds, the Incrmnt_Time is a NUMBER(7,2) column, coupled with an Incrmnt_Time_Unit column, of type VARCHAR(12). The Well_Test_Hdr table specifies the Start_Time, with the Well_Test_Incrmnt providing a specific observation (ordered by the Incrmnt_Obs_No column). Consider though how you would calculate in SQL the starting time of a particular observation. The Incrmnt_Time is a fractional number of units, which must be multiplied by the size of the unit, determined from the name of that unit, then added to the Start_Time. This calculation must be done in terms of fractional days, so it is important that the size of the units are stored in that manner (Oracle8 Server will blithely permit this calculation without this requirement being satisfied, with incorrect results).

3.5.7   UniSQL

UniSQL supports the DATE, TIME, and TIMESTAMP data types. TIMESTAMPs are constrained to a granularity of a second; their range is only January 1, 1970 through 03:14:07 January 19, 2038. TIMEs also are to a granularity of one second. Subtracting two dates yields an integral number of days; subtracting two TIME or TIMESTAMP values will yield an integral number of seconds.

Table 3.8 shows how the facilities in SQL-92 can be simulated, to some degree, in UniSQL. In the SQL-92 column, d denotes a datetime value, i an interval value,

The Start of the Millennium

Given that there is no 0 A.D. (see page 27), the issue is then raised of when the millennium starts, 2000 A.D. or 2001 A.D. Those of a more mathematical constitution generally insist on the latter, an example being the Royal Greenwich Observatory in Cambridge, England, as reported in the *New York Times* on December 8, 1996. Pop culture clearly sides with the former; we can effectively argue that the millennium has already arrived months before the end of 2000 A.D. But in that case, the first millennium was but 999 years long, thereby belying its etymological basis ("one thousand years," in Latin, see page 86), and raising the questions of how long is a century, or even a decade? Stephen Jay Gould is unapologetic on this—it is clear his sentiments lie with the solution that a (nay, every) millennium is 10 centuries long, a century consists of 10 decades, and all decades are 10 years, save the first, which was but 9 years in length, thus, by incontrovertible logic, the first century contained 99 years and the first millennium (but thankfully, not the present one) comprised 999 years.

and n a numeric (exact or approximate) value. In the UniSQL column, d denotes a UniSQL TIMESTAMP value (to the granularity of seconds), and i denotes a UniSQL INTEGER representing an integral number of granules (seconds).

3.6   THE YEAR 2000 PROBLEM*

(We remind you that the asterisk in this section heading—and in some later section headings—indicates advanced material that may be skipped on a first reading.)

The year 2000 problem has often been abbreviated to the "Y2K problem" by those who love acronyms, and termed the "Millennium Bug" by those who want a more catchy name. The problem involves software that stores dates using only two digits for the year. That raises the issue of determining what the year 00 denotes. If it denotes 2000, then everything will generally be fine when that year arrives. On the other hand, if it denotes 1900, then all manner of difficulties will arise. A phone call that starts the night of December 31, 1999, and extends a little past midnight, could be charged for 100 years of air time, resulting in a horrendous bill. A bill due in December 1999 but not paid until the next month could result in an interest fee of gigantic proportions. Or the software might just fail, freezing bank accounts and leaving flight controllers with no information on their tracking screens. Indeed, consumers are already being affected. The expiration date on Mastercard and Visa credit and debit cards is listed as MM/YY. Recently, cards were issued with an expiration date of 01/00, and these cards are being denied, as having expired some 98 years ago. While the larger authorization centers have updated their software, some smaller authorization centers still cannot accept those cards (as of June 1998).

Table 3.8 SQL-92 operations in UniSQL.

SQL-92	UniSQL Equivalent
Types:	
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP (to second granularity)
TIME WITH TIME ZONE	no equivalent
TIMESTAMP WITH TIME ZONE	no equivalent
INTERVAL YEAR TO MONTH	no equivalent
INTERVAL DAY TO SECOND	INTEGER (integer number of seconds or days)
Literals:	
DATE '1997-01-01'	DATE '01/01/1997'
TIME '12:34:56'	TIME '12:34:56'
TIMESTAMP '1997-01-01 12:34:56'	TIMESTAMP '01/01/1997 12:34:56'
INTERVAL '3-4' YEAR TO MONTH	not possible
INTERVAL '1 23:45:12' DAY TO SECOND	171812 (seconds)
Predicates:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 BETWEEN d_2 AND d_3	d_1 BETWEEN d_2 AND d_3
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 BETWEEN i_2 AND i_3	i_1 BETWEEN i_2 AND i_3
d IS NULL	d IS NULL
i IS NULL	i IS NULL
(d_1, i) OVERLAPS (d_2, d_3)	$d_1 < d_3$ AND $d_2 < d_1 + i$
Datetime Constructors:	
$d + i$	$d + i$
$i + d$	$i + d$
$d - i$	$d - i$
d AT i	not supported
d AT LOCAL	not supported
CURRENT_DATE	not supported
CURRENT_TIME	not supported
CURRENT_TIMESTAMP	not supported
Interval Constructors:	
$i_1 + i_2$	$i_1 + i_2$
$i_1 - i_2$	$i_1 - i_2$
$(d_1 - d_2)$ *qual*	$d_1 - d_2$ (result is an integer number at the indicated granularity)
$(d_1 - d_2)$ MONTH	not possible

Table 3.8 (continued)

SQL-92	UniSQL Equivalent
Interval Constructors (cont.):	
$i * n$	$i * n$
$n * i$	$n * i$
i_1 / i_2	j_1 / j_2
i / n	j / n
$+ i$	i
$- i$	$- i$
Other Operators:	
CAST(d AS DATE)	CAST(d AS DATE)
CAST(d AS TIME)	CAST(d AS TIME)
CAST(d AS TIMESTAMP)	CAST(d AS TIMESTAMP)
CAST(i AS INTERVAL YEAR TO MONTH)	not possible
CAST(i AS INTERVAL DAY TO SECOND)	i
CAST(d AS CHAR)	CAST(d AS CHAR)
CAST(i AS CHAR)	CAST(i AS CHAR)
CAST(i AS INTEGER)	i already an integer
EXTRACT(DAY FROM d)	EXTRACT(DAY FROM d)
EXTRACT(DAY FROM i)	not possible
EXTRACT(HOUR FROM i)	not possible

The year 2000 problem is a specific instance of a more general problem of an (often unstated) assumption that will be invalidated purely by the course of time.

The year 2000 problem is but one instance of a more general problem, that of making assumptions that are invalidated purely by the course of time. Here, the assumption was that two digits suffice for the year, which is a valid assumption if all the information is contained in a single century. Indeed, one digit suffices if the information is contained in a particular decade. What has captured the imagination of the public and the press about the year 2000 problem is due to four factors:

1. The underlying assumption was made, sometimes implicitly, in so many software systems.
2. The problem was ignored until (almost) too late, despite being recognized for decades: the 1965 Multics system used a 71-bit microsecond representation.
3. The assumption is invalidated at exactly the same time (well, within a single 24-hour period) for these systems.
4. The systems involved are generally legacy systems, with underlying source code a much-modified mess, or worse, simply unavailable.

These factors conspire to make fixing the problem exceedingly expensive, yet the third factor imposes an unavoidable deadline for coming up with an acceptable solution, with often disastrous consequences if this deadline is not met. The classic movie *High Noon* comes to mind, with Gary Cooper in almost every scene glancing at that pendulum clock inexorably ticking away the minutes. I, as author of this book, which contains material on the Y2K problem, am similarly constrained by this approaching deadline.

The year 2000 problem may be found in hardware, specifically physical clocks, and all manner of software, from programming languages to operating systems, from DBMSs to applications. As the present book considers time-varying applications and SQL, we will limit our discussion to those topics. In particular, we will eschew discussion of the extremely important and challenging task of identifying, repairing, and testing legacy code. At the same time, we will examine the phenomenon more generally, by highlighting *time-dependent assumptions* that involve both the year 2000 and other dates. As we will emphasize, it is effectively impossible to completely avoid these problems, but we can minimize them, and be aware of those that are present.

3.6.1   Year 2000 Compliance and Certification

Each organization must develop its own definition of compliance, termed "year 2000 compliance," or in the general case, "century compliance." The following language is recommended by the Chief Information Officers Council Sub-Committee on the Year 2000 for voluntary use by federal agencies in their solicitations and contracts for year 2000 compliant products.

> The contractor warrants that each hardware, software, and firmware product delivered under this contract and listed below shall be able to accurately process date/time data (including, but not limited to, calculating, comparing, and sequencing) from, into, and between the twentieth and twenty-first centuries, and the years 1999 and 2000 and leap year calculations to the extent that other information technology, used in combination with the information technology being acquired, properly exchanges date/time data with it.

Year 2000 certification is defined by Mitre as "a measure of assurance by a designated Y2K authority (or their representative), that an item is operationally ready. Any company working toward Year 2000 compliance must ultimately be concerned with Year 2000 certification."

For database products, vendors generally certify, viewing themselves as a Y2K authority, that their products are (or are not) year 2000 compliant, generally based on their specific definition of year 2000 compliance.

To address the problem of two-digit dates, vendors generally define a "window" of 100 years within which a two-digit date is interpreted. This interpretation is often called the "implied century rule." The window is sometimes dependent on when the date is interpreted. In many cases, the window is xx00–xx99, meaning

that a two-digit date is interpreted as being in the current century. So, 34 would then be interpreted in 1998 as 1934, and in 2001 as 2034; the window jumped in the year 2000. The year 2000 problem can be rephrased as the "window ending at 99 and jumping at 00 problem," in that the window goes from 1900 to 1999, and jumps to the next century (2000–2099) in the year 2000. Products vary on where this window is located and when it jumps (in the best scenario, the jump and the window boundary should be far apart).

It would have been to the consumer's benefit for vendors to come up with a common approach to the year 2000 problem: specify a common window, specify a common jump point, and specify a common mechanism to ensure backward compatibility. Vendors in their wisdom have taken exactly the opposite tack: every approach is unique. Would it be cynical to think that this is intentional?

Two-digit dates, and indeed, any finite representation, imply that eventually the range will be exhausted, and some kind of discontinuity will result. All that the windowing approaches do is delay the discontinuity at the jumps. In scanning the various extant DBMSs, we see difficulties with the years 2000, 2030, 2050, 2079, 2100, 4712, and 10,000. This implies that programmers will be kept busy both fixing applications as these notable years approach and converting applications from one DBMS to another that takes a different approach.

3.6.2   SQL-92

SQL-92 DATEs and TIMESTAMPs both use four digits to represent the year. Applications using this standard are fine until the year 9999 C.E., and thus exhibit the "year 10,000 problem," but are fortunately not affected by the year 2000 transition.

As will be discussed in the following section, an SQL-92 TIME value is actually a funny kind of interval. An alternative characterization is that TIME has a midnight problem: its meaning changes every midnight. In the above terminology, a TIME value has a window of 24 hours (from midnight to midnight) and a jump time of midnight.

Concerning leap years, the question could be phrased several ways.

- "Is the value of `DATE '2000-02-29'` valid?" The standard states "Within the definition of a ⟨datetime literal⟩, the ⟨datetime value⟩s are constrained by the natural rules for dates and times according to the Gregorian calendar" [44, p. 75]. As the contribution of the namesake of this calendar, Pope Gregory XIII, was to specify that century years not divisible by 400 would no longer be leap years, this clearly indicates that the "natural rules" would consider the year 2000 to be a leap year.
- "Is `(DATE '2000-03-01' - DATE '2000-02-01') DAY` the value 29 days?" The SQL-92 standard specifies datetime subtraction as "a) A [here, `DATE '2000-03-1'`] and B [here, `DATE '2000-02-1'`] are converted to integer scalars $A2$ and $B2$, respectively, in units Y [here, `DAY`] as displacements from some implementation-

dependent start datetime. b) The result is determined by effectively computing *A2–B2...*" [44, p. 137]. Is this year 2000 compliant? It all depends on what is meant by the word "converted." Jim Melton has told me that those on the standards committee would agree that the value 29 is correct and intended, but I counter that the standard itself should be unambiguous and clear, especially on such an important question.

- "Is `DATE '2000-03-01' - INTERVAL '29' DAY` the value `DATE '2000-02-01'`?" The SQL-92 standard specifies such expressions as "Arithmetic is performed so as to maintain the integrity of the datetime data type that is the result of the ⟨datetime value expression⟩. This may involve carry from or to the immediately next more significant ⟨datetime field⟩" [44, p. 133]. Here, we are concerned about the carry from the DAY field to the MONTH field. Presumably the "integrity of the datetime data type" refers back to the definition of a DATE literal, which we saw above treats the year 2000 as a leap year.

My conclusion: since the conversion for date difference is not explicitly spelled out, we don't *know* that the conversion will treat the year 2000 as a leap year, and so the SQL-92 standard should not be considered year 2000 compliant.

What should programmers do to ensure that new code being written does not exhibit the year 2000 problem? Quite simply, use four-digit years, as the SQL-92 standard mandates. Of course, this aphorism ignores the requirement that new code work with and indeed be compatible with existing legacy programs, which might themselves use only two-digit years. As Mark Haselkorn said in an interview published in the February 1998 issue of the *Institute* of the IEEE:

> Y2K is not about hardware, firmware and operating software (platforms). It is not even about application software and even data. It is not even about users, organizations, economies and nations—it's about all of them together. You cannot change your computer to a Y2K-safe one and think you have fixed the problem. You still have software that runs on it and, more importantly, data you have accumulated that has great value to you that must be part of the fix.

With that chastening fresh on our mind, we now turn to specific DBMS products. A critical disclaimer: these products and their level of compliance are highly fluid, with new techniques being developed daily to achieve compliance. The remarks below reflect the situation as this is being written, in June 1998. This material will surely be somewhat out of date when it appears in print. You are urged to contact your vendor for information on year 2000 compliance.

3.6.3   IBM DB2 Universal Database

IBM DB2 UDB DATEs and TIMESTAMPs both use four-digit years, and so are year 2000 compliant. The year 2000 is considered a leap year by DB2 UDB.

- `DATE('2000-02-29')` is a valid DB2 UDB DATE.

- `DATE('2000-02-01') + 1 MONTH` yields March 1, 2000.
- `DATE('2000-03-01') - 1 MONTH` yields February 1, 2000.
- `DATE('2000-02-01') + 00000100.` yields March 1, 2000.
- `DATE('2000-03-01') - 00000100.` yields February 1, 2000.
- `DATE('2000-02-28') + 1 DAY` yields February 29, 2000.
- `DATE('2000-03-01') - 1 DAY` yields February 29, 2000.

3.6.4 Informix

Informix defines "year 2000 compliant" as

> . . . the use or occurrence of the dates on or after January 1, 2000, will not adversely affect the performance of the Informix products with respect to four-digit data dependent data or the ability of such products to correctly create, store, process, and output information related to such date data.

The internal formats of Informix DATE and DATETIME data types both support four digits for the year.

For two-digit input of dates, Informix has added the DBCENTURY environment variable. There are four values for this environment variable: past ('P'), future ('F'), closest ('C'), and present ('R'). If no value is specified, the default is present semantics. Of course, this environment variable is not used if four digits are supplied.

- *Present semantics ('R')*   The present century provides the window. The window is 00–99 and the jump date is the end of the century.
  `DATE("1-1-1")` when entered on June 22, 1998, evaluates to January 1, 1901. When entered in 2002, this literal evaluates to January 1, 2001.
- *Past semantics ('P')*   The past and present centuries provide two windows and produce two expanded date values. The one that is prior to the current date is chosen. If both dates are prior to the current date, the date that is closest to the current date is chosen.
  `DATE("1-1-99")` when entered on June 22, 1998, produces January 1, 1899 and January 1, 1999; January 1, 1899 is chosen. When entered in 2002, this literal evaluates to January 1, 1999. `DATE("1-1-97")` when entered on June 22, 1998, produces January 1, 1897 and January 1, 1997; January 1, 1997 is chosen. While seemingly only two years before `DATE("1-1-99")`, the resulting four-digit date is 98 years *later*. When entered in 2002, `DATE("1-1-97")` evaluates to January 1, 1997.
  In our terminology, the window is the preceding 100 years (for a current date of June 22, 1998, the window extends from June 21, 1898 to June 21, 1998) and the jump date is each day (though the window only moves forward one day). `DATE("6-22-98")` will evaluate to June 22, 1898, exactly 100 years ago today; tomorrow that same literal will evaluate to June 22, 1998.

- *Future semantics ('F')*   The present and future centuries provide two windows and produce two expanded date values. The one that is *after* the current date is chosen. If both dates are after the current date, the date that is closest to the current date is chosen.

   DATE("1-1-99") when entered on June 22, 1998, produces January 1, 1999 and January 1, 2099; January 1, 1999 is chosen. When entered in 2002, this literal evaluates to January 1, 2099. DATE("1-1-97") when entered on June 22, 1998, produces January 1, 1997 and January 1, 2097; January 1, 2097 is chosen. As before, while this literal seems to be only two years before DATE("1-1-99"), the resulting four-digit date is 98 years *later*. When entered in 2002, DATE("1-1-97") evaluates to January 1, 2097.

   The window is the following 100 years (for a current date of June 22, 1998, the window extends from June 23, 1998 to June 22, 2098) and the jump date is each day. DATE("6-22-98") will evaluate to June 22, 2098, exactly 100 years in the future; tomorrow that same literal will evaluate to June 22, 1998.

- *Closest semantics ('C')*   The past, present, and future centuries provide *three* windows and produce three expanded date values. The one that is closest to the current date is chosen.

   DATE("1-1-99") when entered on June 22, 1998, produces January 1, 1899, January 1, 1999, and January 1, 2099 as candidates; January 1, 1999 is chosen. When entered in 2002, this literal evaluates to January 1, 1999. DATE("1-1-97") when entered on June 22, 1998, produces January 1, 1897, January 1, 1997, and January 1, 2097 as candidates; January 1, 1997 is chosen. Unlike with the other semantics, this literal seems to be only two years before DATE("1-1-99"), and in fact the resulting four-digit date is also two years earlier. When entered in 2002, DATE("1-1-97") evaluates to January 1, 1997.

   There is still anomalous behavior with this semantics; it is just distant in time. DATE("1-1-48") when entered on June 22, 1998, evaluates to January 1, 2048. DATE("1-1-49"), seemingly one year later, evaluates to January 1, 1949.

   The window is the 100 years centered on the current date (for a current date of June 22, 1998, the window extends from June 22, 1948 to June 22, 2048) and the jump date is each day. Actually, this brings up a slight unspecification of closest semantics. Because 2000 is a leap year, any 100-year window between 1900 and 2100 will contain an odd number of days (36,525); hence, there will be a (single) closest value. However, when the present date is after the year 2100, the current window will contain an even number of days (36,524), and there may occur two potential dates equidistant from the current date. As an example, if the current date is January 1, 2105, the current semantics provides for the value DATE("1-1-55") January 1, 2055, January 1, 2155, and January 1, 2255. The last date is certainly out of consideration, but the first two are exactly 18,262 days from the current date, and so neither is preferred (or worse, both are preferred). We

simply don't know which date will be returned when this literal is evaluated on that date.

It seems that the closest semantics provides the most intuitive behavior, in that the anomalies occur with distant dates.

The year 2000 is considered a leap year by Informix.

- `DATE("2/29/2000")` is a valid SQLServer DATETIME.
- `DATE("3/1/2000") - DATE("2/1/2000")` yields 29 days.
- `DATE("2/28/2000") + INTERVAL(1) DAY` yields February 29, 2000.
- `DATE("3/1/2000") - INTERVAL(1) DAY` yields February 29, 2000.

3.6.5 Microsoft Access

Microsoft's definition of year 2000 compliance is as follows:

> A Year 2000 Compliant product from Microsoft will not produce errors processing date data in connection with the year change from December 31, 1999 to January 1, 2000 when used with accurate date data in accordance with its documentation and the recommendations and exceptions set forth in the Microsoft Year 2000 Product Guide, provided all other products (e.g., other software, firmware and hardware) used with it properly exchange date data with the Microsoft product. A Year 2000 Compliant product from Microsoft will recognize the Year 2000 as a leap year.

Microsoft classifies its products into five categories.

- **Compliant** The product fully meets Microsoft's standard of compliance. May have prerequisite patch or service pack for compliance.
- **Compliant with minor issues** The product meets Microsoft's standard of compliance with some disclosed exceptions that constitute minor date issues.
- **Not compliant** The product does not meet Microsoft's standard of compliance.
- **Testing yet to be completed** Product test is not yet complete or has not started but will be tested.
- **Will not test** The product will not be tested for compliance.

Microsoft considers both Access 95 and Access 97 to be year 2000 compliant, in terms of the above definition.

Access Date/Time values have a range of 9899 years (100 A.D. to 9999 A.D.). However, Access 95 and Access 97 differ on the interpretation of two-digit dates.

Access 95 by default allows two-digit and four-digit years on input. The user can define formats, via an input mask. Included in the predefined formats is a Short Date format, which forces users to enter dates in a two-digit year format.

Parsing of dates is controlled by the `OLEAUT32.DLL` file in the system folder. The interpretation of the two-digit dates depends on the version of this file.

For version 2.20.4048 and lower, two-digit dates are considered to be in the same century. Hence, `DateValue('01/01/00')` will be interpreted as January 1, 1900.

For version 2.20.2049 and higher, dates 1/1/00 through 12/31/29 are interpreted as being in the next century. So these dates would be interpreted in 1998 as 1/1/2000 through 12/31/2029. Dates 1/1/30 through 12/31/99 are interpreted in the current century, so would be interpreted in 1998 as 1/1/1930 through 12/31/1999. Effectively, this delays the year 2000 problem until the year 2030.

For Access 97, there is only one possible interpretation: the window is from year 30 of this century to year 29 of the next century, and shifts at the year 2000.

The year 2000 is considered a leap year by Access.

- `DateValue('2000-02-29')` is a valid Access Date/Time.
- `DateValue('2000-03-01') - DateValue('2000-02-01')` yields 29 days.
- `DateAdd("m", DateValue('2000-02-01'), 1)` yields March 1, 2000.
- `DateAdd("m", DateValue('2000-03-01'), -1)` yields February 1, 2000.
- `DateAdd("d", DateValue('2000-02-28'), 1)` yields February 29, 2000.
- `DateAdd("d", DateValue('2000-03-01'), -1)` yields February 29, 2000.

3.6.6   Microsoft SQL Server

Microsoft considers SQL Server 6.5 and 7.0 to be year 2000 compliant.

Microsoft SQL Server supports two temporal data types, DATETIME, with a range of 1753 to 9999, and SMALLDATETIME, with a range of January 1, 1900 to June 6, 2079. Either data type allows you to specify only the last two digits of the year, with values less than 50 interpreted as 20yy (e.g., 17 is interpreted as 2017) and values greater than 50 interpreted as 19yy (e.g., 57 is interpreted as 1957). Put another way, the window starts at 1950 and is fixed, representing a "year 2050 problem," as well as a "year 2079 problem," beyond which SMALLDATETIMEs are not defined.

Concerning the year 2000 being considered a leap year, several bugs in this regard were fixed in Service Packs 2 and 5 of SQL Server 6.5.

- `CONVERT(DATETIME, "2000-02-29", 102)` is a valid SQL Server DATETIME.
- `DateDiff(day, CONVERT(DATETIME, "2000-02-01", 102), CONVERT(DATETIME, "2000-03-01", 102))` yields 29 days.
- `DateAdd(month, 1, CONVERT(DATETIME, "2000-02-01", 102))` yields March 1, 2000.
- `DateAdd(month, -1, CONVERT(DATETIME, "2000-03-01", 102))` yields February 1, 2000.
- `DateAdd(day, 1, CONVERT(DATETIME, "2000-02-28", 102))` yields February 29, 2000.
- `DateAdd(day, -1, CONVERT(DATETIME, "2000-03-01", 102))` yields February 29, 2000.

3.6.7 Sybase SQLServer

Sybase's definition of "century compliant" is as follows:

- "General Integrity: No value for the current date will interrupt normal operation: the system returns the correct date accurate to century in response to a request for current date, and the software is unaffected by any value returned."
- "Date Integrity: Correct results are returned in the operation of all legal and calendar operations of dates that span century marks within the range of the software."
- "Explicit Century: The software's internal date storage format explicitly includes the century and reporting formats allow date representation in the full century format."
- "Implicit Century: On encountering data that does not include the century either from transaction input or from an external data source, the century value is unambiguously inferred by the software."

Under this definition, Sybase considers SQLServer to be century compliant.

Sybase SQLServer supports two temporal data types, DATETIME, with a range of 1753 to 9999, and SMALLDATETIME, with a range of January 1, 1900 to June 6, 2079. Either data type allows you to specify only the last two digits of the year, with values less than 50 interpreted as 20yy (e.g., 17 is interpreted as 2017) and values greater than 50 interpreted as 19yy (e.g., 57 is interpreted as 1957). Put another way, the window starts at 1950 and is fixed, representing a "year 2050 problem," as well as a "year 2079 problem," beyond which SMALLDATETIMEs are not defined.

The year 2000 is considered a leap year by Sybase SQLServer.

- `CONVERT(DATETIME, "2000-02-29", 102)` is a valid SQLServer DATETIME.
- `DateDiff(day, CONVERT(DATETIME, "2000-02-01", 102), CONVERT(DATETIME, "2000-03-01", 102))` yields 29 days.
- `DateAdd(month, 1, CONVERT(DATETIME, "2000-02-01", 102))` yields March 1, 2000.
- `DateAdd(month, -1, CONVERT(DATETIME, "2000-03-01", 102))` yields February 1, 2000.
- `DateAdd(day, 1, CONVERT(DATETIME, "2000-02-28", 102))` yields February 29, 2000.
- `DateAdd(day, -1, CONVERT(DATETIME, "2000-03-01", 102))` yields February 29, 2000.

3.6.8 Oracle8 Server

The Oracle8 Server supports the DATE type, which includes a four-digit year, and so accommodates the year 2000. However, Oracle8 Server can store a maximum year of 4712; this raises the impending "year 4712 problem."

Oracle's `TO_DATE` function takes two strings, a value string and a format string. Applications using `YYYY` in the format string are safe; applications using only two

digits (e.g., a format string of `'YY-MM-DD'`) will need to be examined, as the function will interpret such value strings as being in the current century. For example, `TO_DATE('450123', 'YYMMDD')` when evaluated today (June 19, 1998) returns the value denoting January 23, 1945. This means that `TO_DATE('000101', 'YYMMDD')` will evaluate to January 1, 1900.

For such applications, the Oracle7 Server and the Oracle8 Server provide a special year format mask, RR. Values with years between 0 and 49 that are stored in 1998 with the RR format are interpreted to be in the twenty-first century; for example, `TO_DATE('000101', 'RRMMDD')` will evaluate to January 1, 2000. This format still experiences a shift in semantics at the millennium. Say in 1998 an application attempts to store a future date of 2051, using a value string of `'550101'` and a format of RR. This will be interpreted as 1955. Two years later, that same value string will be interpreted as 2055. This raises a "year 2050 problem."

Summarizing, the YY format uses centuries (years having the same first two digits are in the same century) as both the window and the window transition. The RR format has a window starting at year 50 and going to year 49, with the window transition occurring at year 0 of the century.

The year 2000 is considered a leap year by Oracle8 Server.

- `TO_DATE('2000-02-29', 'YYYY-MM-DD')` is a valid Oracle DATE.
- `TO_DATE('2000-03-01', 'YYYY-MM-DD') - TO_DATE('2000-02-01', 'YYYY-MM-DD')` yields 29 Julian days.
- `ADD_MONTHS(TO_DATE('2000-02-01', 'YYYY-MM-DD'), 1)` yields March 1, 2000.
- `ADD_MONTHS(TO_DATE('2000-03-01', 'YYYY-MM-DD'), -1)` yields February 1, 2000.
- `ADD_DAYS(TO_DATE('2000-02-28', 'YYYY-MM-DD'), 1)` yields February 29, 2000.
- `ADD_DAYS(TO_DATE('2000-03-01', 'YYYY-MM-DD'), -1)` yields February 29, 2000.

3.7   SUBTLETIES*

It is critical that the limitations and subtle ramifications of the representation of instants provided by the DBMS be understood. As we'll see, the meaning of a temporal value is somewhat arbitrary, with the application providing some of the semantics.

3.7.1   Datetimes

While an instant is, well, instantaneous, SQL and all DBMSs assume a discrete time line of various *granularities*, such as second, day, and year, and indicate only the particular granule in which the instant is located. The *event* of an individual well sample extraction occurred at a specific instant, but we may care to record only the

B.C., A.D., B.C.E., C.E., and B.P.

The prevailing system before the use of B.C.–A.D., at least in the Western world, was A.U.C. (*ab urbe condita*, literally, "from the foundation of the city," which, being in Latin, of course meant Rome). Dionysius (see page 27) pegged 1 A.D. at 754 A.U.C. There is the slight problem that King Herod died in 750 A.U.C., which translates to 4 B.C. Since Herod was ostensibly alive at the birth of Jesus, we have the interesting oxymoron of Christ being alive in 4 B.C., that is, four years before the birth of Christ. In an effort to be less parochial, B.C. has been retermed B.C.E. ("Before the Christian Era," or even better, "Before the Common Era"), with A.D. renamed C.E. ("Common Era"). Even more PC is B.P. ("Before the Present"), that is, interpreted with reference to the year of publication of the source. This book will have a copyright date of 1999 (as will all books published between July 1, 1998, and June 30, 1999), so the millennium will reputedly end at −1 B.P. (since it is *after* the present). The problem with this scheme is that it carries with it an extra obligation for the author: if this book is delayed but a few months, I will have to adjust that B.P. date above, as well as all others that appear herein.

An instant has no duration, but its representation as a particular granule always does.

day granule in which that event occurred. If multiple tests are performed during the day on a well sample, a time granularity, say, hour or even second, may be appropriate. An instant has no duration, but its representation, as a particular granule, always does (when utilizing a discrete time line).

The concept of an instant is independent of any particular calendar. SQL has chosen the Gregorian calendar for its representation of an instant. The use of a specific calendar, especially one so infused with politics, brings with it subtle difficulties. The Gregorian calendar was proclaimed by Pope Gregory XIII in 1582, with adoption by the Catholic states within a year. However, in some places adoption was very slow. The Protestant German states adopted the Gregorian calendar in 1699, Japan in 1873, and Greece only in 1923. Muslim countries tend to retain calendars based on Islam, and Asian countries generally use lunar or hybrid (solar/lunar) calendars.

Consider an art dealer who has in hand a letter written by the artist Enoch Seeman stating that his "Portrait of the Countess of Berkeley" was completed on "March 23, 1735." The art dealer entering this date in SQL as `DATE '1735-03-23'` would in fact be specifying a day some 11 days *before* the painting was finished. The reason is that before 1752 England and its colonies used the Julian calendar, which differs from the Gregorian calendar only in the presence of century leap years (in fact, this difference between the solar year and the calendar year precipitated the construction of the Gregorian calendar). So the correct denotation in SQL is `DATE '1735-04-03'`. Had the letter been written in, say, Paris, the same day,

rather than at Berkeley Castle, it would have recorded the date "April 3, 1735." The geographical location of the historical reference supplies the calendar in force, which then implies the correction, if any, required before the date can be specified in SQL.

As the Gregorian calendar was undefined before 1582, SQL presumably extrapolates this calendar backwards using its rules to 1 C.E., obtaining what some have called the "proleptic Gregorian calendar" (which as has been noted is a misnomer, because "proleptic" refers to a future act). Problems occur in the opposite direction. Because the tropical year is 365.242,191 days, and the Gregorian year (due to the rather complex leap year rules, see page 37) is 365.2425 days, the calendar year will be one day longer than the astronomical year in 4000 C.E. and two days longer in 8000 C.E. A further refinement to the Gregorian calendar designating years evenly divisible by 4000 as common (not leap) years would ensure accuracy to within one day in 20,000 years. If this refinement was legislated, dates after 4000 C.E. stored in the database would be off by one or two days. As it is, the vernal equinox will occur on `DATE '1997-03-21'`, `DATE '4000-03-22'`, and `DATE '8000-03-23'`.

While my watch tells me in which second (approximately) the current instant occurs, we desire a more precise and universal definition. As mentioned, SQL uses UTC. UTC is based on cesium atomic clocks, which are accurate to within a second in a million years. A step adjustment of a fraction of a second at the beginning of each month correlates UTC with mean solar time (the average time between noons, when the sun is directly overhead). In October 1967, the second in the International System of Weights and Measures was defined to be 9,192,631,770 periods of the radiation emitted by the transition between two hyperfine states of the cesium 133 atom in the ground state. On January 1, 1972, the atomic second became the practical unit of time. The UTC clock runs just a little fast with respect to mean solar time, gaining about a second a year. UTC is adjusted by applying leap seconds on January 1 or July 1 to keep UTC within 0.7 seconds of solar time.

So, what does this mean for an SQL user? The database is a model of reality. An event in reality occurs at a particular instant. The representation of that instant in the database should identify the particular day, or second, or microsecond, in which the instant occurred, to the precision chosen by the user.

Say that `TIMESTAMP(0)` is specified. Then the user is satisfied if the instant can be characterized to within a second. Alternatively, if the database specifies that an event happened at a particular timestamp value, the user would like to identify the second in reality during which the event occurred.

For times between 1958 and about 1998, this correspondence between reality and its representation in an SQL-compliant database is well defined. The problem with future time is that leap seconds are determined by a committee, after reviewing astronomical records indicating how far apart solar time is from atomic time. When the differential gets too great, a leap second is mandated. Since January 1972,

22 leap seconds have been added, the last inserted just before 12:00 A.M., January 1, 1999 (do you remember?). Specifically, the sequence of UTC markers was `'1998-12-31 23:59:59'`, `'1998-12-31 23:59:60'`, `'1999-01-01 00:00:00'`. While it is probable that a leap second will be added in 2000, exactly when future ones will be added is a bureaucratic decision, informed by changes, which cannot be precisely predicted, that are slowing down the earth's spin. The decision is generally made around five months before the leap second is effected.

Leap seconds imply that some minutes, such as the last minute of the year 1995, contain 61 seconds. In fact, UTC allows two leap seconds to be added, so the seconds value of a TIMESTAMP is restricted to 0.0 through 61.999. … UTC also allows the omission of a leap second; such minutes will only contain 59 seconds. However, a leap second has not yet been omitted in the more than two decades since UTC was defined.

Since UTC is coordinated with solar time, the sun will be directly overhead within 0.7 second of `'2222-01-01 12:00:00'`. However, the number of seconds between TIMESTAMP `'1997-01-15 11:35:29.123456'` and that instant is not known. As we will see later, the DBMS will provide a number that should be close, within 300 seconds or so: the DBMS will assume no leap seconds in the intervening time. The number of intervening minutes is known precisely, 118,843,224, because there are no leap minutes. Leap seconds extend the minute to which they are assigned (such as `'1999-12-31 23:59:60'`, above); they do not accumulate into a leap minute.

Before 1958, UTC is not defined. One possibility is that the definition of UTC is extrapolated backward to 1 C.E. The definition of UTC in 1958 is ephemeris seconds

Which second an SQL timestamp before 1958 denotes is not adequately specified in the standard.

as measured with an atomic clock, with adjustments for changes in the earth's rotation. So there are at least two possibilities. One is that the proper adjustment is made each month, in that the first second of each month is a little longer or shorter than the other seconds, so that the solar day is coordinated with UTC. The problem with this approach is that it is impossible to correlate SQL timestamps with any other time, such as unadjusted ephemeris time, used by astronomers. A second possibility is that we can assume that each minute contains exactly 60 (unadjusted) ephemeris seconds, which emphasizes equal-sized seconds but which becomes uncorrelated with the solar day as we go back in time.

The moral is that no one really knows which second *in reality* is denoted by an SQL timestamp before 1958. Returning to the letter written by our artist Enoch Seeman, that day could have started on the second denoted by TIMESTAMP `'1735-03-23 00:00:00'`, or perhaps it started at `'1735-03-23 00:00:10'`, or perhaps even at `'1735-03-22 23:55:04'`. Such uncertainty is frustrating. Ideally, that date *should* start precisely at midnight: `'1735-03-23 00:00:00'`. But the standard is surprisingly silent on precisely what instant in reality such a value means.

Given this imprecision in the standard, it is the responsibility of the application designer to supply the semantics for SQL timestamp values. When a value in a specific system—be it atomic (TAI), barycentric coordinate (TCB), barycentric dynamical (TDB), ephemeris, geocentric coordinate (TCG), mean solar, sidereal, terrestrial (TT), universal (UT0, UT1, UTC), or some other system—is stored in an SQL database, it must be converted to the semantics chosen for the application. When information from two databases, with different semantics, are combined or compared, the timestamp values must be converted to a common representation. It may very well be the case that one timestamp `'1735-03-23 00:00:00'` from a database and a second timestamp `'1735-03-22 23:55:04'` from another database denote the same exact instant!

Recall that an instant is an anchored location on the time line. So, which instant is denoted by `TIME '11:35:29'`? On this, perhaps the most critical question, the standard is entirely silent. Here is the entire specification [44, pp. 24–25]:

> **Section 4.5.1 Datetimes** Therefore, datetime data types that contain time fields (TIME and TIMESTAMP) are maintained in Universal Coordinated Time (UTC), with an explicit or implicit time zone part.... A TIME or TIMESTAMP that does not specify WITH TIME ZONE has an implicit time zone equal to the local time zone for the SQL-session. However, the *meaning* [italics retained] of the time does not change, because it is effectively in UTC.

An aside: the draft Technical Corrigendum 3 replaces this explanation with an equally laconic specification [19, pp. 10–11]:

> **Section 4.5.1 Datetimes, Draft Corrigendum** The surface of the earth is divided into zones, called time zones, in which every correct clock tells the same time, known as *local time* [italics retained]. Local time is equal to UTC (Coordinated Universal Time) plus the *time zone displacement* [italics retained].... A datetime value, of data type TIME or TIMESTAMP, may represent a local time or UTC.

Let's assume that the user is in the Mountain Standard time zone (seven hours behind Greenwich). She specifies in her SQL code the literal `TIME '11:35:29'`. This particular time occurs exactly once each day in the UTC definition. So perhaps the implied meaning is that the meaning of a TIME literal (or data value) is relative to the current day. But this assumption has two unfortunate ramifications. Some TIME values are defined for only about 1 in every 500 days: leap seconds, `TIME '23:59:60'`, have occurred 22 times thus far. So if I happen to retrieve this TIME value on one of those days, everything is well defined, but if I retrieve the same value any other day, the value indicates a nonexistent instant. The second problem is that the particular instant denoted by the value is dependent on when the value is used or retrieved from the database. Consider a transaction that starts a few minutes before midnight on Monday, January 13, 1997, and runs until a few minutes after midnight on Tuesday. The literal `TIME '11:35:29'` when first retrieved by the transaction denotes the instant `TIMESTAMP '1997-01-13 11:35:29'`. Just a few

---

More on the Start of the Millennium

This confusion between anchored and unanchored values also appears in discussions of "the millennium." That word translated from the Latin means simply "one thousand years," and hence is an unanchored duration: an interval. So the years 998 and 1998 differ by a millennium. However, *Webster's Dictionary* notes the connection with "biennium" ("a period of two years"), defining millennium to be "a period of 1000 years," in particular, "the thousand years mentioned in Revelation 20 during which holiness is to prevail and Christ is to reign on earth." Many different interpretations of the millennium have been given [18]: it is unclear when this thousand years will start. This confusion

between interval and period thus seems to underlie the doomsayers who claim that the world will end or other fantastic happenings will occur on January 1, 2000 (or is it January 1, 2001—see page 63).

Others have started the millennial clock at the supposed start of the world. It has been calculated that Jesus was born on 4000 A.M. (*Annus Mundi*, or "year of the world"). Given that Jesus was born in 4 B.C. (see page 75), you may want to ponder where *you* were at the start of the sixth millennium, or 6000 A.M., which started Monday, October 27, 1997. (Surely you remember that momentous transition!)

---

minutes later, the same value denotes a different instant, TIMESTAMP '1997-01-14 11:35:29', that is, an instant 24 hours later.

Values with an explicit UTC offset are safer to use. Consider again Enoch Seeman's letter; say it was written on TIMESTAMP '1735-03-23 13:23:45', in the early afternoon. Now the letter was written in Berkeley Castle, which is in the same time zone as Greenwich (using time zones as they are specified today; there were no time zones in Seeman's day). However, a user in Los Angeles who retrieved this value from the database and printed it relative to GMT would see it as having been written in the wee hours of the morning. On the other hand, had the offset (in this case, +0:00) been stored with the timestamp, the desired instant would have been correctly specified and would print out correctly in GMT.

An SQL-92 TIME value is really an interval that can be added to midnight of a particular day to specify an instant.

In conclusion, only DATE and TIMESTAMP WITH TIME ZONE adequately specify an instant, an anchored location on the time line. TIME is relative to an unspecified midnight, and TIMESTAMP without an associated offset acquires the time zone of the user when the value is manipulated.

As another subtlety, the standard states:

Subclause 6.8 ⟨datetime value function⟩ General Rule 1. The ⟨datetime value function⟩s CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP respectively return the current date, current time, and current timestamp.

However, that subclause goes on to state:

> **Subclause 6.8 ⟨datetime value function⟩ General Rule 3.** If an SQL-statement generally contains more than one reference to one or more ⟨datetime value function⟩s, then all such references are effectively evaluated simultaneously. The time of evaluation of the ⟨datetime value function⟩ during the execution of the SQL-statement is implementation-dependent.

So an implementation is free to adopt whatever definition of "current" it wishes, including perhaps when the statement was presented to the system, or perhaps when the database was first defined.

3.7.2   Time Zones

As we saw above, TIME WITH TIME ZONE values are safer than TIME values. A careful reading of the SQL-92 standard indicates that implicit time zone displacement "is defective, in at least one respect, is imprecisely specified, does not fully implement the approach proposed [in a prior standards meeting] and leaves unsolved a problem that was acknowledged to need a solution as long ago as 1988" [101, pp. 1–2]. In the current SQL-92 standard, for a TIME value without an explicit time zone, either AT LOCAL or GMT was assumed; the standard is unclear on which is to be used.

Use TIME (without time zones) exceedingly carefully, as the standard is imprecise and defective in its application of implicit time zones.

Technical Corrigendum 3 (at the time this is being written, this document is a working draft approaching ISO approval) specifies that a TIME (without the time zone) value does *not* have an implicit time zone; indeed, nothing is assumed about the nonexistent time zone. While some of the identified deficiencies have been addressed in this way by the Technical Corrigendum, it is doubtful that the changes have migrated into commercial products.

Even when the TIME WITH TIME ZONE type is used, the user should be careful. For example, given two values v_1 and v_2 of type TIME WITH TIME ZONE, it is possible that $v_1 = v_2$ while `EXTRACT(TIMEZONE_HOUR FROM `v_1`) <> EXTRACT(TIMEZONE_HOUR FROM `v_2`)`, and

Use TIME WITH TIME ZONE carefully, as the time zone stored in such a value is often ignored.

`CAST(`v_1` TO TIME) <> CAST(`v_2` TO TIME)`. This unintuitive semantics results from the time zone being ignored in the equality, but not in the latter two expressions. Since many other predicates, such as OVERLAPS, are defined in terms of equality, often the time zone stored in a value is ignored.

3.7.3   Intervals

The reason given for the distinction between year-month intervals and day-time intervals is that months are not an integral number of days. Melton and Simon [71] ask the question, "What is the result of 1 year, 3 months, 19 days divided by

3?" They correctly state that the answer cannot be determined unless we know the dates spanned by that interval. However, minutes are not an integral number of seconds, due to leap seconds. What if we ask the question, "What is the result of 1 minute divided by 4, in seconds?" The answer could be 14, 15, or 16, depending on which particular times were spanned by that interval (up to 2 leap seconds can be added or subtracted from a minute). So, what does the standard have to say about this? The expression `INTERVAL '1' MINUTE / 4` evaluates to `INTERVAL '0' MINUTE`; fractional minutes are lost. However, we can explicitly request that the calculation be done in terms of seconds, either with `INTERVAL '1:00' MINUTE TO SECOND / 4` or `CAST(INTERVAL '1' MINUTE TO INTERVAL MINUTE TO SECOND) / 4`, with the cast being implicit or explicit. Now the question is, What does cast return? Here is the entire specification [44, p. 122].

> **Subclause 6.10 ⟨cast specification⟩ General Rule 13.d.** If *SD* [the data type of the source expression *SV*, here `INTERVAL '2' MINUTE`] is interval and *TD* [the target data type, here INTERVAL MINUTE TO SECOND] and *SD* have different interval precisions, then let *Q* be the least significant ⟨datetime field⟩ of *TD* [that is, SECOND]. Let *Y* be the result of converting *SV* to a scalar in units *Q* according to the natural rules for intervals as defined in the Gregorian calendar. Normalize *Y* to conform to the datetime qualifier "*P* TO *Q*" of *TD*.

The rub lies in deciding exactly what the "natural rules for intervals as defined in the Gregorian calendar" are in the presence of leap seconds. Presumably this expression would always evaluate to 15 seconds (assuming that the "average" minute contains 60 seconds), but the specification is not clear. However, using the same logic, "What is the result of 1 year, 3 months, 19 days divided by 3?" could just as easily be interpreted to yield 3 months, 17 days, using an average length, in days, of a year and a month in the Gregorian calendar.

There are at least two ways the SQL could be interpreted in its handling of intervals in a consistent manner. One is to use "average" months, and minutes, in interval conversions, and do away with the distinction between year-month and day-time intervals. The other is to not use average months or minutes, and expand the kinds of intervals to year-month, day-minute, and second (and fractions thereof) variants.

Another problem surfaces as to what values are allowed for SQL intervals. The specification is laconic on this as well [44, p. 75]:

> **Subclause 5.3 ⟨literal⟩ Syntax Rule 23.** Within the definition of an ⟨interval literal⟩, the ⟨datetime value⟩s are constrained by the natural rules for intervals according to the Gregorian calendar.

Whether or not leap seconds are included in day-time intervals is not specified in the SQL-92 standard.

Most days have 24 hours. The day in April that daylight saving time kicks in has only 23 hours; the day in October that daylight saving time ends contains 25 hours. Similarly, minutes can have 62 seconds (though up to 1999 only one leap second has ever been added to any particular minute), as mentioned in this standard [44, p. 25].

Section 4.5.1 Datetimes Note: On occasion, UTC is adjusted by the omission of a second or the insertion of a "leap second" in order to maintain synchronization with sidereal time. This implies that sometimes, but very rarely, a particular minute will contain exactly 59, 61 or 62 seconds.

However, no such mention is made of days with 25 hours. Hence, the standard is not clear as to whether the maximum value of the hours field is 24 or 25, or whether the maximum value of the seconds field is 60, 61, or 62.

3.7.4   Predicates

The OVERLAPS predicate could have been easily generalized, but wasn't. For example, the following forms are *not* permitted, even though they make perfect sense:

```
BirthDate OVERLAPS DATE '1970-01-01'
BirthDate OVERLAPS (DATE '1970-01-01', INTERVAL '0' DAY)
(DATE '1970-01-01', INTERVAL '0' DAY) OVERLAPS BirthDate
```

We could argue that all three are equivalent to

```
BirthDate = DATE '1970-01-01'
```

but the discussion in Section 3.3 shows that orthogonality was not a priority in SQL's design. If the intervals in the last two are replaced with a nonempty interval, say, INTERVAL '7' DAY (within a week), then they are not equivalent to the first. Again, we could argue that those would be equivalent to

```
(BirthDate, INTERVAL '0' DAY)
OVERLAPS (DATE '1970-01-01', INTERVAL '7' DAY)
(DATE '1970-01-01', INTERVAL '7' DAY)
OVERLAPS (BirthDate, INTERVAL '0' DAY)
```

or

```
(BirthDate, NULL) OVERLAPS (DATE '1970-01-01', INTERVAL '7' DAY)
(DATE '1970-01-01', INTERVAL '7' DAY) OVERLAPS (BirthDate, NULL)
```

but it seems less desirable to require an empty or null interval.

In the same vein, it would have been nice to allow equality and inequality comparisons between these *period information* values, such as

```
(BirthDate, INTERVAL '7' DAY)
= (DATE '1970-01-01', INTERVAL '7' DAY)
(DATE '1970-01-01', INTERVAL '7' DAY)
<= (BirthDate, INTERVAL '9' DAY)
```

As it is, SQL-92 introduces these *period information* values with their concomitant complex syntax rules just for the OVERLAP predicate.

3.7.5   Constructors

The CAST function is not symmetric, in the following way. This is being written at 4:57 P.M. on July 23, 1997. The expression `CAST(TIME '12:34:56' AS TIMESTAMP(0))` yields `'1997-07-23 12:34:56'`, that is, that time today, but `CAST(DATE '1997-01-01' AS TIMESTAMP(0))` yields `TIMESTAMP '1997-01-01 00:00:00'`. This asymmetry appears to be a reasonable design decision, as chances are that something that happened at some other date probably did not happen at exactly the same time of day as "now."

3.8   IMPLEMENTATION CONSIDERATIONS*

Here we consider subtleties of instants and intervals in specific DBMSs.

3.8.1   IBM DB2 Universal Database

IBM DB2 UDB will generate an error if a field value (e.g., 60 seconds, 24 hours) was out of range. Hence, every minute in DB2 contains exactly 60 seconds; leap seconds are not accommodated.

3.8.2   Microsoft Access

Access interprets the fractional portion of a DATE as a fraction of a day, effectively dividing each day into 86,400 seconds. Hence, every minute in Access contains exactly 60 seconds; Access DATEs do not take into account leap seconds. Access will generate a runtime error if the field value (e.g., 60 seconds, 24 hours) was out of range.

3.8.3   Oracle8 Server

Oracle8 Server date arithmetic takes into account the (Catholic) switch from the Julian to the Gregorian calendar, which eliminated 10 days in October 1582 (October 5 through October 14). Missing dates can be entered into the database, but are ignored in date arithmetic and treated as the next date. For example, the next day after October 4, 1582, is October 15, 1582, and the day following October 5, 1582, is October 15, 1582. Specifically, all the dates between October 5 and October 14 are mapped identically to October 15.

As the maximum number of seconds in a minute in Oracle8 Server is 60, Oracle DATEs do not take into account leap seconds. Instead, Oracle8 Server will generate a runtime error if the field value (e.g., 60 seconds, 24 hours) was out of range.

The Adoption of the Gregorian Calendar

As the Gregorian calendar was imposed by fiat by a sitting pope (see page 37), adoption was quick in Roman Catholic countries, but decidedly unenthusiastic in Protestant countries. Britain and its colonies (which includes what is now the United States) didn't adopt the Gregorian calendar until 1752. Because it waited so long, Parliament had to drop 11 days (September 3–13, 1752) in order to catch up. George Washington's birthday was recorded at the time as February 11, 1731; this is a Julian date because Britain hadn't yet switched over. However, President George Wash-ington's birthday is celebrated in the United States on February 22, its Gregorian date (at least until a Presidents' Day was instituted covering for both Washington's birthday and Lincoln's birthday, which occurred after the switch).

The U.S.S.R. didn't join the bandwagon until 1918. The "October Revolution" happened in a Julian October; until recently it has been celebrated in Gregorian November. In fact, there are many different switchover dates (e.g., Sweden, 1753; Turkey, 1927), rendering "*the* Gregorian calendar" an oxymoron.

3.8.4   CD-ROM Materials

Detailed explanations of the temporal types in Microsoft Access 2000, Microsoft SQL Server, IBM DB2 UDB, Informix–Universal Server, Oracle8 Server, Sybase SQL-Server, and UniSQL are provided, as well as sample SQL statements illustrating operations on instants and intervals. For some of the operations that are not possible in IBM DB2 strictly in SQL, the equivalents are given as embedded SQL.

A detailed explanation of Ingres is also included on the CD-ROM, but the examples have not been tested.

3.9   SUMMARY

Temporal values are the stuff of which time-varying applications are made. In order to record the history of the modeled reality, it is first necessary to be able to record the "when."

Instants are the most basic data type. An instant is a position on the time line. Virtually all DBMSs support this data type. In SQL-92, five related data types encode instants, to various granularities: DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, and TIMESTAMP WITH TIME ZONE.

Intervals are unanchored, directed portions of the time line; an interval can be added to an instant to displace that instant either into the future or back into the past. SQL-92 supports two kinds of intervals, year-month intervals and day-time intervals.

A rich set of operators and predicates applies to temporal values. SQL-92 provides the following classes of predicates: equality, inequality, is null, and overlaps. It also provides arithmetic operators ('+', '-', '*', '/'), unary plus and minus, time zone conversion (AT), "now" (CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP), a variety of conversions (CAST), and field extraction (EXTRACT).

Specific DBMSs vary greatly in their support of the standard, from quite strict adherence (e.g., IBM DB2) to studied apathy (most DBMSs). While most temporal operations in SQL-92 can be simulated in the facilities of the various DBMSs (and vice versa), that simulation is often unnecessarily convoluted.

Despite the care with which the SQL-92 standard has been developed and documented, it still contains dark corners and seemingly arbitrary design decisions. Most DATE and TIMESTAMP values are undefined, as the standard is based on UTC, which doesn't apply before 1958. SQL-92, and virtually all DBMSs, utilize the Gregorian calendar, which was adopted in different parts of the world over a 350-year period. A TIME value does not actually represent an instant; rather, it represents a special kind of interval. Leap seconds may or may not be accommodated (the standard doesn't say); most DBMSs ignore this subtlety.

3.10   READINGS

Information on the Public Petroleum Data Model can be found at *www.ppdm.org*.

Datetime literals are based on an ISO standard, "Representation of Dates and Times" [43]. This standard uses the Gregorian calendar as well as a 24-hour clock, which also serve as the basis for SQL datetimes.

While the SQL-92 standard [44] is quite lengthy, at 580 pages, only a small portion, about 30 pages, or 5 percent, concerns temporal data types and their operators. However, this portion in some ways is more complex than other parts of SQL, as evidenced by over 12 pages (almost 10 percent) of the Technical Corrigendum 3 [19]. Even with these numerous corrections, many of the deficiencies discussed in Section 3.7 remain. Sykes provides a cogent discussion of the problems, and partial solutions, to time zone support in SQL-92 [101].

The temporal constructs are included in the Intermediate SQL and Full SQL levels of conformance to SQL-92; the Entry SQL level of conformance includes no temporal types. Conformance testing was initially done by the National Institute of Standards and Technology (NIST), a U.S. Department of Commerce agency. As of July 1, 1997, when NIST ceased SQL conformance testing, 11 products had been validated for conformance to FIPS publication 127-2 [73]: IBM (2 configurations), Informix (5 configurations), NCR, and Sybase (3 configurations). Unfortunately, all of these validations were at Entry FIPS 127-2, which includes no time support. The National Software Testing Laboratories (NSTL), an independent organization not

associated with the government, has established a testing and certification program for SQL at *www.nstl.com/html/press__nstl_establishes_sql_conformance_testing.html*. Unfortunately, according to NSTL, "All of NSTL's testing services are conducted on a strictly confidential basis. Clients. . . have used NSTL test results for promotional purposes," so information about which products conform to the standard will come only from the vendors themselves.

Books and articles on the year 2000 problem could fill an entire shelf of your library. A quick search on *amazon.com* turned up over several dozen titles; there is even a *Year 2000 for Dummies* [16]. (Some book titles to the contrary, the proper spelling is "millennium," with two *n*s, from the Latin *mille*, "one thousand," and *annus*, "year," whereby the two *n*s.) The following are some useful Web sites providing further pointers:

- The Year 2000 Information Center: *www.year2000.com*
- U.S. Federal Government Gateway for Year 2000 Information Directories: *www.itpolicy.gsa.gov/mks/yr2000/y2khome.htm*
- National Institute of Standards and Technology: *www.nist.gov/y2k/*
- Information Technology Association of America (ITAA): *www.itaa.org/year2000.htm*
- Mitre's Year 2000 home page: *www.mitre.org/technology/y2k/*
- The Federal Technology Service of the General Service Administration (GSA) of the U.S. Federal Government: *www.fts.gsa.gov*
- The IEEE Technical Activities Board (TAC) New Technologies Development Committee: *www.mindspring.com/~pci-inc/Year2000/y2ktech.htm*
- IBM's Year 2000 home page: *www.ibm.com/IBM/year2000/*
- Microsoft Year 2000 Resource Center: *microsoft.com/year2000/*
- Newsgroup: *comp.software.year-2000*

The year 2000 problem isn't unique to computers. The July 1998 issue of *Consumer Reports* (p. 67) describes a (manual) Mead 10-year date stamp that had been purchased in March of that year. The reader subsequently found out that the date stamp was good only until December 31, 2000. As the packaging copyright says 1993, this was at best an 8-year date stamp when it was manufactured. Perhaps at the turn of the century there will be a run on office supply stores when all of the 10-year date stamps expire.

Jones [59] lists other year 2000–like problems, many of which in a cruel irony just happen to fall right around the same time: the conversion of the euro, which started January 1, 1999, the Global Positioning System (GPS) week-counter rollover, which occurs at midnight on August 21, 1999, the use of the value 9999 as a file termination code, which might be misinterpreted as September 9, 1999, and the use in Unix of the number 999999999 as end-of-file, which can be interpreted as a

Unix date of September 8, 2001. Neumann [75] provided the Multics observation on page 65.

The SQL standards are denoted SQL-86, SQL-89, and SQL-92. You might think that the next standard, due out in 1999, would be named SQL-99. Logically, then, the standard following that might be named SQL-02, which might be confused with SQL2 (the project name under which SQL-92 was developed). The resolution was to term the next standard SQL:1999, thereby avoiding a year 2000 problem with the *name* of the standard [30].

An impressive amount of information on UTC and leap seconds can be found at *tycho.usno.navy.mil/time.html*. Papers by Dyreson, Howse, Quinn, and the author provide details on UTC and ephemeris time [28, 42, 79]. While UTC is defined relative to an atomic clock, an ephemeris second is a constant duration of time: 1/31,556,925.9747 of the period of time between the passage in 1899 and the passage in 1900 of the sun through the vernal equinox, when the duration of sunlight and darkness are the same. While this may seem an odd definition, the ephemeris second is actually the average value of a second calculated from astronomical observations over the 18th and 19th centuries.

HAL was the computer featured prominently in Arthur C. Clarke's *2001: A Space Odyssey*. Its (his?) birth date, January 12, 1997, was occasioned by the release of a book on HAL's legacy [100].

Oracle's temporal support is described well in Koch and Loney's encyclopedic reference book [63, ch. 7].

See *bert.cs.pitt.edu/~tawfig/convert/introduction.html* for a discussion of the Hijri calendar, supported by Microsoft Access.

Dershowitz and Reingold's beautiful book, *Calendrical Calculations* [27], presents in completely algorithmic form a description of 14 calendars: the present civil calendar (Gregorian), the recent ISO commercial calendar (ISO 8061), the old civil calendar (Julian), the Coptic and Ethiopic calendars, the Islamic (Moslem) calendar, the modern Persian (solar) calendar, the Bahá'í calendar, the Hebrew (Jewish) calendar, the Mayan calendar, the French Revolutionary calendar, the Chinese calendar, and both the old (mean) and new (true) Hindu (Indian) calendars. Included is a wealth of historical material. The mere existence of Dershowitz and Reingold's book is illustrative of the inherent complexities of the subject.
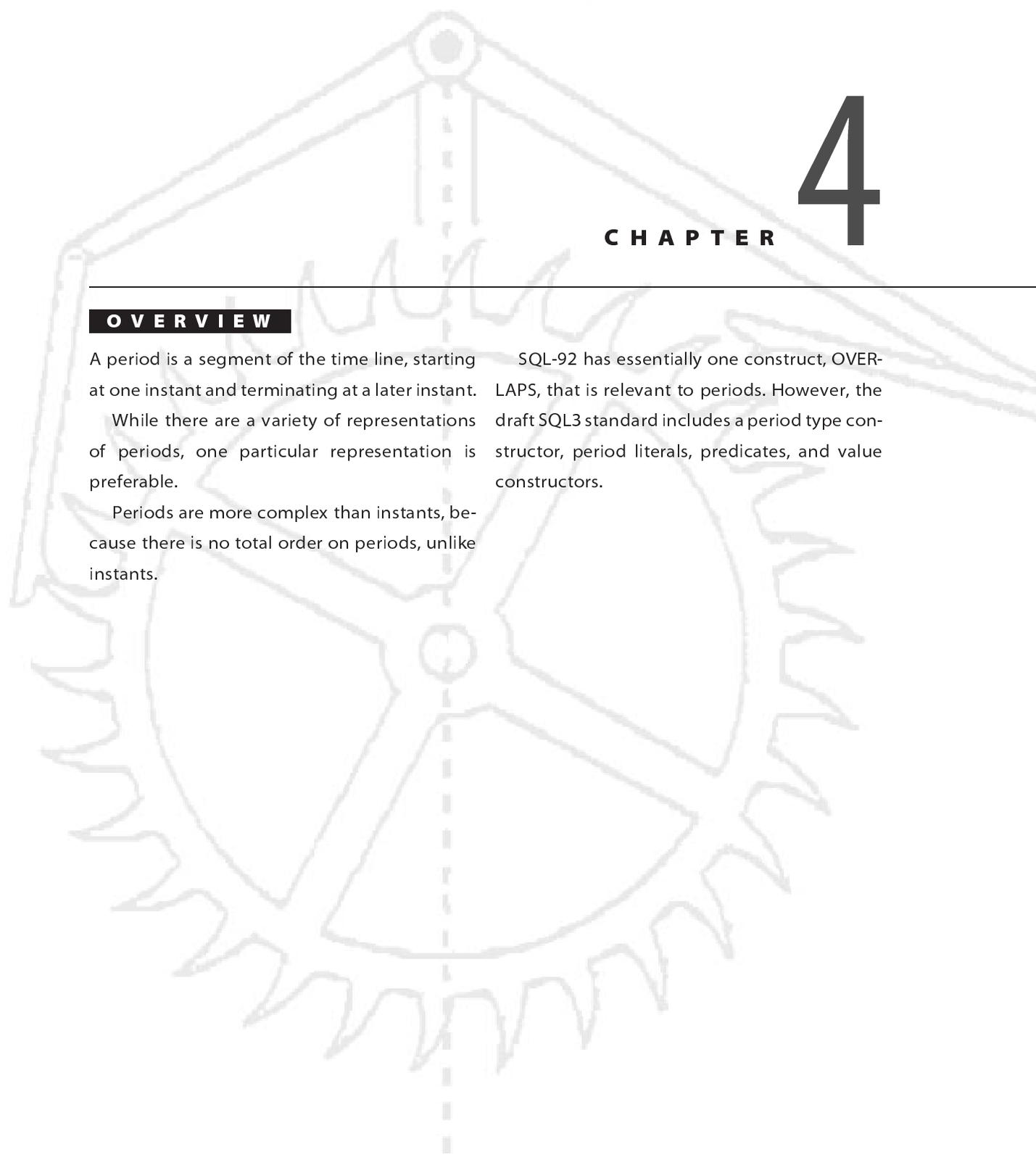
OVERVIEW

A period is a segment of the time line, starting at one instant and terminating at a later instant.

While there are a variety of representations of periods, one particular representation is preferable.

Periods are more complex than instants, because there is no total order on periods, unlike instants.

SQL-92 has essentially one construct, OVER-LAPS, that is relevant to periods. However, the draft SQL3 standard includes a period type constructor, period literals, predicates, and value constructors.

Periods

An instant has no duration. Yet facts in the database are true over a duration of time. To express when a fact holds in the enterprise, a *period* is associated with that fact.

A period is an anchored duration of the time line. The Fall 1997 academic semester at the University of Arizona comprises the period from August 25, 1997 to December 19, 1997. This data type, while quite useful, is not supported directly by any commercial DBMS, nor is it in the SQL-92 standard. However, periods are in the SQL3 draft standard, as will be discussed in Section 12.4.

> A period is an anchored duration of the time line.

Perhaps the reason that periods were not included with the other temporal types in SQL-92 is that they are relatively easy to simulate with datetimes. The most common representation is with a pair of instants, the first specifying the first day (second, microsecond) of the period and the second specifying the last day (second, microsecond) of the period. Generally the delimiting datetimes are of identical granularity.

Jim Barnett utilized periods in several places in the Finder schema. The `Create_Date` and `Last_Update` columns indicate when the data was stored in the database. Many tables also have `Start_Date` and `End_Date` columns to specify when the data was valid in reality. These two, quite different notions are explored in detail in Chapters 8 and 5, respectively.

There are several variants possible even with an instant-pair representation of periods. One common representation is termed a *closed-closed* representation because both delimiting datetimes are in the period. For the Fall 1997 semester, the pair of dates would be `[DATE '1997-08-25', DATE '1997-12-19']`, with the square brackets denoting a closed representation.

An alternative is the *closed-open* representation, in which the second datetime of the pair represents the granule immediately following the last granule of the period. Our example in a closed-open representation is thus `[DATE '1997-08-25',`

DATE '1997-12-20'). The ending parenthesis indicates that the ending is open. We'll examine the relative advantages of each of these representations shortly.

Two less often used alternatives are open-closed and open-open.

Yet another alternative is the starting datetime and an interval specifying the duration of the period. In this approach, the Fall 1997 semester becomes (DATE '1997-08-25', INTERVAL '117' DAY). Finally, for completeness, we might use the terminating datetime and the duration, for example, (INTERVAL '117' DAY, DATE '1997-12-19'). We could also consider open variants thereof, but that is not productive.

> The primary representations of periods are closed-closed and closed-open pairs of datetimes, and a pair of a starting datetime and an interval, with both components of the same granularity.

The delimiting datetime(s) of a period may include a time zone, if their granularity is to the hour or smaller. The Fall 1997 semester to the second granularity in the closed-open representation with a time zone of Mountain Standard Time is [TIMESTAMP '1997-08-25 08:00:00-07:00', TIMESTAMP '1997-12-19 17:00:00-07:00'). This example hints at the utility of the closed-open representation. In the closed-closed representation, the terminating timestamp would be an awkward TIMESTAMP '1997-12-19 16:59:59-07:00'. The two delimiting datetimes of a stored period should have an identical time zone, for the reasons given in Section 3.7.2.

> The time zone of a period, if any, should be stored with the first datetime of the representation.

4.1   LITERALS

As SQL-92 does not provide a period data type, there are no period literals in that language. Periods must instead be specified by their constituent datetime and interval literals.

4.2   PREDICATES

As we saw in the previous chapter, SQL-92 supports only four classes of predicates on datetimes and intervals: equality, less-than, is null, and overlaps.

Equality on periods can be implemented using their underlying components, as shown in Table 4.1. Here, the expression "+ 1" denotes adding one granule at the granularity of the period. At a granularity of day, this expression would be "+ INTERVAL '1' DAY".

> Equality testing on periods is highly dependent on their underlying representation.

Testing for is null is straightforward in any of these representations: simply apply IS NULL to the first component, which is always a datetime. If the first component is null, the value of the second component is irrelevant.

Table 4.1 The equality predicate on periods.

Representation	Equality Predicate
$[a_1, a_2]$ *equals* $[b_1, b_2]$	$a_1 = b_1$ AND $a_2 = b_2$
$[a_1, a_2]$ *equals* $[b_1, b_2)$	$a_1 = b_1$ AND $a_2 + 1 = b_2$
$[a_1, a_2]$ *equals* (b_1, b_i)	$a_1 = b_1 + 1$ AND $a_2 + 1 = b_1 + b_i$
$[a_1, a_2)$ *equals* $[b_1, b_2]$	$a_1 = b_1$ AND $a_2 = b_2 + 1$
$[a_1, a_2)$ *equals* $[b_1, b_2)$	$a_1 = b_1$ AND $a_2 = b_2$
$[a_1, a_2)$ *equals* (b_1, b_i)	$a_1 = b_1 + 1$ AND $a_2 = b_1 + b_i$
(a_1, a_i) *equals* $[a_2, b_1]$	$a_1 + 1 = a_2$ AND $a_1 + a_i = b_1 + 1$
(a_1, a_i) *equals* $[a_2, b_1)$	$a_1 + 1 = a_2$ AND $a_1 + a_i = b_1$
(a_1, a_{i_1}) *equals* (a_2, a_{i_2})	$a_1 = a_2$ AND $a_{i_1} = a_{i_2}$

Unlike datetimes and intervals, periods are *not* ordered. There is only a partial order between periods. Consider the Fall 1997 semester and the calendar year 1997. The calendar year both starts before the semester and ends after the semester. However, the month of July 1997 definitely precedes the Fall 1997 semester.

While datetimes and intervals are totally ordered, periods are only partially ordered, with 13 possible relationships between two periods.

While two datetimes or intervals can be related in three ways (before, equal, and after), two periods can have one of 13 relationships, shown in Figure 4.1. In this figure, time proceeds from left to right. These relationships are disjoint: only one can hold between any two given periods. Note also that *a overlaps b* is more restrictive than SQL's OVERLAPS, which will be discussed shortly.

These relationships can be expressed in terms of comparisons on the components of the underlying periods. Table 4.2 provides the SQL-92 equivalents for the relationships, except for equals, which was provided before, and the inverse relationships (e.g., *before*$^{-1}$), which can be easily derived. This table assumes both argument periods are in the same representation, with the same granularity. Here, a_i is the interval component of the period a, and a_1 and a_2 are its datetime components.

The preferred representation of a period is a closed-open pair of datetimes.

Note the niggling "+ 1" and "<=" that keep popping up with the closed-closed representation, and the interval addition that is present with the interval representation. It is for these reasons that the closed-open representation is generally preferred.

As mentioned in the previous chapter, SQL-92 provides an OVERLAPS predicate, on pairs of two datetimes, corresponding to a closed-open representation, or pairs of a datetime and an interval (see page 35). In our terminology, p OVERLAPS q (the SQL-92 predicate) is equivalent to the following using the basic predicates just introduced: *p overlaps q* $\vee$ *p overlaps*$^{-1}$ q $\vee$ *p starts q* $\vee$ *p starts*$^{-1}$ q $\vee$ *p finishes q* $\vee$ *p finishes*$^{-1}$ q $\vee$ *p during q* $\vee$ *p during*$^{-1}$ q $\vee$ *p equals q*.

a *before* b
b *before* $^{-1}$ a

a *meets* b
b *meets* $^{-1}$ a

a *overlaps* b
b *overlaps* $^{-1}$ a

a *equals* b

a *starts* b
b *starts* $^{-1}$ a

a *finishes* b
b *finishes* $^{-1}$ a

a *during* b
b *during* $^{-1}$ a
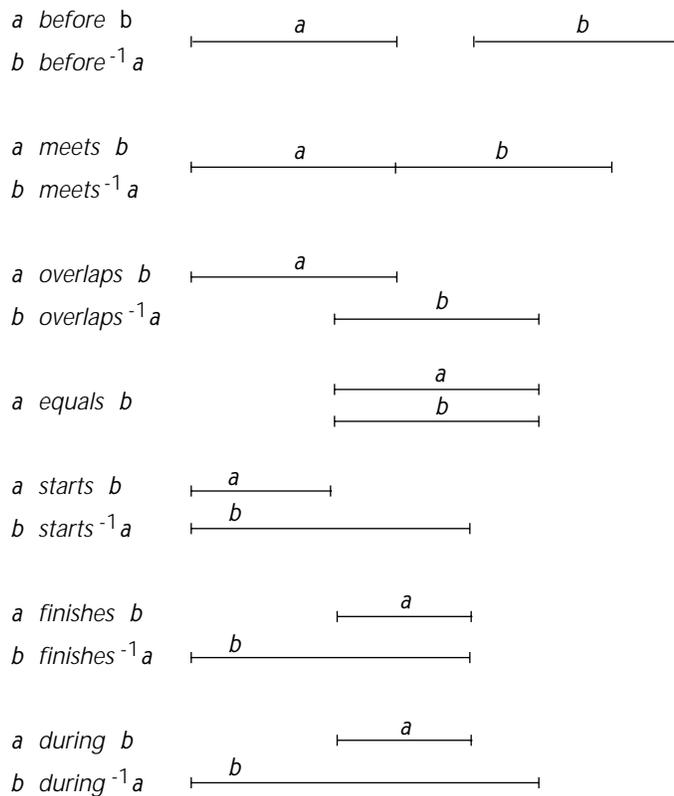
Figure 4.1 Relationships between two periods.

SQL-92 also permits a BETWEEN predicate for datetimes and intervals (see page 35). Such a predicate doesn't strictly apply to periods, as it is defined in terms of less-than. We might wish to define a *between* predicate on periods based on *before*.

A period can also be compared with a datetime of identical granularity. Conceptually, a datetime is simply a very short period, comprising one granule. Five of the predicates are not possible between a datetime and a period: *overlaps*, *overlaps*$^{-1}$, *during*$^{-1}$, *starts*$^{-1}$, and *finishes*$^{-1}$, due to the existence of only one granule in a datetime. The rest are listed in Table 4.3. Note that a datetime is considered to be analogous to a closed-closed period.

> When comparing a datetime with a period, consider the datetime to be a period of a single granule in duration.

In the remainder of this book, we will utilize the preferred representation of a period, that of a closed-open pair of datetimes, of identical granularity.

Table 4.2 The inequality predicates on periods.

Relationship	SQL-92 Predicate
$[a_1, a_2]$ *before* $[b_1, b_2]$	$a_2 + 1 < b_1$
$[a_1, a_2)$ *before* $[b_1, b_2)$	$a_2 < b_1$
(a_1, a_i) *before* (b_1, b_i)	$a_1 + a_i < b_1$
$[a_1, a_2]$ *meets* $[b_1, b_2]$	$a_2 + 1 = b_1$
$[a_1, a_2)$ *meets* $[b_1, b_2)$	$a_2 = b_1$
(a_1, a_i) *meets* (b_1, b_i)	$a_1 + a_i = b_1$
$[a_1, a_2]$ *overlaps* $[b_1, b_2]$	$a_1 < b_1$ AND $a_2 < b_2$ AND $b_1 <= a_2$
$[a_1, a_2)$ *overlaps* $[b_1, b_2)$	$a_1 < b_1$ AND $a_2 < b_2$ AND $b_1 < a_2$
(a_1, a_i) *overlaps* (b_1, b_i)	$a_1 < b_1$ AND $a_1 + a_i < b_1 + b_i$ AND $b_1 <= a_1 + a_i$
$[a_1, a_2]$ *during* $[b_1, b_2]$	$b_1 < a_1$ AND $a_2 < b_2$
$[a_1, a_2)$ *during* $[b_1, b_2)$	$b_1 < a_1$ AND $a_2 < b_2$
(a_1, a_i) *during* (b_1, b_i)	$b_1 < a_1$ AND $a_1 + a_i < b_1 + b_i$
$[a_1, a_2]$ *starts* $[b_1, b_2]$	$a_1 = b_1$ AND $a_2 < b_2$
$[a_1, a_2)$ *starts* $[b_1, b_2)$	$a_1 = b_1$ AND $a_2 < b_2$
(a_1, a_i) *starts* (b_1, b_i)	$a_1 = b_1$ AND $a_i < b_i$
$[a_1, a_2]$ *finishes* $[b_1, b_2]$	$b_1 < a_1$ AND $a_2 = b_2$
$[a_1, a_2)$ *finishes* $[b_1, b_2)$	$b_1 < a_1$ AND $a_2 = b_2$
(a_1, a_i) *finishes* (b_1, b_i)	$b_1 < a_1$ AND $a_1 + a_i = b_1 + b_i$

4.3   CONSTRUCTORS

Periods may participate in, or be returned by, temporal constructors. We organize this discussion along the types that such constructors can return.

4.3.1   Datetime Constructors

The datetime constructors that involve periods are also called *instant extractors*, as they identify the delimiting instants of the argument period. Note especially here that the *semantics* of such constructors (and indeed, of all constructors, and all predicates) is independent of the representation, though their *implementation* is certainly highly dependent on the representation.

Four such delimiting instants are relevant: the *beginning instant*, the *last instant*, the *ending instant* (that immediately following the last instant), and the *previous instant* (that immediately preceding the beginning instant). For our closed-open representation $p = [a_1, a_2)$, these instants can be extracted in the following fashion. Here we use the period of the Fall 1997 semester, p = `[DATE'1997-08-25', DATE '1997-12-20')`.

Table 4.3 Predicates on a period and a datetime.

Relationship	SQL-92 Predicate
a *equals* $[b_1, b_2]$	$a = b_1$ AND $a = b_2$
a *equals* $[b_1, b_2)$	$a = b_1$ AND $a + 1 = b_2$
a *equals* (b_1, b_i)	$a = b_1$ AND $b_i = 1$
a *before* $[b_1, b_2]$	$a + 1 < b_1$
a *before* $[b_1, b_2)$	$a + 1 < b_1$
a *before* (b_1, b_i)	$a < b_1$
a *before*$^{-1}$ $[b_1, b_2]$	$b_2 + 1 < a$
a *before*$^{-1}$ $[b_1, b_2)$	$b_2 < a$
a *before*$^{-1}$ (b_1, b_i)	$b_1 + b_i < a$
a *meets* $[b_1, b_2]$	$a + 1 = b_1$
a *meets* $[b_1, b_2)$	$a + 1 = b_1$
a *meets* (b_1, b_i)	$a = b_1$
a *meets*$^{-1}$ $[b_1, b_2]$	$b_2 + 1 = a$
a *meets*$^{-1}$ $[b_1, b_2)$	$b_2 = a$
a *meets*$^{-1}$ (b_1, b_i)	$b_1 + b_i = a$
a *during* $[b_1, b_2]$	$b_1 < a$ AND $a < b_2$
a *during* $[b_1, b_2)$	$b_1 < a$ AND $a + 1 < b_2$
a *during* (b_1, b_i)	$b_1 < a$ AND $a < b_1 + b_i$
a *starts* $[b_1, b_2]$	$a = b_1$ AND $a < b_2$
a *starts* $[b_1, b_2)$	$a = b_1$ AND $a + 1 < b_2$
a *starts* (b_1, b_i)	$a = b_1$ AND $b_i > 1$
a *finishes* $[b_1, b_2]$	$a = b_2$ AND $b_1 < a$
a *finishes* $[b_1, b_2)$	$a + 1 = b_2$ AND $b_1 < a$
a *finishes* (b_1, b_i)	$a + 1 = b_1 + b_i$ AND $b_i > 1$

- *beginning:* a_1 (the first day in the period)
  beginning(p) = DATE '1997-08-25'
- *previous:* a_1 - 1 (the day immediately preceding the period)
  previous(p) = DATE '1997-08-24'
- *last:* a_2 - 1 (the last day in the period)
  last(p) = DATE '1997-12-19'
- *ending:* a_2 (the day immediately following the period)
  ending(p) = DATE '1997-12-20'

4.3.2   Interval Constructors

There are two useful interval constructors that take a period as an argument. The first, *duration*, can be computed from $[a_1, a_2)$ as $(a_2 - a_1)$ `qual`. As an example, the period of the Fall 1997 semester has a duration of

```
(DATE '1997-12-20' - DATE '1997-08-25') DAY
```

The other useful interval constructor is time zone extraction, which returns the time zone of the argument interval, if the delimiting datetimes have an associated time zone. In SQL, this can be done by extracting the time zone hour and minute from the initial datetime, then converting to an interval:

$$\texttt{CAST(EXTRACT(TIMEZONE\_HOUR}$$
$$\texttt{FROM } a_1 \texttt{) AS HOUR)} +$$
$$\texttt{CAST(EXTRACT(TIMEZONE\_MINUTE}$$
$$\texttt{FROM } a_1 \texttt{) AS MINUTE)}$$

The result is an interval of granularity `HOUR TO MINUTE`.

4.3.3 Period Constructors

There are several constructors that return a period value. Figure 4.2 illustrates some of these.

- `[a, a + 1)` converts the datetime `a` into a period of one granule.
- `[a₁, a₂) + INTERVAL i` yields `[a₁ + i, a₂ + i)`. This shifts the period later by the interval, or earlier, if the interval is negative.
- `INTERVAL i + [a₁, a₂)` yields `[a₁ + i, a₂ + i)`, as addition is symmetric.
- `[a₁, a₂) - INTERVAL i` yields `[a₁ - i, a₂ - i)`. This shifts the period earlier by the specified interval.
- `a₁` *extend* `a₂` yields `[a₁, a₂ + 1)`, that is, the period from `a₁` to `a₂`.
- `[a₁, a₂)` *extend* `[b₁, b₂)` yields `[min(a₁, b₁), max(a₂, b₂))`, that is, the period that starts the earlier of periods `a` and `b` and finishes the later of periods `a` and `b`.
- `a` *extend* `[b₁, b₂)` yields `[min(a, b₁), max(a, b₂))`, that is, the period that starts the earlier of instant `a` and period `b` and finishes the later of instant `a` and period `b`.
- `[a₁, a₂)` *extend* `b` yields `[min(a₁, b), max(a₂, b))`, because *extend* is symmetric.
- By using AT LOCAL and AT TIME ZONE on the initial datetime in the period representation, the time zone offset of a period can be changed.
- By CASTing each of the constituent components, the granularity of a period can be changed.

a
●

b
●

b
●

a

a
●

b

a

b

a extend b

a

b

a - b

$a \cap b$

$a \cup b$
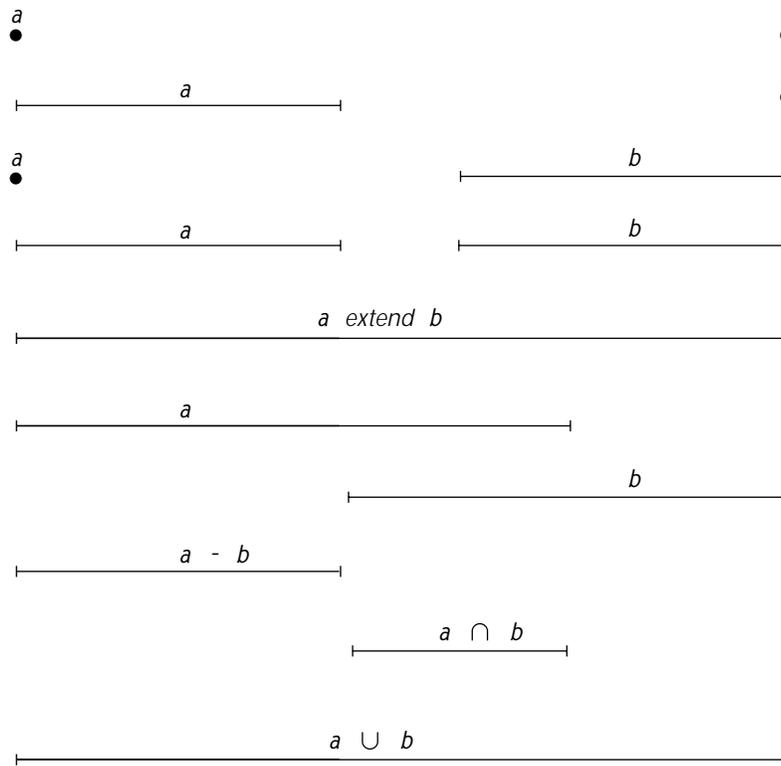
Figure 4.2 Period operations.

The following constructors impose restrictions on the argument periods:

- $[a_1, a_2) \cap [b_1, b_2)$ yields $[max(a_1, b_1), min(a_2, b_2))$, that is, the period that starts the later of the beginning of periods a and b and finishes the earlier of the end of periods a and b. Note that this constructor is defined only if a OVERLAPS b; otherwise, an invalid period is returned, one in which the starting delimiting instant occurs after the ending delimiting instant.

- $[a_1, a_2) - [b_1, b_2)$ yields

    — $[b_2, a_2)$ if a *starts*$^{-1}$ b or a *overlaps*$^{-1}$ b,

    — $[a_1, b_1)$ if a *finishes*$^{-1}$ b or a *overlaps* b, or

    — a, if a *before* b or a *meets* b or a *meets*$^{-1}$ b or a *before*$^{-1}$ b.

Period difference results in the portion of a not overlapping with b. Note that this constructor is undefined if a *during* b because then there are two subperiods

of a that are not in b. This constructor is also undefined if a *equals* b or a *during* b, or a *starts* b, or a *finishes* b because then the result is the empty period.

- $[a_1, a_2) \cup [b_1, b_2)$ yields $[min(a_1, b_1), max(a_2, b_2))$, that is, the period that starts the earlier of the starting delimiters of periods a and b and finishes the later of the ending delimiters of periods a and b. This constructor is undefined if a *before* b or a *before*$^{-1}$ b; otherwise, an invalid period is returned, one that contains instants that are not in either a or b. Note that under this restriction, $[a_1, a_2) \cup [b_1, b_2)$ $= [a_1, a_2)$ *extend* $[b_1, b_2)$.

4.3.4 Character Constructors

A period can be cast into a character string via casts on its constituent parts and string concatenation ("||").

```
'[' || CAST( a₁ AS CHARACTER) || ' - ' || CAST( a₂ AS CHARACTER) || ')'
```

4.4 IMPLEMENTATION CONSIDERATIONS

A period data type is not included in the SQL-92 standard, and no vendor offers support for such a data type. In the following, we show how periods can be simulated using a pair of instants.

4.4.1 IBM DB2 Universal Database

We use the closed-open representation of a pair of IBM DB2 UDB DATEs (or TIMEs, or TIMESTAMPs). Table 4.4 shows how the period operations can be implemented in DB2. In the first column, a and b denote datetime values. In the DB2 column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote DB2 UDB DATE values (we use a day granularity here), and i denotes a DB2 UDB DECIMAL(8,0) representing a date duration. A constraint can be used to ensure that the starting date of a period is *before* the ending date.

4.4.2 Informix–Universal Server

We use the closed-open representation of a pair of DATEs (or DATETIMEs). Table 4.5 shows how the period operations can be implemented in Informix–Universal Server. In the first column, a and b denote instant values. In the Informix–Universal Server column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote Informix DATE values (we use a day granularity here), and i denotes an Informix INTERVAL DAY representing a date duration. Some of the temporal predicates can be conveniently defined as SPL PROCEDUREs. A constraint can be used to ensure that the starting date of a period is *before* the ending date.

Table 4.4 Period operations in IBM DB2 UDB.

Period Operations	IBM DB2 UDB Equivalent
Types:	
period	[DATE, DATE)
Predicates:	
p *equals* q	$p_1 = q_1$ AND $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ AND $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ AND $p_1 < q_2$
p *during* q	$q_1 < p_1$ AND $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ AND $q_2 < p_2$
p *starts* q	$p_1 = q_1$ AND $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ AND $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ AND $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ AND $p_2 = q_2$
p OVERLAPS q	$p_1 < q_2$ AND $q_1 < p_2$
p IS NULL	p_1 IS NULL
Datetime Constructors:	
beginning(p)	p_1
previous(p)	p_1 - 1 DAY
last(p)	p_2 - 1 DAY
ending(p)	p_2
Interval Constructors:	
duration(p)	p_2 - p_1
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	[$p_1 + i$, $p_2 + i$)
$i + p$	[$p_1 + i$, $p_2 + i$)
$p - i$	[$p_1 - i$, $p_2 - i$)
p *extend* q	[CASE WHEN $p_1 < q_1$ THEN p_1 ELSE q_1 END, CASE WHEN $p_2 < q_2$ THEN q_2 ELSE p_2 END)
p *extend* a	[CASE WHEN $p_1 < a$ THEN p_1 ELSE a END, CASE WHEN $p_2 <= a$ THEN a + 1 DAY ELSE p_2 END)
a *extend* p	[CASE WHEN $p_1 < a$ THEN p_1 ELSE a END, CASE WHEN $p_2 <= a$ THEN a + 1 DAY ELSE p_2 END)

Table 4.4 (continued)

Period Operations	IBM DB2 UDB Equivalent
Period Constructors, continued:	

a extend *b*	[CASE WHEN $a < b$ THEN a ELSE b END,
	    CASE WHEN $a < b$
	       THEN b + 1 DAY
	       ELSE a + 1 DAY END)

$p \cap q$	[CASE WHEN p_2 <= q_1 OR q_2 <= p_1
	          THEN NULL
	          WHEN $p_1 < q_1$ THEN q_1
	          ELSE p_1 END,
	       CASE WHEN p_2 <= q_1 OR q_2 <= p_1
	          THEN NULL
	          WHEN $p_2 < q_2$ THEN p_2
	          ELSE q_2 END)

$p - q$	[CASE WHEN q_1 <= p_1 AND p_2 <= q_2
	       THEN NULL
	          WHEN $p_1 < q_1$ AND $q_2 < p_2$
	             THEN NULL
	          WHEN $p_2 > q_2$ AND $p_1 < q_2$
	             THEN q_2
	          ELSE p_1 END,
	       CASE WHEN q_1 <= p_1 AND p_2 <= q_2
	             THEN NULL
	          WHEN $p_1 < q_1$ AND $q_2 < p_2$
	             THEN NULL
	          WHEN $q_1 < p_2$ AND $p_2 > q_1$
	             THEN q_1
	          ELSE p_2 END)

$p \cup q$	[CASE WHEN p_2 <= q_1 OR q_2 <= p_1
	          THEN NULL
	          WHEN $p_1 < q_1$ THEN p_1
	          ELSE q_1 END,
	       CASE WHEN p_2 <= q_1 OR q_2 <= p_1
	          THEN NULL
	          WHEN $p_2 < q_2$ THEN q_2
	          ELSE p_2 END)

p AT TIME ZONE i	not supported
p AT LOCAL	[p_1 + CURRENT TIMEZONE, p_2 + CURRENT TIMEZONE)
Other Operators:	
CAST(a AS PERIOD)	[a, a + 1 DAY)
CAST(p AS CHAR)	'[' CONCAT CHAR(p_1) CONCAT '-' CONCAT CHAR(p_2) CONCAT ')'

Table 4.5 Period operations in Informix–Universal Server.

Period Operations	Informix– Universal Server Equivalent
Types:	
period	`[DATETIME, DATETIME)`
Predicates:	
p *equals* q	$p_1 = q_1$ `AND` $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ `AND` $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ `AND` $p_1 < q_2$
p *during* q	$q_1 < p_1$ `AND` $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ `AND` $q_2 < p_2$
p *starts* q	$p_1 = q_1$ `AND` $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ `AND` $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ `AND` $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ `AND` $p_2 = q_2$
p `OVERLAPS` q	$p_1 < q_2$ `AND` $q_1 < p_2$
p `IS NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(p)	p_1
previous(p)	p_1 `- INTERVAL(1) DAY`
last(p)	p_2 `- INTERVAL(1) DAY`
ending(p)	p_2
Interval Constructors:	
duration(p)	$p_2 - p_1$
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	`[`$p_1 + i$`, `$p_2 + i$`)`
$i + p$	`[`$p_1 + i$`, `$p_2 + i$`)`
$p - i$	`[`$p_1 - i$`, `$p_2 - i$`)`
p *extend* a	not possible
$p \cap q$	not possible
$p - q$	not possible
$p \cup q$	not possible
p `AT TIME ZONE` i	not supported
p `AT LOCAL`	not supported
Other Operators:	
`CAST(`a `AS PERIOD)`	`[`a`, `a `+ INTERVAL(1) DAY)`
`CAST(`p `AS CHAR)`	not possible

4.4.3 Microsoft Access

We use the closed-open representation of a pair of Microsoft Access DATEs, which are at the second granularity. Table 4.6 shows how the period operations can be implemented in Microsoft Access 2000. In the first column, a and b denote date-time values. In the Access column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote Access DATE values, and j denotes an Access NUMBER representing Julian days (and fractional days).

4.4.4 Microsoft SQL Server

We use the closed-open representation of a pair of Microsoft SQL Server DATE-TIMEs, which are at the subsecond granularity (1/300 second). Table 4.7 shows how the period operations can be implemented in SQL Server. In the first column, a and b denote datetime values. In the Microsoft SQL Server column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote SQL Server DATETIME values, and i denotes an SQL Server INTEGER representing an integral number of seconds.

4.4.5 Sybase SQLServer

We use the closed-open representation of a pair of Sybase SQLServer DATETIMEs, which are at the subsecond granularity. Table 4.8 shows how the period operations can be implemented in Sybase. In the first column, a and b denote datetime values. In the Sybase column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote SQLServer DATETIME values, and i denotes an SQLServer INTEGER representing an integral number of seconds.

4.4.6 Oracle8 Server

We use the closed-open representation of a pair of Oracle DATEs, which are at the second granularity. Table 4.9 shows how the period operations can be implemented in Oracle8 Server. In the first column, a and b denote datetime values. In the Oracle8 Server column, p_1 and p_2 denote the two components of the representation of the period p, a and b denote Oracle DATE values, and j denotes an Oracle NUMBER representing Julian days.

4.4.7 UniSQL

We use the closed-open representation of a pair of UniSQL TIMESTAMPs, which are at the second granularity. Table 4.10 shows how the period operations can be implemented in UniSQL. In the first column, a and b denote datetime values. In the UniSQL column, p_1 and p_2 denote the two components of the representation

Table 4.6 Period operations in Microsoft Access 2000.

Period Operations	Microsoft Access 2000 Equivalent
Types:	
period	`[DATE, DATE)`
Predicates:	
p *equals* q	$p_1 = q_1$ `AND` $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ `AND` $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ `AND` $p_1 < q_2$
p *during* q	$q_1 < p_1$ `AND` $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ `AND` $q_2 < p_2$
p *starts* q	$p_1 = q_1$ `AND` $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ `AND` $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ `AND` $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ `AND` $p_2 = q_2$
p `OVERLAPS` q	$p_1 < q_2$ `AND` $q_1 < p_2$
p `IS NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(p)	p_1
previous(p)	`DateAdd("D", -1,` p_1`)`
last(p)	`DateAdd("D", -1,` p_2`)`
ending(p)	p_2
Interval Constructors:	
duration(p)	p_2-p_1 or `DateDiff("D",` p_1`,` p_2`)` (integral number of days)
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	`[DateAdd("D",` j`,` p_1`), DateAdd("D",` j`,` p_2`))`
$i + p$	`[DateAdd("D",` j`,` p_1`), DateAdd("D",` j`,` p_2`))`
$p - i$	`[DateAdd("D", -`j`,` p_1`), DateAdd("D", -`j`,` p_2`))`
p *extend* q	not possible
$p \cap q$	not possible
$p - q$	not possible
$p \cup q$	not possible
p `AT TIME ZONE` i	not supported
p `AT LOCAL`	not supported
Other Operators:	
`CAST(`a `AS PERIOD)`	`[`a`, DateAdd("D", 1,` a`))`
`CAST(`p `AS CHAR)`	`CONCAT("[" & Cstr(`p_1`) & "-" & Cstr(`p_2`) & ")")`

Table 4.7 Period operations in Microsoft SQL Server.

Period Operations	Microsoft SQL Server Equivalent
Types:	
period	`[DATETIME, DATETIME)`
Predicates:	
p *equals* q	$p_1 = q_1$ `AND` $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ `AND` $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ `AND` $p_1 < q_2$
p *during* q	$q_1 < p_1$ `AND` $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ `AND` $q_2 < p_2$
p *starts* q	$p_1 = q_1$ `AND` $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ `AND` $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ `AND` $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ `AND` $p_2 = q_2$
p `OVERLAPS` q	$p_1 < q_2$ `AND` $q_1 < p_2$
p `IS NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(p)	p_1
previous(p)	`dateadd(second, -1, `p_1`)`
last(p)	`dateadd(second, -1, `p_2`)`
ending(p)	p_2
Interval Constructors:	
duration(p)	$p_2 - p_1$
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	`[dateadd(second, `i`, `p_1`),` `   dateadd(second, `i`, `p_2`))`
$i + p$	`[dateadd(second, `i`, `p_1`),` `   dateadd(second, `i`, `p_2`))`
$p - i$	`[dateadd(second, -`i`, `p_1`),` `   dateadd(second, -`i`, `p_2`))`

p *extend* q	`[ CASE WHEN `$p_1 < q_1$
	`      THEN `p_1` ELSE `q_1` END,`
	`   CASE WHEN `$p_2 < q_2$
	`      THEN `q_2` ELSE `p_2` END )`

p *extend* a	`[ CASE WHEN `$p_1 < a$` THEN `p_1` ELSE `a` END,`
	`   CASE WHEN `$p_2 <= a$
	`      THEN dateadd(day, 1, `a`) ELSE `p_2` END`
	`   )`

a *extend* p	`[ CASE WHEN `$p_1 < a$` THEN `p_1` ELSE `a` END,`
	`   CASE WHEN `$p_2 <= a$
	`      THEN dateadd(day, 1, `a`) ELSE `p_2` END`
	`   )`

Table 4.7 (continued)

Period Operations	Microsoft SQL Server Equivalent
Period Constructors, continued:	

a extend b	[CASE WHEN $a < b$ THEN a ELSE b END,
	  CASE WHEN $a < b$
	    THEN dateadd(day, 1, b)
	    ELSE dateadd(day, 1, a) END)

$p \cap q$	[CASE WHEN $p_2 <= q_1$ OR $q_2 <= p_1$
	    THEN NULL
	    ELSE CASE WHEN $p_1 < q_1$
	      THEN q_1 ELSE p_1 END END,
	  CASE WHEN $p_2 <= q_1$ OR $q_2 <= p_1$
	    THEN NULL
	    ELSE CASE WHEN $p_2 < q_2$
	      THEN p_2 ELSE q_2 END END)

$p - q$	[CASE WHEN $q_1 <= p_1$ AND $p_2 <= q_2$
	    THEN NULL
	    ELSE CASE WHEN $p_1 < q_1$ AND $q_2 < p_2$
	      THEN NULL
	      ELSE CASE WHEN $p_2 > q_2$ AND $p_1 < q_2$
	      THEN q_2
	      ELSE p_1 END
	      END
	    END,
	  CASE WHEN $q_1 <= p_1$ AND $p_2 <= q_2$
	      THEN NULL
	    ELSE CASE WHEN $p_1 < q_1$ AND $q_2 < p_2$
	      THEN NULL
	      ELSE CASE WHEN $q_1 < p_2$ AND $p_2 > q_1$
	      THEN q_1
	      ELSE p_2 END
	      END
	    END)

$p \cup q$	[CASE WHEN $p_2 <= q_1$ OR $q_2 <= p_1$
	    THEN NULL
	    ELSE CASE WHEN $p_1 < q_1$
	      THEN p_1 ELSE q_1 END END,
	  CASE WHEN $p_2 <= q_1$ OR $q_2 <= p_1$
	    THEN NULL
	    ELSE CASE WHEN $p_2 < q_2$
	      THEN q_2 ELSE p_2 END END)

p AT TIME ZONE i	not supported
p AT LOCAL	not supported
Other Operators:	
CAST(a AS PERIOD)	[a, dateadd(second,1,a))
CAST(p AS CHAR)	"[" + convert(char(11), p_1) + "-" + convert(char(11), p_2) + ")"

Table 4.8 Period operations in Sybase SQLServer.

Period Operations	Sybase SQLServer Equivalent
Types:	
period	`[DATETIME, DATETIME)`
Predicates:	
p *equals* q	$p_1 = q_1$ `AND` $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ `AND` $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ `AND` $p_1 < q_2$
p *during* q	$q_1 < p_1$ `AND` $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ `AND` $q_2 < p_2$
p *starts* q	$p_1 = q_1$ `AND` $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ `AND` $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ `AND` $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ `AND` $p_2 = q_2$
p q	$p_1 < q_2$ `AND` $q_1 < p_2$
p `IS NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(p)	p_1
previous(p)	`dateadd(second, -1,` p_1`)`
last(p)	`dateadd(second, -1,` p_2`)`
ending(p)	p_2
Interval Constructors:	
duration(p)	p_2 - p_1
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	`[dateadd(second,` i`,` p_1`),` `dateadd(second,` i`,` p_2`))`
$i + p$	`[dateadd(second,` i`,` p_1`),` `dateadd(second,` i`,` p_2`))`
$p - i$	`[dateadd(second,` -i`,` p_1`),` `dateadd(second,` -i`,` p_2`))`
p *extend* q	not possible
$p \cap q$	not possible
p - q	not possible
$p \cup q$	not possible
p `AT TIME ZONE` i	not supported
p `AT LOCAL`	not supported
Other Operators:	
`CAST(`a `AS PERIOD)`	`[`a`, dateadd(second, 1,` a`))`
`CAST(`p `AS CHAR)`	`"[" + convert(char(11),` p_1`) + "-"` `+ convert(char(11),` p_2`) + ")"`

Table 4.9 Period operations in Oracle8 Server.

Period Operations	Oracle8 Server Equivalent
Types:	
period	`[DATE, DATE)`
Predicates:	
p equals *q*	$p_1 = q_1$ AND $p_2 = q_2$
p before *q*	$p_2 < q_1$
p before^{-1} *q*	$q_2 < p_1$
p meets *q*	$p_2 = q_1$
p meets^{-1} *q*	$q_2 = p_1$
p overlaps *q*	$p_1 < q_1$ AND $q_1 < p_2$
p overlaps^{-1} *q*	$q_1 < p_1$ AND $p_1 < q_2$
p during *q*	$q_1 < p_1$ AND $p_2 < q_2$
p during^{-1} *q*	$p_1 < q_1$ AND $q_2 < p_2$
p starts *q*	$p_1 = q_1$ AND $p_2 < q_2$
p starts^{-1} q_2	$p_1 = q_1$ AND $q_2 < p_2$
p finishes *q*	$q_1 < p_1$ AND $p_2 = q_2$
p finishes^{-1} *q*	$p_1 < q_1$ AND $p_2 = q_2$
p q	$p_1 < q_2$ AND $q_1 < p_2$
p `NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(*p*)	p_1
previous(*p*)	p_1 - 1
last(*p*)	p_2 - 1
ending(*p*)	p_2
Interval Constructors:	
duration(*p*)	p_2 - p_1
extract_time_zone(*p*)	not supported
Period Constructors:	
p + *i*	$[p_1 + j, p_2 + j)$
i + *p*	$[p_1 + j, p_2 + j)$
p - *i*	$[p_1 - j, p_2 - j)$
a extend *b*	`[LEAST(`*a*`, `*b*`), GREATEST(`*a*` + 1, `*b*` + 1))`
p extend *q*	`[LEAST(`p_1`, `q_1`), GREATEST(`p_2`, `q_2`))`
p extend *a*	`[LEAST(`p_1`, `*a*`), GREATEST(`p_2`, `*a*` + 1))`
a extend *p*	`[LEAST(`*a*`, `p_1`), GREATEST(`*a*` + 1, `p_2`))`
p ∩ *q*	`[GREATEST(`p_1`, `q_1`), LEAST(`p_2`, `q_2`))`
p - *q*	not possible
p ∪ *q*	`[LEAST(`p_1`, `q_1`), GREATEST(`p_2`, `q_2`))`
p `AT TIME ZONE` *i*	not supported
p `AT LOCAL`	not supported
Other Operators:	
`CAST(`*a* `AS PERIOD)`	$[a, a + 1/86400)$

Table 4.10 Period operations in UniSQL.

Period Operations	UniSQL Equivalent
Types:	
period	`[TIMESTAMP, TIMESTAMP)`
Predicates:	
p *equals* q	$p_1 = q_1$ `AND` $p_2 = q_2$
p *before* q	$p_2 < q_1$
p *before*$^{-1}$ q	$q_2 < p_1$
p *meets* q	$p_2 = q_1$
p *meets*$^{-1}$ q	$q_2 = p_1$
p *overlaps* q	$p_1 < q_1$ `AND` $q_1 < p_2$
p *overlaps*$^{-1}$ q	$q_1 < p_1$ `AND` $p_1 < q_2$
p *during* q	$q_1 < p_1$ `AND` $p_2 < q_2$
p *during*$^{-1}$ q	$p_1 < q_1$ `AND` $q_2 < p_2$
p *starts* q	$p_1 = q_1$ `AND` $p_2 < q_2$
p *starts*$^{-1}$ q_2	$p_1 = q_1$ `AND` $q_2 < p_2$
p *finishes* q	$q_1 < p_1$ `AND` $p_2 = q_2$
p *finishes*$^{-1}$ q	$p_1 < q_1$ `AND` $p_2 = q_2$
p q	$p_1 < q_2$ `AND` $q_1 < p_2$
p `IS NULL`	p_1 `IS NULL`
Datetime Constructors:	
beginning(p)	p_1
previous(p)	p_1 `-` 1
last(p)	p_2 `-` 1
ending(p)	p_2
Interval Constructors:	
duration(p)	p_2 `-` p_1
extract_time_zone(p)	not supported
Period Constructors:	
$p + i$	`[`$p_1 + i$`, `$p_2 + i$`)`
$i + p$	`[`$p_1 + i$`, `$p_2 + i$`)`
$p - i$	`[`$p_1 - i$`, `$p_2 - i$`)`
p *extend* q	not possible
$p \cap q$	not possible
p `-` q	not possible
$p \cup q$	not possible
p `AT TIME ZONE` i	not supported
p `AT LOCAL`	not supported
Other Operators:	
`CAST(`a `AS PERIOD)`	`[`a`, `$a + 1$`)`
`CAST(`p `AS CHAR)`	not possible

of the period *p*, *a* and *b* denote UniSQL TIMESTAMP values, and *i* denotes a UniSQL INTEGER representing seconds.

4.4.8   CD-ROM Materials

The CD-ROM contains example queries in IBM DB2 UDB, Informix–Universal Server, Microsoft Access 2000, Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL illustrating how periods may be simulated using a pair of dates. For some of the operations that are not possible in IBM DB2 UDB strictly in SQL, the equivalents are given as embedded SQL.

A detailed explanation of Ingres is also included, but the examples have not been tested.

4.5   SUMMARY

A period is an anchored duration of the time line. It extends from a beginning instant to a last instant.

Although a variety of representations are possible, including closed-closed, closed-open, open-closed, open-open, a pair of starting instant and duration, and a pair of last instant and duration, the most convenient representation is a closed-open pair of instants.

Predicates are complex for periods, as they are not totally ordered. There are 13 possible relationships between two periods.

The delimiting timestamps can be extracted from a period. Two periods can be unioned and intersected, and a period can be subtracted from another period. A period can be shifted by adding or subtracting an interval.

SQL/Temporal, part of SQL3, provides built-in support for periods, including the period type constructor, period literals, predicates, and value constructors. Additional constructs break a period into its constituent granules and normalize a set of periods into a set of disjoint, nonadjacent periods.

4.6   READINGS

James Allen showed that there are exactly 13 disjoint binary relationships possible between two periods [2]; these are now termed the *Allen relations*.
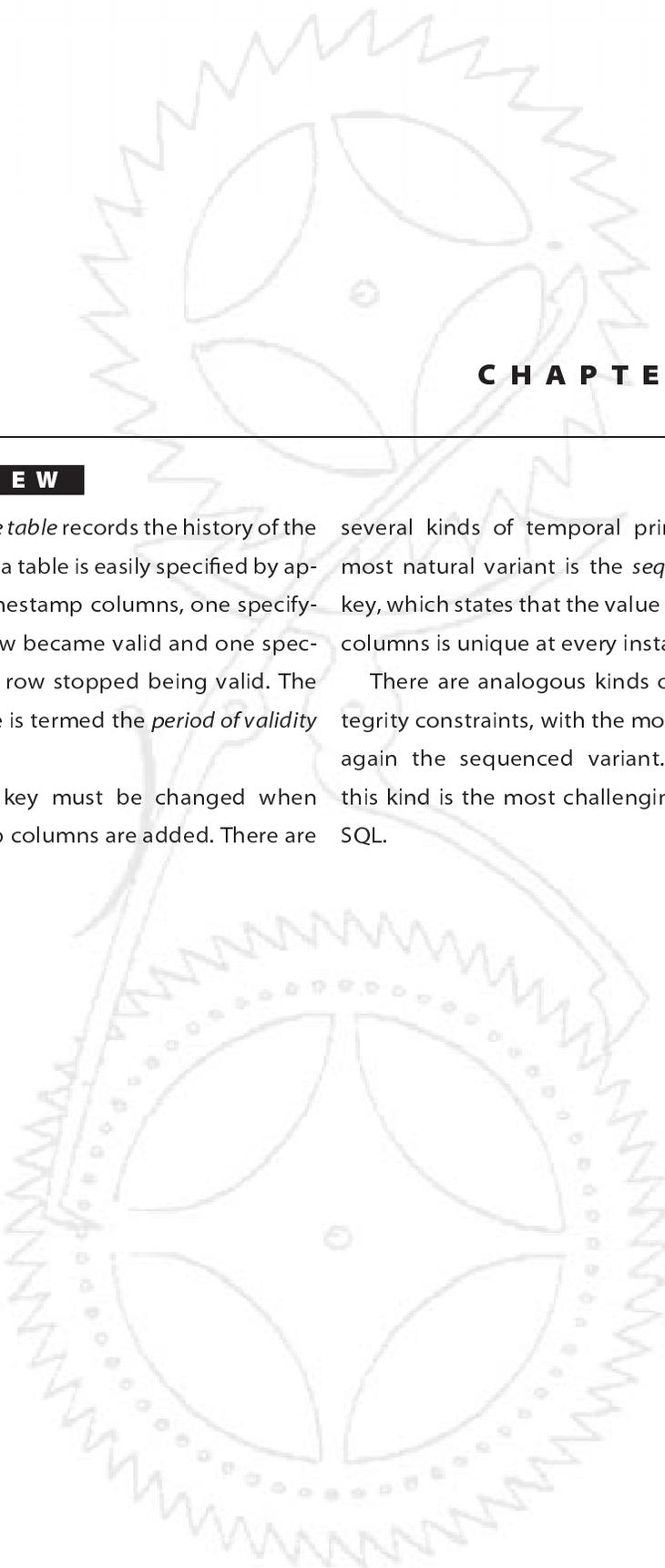
OVERVIEW

A *valid-time state table* records the history of the enterprise. Such a table is easily specified by appending two timestamp columns, one specifying when the row became valid and one specifying when the row stopped being valid. The intervening time is termed the *period of validity* of the row.

The primary key must be changed when these timestamp columns are added. There are several kinds of temporal primary keys. The most natural variant is the *sequenced* primary key, which states that the value of the indicated columns is unique at every instant of time.

There are analogous kinds of referential integrity constraints, with the most natural being again the sequenced variant. Unfortunately, this kind is the most challenging to express in SQL.

Defining State Tables

I n 1992 the University of Arizona was confronted with a knotty dilemma. Its data was managed by a suite of COBOL legacy systems, each with its own underlying DBMS or file structure. Personnel records were (and continue to be) managed by PSOS (Personnel Operating System), using the Transact DBMS. Financial records, including purchasing, reorders, property management, fixed assets, and the ledger, are managed by FRS (Financial Records System), which uses IDMS. Student records are contained in yet another system, SIS (Student Information System), which uses VSAM. To obtain information from a specific system, someone in the Center for Computing and Information Technology (CCIT) who was familiar with that system would be tasked to write a report program, which often took several weeks. Getting an integrated report across two or more of these databases was exceedingly difficult. About the only positive thing that could be said about this situation was that it guaranteed job security for CCIT personnel.

Cheryl Bach, an impassioned and imaginative seven-year CCIT veteran, had a better idea. She proposed that a data warehouse be created in a relational DBMS, gathering sanitized information from all of these systems into a single, consistent database. Perhaps predictably, this suggestion was not embraced by the the CCIT rank and file. So Larry Rapagnani, then CCIT director, let Cheryl start a somewhat clandestine project to develop this system, to be called the University Information System (UIS), using a VAX-2000 and Rdb, which the university had been given by Digital Equipment Corporation. The hope was that this minimally funded project would produce an initial system that demonstrated the many benefits of a unified database, thereby reducing the resistance encountered initially.

Cheryl, working with Htay Lay and later with Chris Janton, eventually defined some 300 tables. Some tables were copied over daily from the legacy systems; others were added each pay period, each month, or even each 12 months (for those tables summarizing the fiscal year). The loading phase requires most of each night; access is permitted from 8 A.M. to 8 P.M.

UIS employs a client-server architecture. Users could connect over the network to the server and run queries directly using interactive SQL, access it via COBOL programs running on their client machine, or use the graphical query facilities of their local machine. The system was initially released in December 1992. University staff were encouraged to use the system, and training was provided in the graphical user interface so that users could write their own queries. Access was also given to CCIT personnel, so that they could try it out on their own. While the response from users of direct access to this information was enthusiastic, not a single person within CCIT tried the system for the first two and a half years. Decentralization can be quite threatening.

5.1   INITIAL SCHEMA

Merging the data from each system into UIS required many technical decisions, especially as the data generally wasn't stored in tabular format in the legacy systems. We start with four tables, the EMPLOYEES table, the INCUMBENTS table, the POSITIONS table, and the JOB_TITLES table. The EMPLOYEES table contains 88 columns, of which 4 are germane to this discussion: SSN, LAST_NAME, FIRST_NAME, and ANNUAL_SALARY. The INCUMBENTS table contains 12 columns, including SSN and PCN (position control number). The POSITIONS table contains 16 columns, including PCN and JOB_TITLE_CODE1. The JOB_TITLES table contains 11 columns. This last table is quite sizable; there are over 7000 job titles defined. The University is certainly prepared for future growth: 10 digits are allocated to the job title code!

We focus on the following columns:

```
EMPLOYEES(SSN, LAST_NAME, FIRST_NAME, ANNUAL_SALARY)
INCUMBENTS(SSN, PCN)
POSITIONS(PCN, JOB_TITLE_CODE1)
JOB_TITLES(JOB_TITLE_CODE, JOB_TITLE)
```

Each row of the first table provides information on one employee, each row of the second table provides information on a job assignment for a current employee, each row of the third table describes a particular position (which can be associated with multiple job titles), and each row of the last table describes a particular job title code. The primary key of EMPLOYEES is SSN, the primary key of INCUMBENTS is (SSN, PCN), and the primary key of POSITIONS is PCN. INCUMBENTS.SSN is a foreign key to EMPLOYEES.SSN, INCUMBENTS.PCN is a foreign key to POSITIONS.PCN, and POSITIONS.JOB_TITLE_CODE1 is a foreign key to JOB_TITLES.JOB_TITLE_CODE.

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

Code Fragment 5.1 **What is Bob's salary?**

```
SELECT ANNUAL_SALARY
FROM EMPLOYEES
WHERE FIRST_NAME = 'Bob'
```

To determine the employee's position, all three tables need to be consulted:

Code Fragment 5.2 **What is Bob's position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
  AND EMPLOYEES.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
```

Cheryl also wished to record the date of birth. To do so, she added a column to the EMPLOYEES table, yielding the following schema:

```
EMPLOYEES(SSN, LAST_NAME, FIRST_NAME, ANNUAL_SALARY, BIRTH_DATE)
```

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

Code Fragment 5.3 **What is Bob's date of birth?**

```
SELECT BIRTH_DATE
FROM EMPLOYEES
WHERE FIRST_NAME = 'Bob'
```

SQL is adequate to handle queries on isolated temporal columns.

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 saw in Chapter 3, other temporal types are available for columns. This chapter will investigate how such temporal columns can be used to indicate the period of validity of the *other* columns.

5.2 ADDING HISTORY

To the INCUMBENTS table, Cheryl appended two columns. The first column indicates when the information in the row became valid, that is, when the employee was assigned to that position. The second column indicates when the information was no longer valid, that is, when the employee was assigned to another position or left the university. (Cheryl would have preferred using a period data type, but as we saw before, such a type is not available in SQL-92, nor in Rdb. So a period is simulated with two DATEs.)

Code Fragment 5.4   **Add a period timestamp to** INCUMBENTS.

```
ALTER TABLE INCUMBENTS ADD START_DATE DATE
ALTER TABLE INCUMBENTS ADD END_DATE DATE
```

To the data model, these new columns are identical to the BIRTH_DATE column, in that they are of data type DATE. However, their meaning is quite different. The BIRTH_DATE column is independent of the rest of the table, except for the primary key. However, the *timestamp columns* (START_DATE and END_DATE) interact closely with the other columns, in that they specify the *period of validity* of the values of these columns. In the INCUMBENTS table, the timestamp columns specify the period of validity of the PCN column. This difference between the semantics of the BIRTH_DATE column and the other timestamp columns has far-ranging consequences and is the primary impetus for this entire book.

You might argue that the BIRTH_DATE column is the start of validity of the *employee*, with a (not included) column DEATH_DATE providing the end of validity of the employee. However, the period of validity is with respect to the fact modeled by the entire row and applies to the information of that row. The EMPLOYEES table records the employee's social security number (SSN), first name, last name, and annual salary. Taking the SSN to be time-invariant, which is a common assumption, a row of this table associates a first name, a last name, an annual salary, and a birth date with that SSN. The period of validity would then record when that combination of four values was valid for that SSN. If any of the four values changed—for example, a salary raise or changing the last name of the employee (say, if he or she got married)—the period of validity would be terminated and another row would state the new value. For this reason, the BIRTH_DATE is simply another column and should not be considered to start the period of validity of the row.

Such a table is called a *valid-time table*. This table records the history of the modeled reality. The original table, without temporal support, is termed a *snapshot table*, as it logically captures the state of the enterprise at a single point in time, much as a photographic snapshot does. Snapshot tables are generally kept up-to-date, and so capture reality "as of now." (Jim Melton's compelling metaphor is the "last" frame of a movie still being shot.)

In contrast with valid time, a column that just happens to be of a datetime data type, but that does not indicate when other columns were valid, is termed a *user-defined time column*. BIRTH_DATE is a user-defined time column in INCUMBENTS.

Table 5.1 is an excerpt of the INCUMBENTS table. The first row specifies that employee 111-22-3333 (Bob) had a position 90025 (Senior Vice President, Research) starting at the beginning of 1996, and extending to June 1 of that year, when he transferred to position 120033. He stayed in that position for a total of four months, until October 1, when he transferred to position 137112, where he continues to this day (as we will see, the special date '3000-01-01' denotes "currently valid"). An-

Table 5.1 **An excerpt from the** INCUMBENTS **valid-time state table.**

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-06-01
111223333	120033	1996-06-01	1996-08-01
111223333	120033	1996-08-01	1996-10-01
111223333	137112	1996-10-01	3000-06-01
444332222	120033	1997-01-01	3000-01-01

other employee, 444-33-2222, has remained in the same position since being hired the beginning of the following year.

This table has several interesting features. First, although Bob appears in four rows, he had only one position at any point in time (note that closed-open periods are used in this table). The candidate keys for this particular table are (SSN, PCN, START_DATE), (SSN, PCN, END_DATE), and (SSN, PCN, START_DATE, END_DATE). As we will see shortly, none of these capture the desired constraint.

Second, neither employee has gaps in their position history (a gap represents times when there was no PCN associated with their SSN); disallowing gaps may be appropriate or unacceptable, depending on the semantics of the application.

The third observation is that this table can be viewed as a compact representation of a sequence of snapshot tables, each valid on a particular day. The snapshot table valid on January 1, 1996, contains one row, (111223333, 900225). The snapshot table valid on September 13, 1996, also contains one row, (111223333, 120033). The table valid on February 22, 1997, contains two rows: (111223333, 137112) and (444332222, 120033). This long sequence of snapshot tables is very efficiently encoded in these five rows, by associating a period with each row.

Also note that the second and third rows have identical SSN and PCN values, yet do not represent a duplicate in any of the snapshot tables, as the periods associated with these two rows do not overlap. Indeed, it might be possible to replace these rows with a single row, associated with the period [1996-06-01 – 1996-10-01).

Water Clocks

A sundial is useless at night and on overcast days. The Mesopotamians invented the *clepsydra*, or water clock, for just such situations. This clock is based on the principle that it takes an approximately fixed amount of time for a given amount of water to drain through a small hole, drop by drop. (This same principle—with a different material, sand— is the basis for the hourglass.) Because the length of a day, and thus an hour (see the *Hours* sidebar in Chapter 2), can vary when measured by a sundial, but is of constant length when measured with a clepsydra, the standard hour was defined based on an *equinoctial day*, in which the daylight and nocturnal portions were equal, and thus the hours were also of equal length.

EMPLOYEES.ANNUAL_SALARY records only the most recent salary the employee received. The table as it is in the personnel system is quite a bit more complicated. It also includes STATUS_BEGIN_DATE and STATUS_END_DATE, which together indicate when another included column, EMPLM_STATUS, was valid, though only the most recent status is retained in the table. EMPLOYEES also includes the PRIOR_SSN, as well as 17 dates, indicating when important events occurred, such as PAY_CHANGE_DATE. As discussed above, none of these date fields constitute the commencement or end of the period of validity. In fact, the EMPLOYEES table doesn't even have a period of validity; it is a snapshot table. When examining a particular table, you must determine which DATE columns apply to the row as a whole, via the period of validity for that row, and which DATE columns merely provide information, such as the employee's date of birth or their most recent promotion date.

The PSOS legacy system, however, does record the employment history. The salary history information is included in UIS in the ZPSOS_COMPENSATION_HISTORY table. The compensation history table is quite difficult to decipher. Its key is (CHRONOLOGY_KEY, SSN), the former a 16-character string specifying the date and time the record was inserted. There are also HISTORY_START_DATE and HISTORY_END_DATE columns, indicating when the information in the record applied (the period of validity).

To get at the information desired, we define a view that eliminates most of the columns. The WHERE clause extracts the most recently stored row for each period of validity. We are interested only in full-time employees (full-time equivalent = 1), for which the hourly rate is actually the annual salary.

Code Fragment 5.5   **Extract the relevant information.**

```
CREATE VIEW SAL_HISTORY
AS SELECT SSN, SALARY_HOUR_RATE AS AMOUNT,
  HISTORY_START_DATE, HISTORY_END_DATE
FROM ZPOS_COMPENSATION_HISTORY AS Z
WHERE CHRONOLOGY_KEY = MAX(SELECT CHRONOLOGY_KEY
      FROM ZPOS_COMPENSATION AS Z2
      WHERE Z2.SSN = Z.SSN
        AND Z2.HISTORY_START_DATE < Z.HISTORY_END_DATE
        AND Z.HISTORY_START_DATE < Z2.HISTORY_END_DATE)
      AND EMPLOYEE_FTE = 1.00
```

A valid-time table records the history of the modeled reality. The history can be retained by adding timestamp column(s).

The value of the column(s) specifying the period of validity of a row is called the *timestamp* of the row; the columns themselves are called the *timestamp columns*, or the *timestamps of the table*. This term, in lowercase letters, should be distinguished from SQL-92's TIMESTAMP data type.

5.3 TEMPORAL KEYS

The first consequence of adding valid-time support to a table is that the primary key of such tables needs to take the timestamp into consideration. The value of the primary key of a table must be unique; that is, each value must be contained in at most one row of the table. For the original INCUMBENTS table, the value of the (SSN, PCN) pair of columns is unique, *at any point in time*. Hence, the following table constraint works fine.

Code Fragment 5.6 **The primary key of** INCUMBENTS **is (**SSN, PCN**).**

```
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN)
```

Informally, this says that no employee (identified by an SSN value) can have the same position more than once simultaneously.

> The original primary key is not, by itself, a primary key of the temporal table.

When history is added, there may well be several rows with the same SSN and PCN. To handle this, one or both of the new temporal columns can be appended to the key. Cheryl specified the primary key of INCUMBENTS as (SSN, PCN, END_DATE); the primary key of SAL_HISTORY is (SSN, HISTORY_START_DATE).

Cheryl could have just as easily used the start date, rather than the end date. She could also have used both the start and end dates in the primary key. Unfortunately, none of these three alternatives is sufficiently restrictive. The problem arises if there are overlapping periods associated with the same SSN. Consider the following table, which is consistent with all three of the possible primary keys just discussed. This table uses a closed-open representation of periods, and so, for example, the period of validity of the first row includes the last day in May, but not the first day in June.

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-06-01
111223333	900225	1996-04-01	1996-10-01

We wish to specify that there can be only one (SSN, PCN) pair, *at any given time*. Put another way, no one can have a particular position twice at the same time. The above table violates this constraint: in May of 1996, Bob has position code 900225 twice. The primary key constraint should have disallowed this because two rows have the same SSN and PCN for any specific day during the months of April and May. Unfortunately, none of the following attempts prevent the above (INCUMBENTS) table.

Code Fragment 5.7   **Attempting to specify a primary key at any point in time.**

```
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN)
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN, START_DATE)
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN, END_DATE)
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN, START_DATE,
    END_DATE)
```

Adding the timestamp does not serve to convert a nontemporal key to a temporal key.

A sequenced *constraint is one that is applied independently at each point in time.*

Including the start date, the end date, or both in the primary key does not prevent Bob from having a position (in this case, position 900225) twice in May 1996 because the rows have different values for both of these columns.

What is needed is a *sequenced* constraint, which is applied at each point in time. The constraint desired is that no two rows have the same value for the SSN and the PCN. Because we want this constraint to be satisfied at *every* point in time, it becomes a sequenced constraint. All constraints specified on a snapshot table have sequenced counterparts, specified on the analogous valid-time state table. A sequenced primary key constraint can be specified in SQL as follows:

Code Fragment 5.8   **(**SSN**,** PCN**)** **is a sequenced primary key for** INCUMBENTS**.**

```
CREATE ASSERTION seq_primary_key
CHECK (NOT EXISTS ( SELECT *
                    FROM INCUMBENTS AS I1
                    WHERE 1 < (SELECT COUNT(SSN)
                      FROM INCUMBENTS AS I2
                      WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                        AND I1.START_DATE < I2.END_DATE
                        AND I2.START_DATE < I1.END_DATE))
      AND NOT EXISTS ( SELECT *
                       FROM INCUMBENTS AS I
                       WHERE I.SSN IS NULL OR I.PCN IS NULL)
)
```

A sequenced primary key can be expressed as an SQL assertion or table constraint.

The last two predicates in the WHERE clause constitute the overlap predicate on two periods defined by their delimiting dates. We use a COUNT aggregate to ensure that I1 and I2 aren't the exact same row. The intuition of this sequenced constraint is "no employee can have the same position more than once simultaneously."

In the nested SELECT, we could have used the SQL-92 OVERLAPS predicate, instead of the last two lines:

```
AND (I1.START_DATE, I1.END_DATE) OVERLAPS (I2.START_DATE, I1.END_DATE)
```

If a closed-closed representation for the period of validity is used (in which case the END_DATE of the above table would be the last day that PCN was valid, or 1996-05-30), the predicate must be altered slightly:

Code Fragment 5.9 **(SSN, PCN) is a sequenced primary key for** INCUMBENTS**, assuming a closed-closed timestamp representation.**

```
CREATE ASSERTION seq_primary_key
CHECK (NOT EXISTS ( SELECT *
                    FROM INCUMBENTS AS I1
                    WHERE 1 < (SELECT COUNT(SSN)
                      FROM INCUMBENTS AS I2
                      WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                        AND I1.START_DATE <= I2.END_DATE
                        AND I2.START_DATE <= I1.END_DATE))
       AND NOT EXISTS ( SELECT *
                        FROM INCUMBENTS AS I
                        WHERE I.SSN IS NULL OR I.PCN IS NULL)
)
```

All the tables shown in this book use a closed-open representation for the period of validity, unless otherwise indicated. The code fragments will also make this assumption. If some other representation for the period of validity is used, the SQL predicates must be examined and possibly changed.

5.4   HANDLING NOW

> It's the fabulous castle of *Now*.
> You can walk in and wander about,
> But it's so very thin,
> Once you *are*, then you've *been*—
> And soon as you're in, you're out.
>
> —Shel Silverstein, "The Castle"

What should the timestamp be for current data? In the PSOS legacy system, current data is indicated with an end date of 00-00-00, which is the code for "not applicable." The extraction utility replaces this with January 1, 1860. (In Rdb, the minimum value for a VMS_DATE type is 1858-11-17. However, the designers wanted this value to be a little more recognizable, so they chose the first day of the following decade. They are thinking about using 1859-12-31 to represent a date that could not be correctly interpreted, such as a YYMMDD format date with

The special value "now" can be stored as a specific instant value that will not occur otherwise.

a value of 990001.) Hence, Bob's current salary in INCUMBENTS has a START_DATE of July 1, 1996, and an END_DATE of January 1, 1860. One advantage is that when records are ordered by end date, current row(s) show up first.

Using this date for current data requires that the applications and queries that use UIS have to treat such values specially. Specifically, to identify the current records, the following predicate can be used:

```
WHERE INCUMBENTS.END_DATE = DATE '1860-01-01'
```

Anyone using the database would have to be told early on about this special, and prevalent, time value. And clearly, taking this at face value is a blatantly false model of the enterprise.

There are several somewhat more appealing alternatives. Rather than resorting to a particular date, NULL can be used. This yields a slightly more readable predicate.

```
WHERE INCUMBENTS.END_DATE IS NULL
```

"Now" can also be represented with a null value, but this complicates queries.

The disadvantages are (1) users sometimes get confused when they encounter a date of NULL, (2) SQL states that any comparison with a null value returns false, with the implication that rows with null values will simply be absent from the result of most queries that contain predicates on the timestamps, which would confuse users, and (3) other uses of NULL are no longer available. We cannot state, for example, that the end date is unknown, which is quite different from stating that the end date is "now."

Another approach, one that is used extensively, is to set the end date to the largest value in the timestamp domain (termed *the end of time*). The legacy financial system, FRS, uses 99-99-99 for this purpose. The extraction process for UIS converts this to 3000-01-01. Why not 9999-12-31? The Sybase DBMS was being considered for UIS, and it doesn't support that date. Chris Janton wanted this date to be within the range of acceptable dates for all the prominent database systems available then and to be visually apparent. So he picked a millennium date that was far into the future, thereby creating the year 3000 problem. However, the designers will be long gone when this problem rears its head. Using a date (far) into the future allows CURRENT_DATE to be used to identify current records:

```
WHERE INCUMBENTS.START_DATE <= CURRENT_DATE
  AND CURRENT_DATE < INCUMBENTS.END_DATE
```

"Now" can be represented with "forever," or a close approximation. However, it still renders the data a rather inaccurate model of reality.

The problem is that this model is also conspicuously fallacious. We can safely state that Bob will not have this position in the year 3000, though that is exactly what the current row records.

With the lack of an entirely acceptable solution, we will use 3000-01-01 in all of our UIS examples to represent both "now" and "forever."

5.5   UNIQUENESS REEXAMINED

A primary key constraint states two things about the associated table. First, the value of each of the key columns in any row cannot be null. Second, there can be no two rows with the same values for the key columns. In fact, a primary key can be expressed in another way, as a pair of constraints, NOT NULL and UNIQUE.

Two rows are *value-equivalent* if the values of their nontimestamp columns are identical. Value equivalence is a weak form of duplication.

Let's examine the uniqueness constraint in the context of the INCUMBENTS table shown in Table 5.2, considering for now the entire row. While in this case all columns are either key columns or timestamp columns, the following definitions hold even in the presence of other, data columns.

There are four kinds of duplicates involved, all but one of which are present in Table 5.2. All of the rows are considered *value-equivalent*, in that the values of all of the columns except for the timestamp columns are identical. Value equivalence is a weak form of duplication.

Two rows are *sequenced duplicates* if they are duplicates at some instant.

The first two rows illustrate a *sequenced* duplicate. Here, the values of nontimestamp columns, in this case SSN and PCN, are value-equivalent and the periods of validity overlap, in this case for the months of April and May of 1996. As with primary keys, the adjective *sequenced* means that the operation or constraint is applied independently at every point in time. Here, Bob has two salaries for these two months, as well as for the month of December 1997.

Two rows are *current duplicates* if they are sequenced duplicates at the current instant.

A variant of sequenced duplicate is a *current duplicate*, in which there are duplicate rows in the current state. It is currently March 20, 1997; there are no current duplicates because only one row, the last, is currently valid. Interestingly, whether a table contains current duplicate rows can change over time, even if no modifications are made to the table. In December 1997, a current duplicate will suddenly appear in Table 5.2.

Table 5.2 A table containing several kinds of duplicates.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-10-01	1998-01-01
111223333	120033	1997-12-01	1998-01-01

Table 5.3 Implications among duplicate variants.

	Sequenced	Current	Value-equivalent	Nonsequenced
Sequenced	√		√	
Current	√	√	√	
Value-equivalent			√	
Nonsequenced	√		√	√

Two rows are nonsequenced duplicates if the values of all columns are identical.

Finally, the second and third rows are *nonsequenced duplicates*, in which the values of all of the columns, including the time-stamp columns, are identical. This adjective emphasizes that the property (in this case, duplicates) is not applied independently at each point in time, but rather is applied to the table as a whole, ignoring its temporal nature.

Table 5.3 indicates how these variants interact. Each entry specifies whether two rows satisfying the variant on the left 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 nonsequenced 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 first two rows of the above INCUMBENTS table.

The least restrictive form of duplication is value equivalence, as it simply ignores the timestamps. Note that in Table 5.3 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.

Code Fragment 5.10   **Prevent value-equivalent rows in** INCUMBENTS**.**

```
CREATE TABLE INCUMBENTS (
   ...
   UNIQUE (SSN, PCN)
)
```

The SQL UNIQUE constraint prevents nonsequenced duplicates.

Intuitively, a value-equivalent constraint (or a value-equivalent primary key constraint) states that "once a position is assigned to an employee, 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 nonsequenced duplicates, by simply including the timestamp columns.

Code Fragment 5.11   **Prevent nonsequenced duplicates in** INCUMBENTS**.**

```
CREATE TABLE INCUMBENTS (
    ...
  UNIQUE (SSN, PCN, START_DATE, END_DATE)
)
```

Nonsequenced uniqueness constraints are easy to specify in SQL, but do not correspond to a naturally stated condition on the modeled reality.

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 (or the closely related nonsequenced primary key constraint) is something like "an employee cannot have the same position twice over identical periods." However, this constraint can be satisfied by simply starting one of the assignments a day earlier or later; the employee can still have two identical positions for overlapping periods.

Current duplicates involve just a little more effort.

Code Fragment 5.12   **Prevent current duplicates in** INCUMBENTS**.**

```
CREATE TABLE INCUMBENTS (
...
  CHECK (NOT EXISTS ( SELECT *
                      FROM INCUMBENTS AS I1
                      WHERE 1 < (SELECT COUNT(SSN)
                        FROM INCUMBENTS AS I2
                        WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                          AND I1.START_DATE <= CURRENT_DATE
                          AND CURRENT_DATE < I1.END_DATE
                          AND I2.START_DATE <= CURRENT_DATE
                          AND CURRENT_DATE < I2.END_DATE)))
)
```

Current uniqueness constraints require an SQL constraint or assertion, and are rather fragile.

Here the intuition is that no employee can have two identical positions. 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 appending the END_DATE.

Code Fragment 5.13   **Prevent current duplicates in** INCUMBENTS**, assuming no future data.**

```
CREATE TABLE INCUMBENTS (
    ...
  UNIQUE (SSN, PCN, END_DATE)
)
```

The SQL UNIQUE constraint with the end date prevents current duplicates, if future data is never stored.

This works because all current data will have the same END_DATE (the special value DATE '3000-01-01').

As a primary key is just a combination of UNIQUE and NOT NULL, we can prevent sequenced duplicates by removing the NOT NULL portion from CF-5.8.

Code Fragment 5.14   **Prevent sequenced duplicates in** INCUMBENTS**.**

```
CREATE TABLE INCUMBENTS (
    ...
  CHECK (NOT EXISTS ( SELECT *
                      FROM INCUMBENTS AS I1
                      WHERE 1 < (SELECT COUNT(SSN)
                        FROM INCUMBENTS AS I2
                        WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                          AND I1.START_DATE < I2.END_DATE
                          AND I2.START_DATE < I1.END_DATE)))
)
```

The intuition behind a sequenced unique (or sequenced primary key) constraint is that "at no time can an employee have two identical positions." This constraint is a natural one. A sequenced constraint is the logical extension of a conventional constraint on a nontemporal table.

Sequenced uniqueness constraints are also specified with an SQL constraint or assertion. Such constraints are analogous to conventional uniqueness constraints on nontemporal tables.

If we know that the application will make only current modifications, that is, will only modify the current state, then we can express a sequenced constraint via a current constraint. There are three cases to consider to see that this holds: past, current, and future. Future data cannot be stored, and so the sequenced constraint will never be violated in the future. For current data, the current constraint will ensure the sequenced constraint. For past data, that data at one time was current data, and so satisfies the constraint. Hence, a current constraint will imply a sequenced constraint, *if* only current modifications were made, and if the current constraint was present from the definition of the table. For sequenced uniqueness, we saw above (CF-5.13) that current uniqueness could be specified by appending the END_DATE. This also works for sequenced duplicates, under the stated assumptions.

Code Fragment 5.15   **Prevent sequenced duplicates in** INCUMBENTS**, assuming only current modifications.**

```
CREATE TABLE INCUMBENTS (
    ...
  UNIQUE (SSN, PCN, END_DATE)
)
```

To review, let's consider briefly the integrity constraint "Each employee has at most one position." (Contrast this with the constraint discussed above: "Each employee can have a position at most once," which allows an employee to have two positions at one time.) On a snapshot table, this is expressed with

```
UNIQUE(SSN)
```

The sequenced variety, "At any time, an employee has at most one position," is violated by the following temporal table:

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	900225	1996-04-01	1996-10-01

To avoid such situations, the following expresses the sequenced constraint.

Code Fragment 5.16   `INCUMBENTS.SSN` **is sequenced unique.**

```
CREATE ASSERTION seq_primary_key
CHECK (NOT EXISTS ( SELECT *
                    FROM INCUMBENTS AS I1
                    WHERE 1 < (SELECT COUNT(SSN)
                        FROM INCUMBENTS AS I2
                        WHERE I1.SSN = I2.SSN
                          AND I1.START_DATE < I2.END_DATE
                          AND I2.START_DATE < I1.END_DATE))
      AND NOT EXISTS ( SELECT *
                       FROM INCUMBENTS AS I
                       WHERE I.SSN IS NULL)
)
```

The nonsequenced variant is "an employee cannot have more than one position over two identical periods." This is expressed as follows:

Code Fragment 5.17   `INCUMBENTS.SSN` **is nonsequenced unique.**

```
UNIQUE(SSN, START_DATE, END_DATE)
```

Finally, the current variant is "an employee has [note present tense] at most one position," expressed as follows:

Code Fragment 5.18   `INCUMBENTS.SSN` **is current unique.**

```
CHECK (NOT EXISTS ( SELECT *
                    FROM INCUMBENTS AS I1
                    WHERE AND START_DATE <= CURRENT_DATE
                      AND CURRENT_DATE < END_DATE
                      AND 1 < (SELECT COUNT(SSN)
                        FROM INCUMBENTS AS I2
                        WHERE I1.SSN = I2.SSN
                          AND I2.START_DATE <= CURRENT_DATE
                          AND CURRENT_DATE < I2.END_DATE)))
```

5.6   REFERENTIAL INTEGRITY

A referential integrity constraint specifies that the value of the specified column in every row of the referencing table appears as the value of a specified column in a row of the referenced column. How such constraints are expressed depends heavily on whether the referencing and referenced tables are temporal tables. There are four cases, depending on whether the referencing table is temporal and whether the referenced table is temporal.

Case 1   Neither table is temporal.

If neither table is temporal, then SQL's constructs are perfectly adequate. Assuming for the moment that the `INCUMBENTS` table has no timestamp columns, the fact that `INCUMBENTS.PCN` is a foreign key for `POSITIONS.PCN` can be expressed as follows:

Code Fragment 5.19   `INCUMBENTS.PCN` **is a foreign key for** `POSITIONS.PCN` **(neither table is temporal).**

```
CREATE TABLE INCUMBENTS (
   ...
      PCN CHAR(6) REFERENCES POSITIONS,
      ...
)
```

Case 2   Only the referencing table is temporal.

If the referencing table is temporal, but the referenced table is not, the same code works as well. Returning to `INCUMBENTS` being a valid-time state table, with columns `START_DATE` and `END_DATE`, the above REFERENCES fragment works fine. Here the assumption is that the nontemporal table contains *time-invariant data*, that is, data that doesn't vary over time.

Case 3 **Both tables are temporal.**

If the *referenced* table is temporal, the situation is considerably more complex. In order to discuss this, we render POSITIONS temporal by adding START_DATE and END_DATE columns. It turns out that in UIS position code numbers are sometimes invalidated, and occasionally a PCN is reused. However, UIS only maintains the current state of the POSITIONS table, so when a PCN is reused, an old value will incorrectly be matched to a new job title code. The same holds for the job title code. In fact, there are (at the time this was written) 881 job titles that appear in POSITIONS.JOB_TITLE_CODE1 that are not associated with any rows in JOB_TITLES. We wish to avoid this situation by defining appropriate referential integrity constraints.

As with keys and duplicates, there are three kinds of referential integrity constraints on temporal tables: current, sequenced, and nonsequenced.

Current referential integrity ("the PCN of all current incumbents must be listed in the current positions") is straightforward, using the trick of converting a predicate of the form $\forall P$ into $\neg\exists(\neg P)$. For a current foreign key, P is "a current incumbent that has a current position," and $\neg P$ is "a current incumbent whose position is not current."

Code Fragment 5.20 INCUMBENTS.PCN **is a current foreign key for** POSITIONS.PCN **(both tables are temporal).**

```
CREATE ASSERTION INCUMBENTS_Current_Referential_Integrity
CHECK (NOT EXISTS (
                  SELECT *
                  FROM INCUMBENTS AS I
                  WHERE I.END_DATE = DATE '3000-01-01'
                    AND NOT EXISTS (
                        SELECT *
                        FROM POSITIONS AS P
                        WHERE I.PCN = P.PCN
                        AND P.END_DATE = DATE '3000-01-01'))
)
```

Here we just extract the current state on which to apply the constraint. Mirroring the current uniqueness constraint, current referential integrity can be satisfied today yet be violated tomorrow, even if no changes are made to either table.

Current referential integrity requires an SQL constraint or assertion.

As with primary keys and duplicates, if future data is never present, current referential integrity still requires the above assertion because the referenced and referencing tables can change independently. For the nonsequenced referential integrity constraint ("for each value of INCUMBENTS.PCN, there existed *at some, possibly different, time* that value in POSITIONS.PCN"), CF-5.19 works perfectly well. The fact that POSITIONS is temporal means that past positions have become

invalid. Nonsequenced referential integrity ignores this time-varying behavior and is content to match current incumbents with out-of-date positions.

Nonsequenced referential integrity is easy to express, but is unnatural.

The temporal analog of a nontemporal referential integrity constraint is the sequenced constraint: "At each point in time, each incumbent's PCN is valid *at that time*." This statement applies the intuition of referential integrity to time-varying information. As with primary keys and duplicates, stating a sequenced referential integrity constraint in English is natural, but stating it in SQL is challenging.

Asserting a sequenced foreign key

The key is a sequenced foreign key if, for all rows *r* in the referencing table,

- there is a row with that key value valid in the referenced table when *r* started,
- there is a row with that key value valid in the referenced table when *r* stopped,
- and there are no gaps when there are no rows in the referenced table, during *r*'s period of validity, that have that key value.

Code Fragment 5.21   `INCUMBENTS.PCN` **is a sequenced foreign key for** `POSITIONS.PCN` **(both tables are temporal).**

```
CREATE ASSERTION INCUMBENTS_Sequenced_Referential_Integrity
CHECK (NOT EXISTS (
    SELECT *
    FROM INCUMBENTS AS I
    -- there was a row valid in POSITIONS  when I started
    WHERE NOT EXISTS (
        SELECT *
        FROM POSITIONS AS P
        WHERE I.PCN = P.PCN
          AND P.START_DATE <= I.START_DATE
          AND I.START_DATE < P.END_DATE)
      -- there was a row valid in POSITIONS  when I ended
    OR NOT EXISTS (
        SELECT *
        FROM POSITIONS AS P
        WHERE I.PCN = P.PCN
          AND P.START_DATE < I.END_DATE
          AND I.END_DATE <= P.END_DATE)
      -- there are no gaps in POSITIONS  during I's period of validity
    OR EXISTS (
```

```
          SELECT *
          FROM POSITIONS AS P
          WHERE I.PCN = P.PCN
            AND I.START_DATE < P.END_DATE
            AND P.END_DATE < I.END_DATE
            AND NOT EXISTS (
                SELECT *
                FROM POSITIONS AS P2
                WHERE P2.PCN = P.PCN
                  AND P2.START_DATE <= P.END_DATE
                  AND P.END_DATE < P2.END_DATE)))
     )
```

Sequenced referential integrity is the natural extension to time-varying tables, but requires a complex SQL assertion.

This assertion may be read as follows. The outermost NOT EX-ISTS states that no row I of INCUMBENTS fails the referential integrity test. The three predicates in the WHERE clause provide ways for row I to fail the test. First, the test fails if there is no row in POSITIONS valid at the start of row I's period of validity. Second, the test fails if there is no row in POSITIONS valid at the end of row I's period of validity. Third, the test fails if there is a gap during row I's period of validity, a time when no row of POSITIONS was valid. A gap exists if there is a row P that ends during row I's period of validity that is not "extended" (towards I.END_DATE) by another row.

If we are assured that the histories in the referenced table are contiguous, that is, that there are no gaps in these histories, then sequenced referential integrity is easier to express as two assertions.

Code Fragment 5.22 POSITIONS.PCN **defines a contiguous history.**

```
CREATE ASSERTION POSITIONS_Contiguous_History
CHECK (NOT EXISTS (
    SELECT *
    FROM POSITIONS AS P, POSITIONS AS P2
    WHERE P.END_DATE < P2.START_DATE
      AND P.PCN = P2.PCN
      AND NOT EXISTS (
          SELECT *
          FROM POSITIONS AS P3
          WHERE P3.PCN = P.PCN
            AND (((P3.START_DATE <= P.END_DATE)
                  AND (P.END_DATE < P3.END_DATE))
                OR ((P3.START_DATE < P2.START_DATE)
                  AND (P2.START_DATE <= P3.END_DATE)))))
    )
```

Exploiting contiguous histories in the referenced table simplifies sequenced referential integrity when both tables are temporal.

Requiring a contiguous history is a nonsequenced constraint. Unlike a sequenced constraint, which must be true independently at each point in time, a nonsequenced constraint requires examining the table at multiple points of time. The absence of a PCN on a particular day constitutes a gap in the history only if there is a PCN for this SSN present both before and after this date.

Given that there are no gaps, we can check for containment of the referencing period of validity by the contiguous history in the referenced table by simply checking the delimiting instants of the referencing period of validity.

Code Fragment 5.23   INCUMBENTS.PCN **is a sequenced foreign key for** POSITIONS.PCN **(both tables are temporal, version 2).**

```
CREATE ASSERTION INCUMBENTS_Sequenced_Referential_Integrity
CHECK (NOT EXISTS (
                    SELECT *
                    FROM INCUMBENTS AS I
                    WHERE NOT EXISTS (
                        SELECT *
                        FROM POSITIONS AS P
                        WHERE I.PCN = P.PCN
                          AND P.START_DATE <= I.START_DATE
                          AND I.START_DATE < P.END_DATE)
                      OR NOT EXISTS (
                        SELECT *
                        FROM POSITIONS AS P
                        WHERE I.PCN = P.PCN
                          AND P.START_DATE < I.END_DATE
                          AND I.END_DATE <= P.END_DATE)))
```

Case 4   **Only the referenced table is temporal.**

The final case to consider is when the referencing table (here, INCUMBENTS) is a nontemporal table and the referenced table (here, POSITIONS) is a temporal table.

The current constraint is easy to express.

Code Fragment 5.24   INCUMBENTS.PCN **is a current foreign key for** POSITIONS.PCN **(only** POSITIONS **is temporal).**

```
CREATE ASSERTION INCUMBENTS_Current_Referential_Integrity
CHECK (NOT EXISTS (
    SELECT *
    FROM INCUMBENTS AS I
    WHERE NOT EXISTS (
```

More Water Clocks

Water clocks have their disadvantages. They are susceptible to frost, and hence are not practical in Europe or other northern climes. Indeed, it has been argued that the mechanical clock had to be invented in Europe, a place where the prior technology of sundials and water clocks was inadequate on cloudy, cold days.

Arguably among the most intricate of water clocks was Su Song's astronomical clock, completed in 1094 C.E. This clock, which weighed several tons and filled a 40-foot tower, reproduced the movements of the sun, the moon, and selected stars, as well as indicating hours and *k'o*, each of which equals 14 minutes and 24 seconds of our time. Mongols invaded some 30 years later and carried away some of the clock, and within 50 years, this magnificent clock was forgotten, to await rediscovery by Joseph Needham in 1954. Although it would be appealing for Song's clock to be a forerunner of the European mechanical clock, no such connection has yet been uncovered.

```
                   SELECT *
                   FROM POSITIONS AS P
                   WHERE I.PCN = P.PCN
                     AND P.END_DATE = DATE '3000-01-01'))
          )
```

The nonsequenced constraint is, again, expressed using CF-5.19.

The case where the referencing table is nontemporal but the referenced table is temporal reduces to the other cases just described.

For the sequenced constraint, we need to be more precise on what "at each point in time" means for a nontemporal table. There are at least two reasonable interpretations. One is that the nontemporal table records *current* data, in which case a sequenced constraint is equivalent to a current constraint. A different interpretation is that the nontemporal table contains time-invariant data. In that case, CF-5.19 works fine.

For the `INCUMBENTS` table, the most appropriate interpretation is that this table records current data, in which case CF-5.20 suffices.

5.7 CONSTRAINT ATTRIBUTES*

SQL-92 and some DBMSs provide attributes on constraints and assertions. A given constraint can be specified as `DEFERRABLE` or `NOT DEFERRABLE`. It is critical that the constraints (and assertions) exemplified in this chapter be `DEFERRABLE`. Otherwise, the constraint is checked at the end of every SQL statement, which is undesirable

Temporal constraints and assertions should be DEFERRABLE INITIALLY DEFERRED, with each transaction containing a modification resetting this via a SET CONSTRAINTS ALL DEFERRED.

because most modifications on temporal tables require several SQL statements to implement. As we shall see in Chapter 7, a single logical deletion requires one or two UPDATE statements, a DELETE statement, and possibly an INSERT statement. While the database may be consistent at the end of this series of statements, it most likely will *not* be at the end of the intermediate SQL statements.

Similarly, the constraint must be set to INITIALLY DEFERRED for the same reason (INITIALLY IMMEDIATE requests per-statement checking).

Neither of these is the default. One must request DEFERRABLE INITIALLY DEFERRED for each constraint on a temporal table.

5.8   IMPLEMENTATION CONSIDERATIONS

The code fragments in this chapter were implemented on a variety of DBMSs.

5.8.1   IBM DB2 Universal Database

IBM DB2 supports a UNIQUE constraint (see CF-5.10 and CF-5.11) only in version 5 (Universal Database); in prior versions, such constraints must be expressed as triggers, or as PRIMARY KEY constraints if it is also desired that the indicated columns are not nullable. Here is the DB2 UDB version of CF-5.10. Even in IBM UDB 5, UNIQUE requires NOT NULLABLE, and so is equivalent to a primary key constraint.

Code Fragment 5.25   **Prevent value-equivalent rows in** INCUMBENTS**, in DB2 UDB 5.**

```
CREATE TABLE INCUMBENTS (
  SSN DECIMAL(9,0) NOT NULL,
  PCN DECIMAL(6,0) NOT NULL,
  START_DATE DATE,
  END_DATE DATE,
  PRIMARY KEY (SSN, PCN)
)
```

IBM DB2 UDB also does not support assertions, nor a SELECT statement within a CHECK constraint; triggers are used to implement these checks. As an example, CF-5.8 may be implemented as the following DB2 UDB trigger.

Code Fragment 5.26   (SSN, PCN) **is a sequenced primary key for** INCUMBENTS**, in DB2 UDB.**

```
CREATE TRIGGER seq_primary_key NO CASCADE
BEFORE INSERT ON INCUMBENTS REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS ( SELECT *
               FROM INCUMBENTS AS I
               WHERE I.SSN = N.SSN AND I.PCN = N.PCN
                 AND I.START_DATE < N.END_DATE
                 AND N.START_DATE < I.END_DATE)
      OR N.SSN IS NULL
      OR N.PCN IS NULL)
SIGNAL SQLSTATE '70003' ('Violates Sequenced primary key')
```

Using triggers instead of CHECK constraints can significantly complicate matters. As an example, to specify current referential integrity (CF-5.20) in DB2 UDB requires four triggers—INSERT and UPDATE triggers on INCUMBENTS and DELETE and UPDATE triggers on POSITIONS—instead of the single fairly simple CHECK constraint given earlier.

5.8.2   Microsoft Access

"Forever" in Microsoft Access 97 and Access 2000 is 23:59:59 December 31, 9999 C.E.

Neither version of Microsoft Access supports assertions. Assertion checking can be implemented in Access via Visual Basic functions that are activated by events provided in Access forms, in particular, the After Insert event. As an example, the following is the Visual Basic code to implement a sequenced primary key (CF-5.8).

Code Fragment 5.27   (SSN, PCN) **is a sequenced primary key for** INCUMBENTS**, in Access.**

```
Function Sequenced_Primary_Key()
Dim dbs As Database, rst As Recordset
Dim rst2 As Recordset

Set dbs = CurrentDb

Set rst = dbs.OpenRecordset("SELECT I1.SSN " _
        & "FROM INCUMBENTS AS I1 " _
        & "WHERE 1 < ( SELECT COUNT (SSN) " _
        & "FROM INCUMBENTS AS I2 " _
        & "WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN " _
        & "AND I1.START_DATE < I2.END_DATE " _
        & "AND I2.START_DATE < I1.END_DATE); ")
```

```
Set rst2 = dbs.OpenRecordset("SELECT * " _
        & "FROM INCUMBENTS AS I " _
        & "WHERE I.SSN IS NULL " _
        & "OR I.PCN IS NULL ;")

If (rst1.EOF And rst1.BOF) Or (rst2.EOF And rst2.BOF) Then
  MsgBox ("Insertion completed")
Else
  MsgBox ("You entered an invalid input")
End If

End Function
```

If the result of the query (the `OpenRecordset`) is empty, then the assertion is satisfied.

Microsoft Access 97 also doesn't support full nesting of SQL statements. Some complex queries have to be divided into several one-statement queries, then combined in Visual Basic functions. For example, here is the Access equivalent of CF-5.21.

Code Fragment 5.28   `INCUMBENTS.PCN` **is a sequenced foreign key for** `POSITIONS.PCN` **(both tables are temporal), in Access.**

```
Function Sequenced_Foreign_Key( )

Dim dbs As Database
Dim rst1 As Recordset
Dim rst2 As Recordset
Dim rst3 As Recordset
Dim Record_Number As Long

Set dbs = CurrentDb

Set rst1 = dbs.OpenRecordset("SELECT * " _
        & "FROM INCUMBENTS AS I " _
        & "WHERE NOT EXISTS ( SELECT * " _
        & "FROM POSITIONS AS P " _
        & "WHERE I.PCN = P.PCN " _
        & "AND P.START_DATE <= I.START_DATE " _
        & "AND I.START_DATE < P.END_DATE ) ")

Set rst2 = dbs.OpenRecordset("SELECT * " _
        & "FROM INCUMBENTS AS I " _
        & "WHERE NOT EXISTS ( SELECT * " _
        & "FROM POSITIONS AS P " _
```

```
        & "WHERE I.PCN = P.PCN " _
        & "AND P.START_DATE < I.END_DATE " _
        & "AND I.END_DATE <= P.END_DATE ) ")

Set rst3 = dbs.OpenRecordset("SELECT * " _
        & "FROM INCUMBENTS AS I " _
        & "WHERE EXISTS ( SELECT * " _
        & "FROM POSITIONS AS P " _
        & "WHERE I.PCN = P.PCN " _
        & "AND I.START_DATE < P.END_DATE " _
        & "AND P.END_DATE < I.END_DATE " _
        & "AND NOT EXISTS ( " _
        & "SELECT * " _
        & "FROM POSITIONS AS P2 " _
        & "WHERE P2.PCN = P.PCN " _
        & "AND P2.START_DATE <= P.END_DATE " _
        & "AND P.END_DATE < P2.END_DATE )) ")

If (rst1.EOF And rst1.BOF) Or (rst2.EOF And rst2.BOF)
    Or (rst3.EOF And rst3.BOF) Then
  MsgBox ("Insertion completed")
Else
  MsgBox ("You entered an invalid input")
End If

End Function
```

Each `OpenRecordset` handles one of the NOT EXISTS clauses of CF-5.21. Microsoft Access 2000 does support nested SQL statements. While the above function will work in Access 2000, a single `OpenRecordset` statement is also acceptable.

5.8.3 Microsoft SQL Server

Microsoft SQL Server does not support ASSERTIONs; these must be implemented as triggers. As an example, CF-5.8 may be implemented as the following SQL Server trigger.

Code Fragment 5.29 (SSN, PCN) **is a sequenced primary key for** INCUMBENTS**, in Microsoft SQL Server.**

```
CREATE TRIGGER Seq_Primary_Key ON INCUMBENTS
FOR INSERT, UPDATE, DELETE AS
```

```
BEGIN
  IF (( EXISTS ( SELECT I1.SSN
                 FROM INCUMBENTS AS I1
                 WHERE 1 < (SELECT COUNT(I2.SSN)
                            FROM INCUMBENTS AS I2
                            WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                              AND I1.START_DATE < I2.END_DATE
                              AND I2.START_DATE < I1.END_DATE)))
    OR ( EXISTS ( SELECT *
                  FROM INCUMBENTS AS I
                  WHERE I.SSN IS NULL OR I.PCN IS NULL))
    )

  RAISERROR('Transaction violates sequenced constraint', 1, 2)
            ROLLBACK TRANSACTION
END
```

5.8.4   Sybase SQLServer

Sybase SQLServer does not support ASSERTIONs; these must be implemented as triggers. As an example, CF-5.8 may be implemented as the following Sybase trigger.

Code Fragment 5.30   **(**SSN**, PCN) is a sequenced primary key for** INCUMBENTS**, in Sybase.**

```
CREATE TRIGGER Seq_Primary_Key ON INCUMBENTS
FOR INSERT, UPDATE, DELETE AS
BEGIN
  IF (( EXISTS ( SELECT I1.SSN
                 FROM INCUMBENTS AS I1
                 WHERE 1 < (SELECT COUNT(I2.SSN)
                            FROM INCUMBENTS AS I2
                            WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                              AND I1.START_DATE < I2.END_DATE
                              AND I2.START_DATE < I1.END_DATE)))
    OR ( EXISTS ( SELECT *
                  FROM INCUMBENTS AS I
                  WHERE I.SSN IS NULL OR I.PCN IS NULL))
    )

  RAISERROR('Transaction violates sequenced constraint', 1, 2)
            ROLLBACK TRANSACTION
END
```

Another approach is to use the `syb_identity` function to differentiate rows, similar to `rowid` in Oracle (compare with CF-5.31, below). To use this, "auto identity" must be turned on via

```
sp_dboption  database_name, "auto identity", "true"
```

Otherwise, a column must be added to the table.

```
ALTER TABLE INCUMBENTS ADD row_id NUMERIC(5,0) IDENTITY
```

5.8.5   Oracle8 Server

"Forever" in Oracle8 Server is 23:59:59 December 31, 4712 C.E. This avoids the year 2000 problem and the year 3000 problem, and even the year 4000 problem.

Oracle8 Server does not support assertions nor complex CHECK constraints; triggers must be used to implement these. As an example, CF-5.8 may be implemented as the following Oracle trigger.

Code Fragment 5.31   (SSN, PCN) **is a sequenced primary key for** INCUMBENTS, **in Oracle.**

```
CREATE OR REPLACE TRIGGER seq_primary_key
AFTER INSERT OR UPDATE ON INCUMBENTS
DECLARE
  valid INTEGER;
BEGIN
  SELECT 1
  INTO valid
  FROM DUAL
  WHERE NOT EXISTS ( SELECT *
      FROM INCUMBENTS I1 , INCUMBENTS I2
      WHERE I1.SSN = I2.SSN
        AND I1.PCN = I2.PCN
        AND I1.START_DATE < I2.END_DATE
        AND I2.START_DATE < I1.END_DATE
        AND I1.rowid <> I2.rowid )
    AND NOT EXISTS ( SELECT *
      FROM  INCUMBENTS I1
      WHERE I1.SSN IS NULL OR I1.PCN IS NULL);

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR ( -20001 ,
      'SSN and PCN are sequenced primary keys' );

END;
```

In this trigger, if the WHERE clause is not satisfied, then the exception will be raised, causing the transaction to abort. (`DUAL` is a dummy system table provided by Oracle8 Server for exactly this kind of situation.)

This example also illustrates the use of Oracle8 Server's `rowid` facility, which eliminates the need of COUNT (which appeared in the original CF-5.8) to ensure no duplicates.

Another relevant limitation is that there can be only one INSERT/UPDATE trigger per table in Oracle8 Server, so multiple CHECK constraints or ASSERTIONs on a table must be merged into a single trigger.

5.8.6   UniSQL

UniSQL does not support ASSERTIONs; these must be implemented as triggers. As an example, CF-5.8 may be implemented as the following UniSQL trigger.

Code Fragment 5.32   (SSN, PCN) **is a sequenced primary key for** INCUMBENTS**, in UniSQL.**

```
CREATE TRIGGER seq_primary_key
BEFORE COMMIT
IF (EXISTS ( SELECT I1.SSN
                FROM Incumbents AS I1
                WHERE 1 < (SELECT COUNT(SSN)
                              FROM INCUMBENTS AS I2
                              WHERE I1.SSN = I2.SSN AND I1.PCN = I2.PCN
                                AND I1.StartDate < I2.EndDate
                                AND I2.StartDate < I1.EndDate))
   OR EXISTS ( SELECT *
                FROM INCUMBENTS AS I
                WHERE I.SSN IS NULL OR I.PCN IS NULL)
   )
EXECUTE REJECT;
```

5.8.7   CD-ROM Materials

The CD-ROM contains all the code fragments in this chapter in IBM DB2 UDB, Microsoft Access 2000, Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL.

5.9 SUMMARY

A *valid-time state table* retains the history of the modeled reality. A table is rendered temporal by appending one or more timestamp columns, to specify the period of validity of each row. As SQL doesn't (yet) provide a period data type, generally two datetime columns are appended, indicating the start and end of the period. The representation can be closed-closed or closed-open, with the latter preferred.

The end time for current data should be the logical value "now." SQL supports CURRENT_DATE in queries, but not as a stored value. Possibilities include using a distinguished value (UIS uses 1860-01-01 and 3000-01-01) or the value NULL. The best approach is to use a value approximating "forever." In Section 7.5 we will see another approach that obviates the need to store "now."

The original primary key of the table is not the primary key of the temporal table. Unfortunately, adding the start column, or the end column, or both, generally does not suffice to define the primary key. The central notion of a key is the absence of duplicates. There are four kinds of uniqueness constraints:

1. Current: No two currently valid rows have the same value for the key columns.
2. Value-equivalent: No two rows have the same value for their nontimestamp columns.
3. Sequenced: No two rows have the same value for the key columns at any instant.
4. Nonsequenced: No two rows have the same value for all their columns.

The database designer should first determine which uniqueness constraint is required by the application. Then the appropriate code fragment can be used as a template. Value-equivalent and nonsequenced uniqueness can be specified using the UNIQUE construct. An SQL constraint or assertion is required to specify current and sequenced constraints. (In the restricted case where future information will never be stored, adding the end column suffices for a current primary key.)

A sequenced primary key is the natural analog of a primary key on a nontemporal table and is what is usually desired. If the application only modifies the current state, then column(s) that comprise a current primary key will also satisfy the sequenced primary key constraint. In that case, the end time should be included in the constraint (see CF-5.13). If any application will store future or past data, then the general constraint for sequenced primary keys (CF-5.8) or sequenced uniqueness (CF-5.14) is required.

Table 5.4 Referential integrity code fragments.

		Nontemporal	*Referenced Table*		
			Temporal		
			Current	Nonsequenced	Sequenced
Referencing	Nontemporal	5.19	5.24	5.19	5.20 or 5.19
Table	Temporal	5.19	5.20	5.19	5.21 or 5.22 + 5.23

How to express a referential integrity constraint depends on whether the referencing and referenced tables are temporal tables. Six code fragments were given to illustrate the possibilities. Table 5.4 identifies which code fragment applies in each situation.

In many cases, a simple constraint suffices. However, when both tables are temporal, the most natural referential constraint is a sequenced one, where a complex assertion (CF-5.21) or pair of assertions (CF-5.22, CF-5.23) is required. All such constraints should be specified as `DEFERRABLE INITIALLY DEFERRED`.

5.10   READINGS

Ensor and Stevenson [32] recommend storing a distant date for *now*, as do we in Section 5.4. They recommend that the end date column be used for a primary key. As we saw in Section 5.5, this works only when future data will never be inserted into the table, and then, only to enforce a current primary key. They also recommend that the end date column be added for referential integrity, which has the same problems as using this approach with the primary key. They also consider the related problem of avoiding gaps in the history of an entity. If gaps cannot occur, then referential integrity reduces to ensuring that the combined period of validity for the entity in the referenced table (identified by the foreign key) contains the period of validity of the row that references it (see CF-5.23).

Clifford et al. show how CURRENT_DATE can be stored directly as a column value, thereby providing support for "now" [23]. This approach requires a few changes to the underlying DBMS.

Böhlen differentiates between *intrastate* integrity constraints, which "enforce the consistency of (all) snapshots of a valid time database," and *interstate* integrity constraints, which "relate and restrict arbitrary snapshots of a valid time database," [11]. (A *snapshot* of a valid-time table at a specific time instant consists of the rows that were valid at that time.) A sequenced constraint is thus an intrastate constraint, and a nonsequenced constraint is thus an interstate constraint. We discuss Böhlen's taxonomy in more detail on page 341.

Barnert et al. provide an analysis of the checking required during various kinds of modifications to ensure sequenced referential integrity [4].

David Landes tells the exciting story of Needham's discovery of Su Song's clepsydra [65].

OVERVIEW

With the history stored in a valid-time state table, the most prevalent queries are still of the form "What is true now?" Such *current* queries are easy to convert into SQL. A related kind of query is, "What is true at a point in time other than now?" These *time-slice* queries are also easy to convert.

For each current query, for example, "Who makes more than $50,000 annually?" there is an analogous *sequenced* query that asks for the history, for example, "Who makes or has made more than $50,000 annually, and when?" Converting such queries to SQL can be challenging.

Querying State Tables

I n the previous chapter, we saw that there are three kinds of constraints you can specify for a temporal table: current, sequenced, and nonsequenced, with nonsequenced constraints the easiest to specify yet the least useful. A similar situation exists for queries and modifications applied to temporal tables.

Here we examine common queries over temporal tables, from extracting the current state, to extracting previous states, to evaluating several kinds of sequenced queries. In the following chapter, we then turn to modifications, examining the three variants there. We assume that the `INCUMBENTS` and `SAL_HISTORY` tables are valid-time state tables.

6.1   EXTRACTING THE CURRENT STATE

Executing a query on the current state of a temporal table requires an additional predicate.

Queries on the original table (before it was timestamped to render it as a valid-time table) correspond to queries that extract the current state of the valid-time table. To determine Bob's (current) position, we take the original query (CF-5.2) and add a predicate to the WHERE clause.

Code Fragment 6.1   **What is Bob's current position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
    AND END_DATE = DATE '3000-01-01'
```

Note that only one of the tables used in this query is temporal: INCUMBENTS. Hence the test for the particular date need only be applied to that table. We could use the same approach when "now" is represented with NULL.

The query on a table using the end of time ("forever") to represent "now" is more general, as it will work even after the year 3000, thus avoiding the Y3K problem.

Code Fragment 6.2   **What is Bob's current position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
    AND START_DATE <= CURRENT_DATE
      AND CURRENT_DATE < END_DATE
```

The last two lines of the predicate are equivalent to requiring that the current date overlap the period of validity. Note that BETWEEN cannot be used here, because it would permit the END_DATE to equal CURRENT_TIMESTAMP, which is incorrect because we are using a closed-open representation for period timestamps.

Current joins over two temporal tables are not that much harder. Joins over the current state of temporal tables can be handled similarly to queries involving one temporal table. To obtain the employee's (current) position and salary, the current state of both the INCUMBENTS and the SAL_HISTORY tables must be used. (We could have also gotten the current salary from the EMPLOYEES table, but that wouldn't have been as fun.)

Code Fragment 6.3   **What is Bob's current position and salary?**

```
SELECT JOB_TITLE_CODE1, AMOUNT
FROM EMPLOYEES, INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
    AND START_DATE <= CURRENT_DATE
      AND CURRENT_DATE < END_DATE
    AND HISTORY_START_DATE <= CURRENT_DATE
      AND CURRENT_DATE < HISTORY_END_DATE
    AND SAL_HISTORY.SSN = EMPLOYEES.SSN
```

This same trick works for more complex queries, including those with subqueries.

Code Fragment 6.4   **What employees currently have no position?**

```
SELECT FIRST_NAME
FROM EMPLOYEES
WHERE NOT EXISTS ( SELECT *
```

```
                    FROM INCUMBENTS
                    WHERE EMPLOYEES.SSN = INCUMBENTS.SSN
                      AND START_DATE <= CURRENT_DATE
                      AND CURRENT_DATE < END_DATE)
```

In such queries, every temporal table has to be restricted to the current state.

6.2   EXTRACTING PRIOR STATES

The queries just discussed are termed *current time-slice queries*. More accurately, the class is termed a *current valid time-slice query*, as such queries select the state valid at a particular time. Time-slice queries need not be restricted to the current state. A common query requests information valid at the beginning of the year.

Code Fragment 6.5   **What was Bob's position at the beginning of 1997?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
     AND EMPLOYEES.SSN = INCUMBENTS.SSN
     AND INCUMBENTS.PCN = POSITIONS.PCN
     AND START_DATE <= DATE '1997-01-01'
       AND DATE '1997-01-01' < END_DATE
```

If "now" is represented with a particular value, we have to augment the predicate, replacing the last line with

```
AND (DATE '1997-01-01' <= END_DATE OR END_DATE = DATE '1860-01-01')
```

Time-slice queries, over a previous state, require an additional predicate for each temporal table.

Using NULL as a proxy for "now" is handled analogously in such queries:

```
AND (DATE '1997-01-01' <= END_DATE OR END_DATE IS NULL)
```

For queries involving multiple temporal tables, the WHERE clause most closely associated with the FROM clause that defines a correlation name over a temporal table should be augmented as shown above to select the previous state of that table.

6.3   SEQUENCED QUERIES

The above queries take time-varying tables and extract a state at a particular point in time. Once that state is available, it can be manipulated conventionally.

We now consider queries in which the resulting table is a valid-time table. The query will be over one or more temporal tables and produce a temporal result. Here we consider sequenced variants of basic operations: selection, projection, union, sorting, join, difference, and duplicate elimination.

Sequenced selection is particularly easy: no change is necessary.

Code Fragment 6.6   **Who makes or has made more than $50,000 annually?**

```
SELECT *
FROM SAL_HISTORY
WHERE AMOUNT > 50000
```

In this code fragment we focus on selection by using a target list of '*', which will return all the columns, including the desired timestamp columns. This is in contrast to a current query, which should return a snapshot state, without timestamp columns (compare with CF-6.3 and CF-6.4), or with a prior valid time-slice query (e.g, CF-6.5, which also doesn't include the timestamp columns).

A selection (a predicate over a nontimestamp column) is a sequenced selection on a temporal table.

Sequenced projection is also easy: simply include the timestamp column(s) in the select list. The following query performs a projection on SSN.

Code Fragment 6.7   **List the social security numbers of current and past employees.**

```
SELECT SSN, HISTORY_START_DATE, HISTORY_END_DATE
FROM SAL_HISTORY
```

A sequenced projection of specified columns in the SELECT clause can be effected by including the timestamp columns.

Note that duplicates resulting from the projection are retained. Adding DISTINCT will remove these duplicates, but is probably not what is desired. Section 6.5 will examine the fascinating subtleties of removing duplicates from temporal tables, including describing why adding DISTINCT is not sufficient.

Sequenced sorting requires the result to be ordered, at each point in time. This can easily be accomplished by appending the start and end times to the sort columns in the ORDER BY clause.

Code Fragment 6.8   **Sequenced sort** INCUMBENTS **on the position code (first version).**

```
SELECT *
FROM INCUMBENTS
ORDER BY PCN, START_DATE, END_DATE
```

Table 6.1 shows the result for five sample rows. For each point in time, the state at that time, taken in the same sequence as the underlying table, will be ordered by PCN.

Table 6.1 An excerpt of `INCUMBENTS`.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-10-01	1997-06-01
111223333	137112	1996-01-01	1996-10-01
111223333	341288	1995-03-01	1996-01-01
111223333	723401	1997-10-01	1998-01-01
111223333	908654	1997-06-01	1998-01-01

Table 6.2 A sorted version of Table 6.1.

SSN	PCN	START_DATE	END_DATE
111223333	341288	1995-03-01	1996-01-01
111223333	137112	1996-01-01	1996-10-01
111223333	120033	1996-10-01	1997-06-01
111223333	908654	1997-06-01	1998-01-01
111223333	723401	1997-10-01	1998-01-01

The valid time-slice over this table on November 19, 1997 will be the following, which indeed is ordered by PCN:

SSN	PCN
111223333	723401
111223333	908654

Putting the timestamp columns first,

```
ORDER BY START_DATE, END_DATE, PCN
```

will not work. Applying such an order to Table 6.1 results in Table 6.2. Now take a valid time-slice on November 19, 1997, preserving the underlying order. Two rows result,

SSN	PCN
111223333	908654
111223333	723401

which are not correctly sorted.

So, our conclusion is that appending the timestamp columns to the end of the composite sort key results in a sequenced sort. Interestingly, sequenced sorting can also be accomplished by omitting the timestamp columns.

Code Fragment 6.9   **Sequenced sort** INCUMBENTS **on the position code (second version).**

```
SELECT *
FROM INCUMBENTS
ORDER BY PCN
```

A query using ORDER BY is automatically sequenced, whether or not the timestamp columns are retained.

The result is still ordered by the sort columns at all points in time. This can be argued by contradiction. Take a valid time-slice at any point in time of the result of CF-6.9, respecting the order of that table. If the PCN of two successive rows of that time-slice are *not* ordered by PCN, then the associated rows of the result of CF-6.9 must also not be ordered by PCN, which is impossible.

The same considerations hold for sequenced union (if duplicates are retained).

Code Fragment 6.10   **Who makes or has made more than $50,000 annually or less than $10,000?**

```
SELECT *
FROM SAL_HISTORY
WHERE AMOUNT > 50000
UNION ALL
SELECT *
FROM SAL_HISTORY
WHERE AMOUNT < 10000
```

A UNION ALL over temporal tables is automatically sequenced if the timestamp columns are retained.

It is appealing that the selection, projection (without duplicate elimination), sorting, and union-all queries are all automatically sequenced, without change.

A UNION without ALL eliminates duplicates but is surprisingly difficult to express in SQL. Section 6.5 will show how to do that. But before we confront that challenging query, let's consider sequenced join queries.

6.3.1   Sequenced Joins

In Section 6.1, we showed how you can produce the join of the current states of two valid-time tables (see CF-6.3). A more challenging query is to perform the join itself in a temporal fashion. What is desired here is to combine the history from the two tables, termed a *sequenced join*.

As an example, to determine the salary and position history for each employee, we must determine, for each point in time, the employee's salary and position.

Mean Solar Time

One problem with true solar time (see page 95) is that the motion of the sun is not uniform because the orbit of the earth is an ellipse rather than a circle. Sometimes the earth moves faster, and sometimes it moves slower, with the result that some days are slightly shorter or longer than others. This variability is vexing to those who use mechanical clocks, which when working well tick off hours that are very similar in duration. *Mean solar time*, or *mean time*, is a calculated time, determined by averaging true solar time over the year.

The difference between mean solar time and true solar time, termed the *equation of time*, peaks at +14 minutes in February, and is smallest at −16 minutes at the beginning of November (see Figure 6.1). To calibrate your watch, take the true time indicated by a sundial and add the value for the equation of time for the current day. Without this correction, a sundial is absolutely accurate only four times a year—around April 16, June 14, September 1, and December 25.
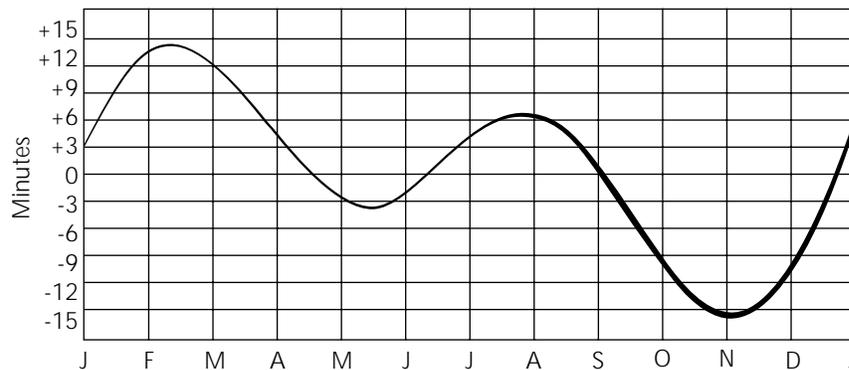
Figure 6.1 The equation of time. (Redrawn from *Sundials: History, Theory, and Practice* by René R.J. Rohr. Dover Publications, 1996.)

The salary comes from `SAL_HISTORY.AMOUNT`, and the position is available in `INCUMBENTS.PCN`. However, to do this on a point-by-point basis would be extremely inefficient, as well as wasteful, because the salary and position remain unchanged for many consecutive days. So we will instead compute the history using the periods themselves.

We initially assume that there are no duplicate rows in either of the underlying temporal tables. This is probably the case for the `SAL_HISTORY` table; it is doubtful that an employee has two salaries at one time. The fact that an employee can have multiple positions (e.g., a department head who is also a faculty member) is accommodated via the position control number. The `POSITIONS` table maps each `PCN` to `JOB_TITLE_CODE1` through `JOB_TITLE_CODE4`; indeed, this is the primary rationale

SAL_HISTORY

INCUMBENTS

Result
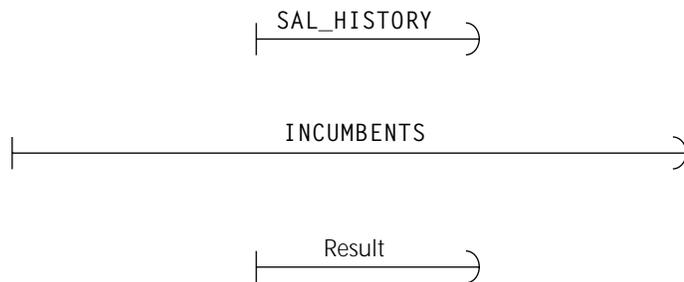
Figure 6.2 First case of a sequenced join.

SAL_HISTORY

INCUMBENTS

Result
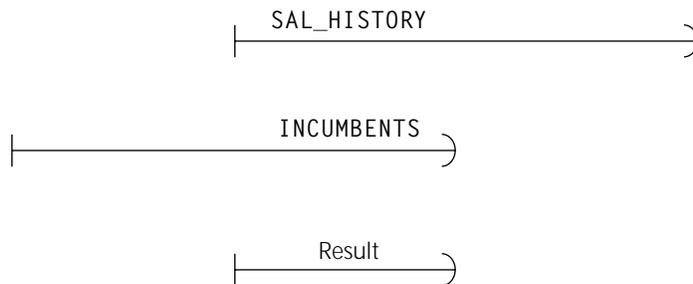
Figure 6.3 Second case of a sequenced join.

for that table. On page 112 we listed only the JOB_TITLE_CODE1 column; we now mention these other three columns to make the point that no employee has two PCN values at the same time in the POSITIONS table. In summary, in the case of a sequenced join between the SAL_HISTORY and POSITIONS tables, there are no duplicate rows to contend with.

Using SQL, the query must do a case analysis of how the period of validity of each row of SAL_HISTORY overlaps the period of validity of each row of INCUMBENTS; there are four possible cases.

A sequenced join requires four SELECT statements and complex inequality predicates.

In the first case, the period associated with the SAL_HISTORY row is entirely contained in the period associated with the INCUMBENTS row. 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 S.HISTORY_START_DATE to S.HISTORY_END_DATE. We illustrate this case in Figure 6.2, with the right end emphasizing the closed-open representation. In the second case, neither period contains the other (shown in Figure 6.3). The other cases similarly identify the overlap of the two periods.

Code Fragment 6.11   **Provide the salary and position history for all employees.**

```
SELECT S.SSN, AMOUNT, PCN,
       S.HISTORY_START_DATE, S.HISTORY_END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.START_DATE <= S.HISTORY_START_DATE
  AND S.HISTORY_END_DATE <= INCUMBENTS.END_DATE
UNION ALL
SELECT S.SSN, AMOUNT, PCN,
       S.HISTORY_START_DATE, INCUMBENTS.END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
  AND S.HISTORY_START_DATE >= INCUMBENTS.START_DATE
  AND INCUMBENTS.END_DATE < S.HISTORY.END_DATE
  AND S.HISTORY_START_DATE < INCUMBENTS.END_DATE
UNION ALL
SELECT S.SSN, AMOUNT, PCN,
       INCUMBENTS.START_DATE, S.HISTORY_END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.START_DATE > S.HISTORY_START_DATE
  AND S.HISTORY.END_DATE <= INCUMBENTS.END_DATE
  AND INCUMBENTS.START_DATE < S.HISTORY_END_DATE
UNION ALL
SELECT S.SSN, AMOUNT, PCN,
       INCUMBENTS.START_DATE, INCUMBENTS.END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.START_DATE > S.HISTORY_START_DATE
  AND INCUMBENTS.END_DATE < S.HISTORY_END_DATE
```

When a sequenced join is applied to the INCUMBENTS excerpt in Table 6.1 and the SAL_HISTORY excerpt in Table 6.3, Table 6.4 results. The first row, for example, results from the intersection of the period [1995-03-01 – 1996-01-01) from the INCUMBENTS table and [1995-03-01 – 1996-05-01) from the SAL_HISTORY table.

As an alternative way of viewing the result, consider the date January 1, 1997. One row in the INCUMBENTS table is valid on that day, with a PCN of 120033. One row from the SAL_HISTORY table is also valid on that date, with an amount of 16.25. This date thus appears once in the result, in the period of validity of row four.

This query requires care to get the 10 inequalities and the four select lists correct. The cases are designed to partition the possible interactions between the SAL_HISTORY and INCUMBENTS rows. Specifically, the first case covers *s finishes i* ∨ *s during i* ∨ *s starts i* ∨ *s equals i*, the second case *s overlaps* $^{-1}$ *i* ∨ *s starts* $^{-1}$ *i*, the third case *s overlaps i* ∨ *s finishes* $^{-1}$ *i*, and the last case *s during* $^{-1}$ *i* (see

Table 6.3 An excerpt from the `SAL_HISTORY` **valid-time state table.**

SSN	AMOUNT	HISTORY_START_DATE	HISTORY_END_DATE
111223333	15.75	1995-03-01	1996-05-01
111223333	16.25	1996-05-01	1997-06-01
111223333	17.00	1997-06-01	1998-01-01

Table 6.4 Result of the sequenced join.

SSN	AMOUNT	PCN	START_DATE	END_DATE
111223333	15.75	341288	1995-03-01	1996-01-01
111223333	15.75	137112	1996-01-01	1996-05-01
111223333	16.25	137112	1996-05-01	1996-10-01
111223333	16.25	120033	1996-10-01	1997-06-01
111223333	17.00	908654	1997-06-01	1998-01-01
111223333	17.00	723401	1997-10-01	1998-01-01

page 92 for an illustration of these operators). Because none of these relationships is covered by more than one case, no duplicates will be generated. For this reason, we use UNION ALL, which is generally more efficient than UNION, which does a lot of work to remove the (nonoccurring) duplicates.

The above code fragments assume that the underlying temporal tables contain no duplicates. If that assumption does not hold, we have three choices:

1. Retain duplicates in the result.
2. Use CF-6.11, replacing UNION ALL with UNION, and eliminate the duplicates after the join, as discussed in the next section.
3. Eliminate duplicates before the join, then use CF-6.11.

The SQL-92 CASE expression allows this query to be written as a single statement, thereby avoiding the complexities of UNION versus UNION ALL.

Code Fragment 6.12 **Provide the salary and position history for all employees, using** `CASE`**.**

```
SELECT S.SSN, AMOUNT, PCN,
     CASE
       WHEN S.HISTORY_START_DATE > INCUMBENTS.START_DATE
         THEN S.HISTORY_START_DATE
       ELSE INCUMBENTS.START_DATE
     END,
     CASE
       WHEN S.HISTORY_END_DATE > INCUMBENTS.END_DATE
         THEN INCUMBENTS.END_DATE
```

```
                   ELSE S.HISTORY_END_DATE
              END
       FROM SAL_HISTORY AS S, INCUMBENTS
       WHERE S.SSN = INCUMBENTS.SSN
          AND (CASE
                 WHEN S.HISTORY_START_DATE > INCUMBENTS.START_DATE
                    THEN S.HISTORY_START_DATE
                    ELSE INCUMBENTS.START_DATE
               END) <
               (CASE
                 WHEN S.HISTORY_END_DATE > INCUMBENTS.END_DATE
                    THEN INCUMBENTS.END_DATE
                    ELSE S.HISTORY_END_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 we will see later, the *first_instant* and *last_instant* functions are available (under different names) from some vendors as SQL extensions. They can also be implemented as SQL/PSM (persistent stored module) FUNCTIONs.

Code Fragment 6.13   **Define a** `first_instant` **function.**

```
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 be 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.

Code Fragment 6.14   **Provide the salary and position history for all employees, using** `first_instant` **and** `last_instant`**.**

```
SELECT S.SSN, AMOUNT, PCN,
       last_instant(S.HISTORY_START_DATE, INCUMBENTS.START_DATE),
       first_instant(S.HISTORY_END_DATE, INCUMBENTS.END_DATE)
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
   AND last_instant(S.HISTORY_START_DATE, INCUMBENTS.START_DATE)
       < first_instant(S.HISTORY_END_DATE, INCUMBENTS.END_DATE)
```

6.3.2   Sequenced EXCEPT

A join of two tables repeatedly takes a row from each and applies a predicate to both to determine whether the rows contribute to the result. The NOT EXISTS construct of SQL is similar, with the critical distinction that the row from the second table must not exist. This distinction renders the sequenced NOT EXISTS more challenging than a sequenced join.

As before, let's start with a nontemporal query and convert it to its sequenced analog. Assume that employees have multiple positions, as an example, a department head (a PCN of 455332) who is also a professor (a PCN of 821197). We wish to identify the exceptions, that is, the department heads who are not also professors.

Code Fragment 6.15   **List the employees who are department heads but are not also professors (nontemporal version).**

```
SELECT SSN
FROM INCUMBENTS AS I1
WHERE PCN = 455332
  AND NOT EXISTS (SELECT *
                    FROM INCUMBENTS AS I2
                    WHERE I2.SSN = I1.SSN
                      AND I2.PCN = 821197)
```

For the sequenced version, we wish to identify *when* the department heads were not professors. Employees are promoted to department head, remain a few years, then someone else becomes department head. Somewhat independently, employees are promoted to the rank of professor; very occasionally, they are demoted to associate professor. Hence, the two SELECT clauses in the above code fragment evaluate to time-varying tables; the NOT EXISTS must perform the relational difference conceptually at each point in time. The result will be a valid-time state table; the challenge is to restate the query so that it computes the timestamps of this table correctly.

As an aside, the SQL NOT EXISTS is closely tied to the EXCEPT construct:

Code Fragment 6.16   **List the employees who are department heads but are not also professors (an equivalent nontemporal version).**

```
SELECT SSN
FROM INCUMBENTS
WHERE PCN = 455332
EXCEPT
SELECT SSN
FROM INCUMBENTS
WHERE PCN = 821197
```

This is the relational difference operator, expressed in SQL. The transformation we give for NOT EXISTS is also applicable for EXCEPT.

NOT EXISTS is also closely related to NOT IN:

Code Fragment 6.17   **List the employees who are department heads but are not also professors (an equivalent nontemporal version).**

```
SELECT SSN
FROM INCUMBENTS AS I1
WHERE PCN = 455332
  AND 821197 NOT IN (SELECT PCN
                     FROM INCUMBENTS AS I2
                     WHERE I1.SSN = I2.SSN)
```

A sequenced NOT EXISTS (EXCEPT, NOT IN) requires four SELECT statements, each with a nested NOT EXISTS.

Returning to the EXCEPT query (CF-6.15), let's focus on a row output by the sequenced version of this query. As illustrated in Figure 6.4, there are four ways in which an output row could result. In the first case, the employee starts being a department head when he or she is still an associate professor; the output row ends when the person is promoted to professor. In the second case, the department head, who was a professor, was for some reason demoted. The third case is a combination of the first two: the department head was demoted, then subsequently promoted back to professor. In all three cases, there must not exist another row with a rank of professor that overlaps the resulting row (otherwise, the NOT EXISTS would not hold). In the last case, the department head was never a professor during his or her tenure, so the entire period of validity should be returned.

These cases allow us to compute the delimiting timestamps of the resulting row. Each of these cases requires a separate SELECT statement in the sequenced version.

Code Fragment 6.18   **List the employees who are or were department heads but were not also professors (sequenced version).**

```
SELECT I1.SSN, I1.START_DATE, I3.START_DATE AS END_DATE
FROM INCUMBENTS AS I1, INCUMBENTS AS I3
WHERE I1.PCN = 455332
  AND I3.PCN = 821197
  AND I1.SSN = I3.SSN
  AND NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I4
                   WHERE I4.SSN = I1.SSN
                     AND I4.PCN = 821197
                     AND I1.START_DATE < I4.END_DATE
                     AND I4.START_DATE < I3.START_DATE)
```

```
UNION
SELECT I1.SSN, I2.END_DATE AS START_DATE, I1.START_DATE AS END_DATE
FROM INCUMBENTS AS I1, INCUMBENTS AS I2
WHERE I1.PCN = 455332
  AND I2.PCN = 821197
  AND I1.SSN = I2.SSN
  AND NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I4
                   WHERE I4.SSN = I1.SSN
                     AND I4.PCN = 821197
                     AND I2.END_DATE < I4.END_DATE
                     AND I4.START_DATE < I1.END_DATE)
UNION
SELECT I1.SSN, I1.END_DATE AS START_DATE, I3.START_DATE AS END_DATE
FROM INCUMBENTS AS I1, INCUMBENTS AS I2, INCUMBENTS AS I3
WHERE I1.PCN = 455332
  AND I2.PCN = 821197 AND I3.PCN = 821197
  AND I1.SSN = I2.SSN AND I1.SSN = I3.SSN
  AND NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I4
                   WHERE I4.SSN = I1.SSN
                     AND I4.PCN = 821197
                     AND I2.END_DATE < I4.END_DATE
                     AND I4.START_DATE < I3.START_DATE)
UNION
SELECT SSN, START_DATE, END_DATE
FROM INCUMBENTS AS I1
WHERE PCN = 455332
  AND NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I4
                   WHERE I4.SSN = I1.SSN
                     AND I4.PCN = 821197
                     AND I1.START_DATE < I4.END_DATE
                     AND I4.START_DATE < I1.END_DATE)
```

The cases are quite similar; they differ in the specific timestamps chosen in the target lists and in the timestamps used in the NOT EXISTS subqueries.

6.4   NONSEQUENCED VARIANTS

We have seen current and sequenced versions of selection, projection, join, union, and sorting. As with the integrity constraints discussed in the previous chapter, nonsequenced versions of these operators on temporal tables are straightforward. Such queries ignore the time-varying nature of the tables.

Output row

Case 1   Department head

                                          Professor

Case 2   Department head

          Professor

Case 3   Department head

          Professor   Professor

Case 4   Department head
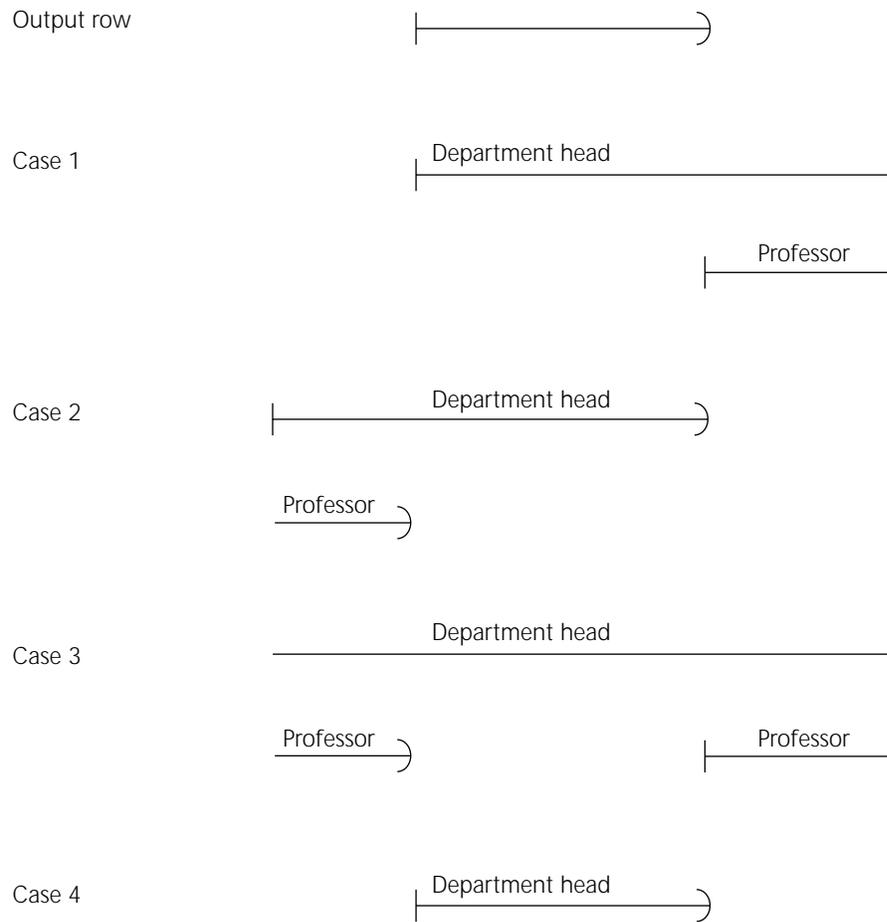
Figure 6.4 Current update cases, with the period of validity of the row shown.

Code Fragment 6.19   **List all the salaries, past and present, of employees who had been a hazardous waste specialist at some time.**

```
SELECT AMOUNT
FROM INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE INCUMBENTS.SSN = SAL_HISTORY.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
  AND JOB_TITLE_CODE1 = 20730
```

The phrases "past and present" and "at some time" indicate that the query is a nonsequenced one. The periods of validity of both INCUMBENTS and SAL_HISTORY are simply ignored in this query.

Some nonsequenced queries do examine the timestamps. However, they are evaluated all at once, rather than at a specified time (a time-slice query) or at each point in time (a sequenced query). A common example is determining when change occurred by observing consecutive periods that signify that change.

Code Fragment 6.20 **When did employees receive raises?**

```
SELECT S2.SSN, S2.HISTORY_START_DATE AS RAISE_DATE
FROM SAL_HISTORY AS S1, SAL_HISTORY AS S2
WHERE S2.AMOUNT > S1.AMOUNT
  AND S1.SSN = S2.SSN
  AND S1.HISTORY_END_DATE = S2.HISTORY_START_DATE
```

A nonsequenced query considers the timestamp columns as just additional columns.

This is not a sequenced query because each result row is derived from information from two different times. The query does not view the underlying table as a sequence of states; rather, it views the underlying table as one with additional timestamp columns that can be used in the query. When evaluated on the SAL_HISTORY excerpt in Table 6.3, the following result is returned:

SSN	RAISE_DATE
111223333	1996-05-01
111223333	1997-06-01

Sequenced, current, and nonsequenced variants also exist for other operators. As shown in Section 6.1, any conventional query can be converted into a current query by adding a predicate for each correlation name. Sequenced variants of many operations were given in Sections 6.3 and 6.3.1. For selection, projection, union, and sorting, the sequenced and nonsequenced variants are identical. For joins, difference (EXCEPT, NOT EXISTS, NOT IN), aggregates, and subqueries, the sequenced and nonsequenced variants are quite different. For duplicate elimination, the analysis is more subtle; we now turn to this intriguing topic.

6.5 ELIMINATING DUPLICATES

Duplicates may be present in the underlying table or may arise in the result of operations such as union, projection, and some implementations of sequenced join.

In some cases, the presence of duplicates will change the result of the query, as in the COUNT aggregate.

One approach to duplicate elimination is to use the SQL facilities directly. However, that didn't work for the analogous situation of defining a primary key, which as a side effect prevents duplicates. In Section 5.3 we saw that the primary key must be changed when valid-time support is added to a table. We also saw that it wasn't sufficient to simply add either the start time or the end time, or both, to the primary key. Section 5.5 amplified on this, by showing that there were four different kinds of uniqueness constraints.

Such considerations also come into play in duplicate elimination.

6.5.1   Easy Duplicate Elimination

SQL is perfectly adequate to remove some duplicate variants.

Code Fragment 6.21   **Remove nonsequenced duplicates from** INCUMBENTS**.**

```
SELECT DISTINCT *
FROM INCUMBENTS
```

Value equivalence, which is more useful, is handled similarly, except that the timestamps are not included.

Code Fragment 6.22   **Remove value-equivalent rows from** INCUMBENTS**.**

```
SELECT DISTINCT SSN, PCN
FROM INCUMBENTS
```

Current duplicates involve just a little more effort. Recall that a row is currently valid in the INCUMBENTS table if its END_DATE is the special value DATE '3000-01-01'.

Code Fragment 6.23   **Remove current duplicates from** INCUMBENTS**.**

```
SELECT DISTINCT SSN, PCN
FROM INCUMBENTS
WHERE END_DATE = DATE '3000-01-01'
```

Removing sequenced duplicates turns out to be quite difficult. Before we show how to do this, we discuss coalescing, which is closely related to duplicate elimination.

6.5.2   Coalescing

Consider again Table 5.2, copied here as Table 6.5. All five rows have the same values for the SSN and PCN columns, and hence are value-equivalent. The second and third rows also have the same values for the START_DATE and END_DATE columns, making them nonsequenced duplicates.

Table 6.5 A table containing several kinds of duplicates.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-10-01	1998-01-01
111223333	120033	1997-12-01	1998-01-01

Coalescing reduces the number of rows by merging the periods of validity of value-equivalent rows.

Now consider the third and fourth rows. As just observed, these rows are value-equivalent, yet are neither sequenced duplicates (the periods of validity do not overlap) nor nonsequenced duplicates (their periods of validity are not identical) nor current duplicates. We focus on these rows because their periods of validity meet (that is, the END_DATE of the third row equals the START_DATE of the fourth row), and so the two rows could be merged. *Coalescing* combines value-equivalent rows into a single row with a combined period of validity. Coalescing would merge these two rows into a single row valid from April 1, 1996 to December 31, 1997.

Duplicate elimination and coalescing are somewhat orthogonal because in most cases eliminating duplicates (of whatever variant) does not result in a coalesced table, and coalescing can be done without removing duplicates.

Coalescing may remove or retain sequenced duplicates.

There are two variants of coalescing: removing and retaining (sequenced) duplicates. After the former is done, there will be no sequenced, current, or nonsequenced duplicates. The latter retains the sequenced duplicates present at each point in time. It changes the timestamps, eliminating value-equivalent rows whose periods of validity meet. The objective is to pare down the table to the minimum number of rows necessary to represent the number of duplicates present at each point in time.

To illustrate this, let's apply coalescing and duplicate elimination separately on Table 6.5.

Coalescing Table 6.5 while removing duplicates results in the following table, merging the five rows into one:

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1998-01-01

Note that the resulting table happens to have no duplicates of any kind, though in general value-equivalent rows are possible (such rows will not overlap or meet).

Coalescing Table 6.5 while *retaining* duplicates results instead in Table 6.6.

Table 6.6 Retaining duplicates while coalescing.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-04-01	1998-01-01
111223333	120033	1997-12-01	1998-01-01

Table 6.7 Another way to retain duplicates while coalescing.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1998-01-01
111223333	120033	1996-04-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1997-12-01	1998-01-01

The original table and the coalesced table are *snapshot-equivalent*: all of their snapshots, resulting from a time-slice at a single instant, are identical. Since duplicates are retained, the snapshot of the result at each instant will have the same number of duplicates as the snapshot of the original table at that instant.

Consider a time-slice at January 3, 1996, of Table 6.5. The snapshot contains a single row, (111223333, 120033). The time-slice at that day on Table 6.6 also contains that single row. Now consider May 25, 1996. The time-slice on that date of Table 6.5 contains three rows (you should verify this), as does the time-slice of Table 6.6.

In the original table, Table 6.5, and in the coalesced table, Table 6.6, there is one row in the snapshots valid from January 1, 1996 to March 30, 1996, then three rows valid until May 31, 1996, then two rows valid until September 30, 1996, then one row valid until November 30, 1997, then two rows valid until December 31, 1997. These two tables are snapshot-equivalent. Informally, they model the same information, the same history of the enterprise. The only difference: Table 6.6 has fewer rows and thus requires less space on disk.

When duplicates are retained, there will be at least as many rows in the resulting table as the maximum number of rows valid at any one time. In this case, the result does not happen to have any nonsequenced duplicates, though such duplicates are possible.

When coalescing while removing sequenced duplicates, only one resulting table is possible. However, when coalescing while retaining duplicates, the resulting table is not always unique. Table 6.7 retains the duplicates from Table 6.5 and also contains the minimum number of rows possible: four.

Table 6.6 resulted from merging pairs of value-equivalent rows that met. In Table 6.7, the timestamps experience a more radical adjustment. The one remaining question is, How do you guarantee that the coalesced table is minimized and thus contains the smallest number of rows possible? The answer is simple. It turns out that (repeatedly) merging pairs of value-equivalent rows that meet will *always* result in a coalesced table with the minimum number of rows.

To summarize, we have described three operations that minimize the number of rows: remove sequenced duplicates, coalesce while removing sequenced duplicates, and coalesce while retaining duplicates. If you are going to eliminate duplicates in the first place, it is just as easy to coalesce at the same time.

6.5.3   Coalescing While Removing Duplicates

One intuition behind coalescing is that we identify those time periods with the same SSN and PCN values that overlap or are adjacent, and merge those periods by extending the earlier period. This process is repeated until maximal periods are constructed. The nonmaximal periods are then removed.

Coalescing while removing duplicates

Repeat, until no rows are updated,

Change a row's end date to that of the value-equivalent row that overlaps it that extends the furthest in the future.

Remove those rows whose period of validity is entirely contained in that of a value-equivalent row.

Remove those rows that are nonsequenced duplicates.

Code Fragment 6.24   **Coalesce** INCUMBENTS **while removing duplicates (in PSM).**

```
PROCEDURE Do_COALESCE () LANGUAGE SQL;
  CREATE TABLE Temp(SSN CHAR(9), PCN INT,
      START_DATE DATE, END_DATE DATE);
  INSERT INTO Temp
  SELECT *
  FROM INCUMBENTS;

      -- Extend row's end date
  BEGIN
    DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN END;
```

```
            LOOP
              UPDATE Temp AS T1
              SET (T1.END_DATE) = (SELECT MAX(T2.END_DATE)
                                     FROM Temp AS T2
                                     WHERE T1.SSN = T2.SSN AND T1.PCN = T2.PCN
                                       AND T1.START_DATE < T2.START_DATE
                                       AND T1.END_DATE >= T2.START_DATE
                                       AND T1.END_DATE < T2.END_DATE)
                  -- Make sure there is at least one end date to extend the row
              WHERE EXISTS (SELECT *
                              FROM Temp AS T2
                              WHERE T1.SSN = T2.SSN AND T1.PCN = T2.PCN
                                AND T1.START_DATE < T2.START_DATE
                                AND T1.END_DATE >= T2.START_DATE
                                AND T1.END_DATE < T2.END_DATE)
            END LOOP;
          END;
              -- Remove wholly contained rows
          DELETE FROM Temp AS T1
          WHERE EXISTS (SELECT *
                          FROM Temp AS T2
                          WHERE T1.SSN = T2.SSN AND T1.PCN = T2.PCN
                            AND ((T1.START_DATE > T2.START_DATE
                              AND T1.END_DATE <= T2.END_DATE)
                            OR (T1.START_DATE >= T2.START_DATE
                              AND T1.END_DATE < T2.END_DATE)));
          CREATE TABLE Temp2 (SSN CHAR(9), PCN INT,
              START_DATE DATE, END_DATE DATE);
          INSERT INTO Temp2
          SELECT DISTINCT *;
          FROM Temp;
        END;
```

The loop maximally extends a row's end date. This loop terminates when no row is updated (the exit handler accomplishes this).

To illustrate, we apply this algorithm to Table 6.5. The `Temp` table would initially contain the value-equivalent rows shown in Figure 6.5.

After the first iteration of the *repeat-until* loop, the `Temp` table would contain the rows shown in Figure 6.6. Note how the end time is extended when a value-equivalent row meets or overlaps it. After the second iteration, some periods are further extended (Figure 6.7).

The next iteration does not change any end time, and so the *repeat-until* loop is terminated. The DELETE statement removes the nonmaximal value-equivalent periods, retaining only the first one shown in Figure 6.7.

96-1   96-6

96-4   96-10

96-4   96-10

96-10   98-1

97-12 98-1
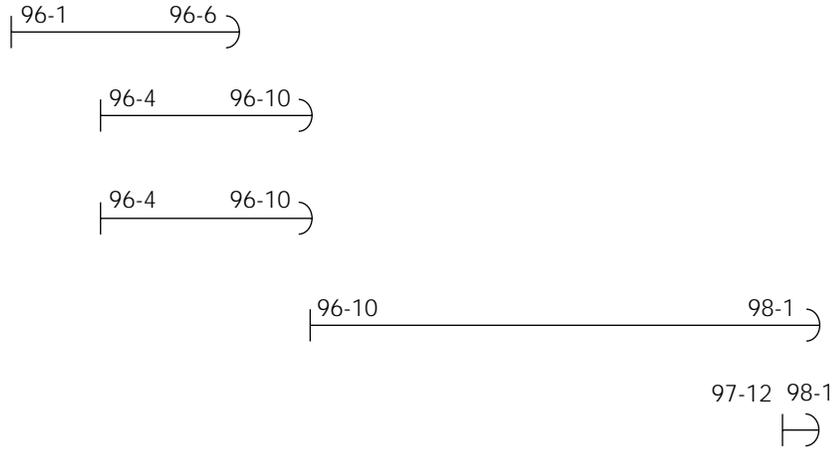
Figure 6.5 Initial Temp **table.**
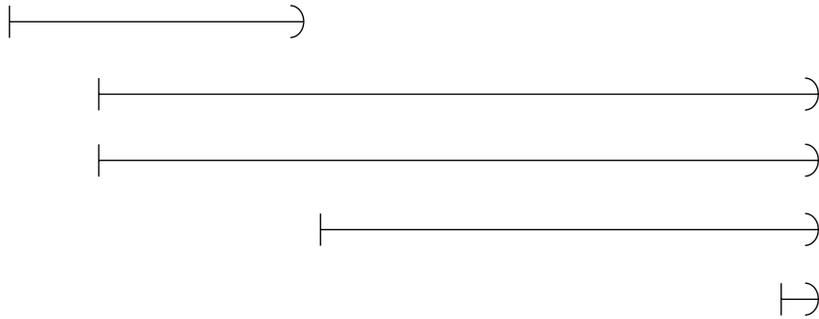
Figure 6.6 After the first iteration.
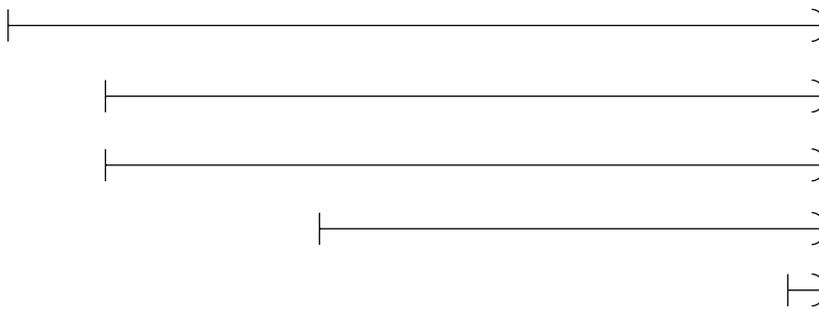
Figure 6.7 After the second iteration.

One problem with this approach is that it uses a PSM PROCEDURE. For a time, it was thought impossible to express this query completely in SQL. A solution was discovered independently by several people just a few years ago, involving complex, multiply nested NOT EXISTS subclauses.

Coalesce entirely in SQL

Select those start and end dates such that

- there are no gaps between these dates,
- no value-equivalent row overlaps the period between the selected start and end dates and has an earlier start date or a later end date.

Code Fragment 6.25   **Coalesce** INCUMBENTS **while removing duplicates (entirely in SQL).**

```
CREATE TABLE Temp(SSN CHAR(9), PCN INT,
    START_DATE DATE, END_DATE DATE)
INSERT INTO Temp
SELECT *
FROM INCUMBENTS;

SELECT DISTINCT F.SSN, F.PCN, F.START_DATE, L.END_DATE
FROM Temp AS F, Temp AS L
WHERE F.START_DATE < L.END_DATE
  AND F.SSN = L.SSN AND F.PCN = L.PCN
    -- There are no gaps between  F.END_DATE and   L.START_DATE
  AND NOT XISTS (SELECT *
                  FROM Temp AS M
                  WHERE M.SSN = F.SSN AND M.PCN = F.PCN
                    AND F.END_DATE < M.START_DATE
                    AND M.START_DATE < L.START_DATE
                    AND NOT EXISTS (SELECT *
                      FROM Temp AS T1
                      WHERE T1.SSN = F.SSN AND T1.PCN = F.PCN
                        AND T1.START_DATE < M.START_DATE
                        AND M.START_DATE <= T1.END_DATE))
    -- Can't be extended further
```

Figure 6.8 A common scenario.

```
AND NOT EXISTS (SELECT *
                FROM Temp AS T2
                WHERE T2.SSN = F.SSN AND T2.PCN = F.PCN
                  AND ((T2.START_DATE < F.START_DATE
                          AND F.START_DATE <= T2.END_DATE)
                      OR (T2.START_DATE <= L.END_DATE
                          AND L.END_DATE < T2.END_DATE)))
```

In this query, we search for two (possibly the same) value-equivalent rows (represented by the correlation names F, for *first*, and L, for *last*) defining start point F.START_DATE and end point L.END_DATE of a coalesced row. The first NOT EXISTS ensures that there are no gaps between F.END_DATE and L.START_DATE (i.e., no time points where the respective fact does not hold). This guarantees that all start points M.START_DATE between F.END_DATE and L.START_DATE of value-equivalent rows are extended (towards F.END_DATE) by a value-equivalent row, T1. This is illustrated in Figure 6.8. 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. Finally, M need not overlap L, in which case there must exist another M that does.

> Coalescing with duplicate removal can be done with a single (complex) SQL statement.

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

It is instructive to compare this code fragment with CF-5.21 on page 128. In both cases, a nested NOT EXISTS is used to detect a gap in time.

Yet another approach uses a COUNT aggregate in place of the nested NOT EXISTS.

Code Fragment 6.26   **Coalesce** INCUMBENTS **while removing duplicates (entirely in SQL, using COUNT).**

```
CREATE VIEW V1 (SSN, PCN, START_DATE, END_DATE)
AS SELECT F.SSN, F.PCN, F.START_DATE, L.END_DATE
FROM INCUMBENTS AS F, INCUMBENTS AS L, INCUMBENTS AS E
WHERE F.END_DATE <= L.END_DATE
  AND F.SSN = L.SSN AND F.SSN = E.SSN
```

```
      AND F.PCN = L.PCN AND F.PCN = E.PCN
GROUP BY F.SSN, F.PCN, F.START_DATE, L.END_DATE
HAVING COUNT(CASE
               WHEN (E.START_DATE < F.START_DATE
                       AND F.START_DATE <= E.END_DATE)
                    OR (E.START_DATE <= L.END_DATE
                          AND L.END_DATE < E.END_DATE)
                  THEN 1 END) = 0

CREATE TABLE Temp(SSN CHAR(9), PCN INT,
      START_DATE DATE, END_DATE DATE)
INSERT INTO Temp
SELECT SSN, PCN, START_DATE, MIN(END_DATE)
FROM V1
GROUP BY SSN, PCN, START_DATE
```

The correlation names in the view denote "first" (F), "last" (L), and "extends" (E). The view collects periods that are maximal, in that there is no row E that extends it either to the left (the first part of the WHEN predicate) or to the right (the second part of the WHEN predicate). COUNT = 0 is equivalent to NOT EXISTS. The WHERE predicate ensures that the period is a correct one and that we only consider rows with the appropriate SSN and PCN. The INSERT command then ensures that there are no gaps *within* the period, by selecting the minimum end date.

While this solution, at 19 lines, is somewhat shorter than CF-6.25 at 27 lines, the three-way join and the grouped aggregate may impact performance.

A fourth alternative is to use SQL only to open a sorted cursor on the table.

Coalesce via a cursor

Retain the initial row in the cursor as the *previous row*.

While the cursor returns a row:

> Output the previous row if there is a gap between it and the start of the row in the cursor.
> Make the row associated with the cursor the previous row.

Output the previous row.

Here we use Oracle's PL/SQL to express the cursor and the WHILE loop.

Code Fragment 6.27 **Coalesce** INCUMBENTS **while removing duplicates (using a cursor in Oracle PL/SQL).**

```
CREATE TABLE Temp(SSN, PCN, START_DATE, END_DATE);
DECLARE CURSOR INC_CURSOR IS
SELECT *
FROM INCUMBENTS
ORDER BY SSN, PCN, START_DATE;

  StartRow       INC_CURSOR%ROWTYPE;
  PrevRow        INC_CURSOR%ROWTYPE;
  CurrRow        INC_CURSOR%ROWTYPE;

BEGIN
  IF NOT INC_CURSOR%ISOPEN
  THEN
    OPEN INC_CURSOR;
  END IF;

  FETCH INC_CURSOR INTO PrevRow;
  StartRow := PrevRow;

  FETCH INC_CURSOR INTO CurrRow;

  WHILE INC_CURSOR%FOUND
  LOOP
    IF (StartRow.SSN <> CurrRow.SSN OR StartRow.PCN <> CurrRow.PCN)
         OR (PrevRow.END_DATE < CurrRow.START_DATE)
    THEN
      INSERT INTO Temp
      VALUES (StartRow.SSN, StartRow.PCN,
        StartRow.START_DATE, PrevRow.END_DATE);
      StartRow := CurrRow;
    END IF;
    PrevRow := CurrRow;
    FETCH INC_CURSOR INTO CurrRow;
  END LOOP;

  INSERT INTO Temp
      VALUES (StartRow.SSN, StartRow.PCN,
        StartRow.START_DATE, PrevRow.END_DATE);

  CLOSE INC_CURSOR;
END;
```

This loop extracts a row from the sorted `INCUMBENTS` table. If this row (`CurrRow`) is value-equivalent with the start row and if the current row overlaps or meets the previous row, then it should be coalesced with the previous row, which is the default action. Otherwise, the coalesced row (from `StartRow.START_DATE` to `PrevRow.END_DATE`) is added to `Temp`. Because the cursor is sorted, coalescing with duplicate removal requires but a single scan of the underlying table.

6.5.4   Coalescing While Retaining Duplicates

To coalesce while retaining duplicates, it is necessary to merge value-equivalent rows whose periods of validity meet.

Coalescing while retaining (sequenced) duplicates turns out to be conceptually simpler, yet is more difficult to implement. As mentioned in Section 6.5.2, a table is coalesced with duplicates if it has the minimum possible number of rows, while still retaining the duplicates at each point in time. As there may be many coalesced versions of a temporal table, our task is to produce one of those versions.

It should be clear that if two value-equivalent rows meet, the table is not coalesced (with duplicates) because these two rows can be merged, thereby reducing the number of rows by one. More surprising is the stronger result that a coalesced table results if all pairs of value-equivalent rows that meet (are adjacent) are merged.

Coalescing while retaining duplicates requires iterating over the table several times via a cursor. Doing so efficiently requires highly system-dependent approaches. As an example, the Oracle8 Server PL/SQL code is some 80 lines long.

6.6   IMPLEMENTATION CONSIDERATIONS

We implemented these code fragments on several DBMSs.

6.6.1   IBM DB2 Universal Database

UDB's recursive queries provide yet another way to effect coalescing with duplicate elimination.

Code Fragment 6.28   **Coalesce** `INCUMBENTS` **while removing duplicates, in DB2 UDB.**

```
WITH DTR (SSN, PCN, START_DATE, END_DATE) AS
  (SELECT SSN, PCN, START_DATE, END_DATE
   FROM INCUMBENTS
   UNION ALL
   SELECT I.SSN, I.PCN, I.START_DATE, J.END_DATE
   FROM INCUMBENTS I, DTR J
```

```
        WHERE I.SSN = J.SSN AND I.PCN = J.PCN
          AND ((I.START_DATE < J.START_DATE
                AND J.START_DATE < I.END_DATE
                AND I.END_DATE < J.END_DATE)
             OR I.END_DATE = J.START_DATE)
        UNION ALL
        SELECT I.SSN, I.PCN, I.START_DATE, J.END_DATE
        FROM INCUMBENTS J, DTR I
        WHERE I.SSN = J.SSN AND I.PCN = J.PCN
          AND ((I.START_DATE < J.START_DATE
                AND J.START_DATE < I.END_DATE
                AND I.END_DATE < J.END_DATE)
             OR I.END_DATE = J.START_DATE))
SELECT DISTINCT I.SSN, I.PCN, I.START_DATE, I.END_DATE
FROM DTR I
WHERE NOT EXISTS (SELECT 1
                  FROM DTR J
                  WHERE I.SSN = J.SSN AND I.PCN = J.PCN
                    AND ((J.START_DATE <= I.START_DATE
                          AND I.END_DATE < J.END_DATE)
                      OR (J.START_DATE < I.START_DATE
                          AND I.END_DATE <= J.END_DATE)))
```

So, what is going on here? The base part of the recursion is the initial SELECT. The second SELECT is the first recursive part, the third SELECT is the second recursive part, and the final SELECT is the main query, evaluated over the result of the recursion. The first recursive part combines the periods of value-equivalent rows where the I row starts *before* the common row (from the base part, J). The second recursive part combines the periods where the I row starts *after* the base part. The recursion keeps extending the ending time of the base tuples until they change no more. Then the main query removes those periods that are contained in another period.

The PERIOD abstract data type in IBM DB2 UDB could simplify many of the algorithms in this chapter, because the overlaps predicate can be done directly.

6.6.2   Informix–Universal Server

While Informix–Universal Server does not support SQL-92's CASE expression, it does support the user-defined procedures `first_instant` and `last_instant`, which were shown earlier as SQL/PSM functions defined in CF-6.13.

Code Fragment 6.29   **Define** `first_instant` **and** `last_instant` **in Informix.**

```
CREATE PROCEDURE first_instant (one DATE, two DATE)
```

```
    RETURNING DATE;
    IF one < two THEN RETURN one;
    ELSE RETURN two;
    END IF
END PROCEDURE;

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

A PERIOD datablade in Informix–Universal Server would simplify many algorithms in this chapter because the OVERLAPS constructor can then be done directly within the SQL statement.

6.6.3   Microsoft SQL Server

Microsoft SQL Server does not support user-defined functions, so CF-6.14 and CF-6.24 are not possible. A cursor-based approach works for coalescing.

Code Fragment 6.30   **Coalesce** INCUMBENTS **while removing duplicates, in Microsoft SQL Server.**

```
CREATE TABLE Tempo
(SSN CHAR(11),
  PCN CHAR(6),
  START_DATE DATETIME,
  END_DATE DATETIME)

DECLARE Inc_Cursor CURSOR FOR
  SELECT *
  FROM INCUMBENTS
  ORDER BY SSN, PCN, START_DATE

DECLARE @S_SSN CHAR(11)
DECLARE @S_PCN CHAR(6)
DECLARE @S_START_DATE DATETIME
DECLARE @S_END_DATE DATETIME

DECLARE @P_SSN CHAR(11)
DECLARE @P_PCN CHAR(6)
DECLARE @P_START_DATE DATETIME
DECLARE @P_END_DATE DATETIME

DECLARE @C_SSN CHAR(11)
DECLARE @C_PCN CHAR(6)
DECLARE @C_START_DATE DATETIME
```

```
DECLARE @C_END_DATE DATETIME

BEGIN
  OPEN Inc_Cursor

  FETCH NEXT FROM Inc_Cursor
    INTO @P_SSN, @P_PCN, @P_START_DATE, @P_END_DATE

  SELECT @S_SSN = @P_SSN
  SELECT @S_PCN = @P_PCN
  SELECT @S_START_DATE = @P_START_DATE
  SELECT @S_END_DATE = @P_END_DATE

  FETCH NEXT FROM Inc_Cursor
    INTO @C_SSN, @C_PCN, @C_START_DATE, @C_END_DATE

  WHILE (@@FETCH_STATUS <> -1)
  BEGIN
    IF ((@S_SSN <> @C_SSN OR @S_PCN <> @C_PCN) OR
        (@P_END_DATE < @C_START_DATE))
    BEGIN
      INSERT INTO Tempo
        VALUES (@S_SSN, @S_PCN, @S_START_DATE, @P_END_DATE)
      SELECT @S_SSN = @C_SSN
      SELECT @S_PCN = @C_PCN
      SELECT @S_START_DATE = @C_START_DATE
      SELECT @S_END_DATE = @C_END_DATE
    END
    SELECT @P_SSN = @C_SSN
    SELECT @P_PCN = @C_PCN
    SELECT @P_START_DATE = @C_START_DATE
    SELECT @P_END_DATE = @C_END_DATE

    FETCH NEXT FROM Inc_Cursor INTO
      @C_SSN, @C_PCN, @C_START_DATE, @C_END_DATE
  END

  INSERT INTO Tempo
    VALUES (@S_SSN, @S_PCN, @S_START_DATE, @P_END_DATE)

  CLOSE Inc_Cursor
END
```

6.6.4 Sybase SQLServer

Sybase SQLServer does not support user-defined functions within SQL statements, so CF-6.14 and CF-6.24 are not possible.

6.6.5 Oracle8 Server

While Oracle8 Server 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 in CF-6.13.

Code Fragment 6.31 **Provide the SSN and position history for all employees, using GREATEST and LEAST.**

```
SELECT S.SSN, AMOUNT, PCN,
       GREATEST(S.HISTORY_START_DATE, INCUMBENTS.START_DATE),
       LEAST(S.HISTORY_END_DATE, INCUMBENTS.END_DATE)
FROM SAL_HISTORY S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
  AND GREATEST(S.HISTORY_START_DATE, INCUMBENTS.START_DATE)
        < LEAST(S.HISTORY_END_DATE, INCUMBENTS.END_DATE)
```

6.6.6 UniSQL

UniSQL does not provide CURRENT_DATE, so the actual date must be supplied, via a parameter, in such queries as CF-6.1.

6.6.7 CD-ROM Materials

The CD-ROM contains the code fragments in this chapter in IBM DB2 UDB, Microsoft Access 97 and Access 2000, Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL.

6.7 SUMMARY

This chapter examined a wide variety of useful queries on valid-time state tables. We first naturally ask, What is true now? Queries over the current state, even complex ones, can be converted to apply to temporal tables with the addition of a predicate for each correlation name, requesting the overlap with "now." Queries over a single prior, or future, state also follow, with the overlap requested at a specified point in time. Such queries are termed *time-slice* queries; current queries are time-slice queries at "now."

Sequenced queries are also natural ones. For every query over the current state, such as "list the salary and position for all employees," an analogous sequenced query that asks for the history of that relationship can be specified, such as "provide the salary and position history for all employees" (CF-6.11). To convert a current query to a sequenced query requires decomposing that query into its underlying operations. Each operation is converted, then the query is stitched back together. Some basic operations are easy to convert: selection (no change is necessary), projection (simply include the timestamp columns in the SELECT list), sorting (no change is necessary), and union (no change is necessary). Sequenced join requires unioning four SELECT statements; sequenced difference (EXCEPT, NOT EXISTS) requires unioning four SELECT statements, each containing its own NOT EXISTS subquery.

Duplicate elimination is a topic unto itself. Removing current, value-equivalent, and nonsequenced duplicates is easy. Coalescing is related to duplicate elimination, in that both reduce the number of rows by eliminating "extraneous" rows. But you can coalesce while retaining (sequenced) duplicates, or you can coalesce while removing such duplicates. Both are challenging in SQL. The cursor-based approach (CF-6.27) is preferred.

Nonsequenced queries, which treat the timestamps as regular columns, are generally difficult to express in English, but relatively straightforward to express in SQL. Such queries do not have current analogs.

6.8	READINGS

SQL-92 includes an unrelated COALESCE operator that is shorthand of CASE that replaces NULL values with other values [71]. SQL's stored procedures (SQL/PSM) is now an ISO standard [45] and is thoroughly described by Melton [70].

Ensor and Stevenson discuss extracting the current state in Oracle, examining both SQL and procedural variants [32]. They conclude that the biggest advantage of the procedural approach is that it can stop when the (single) qualifying row is found.

Current queries were termed *temporally upward compatible queries* by Bair et al. [3] in their discussion of supporting legacy applications when time support was added. The notion of *value equivalence* was introduced by the author [86]. That paper also introduced the notion of *snapshot reducibility*, which relates two queries, a query q^t on a temporal table and a query q^s on a time-slice of that table. As an example, let q^s be CF-6.17 and let q^t be CF-6.18. q^t is said to be snapshot-reducible to q^s if the time-slice of the result of q^t at a time instant i is identical to the result of q^s on the time-slice of the underlying table at instant i, for all possible instants i. This is

precisely the notion of a sequenced query. The sequenced EXCEPT simulates the nontemporal EXCEPT at each instant of time.

The terms "sequenced" and "nonsequenced" first appeared in a change proposal for SQL3 [92].

Coalescing was first studied in depth by Böhlen et al. [15], though many temporal data models and temporal query languages have implicitly or explicitly assumed or provided coalescing. CF-6.25 uses an idea discovered independently by Böhlen [10] and by Rozenshtein, Abramovich, and Birger [82]. CF-6.26 uses an idea proposed by Romley [21]. The recursive query solution using DB2 UDB (CF-6.18) was suggested by Leung and Pirahesh [66].

There has been some work on implementing temporal joins within the DBMS [97].

Milton Stoneman's *Easy-to-Make Wooden Sundials* [99] provides five delightful sundial designs, explaining how to construct these sundials and how to adjust the sundial to your latitude.

OVERVIEW

A major thread of this exposition is the classification of statements (queries, integrity constraints) into current, sequenced, and non-sequenced variants. This taxonomy also benefits modifications.

SQL has three modification statements: IN-SERT, DELETE, and UPDATE. Interestingly, a current deletion is implemented as an UPDATE followed by a DELETE, and a sequenced deletion requires an INSERT, two UPDATEs, and a DELETE. Updates are more complex: a current update is implemented as three SQL statements; sequenced updates require five SQL statements. Mentioning other temporal tables makes things even more exciting. Finally, we consider the pros and cons of breaking up a valid-time table into a current portion and a historical portion, each a separate table.

Modifying State Tables

We now turn to implementing modifications to a valid-time table. We consider separately current modifications (those on the current, and future, states), sequenced modifications (those evaluated, conceptually at least, at each point in time), and nonsequenced modifications (those explicitly mentioning the timestamp columns). We apply these modifications to the `INCUMBENTS` table.

We also examine how primary key and referential integrity constraints can be maintained across these modifications. One approach, exemplified in Chapter 5, is to specify an assertion. Any violating modification results in the rollback of the transaction (assuming the assertion is immediate; see Section 5.7), thereby guaranteeing that the database remains consistent with the integrity constraint. Of course, the problem is that this is an all-or-nothing proposition: one improper modification derails the entire transaction.

A second approach is to code the modification to ensure that the integrity constraint is not violated, perhaps reducing the execution time required to check the constraint.

In this chapter, we first show how to perform the modification, then consider additions that also ensure uniqueness, primary key, and referential integrity constraints remain satisfied. We continue with the `INCUMBENTS` table, repeated as Table 7.1.

7.1 CURRENT MODIFICATIONS

A current modification concerns something that happens right now.

A *current modification* concerns something that happens now and applies into the future. In a conventional, nontemporal table, all modifications are current modifications.

Table 7.1 An excerpt from the `INCUMBENTS` **valid-time state table.**

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-06-01
111223333	120033	1996-06-01	1996-08-01
111223333	120033	1996-08-01	1996-10-01
111223333	137112	1996-10-01	3000-06-01
444332222	120033	1997-01-01	3000-01-01

In showing how integrity constraints can be ensured within the modification, we consider two cases: the general case where any modification is allowed and the restricted case where only current modifications are performed on the table. The cases differentiate the data upon which the modification is performed and consider whether a noncurrent 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 we can exploit in the (current) modification now being performed.

7.1.1   Current Insertions

A current insertion requires only that the start date and end date be specified.

Code Fragment 7.1   Bob joins as associate director of the Computer Center.

```
INSERT INTO INCUMBENTS
VALUES (111223333, 999071, CURRENT_DATE, DATE '3000-01-01')
```

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

Maintaining a sequenced primary key is easy if only the current state of the table is ever modified.

Code Fragment 7.2   Bob joins as associate director of the Computer Center, ensuring the primary key, in the restricted case.

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT 111223333, 999071, CURRENT_DATE, DATE '3000-01-01'
FROM DUAL
WHERE NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I2
                     WHERE SSN = I2.SSN
                       AND PCN = I2.PCN
                       AND I2.END_DATE = DATE '3000-01-01')
```

Babylonic and Italic Hours

Babylonian sundials counted the hours since sunrise. The Italians of the Middle Ages started a new day at sunset. In Germany during the Renaissance, the custom was to count in Babylonic hours during the day and Italic hours during the night. On many cathedrals, in particular in Strasbourg and Basel, sundials indicate both Italic and Babylonic hours. In fact, with such sundials, you can determine (1) the number of hours elapsed since sunrise: the Babylonic hour, (2) the number of hours left till sunset: the Italic hour, (3) the length of the day, by adding the two together, (4) the day of the year, from the position of the shadow at high noon, (5) the true hour of sunrise, from the noon line, and (6) the true hour of sunset, also from the noon line.

Ensuring uniqueness requires a WHERE predicate, an augmented primary key constraint, or a uniqueness constraint.

This works because all rows that are current now extend to "forever"; there are no rows that start in the future and end before "forever." As mentioned on page 138, `DUAL` is a dummy system table provided by Oracle. It can be simulated in other DBMSs for use here by creating a one-column table and inserting a single row into it.

Alternatively, a sequenced primary key in the restricted case can be stated as a primary key constraint, appending the `END_DATE`, as exemplified in CF-5.15.

This statement doesn't insert a row if Bob currently has that position. An alternative approach would be to first (current) delete his position, then perform the insertion. Yet another approach would be to (current) update his position. Both options will be discussed shortly.

Ensuring referential integrity in the restricted case is also straightforward. Only an insertion in the referencing table can possibly violate referential integrity. (We assume in this discussion that both tables, referencing and referenced, are temporal.)

Ensuring uniqueness and referential integrity in the restricted case

Add this row to the table

if a duplicate row is not already present (i.e., no key violation)
and if there is a corresponding row in the referenced table (i.e., no foreign key violation).

We use as an example the referential integrity constraint from `INCUMBENTS.PCN` to `POSITIONS.PCN`, for which an insertion into the referencing table (here, `INCUMBENTS`) can violate the constraint.

Code Fragment 7.3   **Bob joins as associate director of the Computer Center, also ensuring referential integrity, in the restricted case.**

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT 111223333, 999071, CURRENT_DATE, DATE '3000-01-01'
FROM POSITIONS
WHERE PCN = 999071
  AND END_DATE = DATE '3000-01-01'
  AND NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I2
                   WHERE SSN = I2.SSN
                     AND PCN = I2.PCN
                     AND I2.END_DATE = DATE '3000-01-01')
```

Ensuring referential integrity with a current insertion in the restricted case requires an additional WHERE predicate.

If the position is valid at "forever," it is also valid now, and so fully covers the period being inserted into INCUMBENTS.

If the referential integrity would have been violated (because the position isn't current in POSITIONS), the statement inserts nothing. An alternative strategy is to fill the gap in the *referenced* table (where the PCN is not present in POSITIONS) with null values for the other columns, effectively defining a new position with a null JOB_TITLE_CODE1, so that Bob's position can be inserted into INCUMBENTS.

Code Fragment 7.4   **Fill the gap (in the restricted case) in the** POSITIONS **table for the position of associate director of the Computer Center.**

```
INSERT INTO POSITIONS (PCN, JOB_TITLE_CODE1, START_DATE, END_DATE)
SELECT DISTINCT 999071, NULL, CURRENT_DATE, DATE '3000-01-01'
FROM DUAL
WHERE NOT EXISTS ( SELECT *
                   FROM POSITIONS
                   WHERE PCN = 999071
                     AND END_DATE = DATE '3000-01-01')
```

Filling the gap in the referenced table is an easy way to ensure referential integrity for current insertions into the referencing table.

We need to fill the period between "now" and "forever" with this row, but only if there isn't a row already there. If the gap is first filled, then there is no need to check for referential integrity in the original INSERT statement (into INCUMBENTS).

We have thus far looked at the restricted case where only current modifications are allowed. Such a situation is quite common. Envision an application that operates on a nontemporal table. Now the table is converted to retain the history, say, by adding a start and an end date column, and the modifications and queries in the application are also converted. Since the application did not before concern itself with history, all the modifications are current modifications, on the current state.

The Fundamental Insight

The design of early sundials and water clocks reflected the seemingly continuous nature of time. The gnomon's shadow traced a path across the etchings of the sundial, and the water (or sand) flowed in a steady stream. Su Song's water clock, while simultaneously intricate and massive, was hobbled in its accuracy by this flow, which lacked both precision and power. The European insight, by some as yet unknown genius, was to abandon reliance on an *accumulative* approach, and instead focus on periodically *stopping* the fall of a weight that pulled a rope wound round a cylinder. In this way, the weight doesn't continue to accelerate in its descent. If the stops were timed correctly, the cylinder would rotate slowly, a revolution every hour or every day. This transition from an analog model to a digital model was every bit as invigorating to the society of the 15th century as the transition from analog electronics (i.e., vacuum tubes) to digital electronics (i.e., transistors and integrated circuits) in the 20th century.

Now let's consider the more general case. In the restricted case, all modifications are current modifications. This implies that there are only two kinds of rows in the table: those that started in the past and ended in the past, and those that started in the past and are still valid, that is, have an END_DATE of "forever."

The general case allows other kinds of modifications, thereby permitting two additional kinds of rows: those that started in the past and end sometime in the future (but before "forever"), and those that start sometime in the future and end sometime in the future, including "forever." Such rows represent planning information, or scheduled changes.

These two new kinds of rows are problematic, in that they introduce gaps in the future behavior. The possibility of gaps in the future impacts the modifications that are applied to the table.

So, the task before us is to implement a current insertion in the general case. We first consider ensuring solely the primary key constraints.

Code Fragment 7.5 **Bob was assigned the position of associate director of the Computer Center, ensuring the primary key, in the unrestricted case.**

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT 111223333, 999071, CURRENT_DATE, DATE '3000-01-01'
FROM DUAL
WHERE NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I2
                   WHERE SSN = I2.SSN
                     AND PCN = I2.PCN
                     AND I2.END_DATE > CURRENT_DATE)
```

We don't perform the insertion if doing so would violate the primary key constraint. I2 violates this constraint (in the context of the row being inserted) if its period of validity overlaps that of the inserted row. This will occur if I2 stops in the future or is currently valid, in which case the end date is "forever," in this case the year 3000. (If "forever" was represented with a particular date in the past, such as 1860, the above predicate would not be sufficient.)

We now turn our attention to the referential integrity constraint. As before, only an insertion in the referencing table can possibly violate referential integrity. In this case, a predicate must be added to the WHERE clause of the above code fragment asserting that there are no gaps in the POSITIONS table for the position code of 999071 for "now" to "forever." This predicate is a simple modification of CF-5.21 (see also CF-7.14, below).

When unrestricted modifications are possible, such modifications may generate gaps that must be filled to ensure referential integrity.

Alternatively, the gap(s) can be filled, in which case no such predicate is required in the current insertion. In the restricted case, either a row was valid for the entire period from "now" to "forever," or the PCN was not valid for the entire period. In the general case, there can be gaps, due to the possibility of periods starting or ending sometime in the future. Recall that COALESCE in SQL-92 evaluates to the value of the first argument, unless that value is NULL, in which case it evaluates to the value of the second argument, and so on.

Code Fragment 7.6   **Fill gaps in the** POSITIONS **table for the position of associate director of the Computer Center, in the general case.**

```
INSERT INTO POSITIONS
SELECT 999071, NULL, END_DATE AS START_DATE,
  COALESCE((SELECT MIN(START_DATE)
            FROM POSITIONS AS P2
            WHERE P2.PCN = 999071
              AND P2.START_DATE > P.START_DATE),
         DATE '3000-01-01') AS END_DATE
FROM POSITIONS AS P
WHERE P.PCN = 999071
  AND P.END_DATE > CURRENT_DATE
  AND P.END_DATE < DATE '3000-01-01'
  AND NOT EXISTS ( SELECT *
                   FROM POSITIONS AS P3
                   WHERE P3.PCN = 999071
                     AND P3.START_DATE <= P.END_DATE
                     AND P.END_DATE < P3.END_DATE)
```

The start of a gap is identified where a row ends, but there is no overlapping row that extends the period of validity. The end of the gap is either the start of the first period or, in the absence of such a period, "forever."

7.1.2   Current Deletions

Current deletions apply from "now" to "forever." In the restricted case of only current modifications being allowed on tables, a current deletion is translated into an update.

Code Fragment 7.7 **Bob was just fired as associate director of the Computer Center (only current modifications assumed).**

```
UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE SSN = 111223333
  AND PCN = 999071
  AND END_DATE = DATE '3000-01-01'
```

In the restricted case, a current deletion is converted into an update of the end date.

Let's say Bob was previously hired as associate director of the Computer Center on January 1, 1998. Assuming only current modifications, there cannot be future appointments in the table, so the most recent appointment in the INCUMBENTS table is

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	3000-01-01

Bob was fired on March 13, 1998 (Friday!). This logical deletion updates that row to

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	1998-03-13

In the general case, deletions are implemented as an update and a delete.

Deletion in the general case

For those rows that started in the past and end in the future, reset the end date to "now."

Delete entirely those rows that start in the future.

Code Fragment 7.8   **Bob was just fired as associate director of the Computer Center.**

```
UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < CURRENT_DATE
  AND END_DATE > CURRENT_DATE

DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE >= CURRENT_DATE
```

In the general case, a current deletion is implemented as an update, for those currently valid periods, and a delete, for those periods starting in the future.

These two statements can be performed in either order, as they affect disjoint sets of rows.

The DELETE statement eliminates those states that become true in the future. If only current modifications are applied to the `INCUMBENTS` table, such states cannot occur, and the DELETE statement is not then necessary. Note also that in the general case, the UPDATE predicate is more complex.

Ensuring the primary key constraints (specifically, no duplicates) is trivial, because deletions cannot cause duplicates. There is also no problem ensuring referential integrity for the referencing table. For the *referenced* table (e.g., deleting a row of the `POSITIONS` table), a cascading current delete on the referencing table (in this case, on the `INCUMBENTS` table) of all rows referencing that `PCN` value will ensure referential integrity.

7.1.3   Current Updates

A current update in the restricted case is implemented by an update to end the current row at "now" and an insertion of the new values.

An update is logically a delete coupled with an insert. (Things get complicated in the presence of triggers. In this discussion, we assume no triggers have been defined on the temporal table.)

A current update changes the value for the period of validity from "now" to "forever." A current update is the analog of a nontemporal update, such as the following:

Code Fragment 7.9   **Today Bob was promoted to director of the Computer Center (non-temporal version).**

```
UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

If only current modifications are applied to `INCUMBENTS`, then all the affected rows extend to "forever." We terminate the current position at "now" and insert the new position, from "now" to "forever."

Code Fragment 7.10   **Today Bob was promoted to director of the Computer Center (assuming only current modifications).**

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT SSN, 908739, CURRENT_DATE, DATE '3000-01-01'
FROM INCUMBENTS
WHERE SSN = 111223333
  AND END_DATE = DATE '3000-01-01'

UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE SSN = 111223333
  AND START_DATE < CURRENT_DATE
```

The update must occur after the insertion. Alternatively, the portion up to "now" could be inserted and the update could change the explicit column(s) and the start date to "now."

As we'll see, simple modifications such as CF-7.9 transmute into a series of modifications. Here, the temporal version of an update consists of two modification statements. Later we'll see a six-line nontemporal UPDATE explode into eight (!) statements and 77 lines of SQL. These statements in concert effect the desired results. Unfortunately, it is often the case that intermediate states, after some of the statements have executed but before the rest of the statements, may not satisfy stated integrity constraints.

As a specific example, the sequenced primary key constraint of CF-5.8 on page 118 is violated between the INSERT and the UPDATE statements of CF-7.10. For this reason, as discussed in Section 5.7, assertion checking should be delayed to the end of the transaction, or at least until after all the modification statements implementing a temporal modification complete.

Let's return to the previous example table, with the following row:

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	3000-01-01

We promote Bob to director on Friday, March 13 (now this is his lucky day!).

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	1998-03-13
111223333	908739	1998-03-13	3000-01-01

If arbitrary modifications are permitted on INCUMBENTS, then there may exist rows that start in the future, as well as rows that end before "forever." For the former, only the PCN need be changed. For the latter, the END_DATE must be retained on the inserted row.

We emphasize that a current update has a period of applicability of "now" to "forever." A row that starts in the future may be a planned transfer that shouldn't be impacted by the update, in which case a sequenced update, to be discussed in Section 7.2.3, with a shorter period of applicability, is more appropriate.

Figure 7.1 shows the three cases. If a row's period of validity terminates in the past, then the update will not affect that row. If the row is currently valid, then the portion before "now" must be terminated at "now," and a new row, with the updated values, inserted, with the period of validity starting at "now" and terminating when the original row did. If the row starts in the future, the row can be updated as usual.

Current update in the general case

Insert new information valid from "now" until the row ended.
Terminate the current row at "now."
Update any rows that start in the future with the new values.

Code Fragment 7.11   **Today Bob was promoted to director of the Computer Center.**

```
INSERT INTO INCUMBENTS
SELECT SSN, 908739, CURRENT_DATE, END_DATE
FROM INCUMBENTS
WHERE SSN = 111223333
  AND START_DATE <= CURRENT_DATE
  AND END_DATE > CURRENT_DATE

UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE SSN = 111223333
  AND START_DATE < CURRENT_DATE
```

```
    AND END_DATE > CURRENT_DATE

UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
  AND START_DATE > CURRENT_DATE
```

A current update in the general case is implemented by two updates and an insertion.

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

The impact of an update on a primary key constraint is best judged by viewing the update as a delete followed by an insert. The delete cannot violate the constraint, but the insertion can. The approaches described in Section 7.1.1 apply to updates as well.

When examining the impact of an update on a referential integrity constraint, the insertion portion of an update on the *referencing* table is of concern, as is the deletion portion of an update on the *referenced* table.
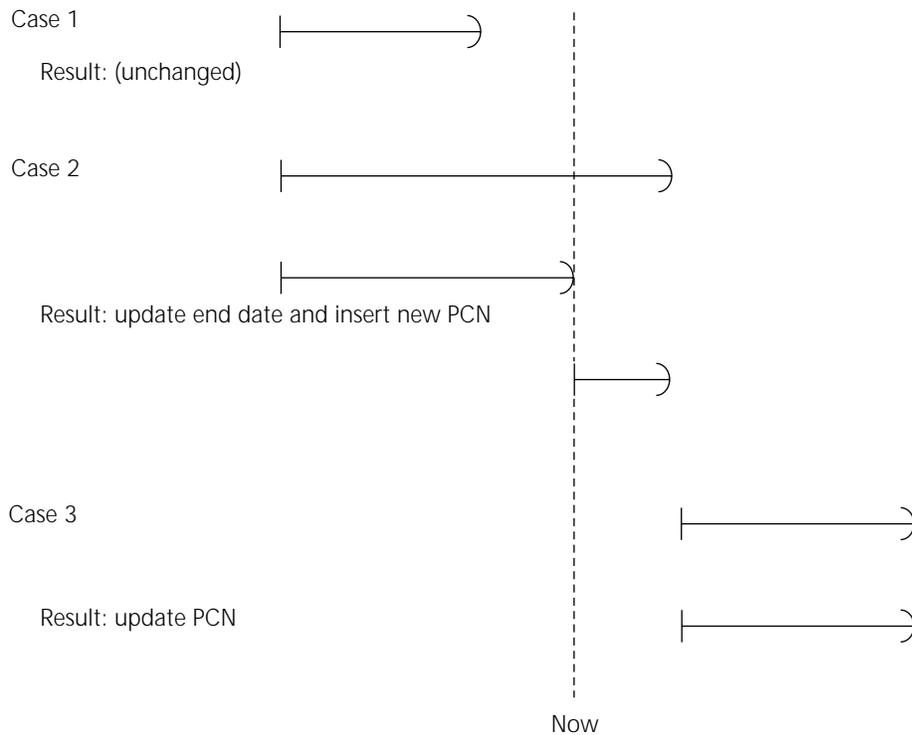
Figure 7.1 Current update cases, with the period of validity of the row shown.

7.2   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."

A current modification is simply a sequenced modification with a period of applicability of "now" to "forever."

Most of the previous material applies to sequenced modifications, with CURRENT_DATE replaced with the start of the period of applicability of the modification and with `DATE '3000-01-01'` replaced with the end of the period of applicability. Here we discuss only the significant differences between current and sequenced modifications, and will discuss the unrestricted case, where periods may start or end in the future.

7.2.1   Sequenced Insertions

In a sequenced insertion, the application provides the period of applicability.

Code Fragment 7.12   **Bob was assigned the position of associate director of the Computer Center for 1997.**

```
INSERT INTO INCUMBENTS
VALUES (111223333, 999071, DATE '1997-01-01', DATE '1998-01-01')
```

Note that as usual we are using a closed-open representation for periods.

To ensure a primary key, we need to look for duplicates anytime during the period of applicability. This requires a slight generalization of CF-7.5.

Code Fragment 7.13   **Bob was assigned the position of associate director of the Computer Center for 1997, ensuring the primary key.**

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT 111223333, 999071, DATE '1997-01-01', DATE '1998-01-01'
FROM DUAL
WHERE NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I2
                   WHERE SSN = I2.SSN
                     AND PCN = I2.PCN
                     AND I2.START_DATE < DATE '1998-01-01'
                     AND DATE '1997-01-01' < I2.END_DATE)
```

Here we just check for overlap with the period of applicability.

For preserving referential integrity, we again have two choices. We can disallow the insertion if the constraint would be violated, or we can fill the gaps before the insertion.

Disallowing a violation of the constraint requires a hefty predicate, drawn from CF-5.21.

Sequenced insertion ensuring uniqueness and referential integrity

Insert a row

if no duplicate exists during the period of applicability,
and if there is a row in the referenced table at the start of the period of applicability,
and if there is a row at the end of the period of applicability,
and if there are no gaps during the period of applicability.

Code Fragment 7.14   **Bob was assigned the position of associate director of the Computer Center for 1997, also ensuring referential integrity.**

```
INSERT INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT DISTINCT 111223333, 999071, DATE '1997-01-01', DATE '1998-01-01'
FROM POSITIONS
WHERE NOT EXISTS ( SELECT *
                   FROM INCUMBENTS AS I2
                   WHERE SSN = I2.SSN
                     AND PCN = I2.PCN
                     AND I2.START_DATE < DATE '1998-01-01'
                     AND DATE '1997-01-01' < I2.END_DATE)
  AND EXISTS ( SELECT *
               FROM POSITIONS AS P
               WHERE P.PCN = 999071
                 AND P.START_DATE <= DATE '1997-01-01'
                 AND DATE '1997-01-01' < P.END_DATE)
  AND EXISTS ( SELECT *
               FROM POSITIONS AS P
               WHERE P.PCN = 999071
                 AND P.START_DATE < DATE '1998-01-01'
                 AND DATE '1998-01-01' <= P.END_DATE)
  AND NOT EXISTS ( SELECT *
                   FROM POSITIONS AS P
                   WHERE P.PCN = 999071
                     AND DATE '1997-01-01' < P.END_DATE
                     AND P.END_DATE < DATE '1998-01-01'
                     AND NOT EXISTS ( SELECT *
                           FROM POSITIONS AS P2
                           WHERE P2.PCN = 999071
                             AND P2.START_DATE <= P.END_DATE
                             AND P.END_DATE < P2.END_DATE))
```

There are four components to the predicate. The first states that there are no duplicates over the period of applicability. The second states that the new position is valid at the start of the period of applicability. The third states that the new position is valid at the end of the period of applicability. The last component (the NOT EXISTS) states that there is no gap during the period of applicability. As before, a gap exists if there is a row `P` that ends during the period of applicability that is not extended (towards the beginning of 1998) by another row of `POSITIONS`.

The other approach, of filling gaps, requires modifying CF-7.6, substituting `CURRENT_DATE` with `DATE '1997-01-01'` and `DATE '3000-01-01'` with `DATE '1998-01-01'`.

7.2.2 Sequenced Deletions

We start with the following nontemporal deletion:

Code Fragment 7.15 **Bob was removed as associate director of the Computer Center (nontemporal version).**

```
DELETE FROM INCUMBENTS
WHERE SSN=111223333
  AND PCN = 999071
```

We now wish to convert this to a sequenced deletion, over a period of applicability of the year 1997: "Bob was removed as associate director of the Computer Center for 1997."

Recall that a current deletion in the general case is implemented as an update, for the currently valid periods, and a delete, for periods starting in the future. For a sequenced deletion, there are four cases, as shown in Figure 7.2. In each case, the period of validity of the original tuple is shown above the period of applicability 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 end 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.

Sequenced deletion

Insert the old values from the end of the period of applicability to the end of the period of validity of the original row.

Update the end date to end at the beginning of the period of applicability.

Update the start date to begin at the end of the period of applicability.

Delete entirely rows that are covered by the period of applicability.
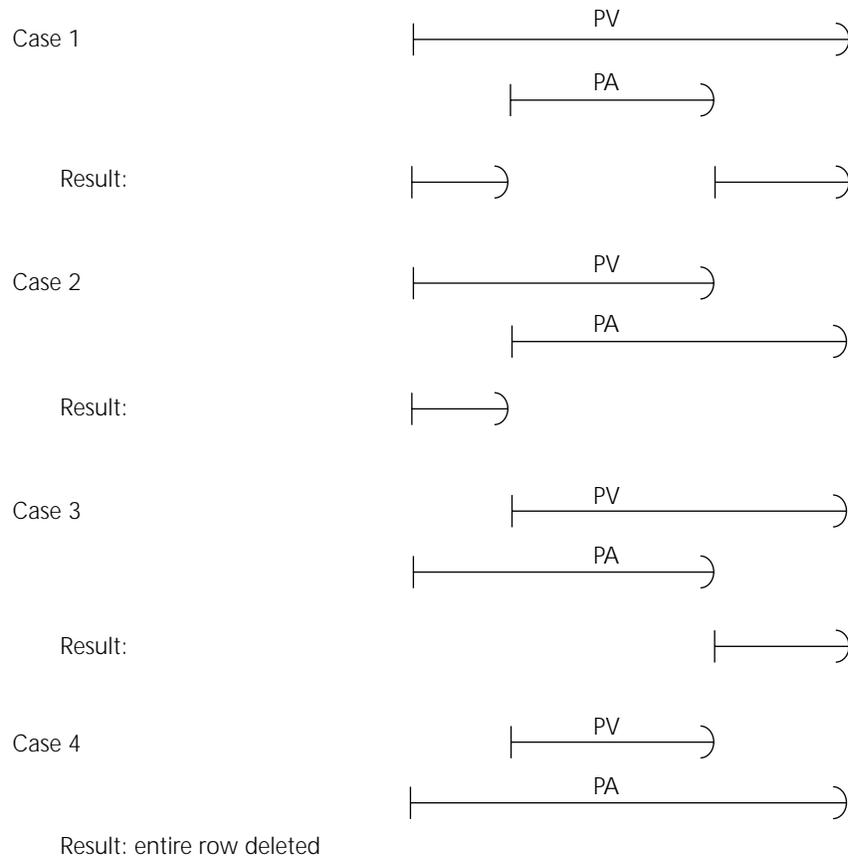
Figure 7.2 Sequenced deletion cases, with the period of validity (PV) and period of applicability (PA) shown.

Code Fragment 7.16   **Bob was removed as associate director of the Computer Center for 1997.**

```
INSERT INTO INCUMBENTS
SELECT SSN, PCN, DATE '1998-01-01', END_DATE
FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1998-01-01'
```

```
UPDATE INCUMBENTS
SET END_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE >= DATE '1997-01-01'

UPDATE INCUMBENTS
SET START_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE >= DATE '1998-01-01'

DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE >= DATE '1997-01-01'
  AND END_DATE <= DATE '1998-01-01'
```

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.

As a simple example, we start with Bob having been promoted on March 13 to director:

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	1998-03-13
111223333	908739	1998-03-13	3000-01-01

He is given a leave of absence for the month of March 1998, which corresponds to a sequenced deletion with a period of applicability of [1998-03-01 – 1998-04-01). The result is

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	1998-03-01
111223333	908739	1998-04-01	3000-01-01

The first row is handled by the first UPDATE (Case 2), and the second row is handled by the second UPDATE (Case 3).

Table 7.2 The leave of absence was for the month of April.

SSN	PCN	START_DATE	END_DATE
111223333	999071	1998-01-01	1998-03-13
111223333	908739	1998-03-13	1998-04-01
111223333	908739	1998-05-01	3000-01-01

Now let's consider instead that the leave of absence is the month of *April*. The result then is Table 7.2. This utilizes only Case 1, which is implemented as an INSERT and an UPDATE.

A sequenced deletion is implemented by four statements: an insertion, two updates, and a deletion.

Note that a sequenced primary key will be temporarily violated (by the INSERT statement), but will be satisfied after the final DELETE statement, which implies that the constraint should be temporarily deferred during this modification. All four statements must be evaluated in the order shown. They have been carefully designed to cover each case exactly once.

A sequenced deletion can violate a nonsequenced uniqueness constraint. It cannot violate a current or sequenced uniqueness constraint.

Concerning referential integrity, a sequenced deletion may introduce a gap, violating a foreign key referencing the table on which the deletion is applied. Concerning duplicates, a deletion applied on a snapshot table cannot cause a uniqueness constraint to be violated. The same holds for a sequenced uniqueness constraint. However, a sequenced deletion can violate nonsequenced uniqueness because deleting the middle out of a single period results in two disconnected periods.

As an example, the following table contains no nonsequenced duplicates (it *does* contain value-equivalent rows and sequenced duplicates, but that is not the focus here):

SSN	PCN	START_DATE	END_DATE
111223333	120033	1998-04-01	1998-06-01
111223333	120033	1998-04-01	1998-10-01

If a sequenced deletion with a period of applicability of June 1, 1998 through July 31, 1998, is applied, the resulting table (Table 7.3) *does* violate nonsequenced uniqueness!

Table 7.3 June and July 1998 were deleted.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1998-04-01	1998-06-01
111223333	120033	1998-04-01	1998-06-01
111223333	120033	1998-08-01	1998-10-01

7.2.3   Sequenced Updates

A sequenced update is the temporal analog of a nontemporal update, with a specified period of applicability. Let us again consider the following update:

Code Fragment 7.17   **Bob was promoted to director of the Computer Center (nontemporal version).**

```
UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

We now convert this to a sequenced update: Bob was promoted only for the calendar year 1997.

A sequenced update is implemented by five statements: two insertions and three updates.

As with sequenced deletions, there are more cases to consider for sequenced updates as compared with current updates. The four cases in Figure 7.2 are handled differently in an update. In Case 1 of Figure 7.3, the initial and final portions of the period of validity are retained (via two insertions), and the affected portion is updated. In Case 2, only the initial portion is retained; in Case 3, only the final portion is retained. In Case 4, the period of validity is retained, as it is covered by the period of applicability.

Sequenced update

Insert the old values from the start date to the beginning of the period of applicability.
Insert the old values from the end of the period of applicability to the end date.
Update the explicit columns of rows that overlap the period of applicability.
Update the start date to begin at the beginning of the period of applicability of rows that overlap the period of applicability.
Update the end date to end at the end of the period of applicability of rows that overlap the period of applicability.

Code Fragment 7.18   **Bob was promoted to director of the Computer Center for 1997.**

```
INSERT INTO INCUMBENTS
SELECT SSN, PCN, START_DATE, DATE '1997-01-01'
FROM INCUMBENTS
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'
```

```
INSERT INTO INCUMBENTS
SELECT SSN, PCN, DATE '1998-01-01', END_DATE
FROM INCUMBENTS
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'

UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS
SET START_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS
SET END_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'
```

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.

Returning again to our simple example,

SSN	PCN	START_DATE	END_DATE
111223333	341288	1998-01-01	1998-03-13
111223333	908739	1998-03-13	3000-01-01

Bob was given a temporal assignment of PCN = 999071 for the month of March 1998. The result (Table 7.4) contains four rows. Study this result carefully to convince yourself that it is indeed what is desired.

Case 1

Old value retained:

Updated portion:

Case 2

Old value retained:

Updated portion:

Case 3

Old value retained:

Updated portion:

Case 4

Result: entire row updated
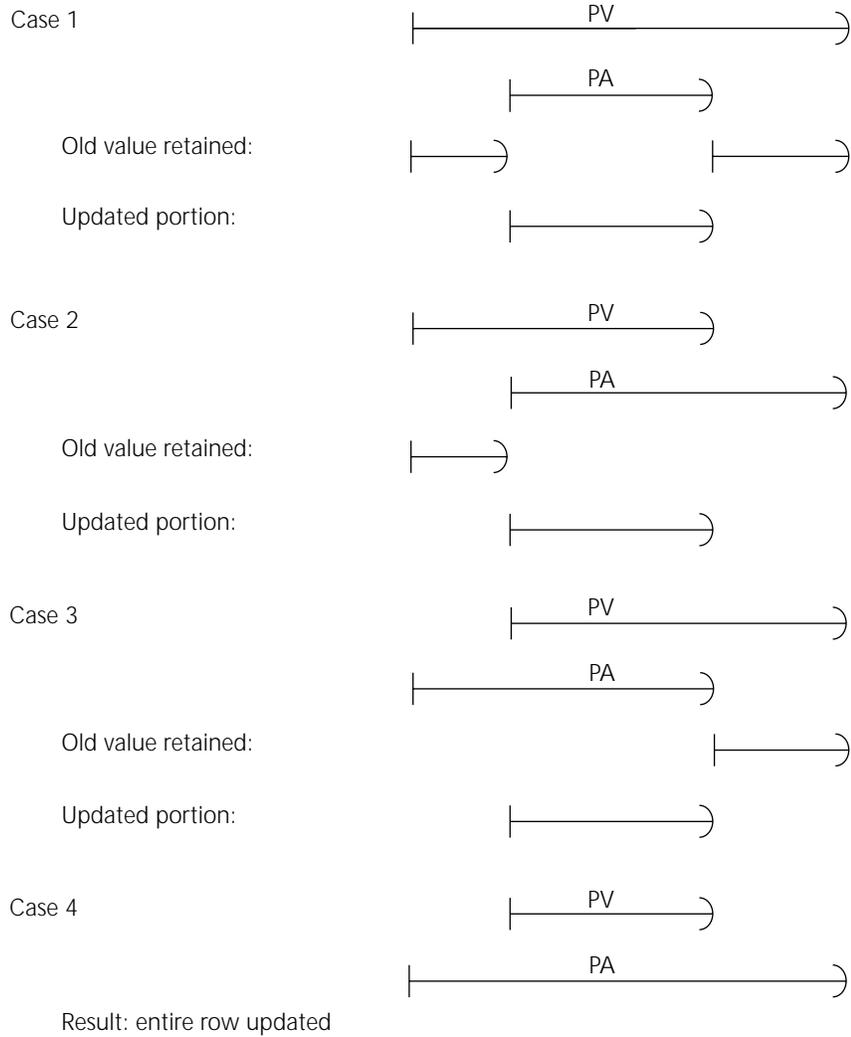
Figure 7.3 Sequenced update cases, with the period of validity (PV) and period of applicability (PA) shown.

Table 7.4 Result of the sequenced update.

SSN	PCN	START_DATE	END_DATE
111223333	*341288*	*1998-01-01*	*1998-03-01*
111223333	999071	*1998-03-01*	1998-03-13
111223333	*999071*	1998-03-13	*1998-04-01*
111223333	*908739*	*1998-04-01*	*3000-01-01*

Effecting this result requires the following changes to the original two rows:

1. Insert the first row of the result (the first INSERT of CF-7.18, Case 2 of Figure 7.3, old value retained).
2. Insert the fourth row of the result (the second INSERT, Case 3, old value retained).
3. Change the PCN of both of the original rows to 999071, forming new rows 2 and 3 (the first UPDATE, Cases 2 and 3, updated portion).
4. Change the start date of the first original row to 1998-03-01, to form the second row of the result (the second UPDATE, Case 2, old value retained).
5. Change the end date of the second original row to 1998-04-01, to form the third row of the result (the third UPDATE, Case 3, old value retained).

Amazingly, this *does* produce the correct result, in all cases.

Concerning duplicates and referential integrity, a sequenced update can again be considered to be a combination of a sequenced deletion followed by a sequenced insertion. The deletion is relevant for referenced tables (e.g., of concern if we updated the POSITIONS table); the insertion is relevant for uniqueness and for referencing tables (e.g., for an insertion we would need to check that the PCN was in the POSITIONS table).

7.3 NONSEQUENCED MODIFICATIONS

As with constraints and queries, a nonsequenced modification treats the timestamps identically to the other columns.

Code Fragment 7.19   **Delete Bob's records that include 1997 stating that he was associate director of the Computer Center.**

```
DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE <= DATE '1997-12-31'
  AND DATE '1997-01-01' < END_DATE
```

Nonsequenced modifications are usually difficult to state in English because they are expressed in terms of the representation, but easier to express in SQL for the same reason.

Let's compare this statement with the current variant, CF-7.8, "Bob was fired as associate director of the Computer Center," and with the sequenced variant, CF-7.16, "Bob was removed as associate director of the Computer Center for 1997." 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.

Nonsequenced modifications are rare.

The ramifications (e.g., ensuring uniqueness and referential integrity) of a current modification impact the period from "now" to "forever"; those for sequenced modifications impact the period of applicability. Since nonsequenced modifications manipulate the timestamps in arbitrary ways, their impact must be judged on a case-by-case basis. Consider the following simple SQL update:

Code Fragment 7.20   **Extend Bob's position as associate director of the Computer Center for an additional year.**

```
UPDATE INCUMBENTS
SET END_DATE = END_DATE + INTERVAL '1' YEAR
WHERE SSN = 111223333
  AND PCN = 999071
```

Correctly implementing a nonsequenced modification in the presence of sequenced constraints is difficult and must be done on a case-by-case basis.

Note that Bob might have had this position multiple times (perhaps he went on leave or had a temporary assignment elsewhere); this statement will change all such records. Unlike the sequenced update (CF-7.18, "Bob was promoted to director of the Computer Center for 1997"), it is probable that the nonsequenced update just listed will result in (sequenced) duplicates (Bob having two positions at a point in time). How do we avoid such integrity violations? In this case, we would need to either augment the UPDATE statement to check for sequenced duplicates (and not do the update if one was found), or first delete the offending duplicates before doing the update. Either way, it is not possible to give a general algorithm; the specific code depends entirely on how the modification manipulates the timestamp column(s).

7.4   MODIFICATIONS THAT MENTION OTHER TABLES*

The examples given in this chapter have been simple ones, with predicates and the SET clause limited to simple equalities involving other columns in the table being modified. What if predicate or SET clause(s) involve subqueries, or mention other tables? To convert such complex modifications, a combination of the techniques from this chapter and the previous chapter on queries must be applied.

7.4.1   Complex Current Modifications

Recall from Chapter 6 that a current query merely requires an additional predicate:

```
START_DATE <= CURRENT_DATE AND CURRENT_DATE < END_DATE
```

Predicates and SET clauses in a current modification that mention other temporal tables can use this same comparison.

The following query assumes we don't know the `PCN`, but can look it up in the `POSITIONS` table, given the `JOB_TITLE_CODE1`, which is available from the (nontemporal) table `JOB_TITLES`. We first show the modification ignoring time.

Code Fragment 7.21   **Bob is promoted to director of the Computer Center getting the** `PCN` **from** `POSITIONS` **(nontemporal version).**

```
UPDATE INCUMBENTS
SET PCN = (SELECT PCN
           FROM POSITIONS, JOB_TITLES
           WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
            AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER')
WHERE SSN = 111223333
```

The following is the current version of this modification; compare it with CF-7.11.

Code Fragment 7.22   **Bob is promoted to director of the Computer Center getting the** `PCN` **from** `POSITIONS` **(current version).**

```
INSERT INTO INCUMBENTS
SELECT SSN, I.PCN, CURRENT_DATE, I.END_DATE
FROM INCUMBENTS AS I, POSITIONS, JOB_TITLES
WHERE SSN = 111223333
  AND I.START_DATE <= CURRENT_DATE
  AND I.END_DATE > CURRENT_DATE
  AND POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
  AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
  AND POSITIONS.START_DATE <= CURRENT_DATE
  AND CURRENT_DATE < POSITIONS.END_DATE

UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE SSN = 111223333
  AND PCN <> (SELECT PCN
               FROM POSITIONS, JOB_TITLES
               WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
                AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
```

```
                        AND POSITIONS.START_DATE <= CURRENT_DATE
                        AND CURRENT_DATE < POSITIONS.END_DATE)
     AND INCUMBENTS.START_DATE < CURRENT_DATE
     AND INCUMBENTS.END_DATE > CURRENT_DATE

UPDATE INCUMBENTS
SET PCN = (SELECT PCN
              FROM POSITIONS, JOB_TITLES
             WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
               AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
               AND POSITIONS.START_DATE <= CURRENT_DATE
               AND CURRENT_DATE < POSITIONS.END_DATE)
WHERE SSN = 111223333
   AND START_DATE >= CURRENT_DATE
```

Current modifications that mention other tables require an additional overlap predicate for each correlation name.

Two changes were made to CF-7.11. The first replaced the two occurrences of a PCN of 908739 with the SELECT statement that looks up this value from the POSITIONS and JOB_TITLES tables. The second change added the overlap test with CURRENT_DATE for the POSITIONS correlation name. No such test is required of JOB_TITLES because this table doesn't record history.

7.4.2   Complex Sequenced Modifications

Sections 6.3 and 6.3.1 showed how to express sequenced selection, projection, union, sorting, join, and duplicate elimination. Section 7.2 showed how to implement sequenced insertions, deletions, and updates. We now consider implementing sequenced modifications that refer to other tables. Doing so requires combining these techniques.

We warn you that sequenced modifications over multiple time-varying tables are exceedingly knotty. That is little consolation to the application developer who is expected to produce such queries. Here we walk through the steps taking a nontemporal multitable update (at 6 lines) to its sequenced equivalent (at 76 lines!). This section can be safely skipped; you can return here when required to write such an update.

If a SET or WHERE clause mentions another temporal table, the predicate in the context of a sequenced modification will be valid for the periods of validity of the relevant rows of that table. Multiple rows contribute various periods of validity, which must be accounted for.

There are three periods that are involved. The first is the period of validity contributed by the row being modified. The second is the period of applicability for the

Escapements

The mechanism that periodically stops the unwinding of the clock, and thus keeps the time, is termed the "escapement." The first escapement, invented in the 15th century, was the *verge-and-foliot*, or *recoil* escapement. The rotating cylinder has notches or teeth on it, which push (hence the name, "recoil") a weighted bar, the foliot, in one direction, twisting a wire. When the wire untwists, it pushes the foliot in the other direction, releasing the cylinder (called the *crown wheel*) to turn until the escapement hits the next notch. Twist-untwist, tick-tock. The escapement ensures that the crown only advances one unit at a time and serves to impulse the foliot, keeping the clock going.

All escapements have these two tasks: regulate the advancement, and "kick" the oscillating body. A wonderous diversity of escapements has been invented over the last 600 years.

modification. The third is the period of validity of the row of the table mentioned in the SET or WHERE clause.

We are primarily interested in regions where these three periods intersect. Times outside the period of validity of the row being modified are obviously of no concern. The SET clause or WHERE clause is defined only for the period of validity of the mentioned table. The modification affects those times within the period of applicability. Note however that those times outside the period of applicability but within the period of validity of the modified row are still relevant, for we wish to retain the old values for the columns during these times.

Hence, to convert a query mentioning other table(s) to be a sequenced query (the most useful kind), a case analysis on the interaction of these three periods is required. Figures 7.2 and 7.3 illustrated the possible ways that the period of validity of the modified row and the period of applicability can interact. To this analysis we must add the period of validity of the mentioned rows. Each type of interaction generally necessitates an insertion or update. Periods outside the period of applicability require inserts to retain the old column values. Before, there was only one period from the row being modified within the period of applicability; here, because of the other referenced tables, there may be many such periods. These periods are also inserted (for a sequenced insertion or update). The exception is the first period within the period of applicability, which is updated (again, for a sequenced update), both of the column values, to effect the update, and the start and end timestamp columns.

As an example, we convert the above nontemporal update (CF-7.21) to a sequenced update, using a period of applicability of the calendar year 1997. Doing so effectively requires a sequenced join between INCUMBENTS and POSITIONS. Here is the original, nontemporal update.
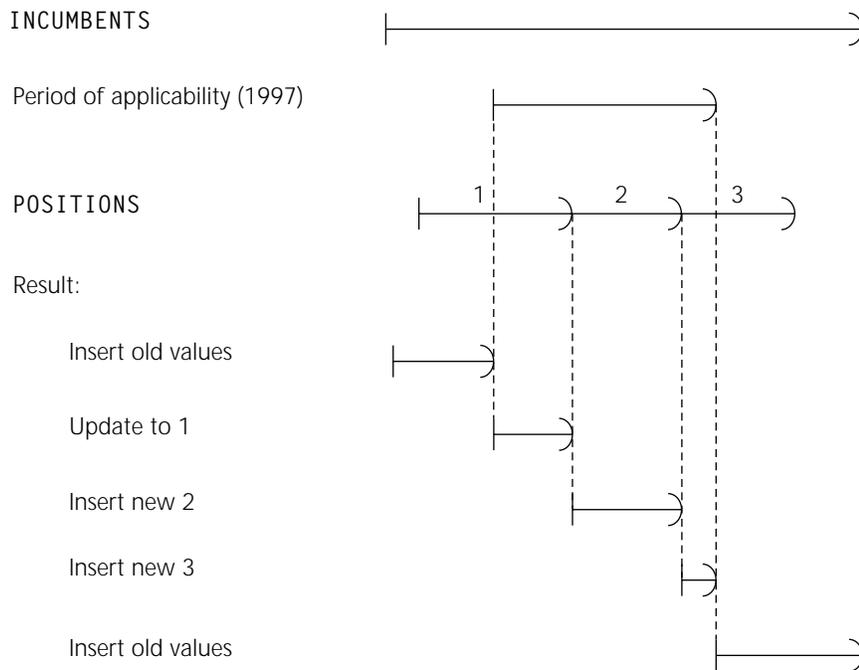
Figure 7.4 Case 1 of sequenced update, mentioning another temporal table.

Code Fragment 7.23   **Bob was promoted to director of the Computer Center.**

```
UPDATE INCUMBENTS
SET PCN = (SELECT PCN
           FROM POSITIONS, JOB_TITLES
           WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
             AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER')
WHERE SSN = 111223333
```

As just mentioned, there are four base cases for sequenced updates (see Figure 7.3). Let's examine how the period of validity for the POSITIONS table interacts with the period of validity for INCUMBENTS and with the period of applicability. In Figure 7.4, we focus on Case 1. In this example, the PCN in POSITIONS for director of the Computer Center changes several times (from 1 to 2 to 3) over the course of 1997.

This particular update will result in a single row from INCUMBENTS being replaced with five rows. The first and last rows will be identical to the original row except for the start and end times. The PCN column of the second row will be 1, of the third row, 2, and of the fourth row, 3, all drawn from the POSITIONS table. Since we need

to derive five rows (in this case) from one original row, this modification will be performed via four inserts and four updates (one to update the PCN column, one to update the START_DATE column, and two to update the END_DATE column), a total of eight modification statements.

Sequenced update mentioning another table

Insert old values before period of applicability begins.

Insert old values after period of applicability ends.

Perform update on first period representing the intersection of the period of validity for the row, the period of applicability, and the period of validity of the mentioned row(s).

Insert new values for the remainder of the period of applicability (two subcases).

Update its start date (one update) and end date (two subcases).

Code Fragment 7.24   **Bob was promoted to director of the Computer Center for 1997 (sequenced version).**

```
INSERT INTO INCUMBENTS
SELECT SSN, PCN, DATE '1998-01-01', END_DATE
FROM INCUMBENTS
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'

INSERT INTO INCUMBENTS
SELECT SSN, PCN, START_DATE, DATE '1997-01-01'
FROM INCUMBENTS
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS
SET PCN = (SELECT PCN
            FROM POSITIONS, JOB_TITLES
            WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
              AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
              AND POSITIONS.START_DATE <= DATE '1997-01-01'
              AND DATE '1997-01-01' < POSITIONS.END_DATE)
```

```
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1997-01-01'

INSERT INTO INCUMBENTS
SELECT SSN, POSITIONS.PCN,
    POSITIONS.START_DATE, POSITIONS.END_DATE
FROM INCUMBENTS, POSITIONS, JOB_TITLES
WHERE SSN = 111223333
  AND INCUMBENTS.START_DATE <= DATE '1998-01-01'
  AND INCUMBENTS.END_DATE > DATE '1998-01-01'
  AND POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
  AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
  AND DATE '1997-01-01' <= POSITIONS.START_DATE
  AND INCUMBENTS.START_DATE <= POSITIONS.START_DATE
  AND POSITIONS.END_DATE < DATE '1998-01-01
  AND POSITIONS.END_DATE < INCUMBENTS.END_DATE

INSERT INTO INCUMBENTS
SELECT SSN, POSITIONS.PCN,
    POSITIONS.START_DATE, DATE '1998-01-01'
FROM INCUMBENTS, POSITIONS, JOB_TITLES
WHERE SSN = 111223333
  AND INCUMBENTS.START_DATE <= DATE '1998-01-01'
  AND INCUMBENTS.END_DATE > DATE '1998-01-01'
  AND POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
  AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER'
  AND DATE '1997-01-01' <= POSITIONS.START_DATE
  AND DATE '1998-01-01 < POSITIONS.END_DATE

UPDATE INCUMBENTS
SET START_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS
SET END_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'
  AND NOT EXISTS ( SELECT *
                    FROM INCUMBENTS AS I2
```

```
                        WHERE INCUMBENTS.SSN = I2.SSN
                          AND INCUMBENTS.PCN = I2.PCN
                          AND INCUMBENTS.START_DATE < I2.END_DATE
                          AND I2.START_DATE < INCUMBENTS.END_DATE)

UPDATE INCUMBENTS
SET END_DATE = (SELECT MIN(I2.START_DATE)
                  FROM INCUMBENTS AS I2
                 WHERE INCUMBENTS.SSN = I2.SSN
                   AND INCUMBENTS.PCN <> I2.PCN
                   AND INCUMBENTS.START_DATE < I2.START_DATE)
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'
  AND EXISTS ( SELECT *
                 FROM INCUMBENTS AS I2
                WHERE INCUMBENTS.SSN = I2.SSN
                  AND INCUMBENTS.PCN <> I2.PCN
                  AND INCUMBENTS.START_DATE < I2.END_DATE
                  AND I2.START_DATE < INCUMBENTS.END_DATE)
```

In comparison with the original sequenced version (CF-7.18), we added here mention of the POSITIONS and JOB_TITLES tables, impacting only the latter three UPDATE statements of the original version. Since POSITIONS is time-varying, we are really doing a sequenced join between that table and the table being updated, INCUMBENTS.

The first INSERT statement handles the initial portion (the first row in Figure 7.4); the second handles the final portion (the last row in the figure). The first update changes the PCN to the value found in the POSITIONS table valid at the beginning of the period of applicability (1997). The following two insertions add intermediate periods during the period of applicability, the former from row(s) of the POSITIONS table that end before the end of applicability, the latter from rows that end after the period of applicability. The following three updates adjust the starting and ending dates of the initial portion with the new PCN value.

This case analysis identifies for which periods the old values should be inserted, for which periods the new values should be inserted, and for which single period the columns should be updated to the new values, with the start and end times changed. We illustrated the most complex case, that of update; deletion is simpler, and insertion simpler still.

7.5   TEMPORAL PARTITIONING*

Sequenced updates referring to other tables are complex to convert into SQL.

As discussed on page 120, the value `'3000-01-01'` ("forever") is used in `INCUMBENTS.END_DATE` to represent currently valid data, that is, data valid until "now." What we really desire is to store CURRENT_DATE as the column value. Unfortunately, neither SQL nor any extant DBMS allows this because the value has the habit of changing once each day. While quite inconvenient for the DBMS vendor, this property is what the user wants. Bob's position could change tomorrow, but at least we know that the position recorded in the database is valid today.

As an aside, using CURRENT_DATE is close, but is not entirely accurate. The UIS is updated every night with the previous day's value. So what we really want to model is that we know Bob's position up to *yesterday*; it could change today, in which case we would hear about the change tomorrow. So what is really needed is the *now-relative* value `CURRENT_DATE - INTERVAL '1' DAY`, which is also not supported as a column value by DBMSs.

Temporal partitioning finesses the limitation of CURRENT_DATE not being permitted as column values by splitting a valid-time table into a current store containing only current information and a history store.

Since we can't store the value we want, we come at the problem a different way. By using *temporal partitioning*, we can avoid the necessity of storing an end time of "now." We partition `INCUMBENTS` into two tables, `INCUMBENTS_CURRENT` and `INCUMBENTS_PAST`. The latter table, termed the *history store*, contains no current information; hence, the `END_DATE` is always a specific, known date. In the former table, termed the *current store*, we have no need for that column, so we omit it. The `START_DATE` is still present in the `INCUMBENTS_CURRENT` table, as we need to know when the current information became valid.

These two tables are not as expressive as the original `INCUMBENTS` table: they cannot record future data. Say Bob receives a promotion postactively, that is, to go into effect at some future time. Postactive promotions are common in universities; many are announced in late spring, to go into effect at the beginning of the following fiscal year (which for the University of Arizona is July 1). Bob's new position could be recorded in `INCUMBENTS`, with a `START_DATE` of July 1. Such data cannot be stored in `INCUMBENTS_PAST`, nor is such data appropriate for `INCUMBENTS_CURRENT` because the position is not yet current.

7.5.1   Queries

If temporal partitioning is used, current queries are simpler: just apply the query to the current store.

Code Fragment 7.25   **What is Bob's current position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS_CURRENT, POSITIONS
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS_CURRENT.SSN
    AND INCUMBENTS_CURRENT.PCN = POSITIONS.PCN
```

The disadvantage of temporal partitioning, as we will see below, is that queries over the history must reference both tables. However, sometimes the data is naturally partitioned at the source of that data. Recall that while SAL_HISTORY (see page 116) contains the salary history, the EMPLOYEES table contains the current salary, as well as the PAY_CHANGE_DATE. An important design issue, which we'll just mention now, is whether to replicate current information in the past table.

Current queries are simplified when applied to a partitioned valid-time table.

While all of these approaches are similar in length (around 30 lines of SQL code), the query becomes much more complex when temporal partitioning is used on one or both of the underlying tables. If the INCUMBENTS table is partitioned into INCUMBENTS_PAST and INCUMBENTS_CURRENT, then the temporal join mushrooms to a 50-line query.

Code Fragment 7.26   **Provide the salary and department history for all employees.**

(Insert CF-6.11 from page 151, substituting   INCUMBENTS_PAST for INCUMBENTS)

```
UNION
SELECT S.SSN, AMOUNT, PCN,
    S.HISTORY_START_DATE, S.HISTORY_END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS_CURRENT AS I
WHERE S.SSN = I.SSN
 AND I.START_DATE <= S.HISTORY_START_DATE
UNION
SELECT S.SSN, AMOUNT, PCN,
    I.START_DATE, S.HISTORY_END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS_CURRENT AS I
WHERE S.SSN = I.SSN
 AND I.START_DATE > S.HISTORY_START_DATE
 AND S.HISTORY._END_DATE <= DATE '3000-01-01'
 AND I.START_DATE < S.HISTORY_END_DATE
UNION
SELECT S.SSN, AMOUNT, PCN,
    I.START_DATE, S.HISTORY_END_DATE
FROM SAL_HISTORY AS S, INCUMBENTS_CURRENT AS I
WHERE S.SSN = I.SSN
 AND I.START_DATE > S.HISTORY_START_DATE
 AND DATE '3000-01-01' < S.HISTORY_END_DATE
```

If the history store does not include current data, then sequenced and nonsequenced queries are more complex. Otherwise, such data must be replicated across both tables.

There are two major changes made to CF-6.11 to produce the portion here. `INCUMBENTS.END_DATE` is replaced with `DATE '3000-01-01'`. With this substitution, the second case in the original doesn't apply to `INCUMBENTS_CURRENT`, and so is omitted.

If the `INCUMBENTS_PAST` also includes the current data, then CF-6.11 with `INCUMBENTS_PAST` substituted for `INCUMBENTS` suffices. This approach favors query simplicity, at the expense of modification complexity, as will now be shown.

7.5.2   Modifications

Current inserts into partitioned tables are even easier.

Code Fragment 7.27   **Bob was assigned the position of associate director of the Computer Center (partitioned).**

```
INSERT INTO INCUMBENTS_CURRENT
VALUES (111223333, 6201945234, CURRENT_DATE)
```

Here we only have to record when the fact became valid.

We assume in this code fragment that duplicates were allowed. If duplicates are not allowed, then there are two choices. One approach is to define a constraint or assertion disallowing duplicates, which will then prevent duplicates from being inserted. This works fine for current and nonsequenced duplicates, as well as for value-equivalent duplicates. For sequenced duplicates (where no duplicates are allowed at each point in time), using a constraint, such as that shown in CF-5.14, is overkill because a duplicate present at *any* instant will prevent the insertion at *all* instants.

Current insertions only impact the current store of a partitioned table.

The second approach is to extend the above INSERT statements to insert facts only if there is no duplicate. For current and nonsequenced duplicates, this is easy. For sequenced duplicates, where duplicates must be avoided at each point in time, ensuring this is easy for the partitioned table. If there is a value-equivalent row in the `INCUMBENTS_CURRENT`, then the period of validity of that row wholly contains the row to be inserted, and the entire insertion should not take place.

Code Fragment 7.28   **Bob was assigned the position of associate director of the Computer Center (partitioned, avoiding sequenced duplicates).**

```
INSERT INTO INCUMBENTS_CURRENT
VALUES (111223333, 6201945234, CURRENT_DATE)
WHERE NOT EXISTS ( SELECT *
                   FROM INCUMBENTS_CURRENT AS I2
                   WHERE I2.SSN = 111223333
                     AND I2.PCN = 6201945234 )
```

Similarly, current deletions move a row from the current store to the history store.

Code Fragment 7.29 **Bob was fired as associate director of the Computer Center (partitioned).**

```
INSERT INTO INCUMBENTS_PAST (SSN, PCN, START_DATE, END_DATE)
SELECT SSN, PCN, START_DATE, CURRENT_DATE
FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND PCN = 999071

DELETE FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND PCN = 999071
```

This is slightly shorter than the nonpartitioned current deletion (CF-7.8). Similarly, a current update is logically a current delete coupled with a current insert.

Sequenced modifications over a partitioned table must consider both stores. For sequenced insertions, there are two cases: (1) the period of applicability lies entirely in the past, in which case the row is inserted into the history store, or (2) the period of applicability includes now, in which case the row is inserted into the current store.

For sequenced deletions, recall that a partitioned store cannot record future data, so the period of applicability must either end before now or extend to "forever." In the first situation, only the history store is modified, as described in Section 7.2.2. The second situation involves changes to the history store and to the current store.

Revisiting Figure 7.2 on page 191, all four cases are relevant for the period of validity for rows in both the current and history stores. The insertion, two updates, and the deletion of CF-7.16 apply as is to INCUMBENTS_PAST. For the current store, these cases result in a somewhat different set of statements.

Code Fragment 7.30 **Bob was removed as associate director of the Computer Center for 1997 (partitioned).**

```
INSERT INTO INCUMBENTS_PAST
SELECT SSN, PCN, DATE '1998-01-01', END_DATE
FROM INCUMBENTS_PAST
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE <= DATE '1997-01-01'
  AND END_DATE > DATE '1998-01-01'
```

```
UPDATE INCUMBENTS_PAST
SET END_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE >= DATE '1997-01-01'

UPDATE INCUMBENTS_PAST
SET START_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE >= DATE '1998-01-01'

DELETE FROM INCUMBENTS_PAST
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE >= DATE '1997-01-01'
  AND END_DATE <= DATE '1998-01-01'

UPDATE INCUMBENTS_CURRENT
SET START_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE <= DATE '1998-01-01'

INSERT INTO INCUMBENTS_PAST
SELECT SSN, PCN, START_DATE, DATE '1997-01-01'
FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1997-01-01'

UPDATE INCUMBENTS_CURRENT
SET START_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE < DATE '1998-01-01'

DELETE FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND PCN = 999071
  AND START_DATE >= DATE '1997-01-01'
```

Sequenced updates are tedious due to the extensive case analysis required for the current and history stores separately.

The situation is similar for sequenced updates. The history store is updated identically to the nonpartitioned valid-time state table (copying the five modification statements from CF-7.18). Updating the current store requires some changes to these statements, yielding another six modification statements (whew!). These are best understood by examining Figure 7.3 and considering what changes to the partitioned store are required in each case.

Code Fragment 7.31   **Bob was promoted to director of the Computer Center for 1997 (partitioned).**

```
INSERT INTO INCUMBENTS_PAST
SELECT SSN, PCN, START_DATE, DATE '1997-01-01'
FROM INCUMBENTS_PAST
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'

INSERT INTO INCUMBENTS_PAST
SELECT SSN, PCN, DATE '1998-01-01', END_DATE
FROM INCUMBENTS_PAST
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'

UPDATE INCUMBENTS_PAST
SET PCN = 908739
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS_PAST
SET START_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS_PAST
SET END_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1998-01-01'
```

```
-- Now handle changes to INCUMBENTS_CURRENT

INSERT INTO INCUMBENTS_PAST
SELECT SSN, PCN, START_DATE, DATE '1997-01-01'
FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'

INSERT INTO INCUMBENTS_PAST
SELECT SSN, 908739, DATE '1997-01-01', DATE '1998-01-01'
FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
  AND DATE '1998-01-01' < CURRENT_DATE

INSERT INTO INCUMBENTS_PAST
SELECT SSN, 908739, START_DATE, DATE '1998-01-01'
FROM INCUMBENTS_CURRENT
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND DATE '1998-01-01' < CURRENT_DATE

UPDATE INCUMBENTS_PAST
SET PCN = 908739
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'
  AND END_DATE > DATE '1997-01-01'

UPDATE INCUMBENTS_CURRENT
SET START_DATE = DATE '1998-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1998-01-01'

UPDATE INCUMBENTS_CURRENT
SET START_DATE = DATE '1997-01-01'
WHERE SSN = 111223333
  AND START_DATE < DATE '1997-01-01'
```

As with the nonpartitioned store, nonsequenced modifications are easy to express in SQL and difficult to express in English because they are so intimately coupled with the implementation.

7.6 IMPLEMENTATION CONSIDERATIONS

As before, the code fragments were implemented on several systems.

7.6.1 IBM DB2 Universal Database

All of the code fragments compile without error in IBM DB2 UDB. The `DUAL` table in CF-7.2 can be replaced with a WITH TEMP construct.

Code Fragment 7.32   **Bob was assigned the position of associate director of the Computer Center, ensuring the primary key, in DB2 UDB.**

```
INSERT INTO INCUMBENTS
WITH TEMP(SSN, PCN, START_DATE, END_DATE) AS
(VALUES (111223333, 999071, CURRENT DATE, '3000-01-01'))
SELECT *
FROM TEMP AS T
WHERE (NOT EXISTS ( SELECT *
                    FROM  INCUMBENTS AS I2
                    WHERE T.SSN = I2.SSN
                      AND T.PCN = I2.PCN
                      AND I2.END_DATE = '3000-01-01'))
```

7.6.2 Microsoft Access

CF-7.6 (fill gaps in the `POSITIONS` table) cannot be done in Microsoft Access 97 or Access 2000 because these DBMSs do not support the COALESCE operator.

7.6.3 Microsoft SQL Server

All of the code fragments compile without error in Microsoft SQL Server.

7.6.4 Oracle8 Server

CF-7.6 causes difficulties with Oracle8 Server, as it doesn't allow SELECT statements in the target list of an outer SELECT statement. This code fragment can be converted to the following PL/SQL code:

Code Fragment 7.33   **Fill gaps in the** POSITIONS **table for the position of associate direc-
tor of the Computer Center, in the general case, in Oracle.**

```
CURSOR gaps(pcn2 varchar2) IS
  SELECT END_DATE
  FROM POSITIONS p2
  WHERE p2.PCN = pcn2
    AND p2.END_DATE > SYSDATE
    AND p2.END_DATE < TO_DATE('3000-01-01','YYYY-MM-DD')
    AND NOT EXISTS ( SELECT *
                     FROM POSITIONS p3
                     WHERE p3.PCN = pcn2
                       AND p3.START_DATE <= p2.END_DATE
                       AND p2.END_DATE < p3.END_DATE );
new_start_date DATE;
new_end_date DATE;

BEGIN
  OPEN gaps(in_pcn);
  LOOP
  BEGIN
    FETCH gaps INTO new_start_date;
    EXIT WHEN gaps%NOTFOUND;

    SELECT NVL(MIN(p2.START_DATE),TO_DATE('3000-01-01','YYYY-MM-DD'))
    INTO new_end_date
    FROM POSITIONS p2
    WHERE p2.PCN = in_pcn
      AND p2.START_DATE > new_start_date;

    INSERT INTO POSITIONS
      ( pcn, JOB_TITLE_CODE1, START_DATE, END_DATE )
    VALUES (in_pcn, NULL, new_start_date, new_end_date );
    END;
  END LOOP;
  CLOSE gaps;
END;
```

The cursor is the WHERE clause of the original INSERT statement. The WHERE
clause in the nested select inside the original COALESCE expression is implemented
here as a select on the rows returned by the cursor, with COALESCE implemented
with NVL. The insert is then performed.

7.6.5   UniSQL

CF-7.6 causes slight difficulties with UniSQL, because COALESCE cannot take a SELECT statement as an argument. Also, as UniSQL does not support CURRENT_DATE, we use today's date instead.

Code Fragment 7.34   **Fill gaps in the** POSITIONS **table for the position of associate director of the Computer Center, in the general case, in UniSQL.**

```
INSERT INTO POSITIONS
SELECT 999071, NULL, END_DATE,
     (SELECT COALESCE(MIN(START_DATE), DATE '01/01/3000')
      FROM POSITIONS AS P2
      WHERE P2.PCN = 999071
        AND P2.START_DATE > P.START_DATE)
FROM POSITIONS AS P2
WHERE P2.PCN = 999071
  AND P2.END_DATE > DATE '1/16/1998'
  AND P2.END_DATE < DATE '01/01/3000'
  AND NOT EXISTS ( SELECT *
                   FROM POSITIONS AS P3
                   WHERE P3.PCN = 999071
                     AND P3.START_DATE <= P2.END_DATE
                     AND P2.END_DATE < P3.END_DATE)
```

7.6.6   CD-ROM Materials

The CD-ROM contains the code fragments in this chapter in IBM DB2 UDB, Microsoft Access 97, Microsoft SQL Server, Sybase SQLServer, Oracle8 Server, and UniSQL.

7.7   SUMMARY

We have examined how to convert various nontemporal modifications into current and sequenced modifications. Insertions are not too troublesome. Additional predicates are required to ensure sequenced uniqueness and referential integrity. Alternatively, gaps can be filled in the referenced table, which is analogous to adding the key values in the nontemporal case.

Nontemporal deletions and updates are mapped into a series of SQL statements. Current modifications can be viewed as sequenced modifications with a period of applicability from "now" to "forever." The portion of each row's period of validity that is outside the period of applicability must be retained. Often this involves inserting new rows. Deleting a short period from the middle of a row will result

in two remaining rows; updating such a period will result in an initial unchanged row, an updated row, and a final unchanged row. Keeping all the cases straight requires a sophisticated dance between as many as five statements for a simple update (CF-7.18).

In showing how integrity constraints can be ensured within the modification, we considered for each current modification two cases: the general case, where any modification is allowed on the underlying table, and the restricted case, where only current modifications are ever performed on the table.

The modifications found in nontemporal applications are all current modifications.

The restricted case is interesting because it mirrors the behavior of a nontemporal table. After a series of current modifications, the current state of the temporal table (the rows that are valid at "now") will be identical to a nontemporal table upon which the same series of modifications had been applied.

Many applications initially have no temporal component. The need for retaining the history then arises. To extend such an application, the first step is to make one or more of the tables temporal, generally by adding a period timestamp (e.g., start date and end date columns). Then the modifications, which are all current modifications, are converted to manipulate these new columns. This chapter explained this conversion process in detail.

Later, such applications are extended to allow storage and changing of future information. Such modifications are more general than current modifications. When these modifications are permitted to the temporal tables, the current modifications in the original application must be converted to cover the general case.

If the modification mentions other tables, the interaction between the period of validity of the row undergoing modification, the period of applicability of the modification, and the period(s) of validity of the mentioned table(s) must be taken into consideration when translating the modification.

Temporal partitioning finesses the problem of storing currently valid data by separating this data from old data, storing the former in a table known as the current store and the latter in a table known as the history store. Some queries and updates are simplified under this arrangement; others are lengthened.

7.8   READINGS

Most temporal database research has focused on queries; modification is often handled as an afterthought. Some proposals for temporal query languages have considered extended modification statements: HSQL [84], TQuel [86, 87], and TRM [5]. Implementing modification of temporal tables has received no attention. In fact, Tansel's book [102] has a 170-page part on implementation, with nary a word on modifications.

Myrach et al. [72] discuss how to ensure keys and referential integrity using the constructs of the TSQL2 temporal query language [91]. A later paper [4] provides various mechanisms for ensuring sequenced referential integrity in an extended relational model.

Ahn's work with the author is perhaps the most thorough investigation of temporal partitioning [1]. Clifford et al. propose storing now-relative values (such as `CURRENT_DATE - INTERVAL '1' DAY`) in the database [23].

As discussed in Section 5.7, each transaction containing a modification should reset the mode of temporal assertions to deferred, say, with `SET CONSTRAINTS ALL DEFERRED`.

David Landes argues persuasively that a *digital* perception of time, encouraged by the stepwise movement of the clock hand in the town's clock tower and the subsequent need for skilled craftsmen who could assemble these clockworks, was one of the major advances that "turned Europe from a weak, peripheral, highly vulnerable outpost of Mediterranean civilization into a hegemonic aggressor. Time measurement was at once a sign of new-found creativity and an agent and catalyst in the use of knowledge for wealth and power" [65, p. 12].

OVERVIEW

A tracking log captures the sequence of modifications that have been applied to a single table, the table being tracked. The tracking log allows the monitored table to be reconstructed as of any time in the past. This feature can be used to undo inadvertent modifications or to restore the table to a previous, consistent state. Of course, the tracking log can also be used in informal or formal audit procedures.

Different organizations of the tracking log result from the varying assumptions and constraints on the monitored table. The schema of the tracking log comprises the columns of the monitored table, along with a timestamp column (a single datetime specifying when the modification occurred) and sometimes an operation code (insert, delete, update) and a transaction identifier. Triggers can be used to maintain the tracking log, without necessitating changes to the application code. A tracking log can contain before-images (the row before the indicated change occurred), after-images, or a combination of both.

The different organizations of the tracking log are coupled with various reconstruction algorithms. Achieving fully accurate transaction semantics for the reconstructed table turns out to be surprisingly difficult; various restrictions on how the monitored trail is modified can simplify the reconstruction algorithm dramatically.

Retaining a Tracking Log

Nigel Corbin, a Welshman who speaks the King's English with a clipped precision underscoring his doctorate in theoretical physics, provides technical support for Schlumberger's GeoQuest division. In April 1996, we talked in Jakarta, at a meeting of FINDER support personnel who had flown in from Schlumberger's wide-flung offices: Perth, Kuala Lumpur, Mexico City. Looking out from GeoQuest's 16th-floor offices in the "Golden Triangle" of this sprawling Indonesian capital of approximately 10 million people, the eye encounters office buildings, five-star hotels, and foreign embassies in every direction, emphasizing the explosive growth over the past two decades and obscuring the crushing poverty still found in other parts of the city. A year later, the then-unthinkable would occur: Suharto resigned, and a massive economic and social upheaval commenced.

Nigel is based on the other side of the world, in Gatwick, near London. Actually, his office is *at* the Gatwick airport, which presented challenges to the building's architect. The windows have 6 inches of double-paned glass, to acoustically isolate the inhabitants from the shrill whine of departing jet planes. An enclosing wire mesh absorbs the microwave emissions from the airport radar, which otherwise would impart a vague warmth on each sweep. There are no architectural solutions for near encounters of planes that occasionally make tight turnarounds for emergency landings due to equipment malfunction, the closest thus far being a miniscule 1 meter.

When we met, Nigel had other concerns. One of his clients called him to report that all of the North Sea wells had been relocated to the Pacific Ocean. This was quite disturbing news! Nigel rushed to the customer site and verified that the wells were indeed showing up in the wrong location on the digitized maps generated by FINDER. Upon further investigation, Nigel determined that the wells' coordinates were fine, but that these coordinates were being interpreted using the incorrect units. FINDER includes a PROJECTIONS table, containing the columns PROJECTION_ID (the primary key), PROJECTION_NAME, PROJECTION_TYPE (an integer between 0 and 20), SPHEROID_CODE (the U.S. Geologic Survey (USGS) spheroid code, an integer

between 0 and 18), PROJECTION_UOM (the spheroid units, an integer between 0 and 5), and ZONE_CODE (the USGS zone code, a number between 0 and 5400). Someone had (perhaps inadvertently) changed one of the codes for the projection used by the North Sea wells, by executing an ill-advised update on this table. Finding the incorrect column value was tedious. While this table is small, containing several tens of rows, it took Nigel the better part of the morning to reconstruct the table's correct contents.

With this problem fixed, Nigel then considered the more basic issue: what specific changes had been made to this table, and why? One option would be to revise the permissions on this table to disallow updates on the table, thereby ensuring that the table remained consistent. However, this alternative was unacceptable to the client, as there were valid situations in which one or more rows would need to be changed. Nigel decided that a tracking log for this table would be desirable, should the circumstances recur.

8.1 DEFINING THE TRACKING LOG

A *tracking log* specifies the sequence of modifications to a single table, the *monitored* table. The tracking log records the fact that a modification had been applied, as well as the data involved in the modification. Usually it contains additional information about the change, such as when the modification occurred, or who performed it, or which task or transaction effected the change. The tracking log permits the contents of the monitored table to be *reconstructed* as of any time in the past.

> A tracking log retains the past states of a table without impacting the monitored table. As differentiated from a valid-time table, which models the state of the enterprise over time, a tracking log captures the state of the monitored table itself over time.

A tracking log differs from the valid-time tables discussed in previous chapters. Those tables modeled the state of the enterprise over time. In contrast, a tracking log captures the state *of the modified table itself* over time.

We first define a new table, P_Log, which will contain the tracking log for the PROJECTIONS table.

Code Fragment 8.1   **Create the tracking log table.**

```
CREATE TABLE P_Log (
PROJECTION_ID  INT,
PROJECTION_NAME  CHAR(10),
PROJECTION_TYPE  INT,
SPHEROID_CODE  INT,
PROJECTION_UOM  INT,
ZONE_CODE  INT,
When_Changed  DATE,
PRIMARY KEY (PROJECTION_ID, When_Changed))
```

All but the final column are from the PROJECTIONS table. The When_Changed column indicates the date on which the values in the row were removed or changed in the PROJECTIONS table. Hence, these are the old values; the current values can be found in the monitored table. In fact, the value of the When_Changed column can never exceed "now." That When_Changed is of type DATE implies that each specific projection will be updated at most once per day (we will relax this assumption later).

The schema of a tracking log comprises the columns of the monitored table, along with a single timestamp column. Its key is simply the primary key of the monitored table and the timestamp column.

Recall from Section 5.3 that the (sequenced) primary key of a valid-time table cannot be composed from its columns; an assertion is required. Such is not the case here. The key of the tracking log is simply the key of the monitored table (PROJECTION_ID) along with the When_Changed column. The reason for this is that the primary key constraint on the monitored table implies a current key constraint on the tracking log. Since only current modifications (in fact, current insertions) are applied to the log, a current key constraint becomes a sequenced key constraint.

A tracking log can also contain auxiliary columns, such as Who_Changed, that provide additional information on the transaction that is updating the monitored table. However, if triggers are used to maintain the tracking log, those triggers must have access to this auxiliary information. SQL-92's SYSTEM_USER is helpful here; other, application-dependent information is harder to make available to the trigger.

As an aside, we saw in Chapter 5 that the ZPSOS_COMPENSATION_HISTORY table included a CHRONOLOGY_KEY column, which is effectively a transaction timestamp. That table contained both the current information and the tracking log.

The benefit of a separate P_Log table is that the PROJECTIONS table remains as before, and all the code that uses that table still works as before. FINDER is a large, complex system. If Nigel had to examine all of the code that accessed this table, this additional work would probably have convinced him to abandon adding the tracking log in the first place.

However, the code that *modifies* the PROJECTIONS table is still of concern. We wish to also retain that code as is. This can be done by maintaining the P_Log table via triggers, working behind the scenes. We define two triggers, one each for DELETE and UPDATE, on the monitored table.

Code Fragment 8.2   **Triggers for maintaining the** P_Log **table.**

```
CREATE TRIGGER Delete_PROJECTIONS
AFTER DELETE ON PROJECTIONS FOR EACH ROW
INSERT INTO P_Log VALUES
(OLD.PROJECTION_ID,
OLD.PROJECTION_NAME,
```

```
OLD.PROJECTION_TYPE,
OLD.SPHEROID_CODE,
OLD.PROJECTION_UOM,
OLD.ZONE_CODE,
CURRENT_DATE)

CREATE TRIGGER Update_PROJECTIONS
AFTER UPDATE ON PROJECTIONS FOR EACH ROW
INSERT INTO P_Log VALUES (OLD.PROJECTION_ID,
OLD.PROJECTION_NAME,
OLD.PROJECTION_TYPE,
OLD.SPHEROID_CODE,
OLD.PROJECTION_UOM,
OLD.ZONE_CODE,
CURRENT_DATE)
```

In both cases, the values stored in the tracking log table are the *old* values. In the case of update, the new values may be found in the PROJECTIONS table. The value of the When_Changed column is "now." Again, we assume that only one modification is applied each day. Since inserted values can also be found in the monitored table, there is no need for an INSERT trigger (we also return to this assumption later in this chapter).

Triggers allow the tracking log to be maintained automatically, without necessitating changes to the application code.

These triggers assume that no other modifications are made to the P_Log table, other than through the INSERT, DELETE, and UPDATE statements. If triggers were already present on the PROJECTIONS table, and if the DBMS allowed only one trigger per operation per table, then the existing triggers would need to be merged with those defined above.

With this organization, Nigel can confidently install the tracking log with a simple table definition and two trigger definitions. No existing code is impacted. The tracking log is maintained entirely as a side effect of modifications to the monitored table.

8.2   QUERIES

Assume that the following transactions have been executed on the PROJECTIONS table (here we show the values of only the table's key column and one additional column).

1. Insert projection 1 with a type of 12 on January 1, 1996.
2. Insert projection 2 with a type of 10 on January 1, 1996.

Table 8.1 The `PROJECTIONS` **table.**

PROJECTION_ID	PROJECTION_TYPE
1	12
2	14
3	11

Table 8.2 The `P_Log` **table.**

PROJECTION_ID	PROJECTION_TYPE	When_Changed
5	18	1996-02-03
2	10	1996-03-20
3	15	1996-05-28
2	13	1996-06-17
4	17	1996-07-12

3. Insert projection 3 with a type of 15 on January 1, 1996.
4. Insert projection 4 with a type of 17 on January 1, 1996.
5. Insert projection 5 with a type of 18 on January 1, 1996.
6. Delete projection 5 on February 3, 1996.
7. Update projection 2 to a type of 13 on March 20, 1996.
8. Update projection 3 to a type of 11 on May 28, 1996.
9. Update projection 2 with a type of 14 on June 17, 1996.
10. Delete projection 4 on July 12, 1996.

After these transactions have executed, the current contents of the `PROJECTIONS` table are shown in Table 8.1. The contents of the P_Log table are shown in Table 8.2.

Note that Table 8.2 will be ordered on `When_Changed`: each trigger adds a new row to the table with a `When_Changed` time greater than all existing values. Note also that the tracking log is append-only; no rows are ever updated or deleted. Finally, observe that the tracking log can contain multiple rows for a particular `PROJECTION_ID` if that value was updated several times.

We first consider extracting prior states, then examine other kinds of queries.

8.2.1 Extracting a Prior State

Extracting a prior state involves looking at both the monitored table and the tracking log.

The tracking log table allows us to reconstruct the state of the `PROJECTIONS` table at any day in the past. We wish to reconstruct the table as of April 1, 1996, that is, to see the table as it existed between transactions 7 and 8. Projection 1 was never modified (it does not appear in the tracking log), so the value in

Circadian Clocks

> Across the evening sky
> All the birds are leaving
> But how can they know
> It's time for them to go
>
> . . . .
>
> So come the storms of winter
> And then the birds in spring again
>
> —Sandy Denny, "Who Knows Where the Time Goes"

Barbara Kingsolver, in her book *High Tide in Tucson*, realizes that her pet crab is still responding to the tidal cycle of the California coast from where it was taken, several hundred miles to the west. The crab would become active only at certain times, even though there was no longer any environmental stimuli to indicate the tides. Tidal rhythms are common in many animals living in tide-washed coastlines: crabs (locomotion, oxygen consumption, and color change), fish (swimming), and bivalves (shell gapping).

On most coastlines there are two high and two low tides each *lunar day*, which is the 24 hour and 51 minute average interval between successive moonrises, due to the gravitational and centrifugal forces generated between the moon and the earth. Hence, the average high tide–to–high tide interval is 12.4 hours.

It appears that crabs possess not one 12.4-hour biological clock, but rather two biological clocks, each running at twice that interval, tightly coupled 180 degrees out of phase, to ensure a stable interval of 12.4 hours between peaks. One benefit of two clocks is that deviations of the tides from a 12.4-hour period can be as great as ± 90 minutes, whereas deviations from a 24.8-hour period are much smaller: ± 10 minutes.

the monitored table is fine. Projection 2 was modified twice, on March 20 and on June 17, and so the *old* value stored in the tracking log on June 17 is the desired one. Projection 3 was modified once, on May 28, and so the old value for that date is the desired one. Projection 4 was deleted on July 12 (it is present in the tracking log, but not in the monitored table); we use the old value for that date. Projection 5 was deleted on February 3, and so should not be present in the reconstructed table. The table as of April 1, 1996, is thus as shown in Table 8.3.

There are two basic cases. Here, the date for which the state is to be reconstructed is termed the *as-of date*.

Reconstruct a previous state of a monitored table, as of a specified date

Each row of a prior state

  was not deleted or updated after the as-of date, and thus is in the monitored table,
  or was deleted or updated after the as-of date, and thus is in the tracking log.

Table 8.3 The monitored table as of April 1, 1996.

PROJECTION_ID	PROJECTION_TYPE
1	12
2	13
3	15
4	17

Code Fragment 8.3   **Reconstruct the** PROJECTIONS **table as of April 1, 1996.**

```
SELECT *
FROM PROJECTIONS AS P
WHERE NOT EXISTS (SELECT *
                  FROM P_Log AS A
                  WHERE P.PROJECTION_ID = A.PROJECTION_ID
                    AND A.When_Changed > DATE '1996-04-01')
UNION
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
       SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_Log AS A
WHERE When_Changed = (SELECT MIN(When_Changed)
                      FROM P_Log AS A2
                      WHERE A.PROJECTION_ID = A2.PROJECTION_ID
                        AND A2.When_Changed > DATE '1996-04-01')
```

This example illustrates the structure of the reconstruction process. The row from the monitored table is to be retained if there does not exist a change record in the tracking log with a When_Changed after the as-of date. Otherwise, the old value is extracted from the change record with a When_Changed date that is the first one to appear after the as-of date.

The approach presented thus far in this chapter is adequate for identifying what changes were made to the PROJECTIONS table, and for undoing the changes that were incorrect by first reconstructing the table as of the date it was last known to be correct. It has the benefits of not impacting the existing FINDER code and of adding little to the storage requirements (recall that the PROJECTIONS table is small and that changes to this table are infrequent).

Nigel was called back a few months later to correct the same problem. This time it took him only a few minutes to reconstruct the monitored table (by running the above code) and identify the incorrect values. Additionally, he had in hand the date the incorrect change was effected and could use this information to narrow down who made the change. Nigel has used the tracking log approach several times for other tables that users were incorrectly updating.

8.2.2   Other Queries

We now turn to other kinds of queries. Current queries on tracking logs are trivial: just perform them on the monitored table as before.

Code Fragment 8.4   **List the information on projection 5.**

```
SELECT *
FROM PROJECTIONS
WHERE PROJECTION_ID = 5
```

Future queries are not possible: we have no way of knowing what the monitored table will look like in the future.

Queries on past states of a monitored table are easiest to express via a reconstruction view.

Past queries are quite useful on monitored tables; indeed, tables are audited precisely to enable such queries. The above SELECT statement (CF-8.3) extracts a prior state of the monitored table. More extensive queries on this state (say, to prepare a report as of the end of the calendar year) are best realized by forming a view.

Code Fragment 8.5   **Reconstruct the** PROJECTIONS **table as of April 1, 1996, as a view.**

```
CREATE VIEW April_PROJECTIONS
    ( PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE)
AS (SELECT ...
    )
```

Here, the SELECT statement in CF-8.3 is used to create a (snapshot) view, which can then be used in queries.

Code Fragment 8.6   **List the information on projection type 12 as of April 1, 1996.**

```
SELECT *
FROM April_PROJECTIONS
WHERE PROJECTION_TYPE = 12
```

8.2.3   Converting to a State Table*

One way to perform sequenced and nonsequenced queries on a tracking log is to convert it to a *transaction-time state table*. As with valid-time state tables, we time-stamp each row with a closed-open period, represented with two transaction time-stamp columns (see Table 8.4). Note that the maximum stop time is "now" (January 16, 1998). Rows that are in the current table (and thus are current now) will have a stop time of "now" in the transaction-time state table. Rows that are not in the current table (and thus were deleted) will have a stop time in the past.

Table 8.4 The tracking log as a transaction-time state table, `PROJECTIONS_State`.

`PROJECTION_ID`	`PROJECTION_TYPE`	`Start_Date`	`Stop_Date`
1	12	1996-01-01	1998-01-16
5	18	1996-01-01	1996-02-03
4	17	1996-01-01	1996-07-12
2	10	1996-01-01	1996-03-20
3	15	1996-01-01	1996-05-28
2	13	1996-03-20	1996-06-17
2	14	1996-06-17	1998-01-16
3	11	1996-05-28	1998-01-16

There are four basic cases to consider, differentiated by the timestamp of the resulting row in the state table:

1. The initial row never changed. The state table includes the initial row, valid from January 1, 1996 to the present.
2. The initial row was deleted. The state table includes the initial row, valid from January 1, 1996 to the tracking log entry's `When_Changed` date. This timestamp also results from the case where the initial row was modified at least once, with the earliest row in the audit table contributing a row in the state table valid from January 1, 1996 to the tracking log entry's timestamp.
3. The initial row was modified at least once, with the resulting row drawn from the audited table, valid from the *last* audit entry's timestamp to the present.
4. Here the initial row was modified several times. Each intermediate value results in a state table row valid from the earlier change date to the later change date. This timestamp also results if the row was deleted.

Each case contributes a SELECT statement, with the result being the UNION of these intermediate results, as the cases are disjoint.

Code Fragment 8.7   Convert `P_Log` **to a transaction-time state table.**

```
CREATE VIEW PROJECTIONS_State
  (PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
     SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
     Start_Date, Stop_Date)
```

```
AS (
  -- Case 1
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    DATE '1996-01-01', CURRENT_DATE
FROM PROJECTIONS
WHERE NOT EXISTS ( SELECT * FROM P_Log
                     WHERE P_Log.PROJECTION_ID = PROJECTIONS.PROJECTION_ID)
UNION
  -- Case 2
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    DATE '1996-01-01', When_Changed
FROM P_Log AS A1
WHERE NOT EXISTS ( SELECT * FROM P_Log AS A2
                     WHERE A1.PROJECTION_ID = A2.PROJECTION_ID
                       AND A1.When_Changed > A2.When_Changed)
UNION
  -- Case 3
SELECT A1.PROJECTION_ID, A1.PROJECTION_NAME, A1.PROJECTION_TYPE,
    A1.SPHEROID_CODE, A1.PROJECTION_UOM, A1.ZONE_CODE,
    When_Changed, CURRENT_DATE
FROM P_Log AS A1, PROJECTIONS
WHERE A1.PROJECTION_ID = PROJECTIONS.PROJECTION_ID
  AND NOT EXISTS ( SELECT * FROM P_Log AS A2
                     WHERE A1.PROJECTION_ID = A2.PROJECTION_ID
                       AND A1.When_Changed < A2.When_Changed)
UNION
  -- Case 4
SELECT A1.PROJECTION_ID, A1.PROJECTION_NAME, A1.PROJECTION_TYPE,
    A1.SPHEROID_CODE, A1.PROJECTION_UOM, A1.ZONE_CODE,
    A0.When_Changed, A1.When_Changed
FROM P_Log AS A0, P_Log AS A1
WHERE A0.PROJECTION_ID = A1.PROJECTION_ID
  AND A0.When_Changed < A1.When_Changed
  AND NOT EXISTS ( SELECT *
                     FROM P_Log AS M
                     WHERE M.PROJECTION_ID = A1.PROJECTION_ID
                       AND M.When_Changed < A1.When_Changed
                       AND M.When_Changed > A0.When_Changed)
)
```

Sequenced and nonsequenced queries on a tracking log are best stated on a view that extracts the states as a transaction-time state table.

Given the transaction-time state table defined by this view, current, sequenced, and nonsequenced queries can be performed analogously to such queries on valid-time state tables. The semantics of such queries are somewhat different because these tables utilize transaction time. Rather than being of the form "Give the history of ...," transaction-time sequenced queries are of the form "When was it recorded that ..."

Code Fragment 8.8   **When was it recorded that a projection had a type of 17?**

```
SELECT PROJECTION_ID, PROJECTION_TYPE, Start_Date, Stop_Date
FROM PROJECTIONS_State
WHERE PROJECTION_TYPE = 17
```

This query returns the following table:

PROJECTION_ID	PROJECTION_TYPE	Start_Date	Stop_Date
4	17	1996-01-01	1996-07-12

This transaction sequenced selection and projection query should be compared with the similar valid-time sequenced queries CF-6.6 and CF-6.7.

Other examples of sequenced and nonsequenced queries are given in Section 9.3.

8.3   MODIFICATIONS

Current modifications are trivial: these are modifications on the monitored table, with the triggers behind the scene ensuring that the tracking log is kept consistent.

Code Fragment 8.9   **Insert a projection with an ID of 6.**

```
INSERT INTO PROJECTIONS (PROJECTION_ID, PROJECTION_NAME,
      PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
      ZONE_CODE)
VALUES (6, 'New Projection', 22, 14, 93, 4)
```

Code Fragment 8.10   **Delete projection 2.**

```
DELETE FROM PROJECTIONS
WHERE PROJECTION_ID = 2
```

Code Fragment 8.11   **Change the type of projection 1 to 43.**

```
UPDATE PROJECTIONS
SET PROJECTION_TYPE = 43
WHERE PROJECTION_ID = 1
```

Sequenced and nonsequenced modifications are not allowed on tracking logs.

Sequenced and nonsequenced modifications are not allowed on tracking logs because such modifications destroy the semantics of a tracking log in two senses. If a past state were modified, we no longer would be able to reconstruct the state as it was stored at that time, thereby removing the prime motivation for a tracking log. If a state with a stop date after "now" were inserted, then a future state could not later be reconstructed.

8.4   PERMITTING INSERTIONS

The above code makes several assumptions, which we now address. The first assumption is that the table was originally constituted with insertions, and thereafter the only changes were updates and deletions. Any insertions that do occur will be assumed (in the reconstruction algorithm) to have been executed when the table was defined. Say that an insertion of a new projection, number 6, was made on June 29, 1996. CF-8.3 would include this projection in the state reconstructed as of April 1, 1996, via the first SELECT statement. A second assumption is that insertions are not allowed after a deletion.

To remove the first assumption, we need to include insertions to the tracking log, so that they will be dated correctly. This is done via a trigger on INSERT.

Code Fragment 8.12   **Insert trigger for maintaining the** P_Log **table.**

```
CREATE TRIGGER Insert_PROJECTIONS
 AFTER INSERT ON PROJECTIONS FOR EACH ROW
INSERT INTO P_Log VALUES   (NEW.PROJECTION_ID,
NEW.PROJECTION_NAME,
  NEW.PROJECTION_TYPE,
NEW.SPHEROID_CODE,
  NEW.PROJECTION_UOM,
NEW.ZONE_CODE,
  CURRENT_DATE)
```

Having the insertions in the tracking log increases the size of the tracking log considerably. In the first approach, projections were in P_Log only if the projection was updated or deleted. The tracking log table might be much smaller than

The Sothic Cycle

Early calendars were lunar, perhaps because the lunar cycle, at slightly less than 30 days, is more easily observed than the solar cycle, at about 365.25 days. The Egyptian calendar contained 12 months of 30 days each. However, the annual flooding of the Nile required reconciliation with the solar year, so five intercalated days were included. The lack of a leap day meant that their civil lunar New Year retrogressed by 1/4 day each year with respect to the religious solar New Year (defined as when the Dog Star, named Sothis by the Greeks, rose with the sun). The two New Years coincide once every 1,460 years, termed the *Sothic cycle*.

It is known that in 139 A.D., the two New Years did coincide, so, working backward, historians have concluded that the Egyptian calendar originated in 4241 B.C.E. (This is being written in 1998, during the 399th year of the fifth Sothic cycle.)

Table 8.5 Including insertions in the P_Log **table.**

PROJECTION_ID	PROJECTION_TYPE	When_Changed
1	12	1996-01-01
2	10	1996-01-01
3	15	1996-01-01
4	17	1996-01-01
5	18	1996-01-01
5	18	1996-02-03
2	10	1996-03-20
3	15	1996-05-28
2	13	1996-06-17
4	17	1996-07-12

the monitored table if the monitored table exhibited low volatility. In this new approach, every projection in PROJECTIONS is also in P_Log, as illustrated by Table 8.5. The tracking log is now always larger than the monitored table. However, this additional information allows us to reconstruct the monitored table correctly even in the presence of later insertions.

The time sequence of an object records the evolution over time of that object.

To reconstruct the PROJECTIONS table at a previous time t, there are five cases to consider, shown in Figure 8.1. In this figure, the sequence of operations for a single projection is shown as a progression along time. This progression consists of an insertion, followed by zero or more updates, followed optionally by a deletion. Such a sequence is called the *time sequence* of the projection.

The five cases indicate possible as-of times. Case 1 occurs when the as-of time t is before the initial insertion. The projection should not appear in the reconstructed table. In Case 2, t is between the time of insertion and the first update. Either the
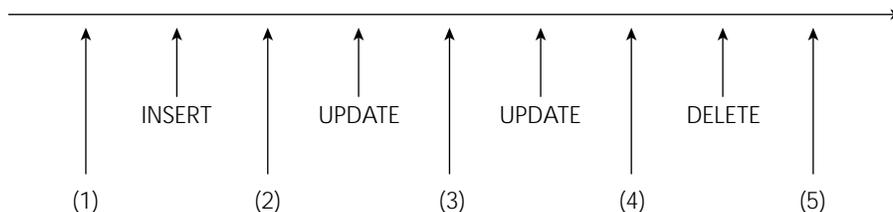
Figure 8.1 Cases for tracking log reconstruction.

first value of the tracking log (from the INSERT trigger) or the second value (from the UPDATE trigger) should be used in the reconstructed table; the two values are identical. In Case 3, *t* occurs between two updates; the value from the tracking log corresponding to the second update should be used. In Case 4, *t* occurs between the last update and the tuple being deleted; the value in the tracking log inserted by the DELETE trigger should be used here. In Cases 2 through 5, the value stored in `PROJECTIONS` may also be used. In Case 5, *t* occurs after the deletion.

   In Cases 4 and 5, the as-of time is after the last time in the tracking log. We differentiate these cases by the presence of the projection in the monitored table. If the projection is not in `PROJECTIONS`, and the timestamp of the last tracking log match is older than the as-of time (Case 5), then that projection was deleted and should not appear in the reconstructed value. Otherwise (Case 4), the value in `PROJECTIONS` should be used.

If arbitrary insertions are allowed, the reconstruction algorithm becomes more complex.

   In the following code fragment, we merge Cases 2 through 4 in the first SELECT. We locate the first entry in the tracking log after the as-of time *t* (the NOT EXISTS and the middle predicate ensure that this entry is the first) and use its values for the columns. The EXISTS ensures that this is not Case 1. The second SELECT is Case 4, where no DELETE or UPDATE followed the as-of time. The result is the same as before, shown in Table 8.3.

Code Fragment 8.13   **Reconstruction algorithm 2, again, as of April 1, 1996.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_Log AS A
WHERE NOT EXISTS ( SELECT *
                   FROM P_Log AS A2
                   WHERE A.PROJECTION_ID = A2.PROJECTION_ID
                     AND DATE '1996-04-01' < A2.When_Changed
AND A2.When_Changed < A.When_Changed)
AND DATE '1996-04-01' < A.When_Changed
  AND EXISTS ( SELECT *
```

```
                        FROM P_Log AS A3
                        WHERE A.PROJECTION_ID = A3.PROJECTION_ID
                          AND A3.When_Changed <= DATE '1996-04-01')
UNION
SELECT *
FROM PROJECTIONS AS P
WHERE DATE '1996-04-01' > (SELECT MAX(When_Changed)
                                 FROM P_Log AS A
                                 WHERE P.PROJECTION_ID = A.PROJECTION_ID)
```

The code to create a transaction-time state table is also complicated by this change.

8.5 BACKLOGS

A *backlog* is a tracking log with the modification operation (insert, delete, update) explicitly identified.

While the trigger in CF-8.12 allows insertions to the `PROJEC-TIONS` table other than at the very beginning, it still does not allow insertions after a deletion. This is because a deletion followed later by an insertion of the same projection is indistinguishable in the tracking log from two sequential updates.

To differentiate a deletion from an update, we add a column to `P_Log` that indicates which operation was done. Such a tracking log is called a *backlog*.

```
ALTER TABLE P_Log ADD COLUMN Operation CHAR(1)
```

The triggers must be changed to also store the `Operation` code, `I`, `D`, and `U`.

Table 8.6 shows an example of the `P_Log` table, with some insertions after deletions. This differs from the list of transactions in Section 8.2 in that the two latest insertions, which were not allowed before, are included here. As before, we show only a few of the columns in the backlog.

There are three basic cases to consider in the reconstruction algorithm.

1. If the as-of time *t* occurs before the first insertion of a projection's time sequence, the projection is not included.
2. If *t* occurs after the last operation of a projection, the projection is included only if the operation is not a deletion, using the current value in the `PROJECTIONS` table.
3. Otherwise, *t* occurs between two operations, and Table 8.7 must be consulted to determine the appropriate action: to use the value from the first operation or to use the value stored from the second operation. The entries marked "illegal" cannot occur. For example, it is impossible for a projection to be reinserted

Table 8.6 The P_Log **backlog.**

PROJECTION_ID	PROJECTION_TYPE	When_Changed	Operation
1	12	1996-01-01	I
2	10	1996-01-01	I
3	15	1996-01-01	I
4	17	1996-01-01	I
5	18	1996-01-01	I
5	18	1996-02-03	D
2	10	1996-03-20	U
5	19	1996-04-09	I
3	15	1996-05-28	U
2	13	1996-06-17	U
4	17	1996-07-12	D
4	18	1996-08-30	I

Table 8.7 Reconstruction action.

		second (youngest after *t*)		
		I	U	D
first	I	illegal	first or second	first or second
	U	illegal	second	second
(oldest before *t*)	D	nothing	illegal	illegal

because PROJECTION_ID is the key for the monitored table. It is also impossible for a deletion to follow a deletion because there is no projection to remove.

This analysis indicates that two SELECT statements are required, the first for Case 2 and the second for the options in Table 8.7. In this table, an illegal entry simply cannot occur, and so needs not be accommodated in the reconstruction algorithm.

Code Fragment 8.14   **Reconstruction algorithm 3.**

```
  SELECT *
FROM PROJECTIONS AS P
WHERE DATE '1996-04-01' > (SELECT MAX(When_Changed)
          FROM P_Log AS A
          WHERE P.PROJECTION_ID = A.PROJECTION_ID)
UNION
SELECT A2.PROJECTION_ID, A2.PROJECTION_NAME, A2.PROJECTION_TYPE,
    A2.SPHEROID_CODE, A2.PROJECTION_UOM, A2.ZONE_CODE
FROM P_Log AS A, P_Log AS A2
WHERE A.When_Changed < A2.When_Changed
```

```
        AND A.PROJECTION_ID = A2.PROJECTION_ID
        AND NOT EXISTS (SELECT *
                       FROM P_Log AS A3
                       WHERE A.PROJECTION_ID = A3.PROJECTION_ID
                         AND A.When_Changed < A3.When_Changed
                         AND A3.When_Changed < A2.When_Changed)
        AND A.When_Changed < DATE '1996-04-01'
        AND DATE '1996-04-01' < A2.When_Changed
        AND A.Operation <> 'D'
```

A tracking log can contain before-images, after-images, or both, with differing implications for reconstruction.

This code is complex because a combination of *before-images*, where the previous values of the row are stored, and *after-images*, where the new values are stored, appear in P_Log. In particular, before-images are stored by the UPDATE and DELETE triggers, and after-images are stored by the INSERT trigger.

8.6   USING AFTER-IMAGES CONSISTENTLY

The reconstruction algorithm can be simplified considerably if the DELETE and UPDATE triggers use after-images consistently. The former needs not store any values because the after-image is not defined for deletions. The INSERT trigger may be retained from CF-8.12, as it already records the after-image.

Code Fragment 8.15   Triggers for maintaining the P_Log **table, version 2.**

```
CREATE TRIGGER Delete_PROJECTIONS
AFTER DELETE ON PROJECTIONS FOR EACH ROW
INSERT INTO P_Log VALUES (OLD.PROJECTION_ID,
NULL, NULL, NULL, NULL, CURRENT_DATE, 'D')

CREATE TRIGGER Update_PROJECTIONS
AFTER UPDATE ON PROJECTIONS FOR EACH ROW
INSERT INTO P_Log VALUES (NEW.PROJECTION_ID,
NEW.PROJECTION_NAME,
NEW.PROJECTION_TYPE,
NEW.SPHEROID_CODE,
NEW.PROJECTION_UOM,
NEW.ZONE_CODE,
CURRENT_DATE, 'U')
```

Table 8.8 Backlog with after-images.

PROJECTION_ID	PROJECTION_TYPE	When_Changed	Operation
1	12	1996-01-01	I
2	10	1996-01-01	I
3	15	1996-01-01	I
4	17	1996-01-01	I
5	18	1996-01-01	I
5	*NULL*	1996-02-03	D
2	*13*	1996-03-20	U
5	19	1996-04-09	I
3	*11*	1996-05-28	U
2	*14*	1996-06-17	U
4	*NULL*	1996-07-12	D
4	18	1996-08-30	I

Table 8.9 Reconstruction action.

		second		
		I	U	D
	I	illegal	first	first
first	U	illegal	first	first
	D	nothing	illegal	illegal

For the same transactions discussed previously, the backlog shown in Table 8.8 results.

In comparison with Table 8.6, note that both have the same number of rows. In fact, the only difference is the value of the PROJECTION_TYPE column (the new values are indicated in italics). The previous representation exhibited redundancy in that consecutive I-U pairs in a time sequence have identical values for that column; the entries for projection 2 on January 1 and March 20 provide an example. The same holds for consecutive I-D pairs, such as the entries for projection 5 on January 1 and February 3.

Using after-images consistently simplifies the reconstruction algorithm considerably.

Consistently using after-images enables a shorter reconstruction algorithm. Let us examine the action table for after-images (Table 8.9). Note that if the Operation is I or U, the same action occurs. Also note that if there is only one operation before the as-of time, the same action occurs here also. Hence, there is no need to consult the PROJECTIONS table, and the reconstruction algorithm becomes a single SELECT statement.

Code Fragment 8.16 **Reconstruction algorithm with after-images.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_Log AS A
WHERE A.When_Changed =(SELECT MAX(A2.When_Changed)
                       FROM P_Log AS A2
                       WHERE A.PROJECTION_ID = A2.PROJECTION_ID
                         AND A2.When_Changed < DATE '1996-04-01')
  AND A.Operation <> 'D'
```

Using only after-images also changes the transaction-time state view.

Convert a backlog containing after-images to a transaction-time state table

Each row of the state table

> starts with an I or U operation and ends with a U or D operation,
> or starts with an I or U operation and is not subsequently modified, in which case its stop time is "now."

Code Fragment 8.17 **Convert the backlog to a transaction-time state table, using after-images.**

```
CREATE VIEW PROJECTIONS_state
  ( PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
     SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
     START_DATE, STOP_DATE)
AS (SELECT A.PROJECTION_ID, A.PROJECTION_NAME, A.PROJECTION_TYPE,
        A.SPHEROID_CODE, A.PROJECTION_UOM, A.ZONE_CODE,
        A.When_Changed, A2.When_Changed
    FROM P_Log AS A, P_Log AS A2
    WHERE A.PROJECTION_ID = A2.PROJECTION_ID
      AND A.When_Changed < A2.When_Changed
      AND A.Operation <> 'D'
      AND NOT  EXISTS ( SELECT *
                        FROM P_Log AS A3
                        WHERE A.PROJECTION_ID = A3.PROJECTION_ID
                          AND A.When_Changed < A3.When_Changed
                          AND A3.When_Changed < A2.When_Changed)
```

```
UNION
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    When_Changed, CURRENT_DATE
FROM P_Log AS A
WHERE A.Operation <> 'D'
  AND NOT  EXISTS ( SELECT *
                    FROM P_Log AS A3
                    WHERE A.PROJECTION_ID = A3.PROJECTION_ID
                      AND A.When_Changed < A3.When_Changed)
)
```

Using after-images consistently also greatly simplifies the conversion of the tracking log to a transaction-time state table.

We make sure each pair of operations in the first SELECT is consecutive by disallowing an intermediate operation.

This code fragment should be compared with CF-8.7, which is over *three* times longer. The comparison emphasizes the value of consistent use of after-images in the tracking log.

As mentioned earlier, once the transaction-time state table is available, sequenced and nonsequenced queries can be performed analogously to such queries on valid-time state tables, as discussed in detail in Chapter 6.

As an example, consider the following query: When was it recorded that a projection had the same USGS zone code as the projection with ID 13447? The first part, "when was it recorded," indicates that we are concerned with transaction time. It also implies that if a particular time is returned, the specified relationship should hold during that time. This indicates a sequenced query, here in transaction time.

Sequenced and nonsequenced queries are best expressed on a transaction-time state table view, rather than on the underlying tracking log.

The fact that two projections are mentioned indicates a (self-) join on the transaction-time state table. CF-6.12 provides the structure for a sequenced join, which we can use here.

Code Fragment 8.18   **List the projections recorded as having the same USGS zone code as the projection with ID 13447.**

```
SELECT S1.PROJECTION_NAME,
    CASE WHEN S1.START_DATE > S2.START_DATE
      THEN S1.START_DATE ELSE S2.START_DATE END,
    CASE WHEN S1.STOP_DATE > S2.STOP_DATE
      THEN S2.STOP_DATE ELSE S1.STOP_DATE END,
FROM PROJECTIONS_state AS S1, PROJECTIONS_state AS S2
WHERE S1.ZONE_CODE = S1.ZONE_CODE
  AND S2.PROJECTION_ID = 13447
  AND S1.PROJECTION_ID <> 13447
  AND (CASE WHEN S1.START_DATE > S2.START_DATE
```

```
      THEN S1.START_DATE ELSE S2.START_DATE END)
 < (CASE WHEN S1.STOP_DATE > S2.STOP_DATE
   THEN S2.STOP_DATE ELSE S1.STOP_DATE END)
```

The resulting query isn't too bad. However, computing this result on the underlying tracking log would have been extremely complex.

When after-images are used consistently, P_Log contains all of PROJECTIONS, which is not true of the prior approaches. It includes this additional information without adding any rows; instead, redundancy is eliminated to provide this information. This allows the current version to be computed even more easily.

Code Fragment 8.19   **Reconstructing the current version.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_Log AS A
WHERE A.When_Changed =(SELECT MAX(A2.When_Changed)
                       FROM P_Log AS A2
                       WHERE A.PROJECTION_ID = A2.PROJECTION_ID)
  AND A.Operation <> 'D'
```

If after-images are used in the tracking log, then the monitored table itself is superfluous.

In fact, the PROJECTIONS table is simply a cached version of the reconstruction as of "now." We could define it as a view, thereby achieving a space savings, as the current information would not be in both the monitored table and the backlog, at the expense of more expensive retrieval of that information.

Code Fragment 8.20   **Defining** PROJECTIONS **as a view on** P_Log**.**

```
CREATE VIEW PROJECTIONS
  ( PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE)
AS (SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
     SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
     FROM P_Log AS A
     WHERE A.When_Changed =(SELECT MAX(A2.When_Changed)
                            FROM P_Log AS A2
                            WHERE A.PROJECTION_ID = A2.PROJECTION_ID)
       AND A.Operation <> 'D'
     )
```

If the PROJECTIONS table is defined as a view, then the user is responsible for updating the backlog directly, as current DBMSs do not permit triggers to be defined on a view.

8.7   TRANSACTION SEMANTICS*

All of the above approaches make three assumptions:

1. No two changes to the same projection occur on a single day, even in different transactions. This assumption derives from the When_Changed column being a DATE.
2. Every transaction commits by the end of the day; that is, a transaction does not span two days. Otherwise the reconstruction algorithm may return a partially completed state—a state that will never be visible to the application, as we will see shortly.
3. Every transaction makes at most one modification to each projection. This assumption is a consequence of PROJECTION_ID and When_Changed forming the primary key for P_Log.

Simple approaches to maintaining the tracking log impose rather harsh constraints on the application.

Violating assumption 1 or 3 will cause the INSERT statement in the trigger to fail, which will abort the entire transaction. In fact, assumption 1 implies no concurrency in the modifications to the PROJECTIONS table, which in many situations is untenable. Violating assumption 2 can cause the reconstruction algorithm to yield an incorrect result, which is also generally untenable.

One obvious attempt to remove these assumptions is to redefine the When_Changed column to be a TIMESTAMP, which has a default precision of microsecond. However, this just changes assumption 1 to one in which no two changes to the same projection can occur on a single microsecond, which, given today's transaction processing rates, is a reasonable one. The concern then arises as to whether the trigger ensures that the When_Changed value is consistent with the serialization order of the transactions that are changing the PROJECTIONS table. To put this another way, was the table reconstructed at a specified time actually visible to transactions executing at that time?

Let us examine two transactions, T_1 and T_2. We assume two-phase row-level locking as our concurrency control mechanism. Transaction T_1 starts on July 1 at 4 P.M. It updates projection 1 at 6 P.M. (18:00) and again at 8 P.M. (Transactions generally complete in a much shorter period of time; here we spread out the time to make this example easier to discuss.) Each update inserts a row into the backlog; that row is write-locked. Transaction T_2 updates projection 2 at 7 P.M. and commits at 9 P.M. T_1 updates projection 3 at 1 A.M. the following day and commits at 2 A.M. Note that T_1 updates projection 1 twice and also spans a day, thereby violating all three assumptions.

Table 8.10 A sample backlog.

	PROJECTION_ID	PROJECTION_TYPE	When_Changed	Operation
T_1	1	10	1996-07-01 18:00	U
T_2	2	12	1996-07-01 19:00	U
T_1	1	14	1996-07-01 20:00	U
T_1	3	13	1996-07-02 01:00	U

Table 8.11 State of the PROJECTIONS **table before 9 P.M.**

PROJECTION_ID	PROJECTION_TYPE
1	5
2	7
3	9

Table 8.12 State of the PROJECTIONS **table between 9 P.M. and 2 A.M.**

PROJECTION_ID	PROJECTION_TYPE
1	5
2	12
3	9

This series of updates will generate the backlog shown in Table 8.10 (here we show the transaction associated with each row, though this information will not be stored in the backlog).

The reconstruction algorithm yields the state of the monitored table at any point in time. Due to concurrent transactions, such as the two above, what we desire is a state that is consistent with the serialization order of the transactions that executed on that table.

Assume that the projection type for projection 1 was initially 5; for projection 2, 7; and for projection 3, 9. At any time before 9 P.M., the state of the PROJECTIONS table is as shown in Table 8.11, reflecting that no (committed) transactions have updated the table.

Recall that transaction T_2 committed at 9 P.M. and T_1 at 2 A.M. So between those two times, the state of the PROJECTIONS table was as shown in Table 8.12, reflecting the changes made by T_2.

After 2 A.M. on July 2, the state of the monitored table has the values shown in Table 8.13, reflecting the changes made by both T_2 and T_1.

Table 8.13 State of the PROJECTIONS **table after 2** A.M.

PROJECTION_ID	PROJECTION_TYPE
1	14
2	12
3	13

Table 8.14 Reconstructed state as of 6:30 P.M.

PROJECTION_ID	PROJECTION_TYPE
1	10
2	7
3	9

The reconstruction algorithm will yield precisely the first state for as-of times before 6 P.M. on July 1 and the third state for times after 2 A.M. on July 2. More interesting is what it does for intermediate as-of times when applied after 2 A.M., July 2. Applying the algorithm with an as-of time of 6:30 P.M., using the backlog shown in Table 8.10, yields Table 8.14.

This differs from the state before 6 P.M. in the type of projection 1, which was changed by transaction T_1. Note, however, that T_1 had not committed by 6:30 P.M. Hence, no other transaction would have seen this state, because it is not consistent with *any* serialization order. We see that reconstruction yields a state containing information from active transactions.

As an aside, what happens if transaction T_1 later aborted, rather than committing? It turns out that we are OK in such situations because all the rows of the backlog inserted by that transaction are automatically removed upon abort, and hence will not be used by the reconstruction algorithm. In the case above, we are running the reconstruction algorithm after July 2, and so we know that T_1 had committed because its updates are in the backlog.

The reconstruction algorithm may yield states inconsistent with serializability.

Returning to 6:30 P.M., what would a concurrent transaction (T_3) see? Transaction T_3 will not have access to projection 1 because that row will be write-locked by T_1 at 6:30 P.M. And we know that T_2 did not read projection 1 because had it tried, it would have blocked on that same lock and would have resumed only at 2 A.M. the next morning, when T_1 committed.

What if transaction T_3 had attempted to read projection 1 using a *dirty read* isolation level, rather than the default *serialization* isolation level? In that case, the write lock on that row is ignored, and T_3 would have seen the projection type as 10.

The Vertical Blanking Interval

That ubiquitous sign of technical ineptitude, the blinking "12:00" on the videocassette recorder, has recently been the source of technological innovation, as VCRs learn how to get the time directly from the video feed. A television frame (30 frames a second) is composed of two fields of 262.5 horizontal scan lines each. The first 21 lines of each field, collectively called the vertical blanking interval (VBI), are hidden (they are the black band you see when your vertical tracking isn't working). The first 9 are used for repositioning the scan, but the remaining 12 lines are available for data transport. Line 21 itself is reserved for closed captioning. PBS stations in the United States use extended data services codes within line 21 to send the current date and time. Some of the newer VCRs scan all frequencies to find the PBS channel, then set their clock automatically from line 21 of the VBI. They also refresh their internal clock, as often as once an hour, from this source, to correct for drift in their internal clock and to make daylight saving time adjustments. This technology renders the VCR the most accurate clock available for home use. Incidentally, the other 11 lines are being considered for VBI data broadcasting; these lines provide an aggregate of about 150 Kbps per channel streaming into every home.

So, to be precise, the reconstruction algorithm yields a state that would be visible to a transaction at the specified time using a dirty read isolation level. This state, though, will contain only changes made by transactions that eventually committed.

What if there are still active transactions? Since the backlog itself is being read by the reconstruction algorithm using the (default) serialization isolation level, the algorithm must place a read lock on the entire backlog, and hence there can be no write locks in place, which implies that there can be no active transactions that have modified the monitored table. Any subsequent transaction that wished to modify the monitored table will attempt to write-lock the backlog, and thus will have to await the completion of the reconstruction algorithm of the other transaction. The effect is to serialize the reconstruction transaction with all other transactions, including other reconstruction transactions.

8.8 REFINEMENTS*

If we wish the reconstruction algorithm to not retrieve dirty data, then we must ensure that such data is not written to the tracking log. The triggers can be modified to delete a previous value, for the projection in question, before inserting the current value. To do so, the trigger needs to know if the value in the tracking log was put there by this transaction, or if it is the committed value by a previous transaction. The former value should be removed, but the latter value must be retained. We add

a transaction identifier to the tracking log to make this distinction. Alternatively, we can augment the reconstruction algorithm to ignore dirty data by using only the last record of each projection, which requires ordering the records inserted by a transaction. The `When_Changed` value serves this purpose if it is of a sufficiently fine precision.

The modifications performed by a transaction should be included in the reconstructed state only for times after the commit time of the transaction. Because the

Achieving fully accurate transaction semantics for the reconstructed table is difficult.

commit time is not stored in the tracking log, the reconstruction algorithm must approximate this time; a reasonable guess is the time of the last modification in the tracking log. We can improve this somewhat by having the application insert a record into the tracking log immediately before committing. Note that this requires scanning the application code to identify COMMITs of transactions that modify the `PROJECTIONS` table, a task not necessitated by any of the other approaches in this chapter. This commit time is also an estimate, in that the commit process may itself take some time, and the actual commit record may be written to the DBMS log (thereby effecting the actual commit) quite a while after the time recorded in `P_Log`. Reconstructing the exact state, utilizing only the transactions that have actually committed by the specified time, is in general not possible because the DBMS does not render the actual commit time accessible.

8.9   IMPLEMENTATION CONSIDERATIONS

Microsoft Access 97 and Access 2000 do not support table-based triggers.

8.9.1   IBM DB2 Universal Database

IBM DB2 UDB triggers differ in some syntactic details from the SQL-92 triggers discussed in this chapter. DB2 UDB requires a referencing clause to create correlation names for the old and new values in the referenced table, for example, `REFERENCING OLD AS O`. DB2 also requires that `MODE DB2SQL` be specified. Finally, when there is more than one action specified for a trigger, these actions must be delimited with BEGIN ATOMIC and END. When there is exactly one action in the trigger, these delimiting clauses are optional. Finally, CURRENT DATE is two words in DB2 UDB (it is one word, with an embedded underscore, in SQL-92).

CF-8.2 can be expressed in DB2 UDB as follows (with the optional ATOMIC clause specified for clarity):

Code Fragment 8.21   **Triggers for maintaining the `P_Log` table in DB2 UDB, assuming no insertions.**

```
CREATE TRIGGER Delete_PROJECTIONS
AFTER DELETE ON PROJECTIONS
```

```
REFERENCING OLD AS O FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  INSERT INTO P_Log VALUES (O.PROJECTION_ID,
    O.PROJECTION_NAME,
    O.PROJECTION_TYPE,
    O.SPHEROID_CODE,
    O.PROJECTION_UOM,
    O.ZONE_CODE,
    CURRENT DATE);
END

CREATE TRIGGER Update_PROJECTIONS
AFTER UPDATE ON PROJECTIONS
REFERENCING OLD AS O FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
  INSERT INTO P_Log VALUES (O.PROJECTION_ID,
    O.PROJECTION_NAME,
    O.PROJECTION_TYPE,
    O.SPHEROID_CODE,
    O.PROJECTION_UOM,
    O.ZONE_CODE,
    CURRENT DATE);
END
```

8.9.2 Microsoft SQL Server

The syntax of the CREATE TRIGGER statement supported by Microsoft SQL Server differs from SQL-92. The implicit `OLD` and `NEW` correlation names are not available; instead, SQL Server provides two special tables, `inserted`, containing the rows that were inserted, and `deleted`, containing the rows that were deleted, by the statement under consideration. "Now" is implemented in SQL Server via `GETDATE()`. CF-8.2 can be expressed in SQL Server as follows:

Code Fragment 8.22   **Triggers for maintaining the** `P_Log` **table in Microsoft SQL Server, assuming no insertions.**

```
CREATE TRIGGER Delete_PROJECTIONS
ON PROJECTIONS FOR DELETE AS
  INSERT P_Log
  SELECT *, GETDATE()
  FROM deleted
```

```
CREATE TRIGGER Update_PROJECTIONS
ON PROJECTIONS FOR UPDATE AS
  INSERT P_Log
  SELECT *, GETDATE()
  FROM deleted
```

The other code fragments can be similarly implemented in Microsoft SQL Server.

8.9.3   Sybase SQLServer

Sybase SQLServer was one of the first DBMSs to support triggers. The syntax of the CREATE TRIGGER statement supported by Sybase SQLServer differs from SQL-92. The implicit OLD and NEW correlation names are not available; instead, Sybase provides two special tables, inserted, containing the rows that were inserted, and deleted, containing the rows that were deleted, by the statement under consideration. "Now" is implemented in Sybase via GETDATE(). CF-8.2 can be expressed in Sybase as follows:

Code Fragment 8.23   **Triggers for maintaining the** P_Log **table in Sybase SQLServer, assuming no insertions.**

```
CREATE TRIGGER Delete_PROJECTIONS
ON PROJECTIONS   FOR DELETE AS
  INSERT P_Log
  SELECT *, GETDATE()
  FROM deleted

CREATE TRIGGER Update_PROJECTIONS
ON PROJECTIONS FOR UPDATE AS
  INSERT P_Log
  SELECT *, GETDATE()
  FROM deleted
```

The other code fragments can be similarly implemented in Sybase SQLServer, with the exceptions of CF-8.7 and CF-8.17, as Sybase does not allow a view to be over a UNION.

8.9.4   Oracle8 Server

All the code fragments work fine under Oracle8 Server.

8.9.5 UniSQL

UniSQL supports triggers, but does not allow expressions such as CURRENTDATE within triggers. However, it does allow triggers to call methods on rows. To store a value of "now" in the tracking log, we'll use two triggers. The first, defined on the monitored table, is the normal one, with the minor change of storing a placeholder date (`DATE '01/01/1800'`) as the timestamp. The second is an insert trigger, defined on the tracking log, which calls a method to replace this value with the current date.

Code Fragment 8.24   Insert UniSQL trigger on the tracking log.

```
EXEC SQLX ALTER CLASS P_Log
    ADD METHOD set_date() FUNCTION set_date FILE './trans.o';


CREATE TRIGGER Insert_P_Log
AFTER INSERT ON P_Log
EXECUTE CALL set_date();
```

The `set_date` function uses the `db_put` operation to replace the date with the current date.

Code Fragment 8.25   `set_date` C function.

```
void set_date(DB_OBJECT *obj)
{
  EXEC SQLX BEGIN DECLARE SECTION;
  DB_VALUE d;
  struct timeval time_sec;
  struct tm *cur_time;
  EXEC SQLX END DECLARE SECTION;

  gettimeofday(&time_sec, NULL);
  cur_time = localtime(&time_sec.tv_sec);

  db_make_date(&d, cur_time->tm_mon+1, cur_time->tm_mday,
      cur_time->tm_year+1900);
  db_put(obj, "When_Changed",&d);
  EXEC SQLX COMMIT WORK;
}
```

8.9.6 CD-ROM Materials

All of the code discussed in this chapter has been implemented in IBM DB2 UDB, Microsoft SQL Server 7.0, Sybase SQLServer, Oracle8 Server, and UniSQL, and is provided on the CD-ROM, along with a test harness and `Makefile`.

Table 8.15 Tracking log organizations and their imposed constraints.

	constraints on updates		
	All insertions performed initially	None allowed after deletions	No constraints
Before-images	✓		
Before- + after-images	✓	✓	
After-images	✓	✓	
Backlog	✓	✓	✓

8.10 SUMMARY

A tracking log contains the history of modifications to the underlying monitored table, allowing previous states of that table to be reconstructed. A tracking log is maintained separately from the monitored table. For the `PROJECTIONS` table, a tracking log is preferable, as the rest of the application need not be modified.

Different organizations of the tracking log result from the varying assumptions and constraints on the monitored table. We considered a variety of situations:

- *All insertions are performed initially.* Only deletions and updates need be stored in the tracking log.
- *Insertions are allowed at any time other than after a deletion.* In this case, insertions must also be stored in the tracking log. The tracking log will be at least as long as the underlying table, even with no updates or deletions.
- *Insertions are allowed at any time.* In such cases a backlog, in which the operation type (INSERT, DELETE, UPDATE) is explicitly recorded, is indicated. Two organizations were discussed, one which stored a combination of before- and after-images, and one which stored only after-images, the latter exhibiting a concomitant simplification of the reconstruction algorithm.

Table 8.15 summarizes the tracking log organizations (listed down the left) and the constraints imposed on when insertions are allowed to the monitored table (listed across the top). A check mark means that the indicated organization will work when the indicated constraint is satisfied.

Different organizations of the tracking log result from various initial assumptions and constraints on the monitored table.

The tracking log structure depends strongly on the application. In the case of the problem with inadvertent changes to the `PROJECTIONS` table, a very simple structure was adequate. Should the application need to make arbitrary modifications to

the monitored table, then a backlog is appropriate. If intermediate, dirty data is undesirable in the reconstructed states, then either more involved triggers or a more involved reconstruction algorithm is required.

Tracking logs support transaction time, which is orthogonal to valid time.

You must be careful to distinguish the information content of a tracking log and of a valid-time table. A conventional table models the current state of a portion of the enterprise. A valid-time table models the time-varying state of a portion of the enterprise. A tracking log is an example of a *transaction-time* table. Such a table records the state not of the modeled reality, but of the monitored table over time. This distinction is important both in the information we can extract from the table and the kinds of updates permitted.

Consider two tables, the valid-time table `Employee` from Chapter 5 and the transaction-time table `P_Log` from this chapter. Requesting the state on April 1, 1996, of the valid-time table will return the employees on that date, as best known. This information is independent of when the `Employee` table was actually created. In fact, this table could have been created yesterday, then populated with historical information. On the other hand, requesting the state on April 1, 1996, of the transaction-time table will return the contents of that table as it was stored on disk on that date. If the table was actually populated yesterday, then its state back on April 1 would be empty.

Information about the past or future can be added or changed in valid-time tables. If we find that information about April 1, 1996, was in error (say, an employee was omitted), we can make that correction; later queries will return that employee. On the other hand, we cannot change what was stored previously on the disk, and so cannot update past states of a transaction-time table. Similarly, we do not know what will be stored on disk in the future, and so cannot update future states of that table. Hence, transaction time is defined only in the past, unlike valid time, which can extend into the future. All we can do is append a new state to the table. For this reason, all the triggers that maintain `P_Log` effect insertions. A transaction-time table is *append-only*. Changes to the monitored table, whether insertions, deletions, or updates, are in actuality insertions into the tracking log. This means that we can add a tracking log to a conventional table, as was described in this chapter, or to a valid-time table, resulting in a *bitemporal* table, to be discussed in Chapter 10.

A transaction-time table has an important property not shared with valid-time tables: queries on past states will return the same result when evaluated at any time in the future. Consider CF-8.3 on page 225, "reconstruct the `PROJECTIONS` table as of April 1, 1996." Whether we execute this query on April 2, 1996, December 31, 1996, July 7, 1998, or September 22, 2009, the resulting states will all be *identical*. The resulting state will be a copy of the `PROJECTIONS` table that was present as magnetic patterns on the disk on April 1, 1996.

We also defined views of the `PROJECTIONS` table; CF-8.5 on page 226 is the reconstructed table as of April 1, 1996 as a view. It makes no difference whether this view is materialized as a stored table or remains as a virtual table because the contents of the view will not change, regardless of the modifications applied to either `PROJECTIONS` or `P_Log`. The triggers defined in this chapter to maintain the tracking log ensure that only the current state is modified; prior states *must* be left intact and unchanged.

The reconstructed state of a transaction-time table as of a point in the past will never change, independent of when that reconstruction query or view is evaluated. The state at a point in time of a valid-time table can change, as new information is received and incorporated into the table.

For this same reason, changes to the future of a transaction-time table are not allowed. There is no way to accurately predict what the magnetic patterns of the disk will be at some point in the future.

Such is not the case with valid-time tables. We are free to revise the stored history in a valid-time table as new information becomes available. Sequenced and nonsequenced modifications can change the past and the future. A view of the enterprise on April 1, 1996, can change as information about that date is received and our knowledge is refined. The valid-time table reflects our understanding of the history of the enterprise *as best known*.

Current queries on a tracking log are much easier on monitored tables than on valid-time state tables: just perform the query on the monitored table, which already records only the current state. Current modifications (the only kind allowed on such tables) are equally simple: just apply it to the table, and the triggers will make sure the tracking log is maintained. Legacy applications need not be modified at all when maintaining a tracking log is initiated on the table.

Extracting a prior state involves looking both at the monitored table and the tracking log in some tracking log organizations; if a backlog is maintained, the monitored table need not be consulted.

Sequenced and nonsequenced queries are best realized by first defining a view that extracts a transaction-time state table from the tracking log. Sequenced and nonsequenced modifications are not permitted on a tracking log, as they would invalidate the semantics of transaction time and would corrupt later reconstructions.

8.11   READINGS

Serialization order of concurrent transactions is discussed in many textbooks; [6, 36] are seminal works on this topic. Transaction time was first covered in detail by Ahn and the author [89, 90] and was later included in the glossary [48]. Tracking logs

have been studied by [8]. Backlog tables have been suggested by several authors [51, 52, 61].

Chamberlin shows how to maintain a list of changes, though not the actual values of the changes, eliminating an opportunity to reconstruct past states via DB2 triggers [22, pp. 358–359].

Leung and Pirahesh show how to access a backlog in which only the changed columns, as well as the primary key, were stored; the columns retaining their old values were represented with NULLs, which incur less storage overhead [66]. A time-slice is obtained by collecting, for each primary key value, the most recent value for each column, using the recursive query facilities of DB2. This is a classic time-for-space trade-off.

Fraser's *Time: the Familiar Stranger* [34] is a superb survey of an expanse of topics related to time.

Palmer provides a highly readable account of detecting two clocks in fiddler crabs, as well as summarizing other biological clocks [78].

[31] is a technical guide for those who wish to provide encoding equipment and/or decoding equipment to produce material with encoded data embedded in line 21 of the vertical blanking interval of the NTSC video signal. The encoded data includes extended data services, such as date and time information.
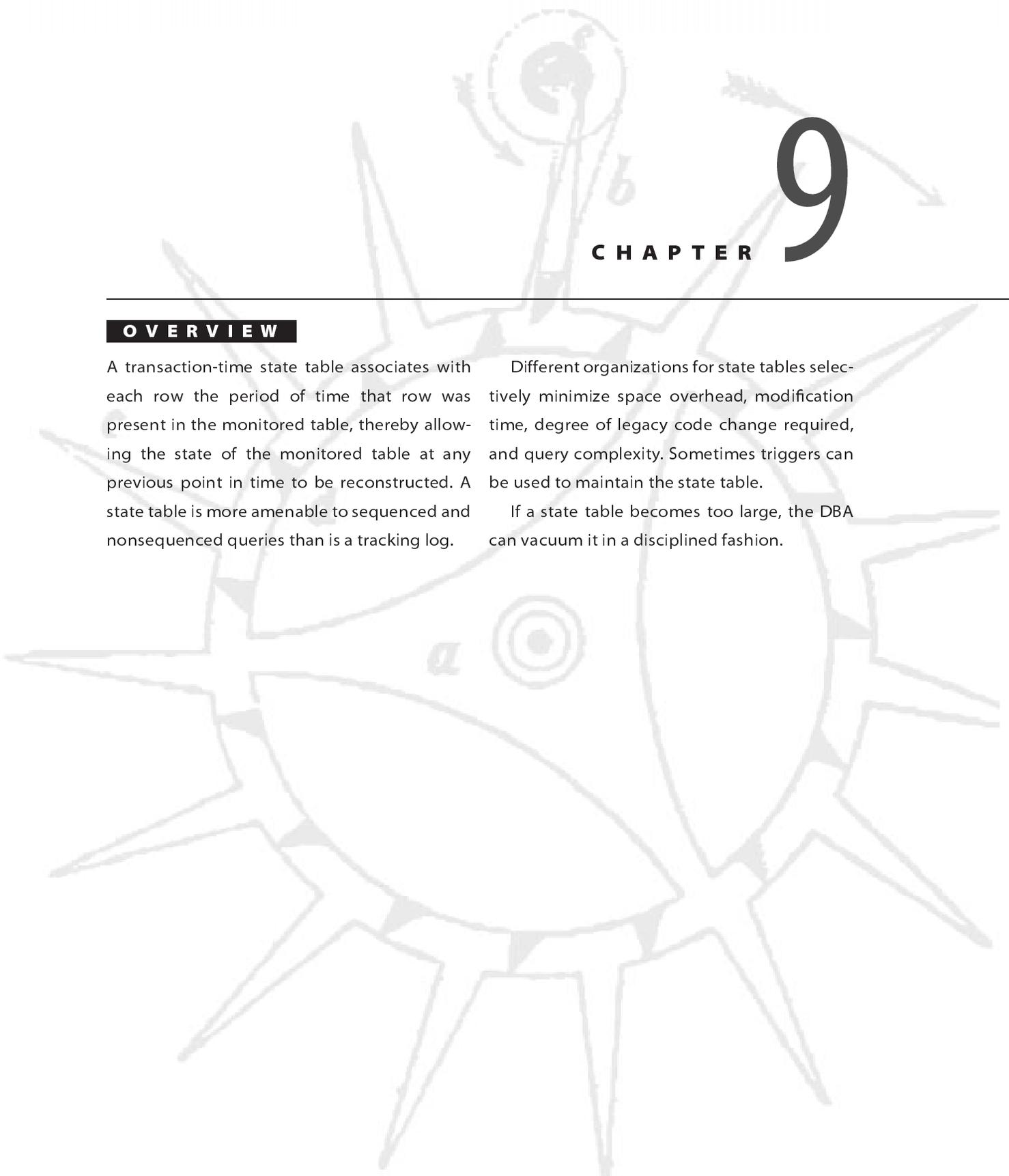
OVERVIEW

A transaction-time state table associates with each row the period of time that row was present in the monitored table, thereby allowing the state of the monitored table at any previous point in time to be reconstructed. A state table is more amenable to sequenced and nonsequenced queries than is a tracking log.

Different organizations for state tables selectively minimize space overhead, modification time, degree of legacy code change required, and query complexity. Sometimes triggers can be used to maintain the state table.

If a state table becomes too large, the DBA can vacuum it in a disciplined fashion.

Transaction-Time State Tables

The previous chapter addressed Nigel's problem of spurious changes in the `PROJECTIONS` table by defining an associated tracking log that captures these changes for later perusal. Several organizations of the tracking log were examined: recording before-images, recording after-images, recording both before- and after-images, and recording the actual modification operations (termed a "backlog"). These variants all utilized a single transaction timestamp, `When_Changed`. To realize some queries, in particular sequenced queries, we recommended that the tracking log be converted into a *transaction-time state table*, with period timestamping.

In this chapter, we adopt exactly the opposite approach. We maintain the transaction-time state table directly, with the monitored (snapshot) table available either as a view or as a regular table.

If space is at a premium, maintaining an instant-stamped tracking log (specifically, the after-image organization) with triggers is the way to go. On the other hand, if sequenced queries or sequenced integrity constraints are important, then the period-stamped approach described here should be considered.

We revisit the issues of the last chapter using this period-stamped organization. This discussion will parallel that chapter, as well as the valid-time state table chapters (Chapters 5–7). While many of the concepts are the same between the two types of state tables, valid-time and transaction-time, it is important to keep in mind the critical difference: valid-time tables model changes in reality, while transaction-time tables model changes in the database. The two kinds of time are orthogonal, and as we will see later, can be combined into one glorious structure, the bitemporal table.

9.1 DEFINITION

Conceptually, a transaction-time state table represents the sequence of snapshot states constituting the states of the monitored table over time. This sequence is represented by timestamping rows with a period. The period begins at the time in which the row was inserted into the monitored table, either directly with an INSERT statement or as a side effect of an UPDATE statement, and ends when the row was deleted from the monitored table, again, either directly via a DELETE statement or as a side effect of an UPDATE statement. This period is termed the *period of presence* of the row, as it specifies when that row was in the monitored table.

We first define a new table, `P_TT`, the transaction-time state table mirroring the `PROJECTIONS` table.

Code Fragment 9.1   **Create the transaction-time state table.**

```
CREATE TABLE P_TT (
  PROJECTION_ID  INT,
  PROJECTION_NAME  CHAR(10),
  PROJECTION_TYPE  INT,
  SPHEROID_CODE  INT,
  PROJECTION_UOM  INT,
  ZONE_CODE  INT,
  Start_Date  DATE,
  Stop_Date  DATE,
  PRIMARY KEY (PROJECTION_ID, Stop_Date))
```

All but the final two columns are from the `PROJECTIONS` table. The `Start_Date` and `Stop_Date` in concert denote the period of presence.

The schema of a transaction-time state table comprises the columns of the monitored table, along with two timestamp columns denoting the period of presence. Its key is simply the primary key of the monitored table and the start timestamp column.

Recall from Section 5.3 that the (sequenced) primary key of a valid-time table cannot be composed from its columns; an assertion is required. Such is not the case here. The key of the transaction-time state table is simply the key of the monitored table (`PROJECTION_ID`) along with the `Stop_Date` column. This works because the primary key constraint on the monitored table implies a current key constraint on its associated transaction-time state table. Since only current modifications are applied to the state table, a current key constraint implies a sequenced key constraint.

9.2   MAINTENANCE

There are two ways to maintain a transaction-time state table: indirectly, as a side effect of triggers defined on the monitored table, and directly, by transforming modifications on the monitored table into modifications on the state table. The benefit of the former approach is that no code need be changed when the state table is defined; legacy applications function as before.

9.2.1   Indirectly via Triggers

In this first approach, the application need not be concerned with maintaining the state table; this will be done behind the scenes via three triggers. In the following, we represent "until changed" with "forever," in this case `DATE '9999-12-31'`.

Code Fragment 9.2   **Triggers for maintaining the** P_TT **table.**

```
CREATE TRIGGER INSERT_P
AFTER INSERT ON PROJECTIONS FOR EACH ROW
  INSERT INTO P_TT(PROJECTION_NAME, PROJECTION_ID,
    PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
    PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
  VALUES (NEW.PROJECTION_NAME, NEW.PROJECTION_ID,
      NEW.PROJECTION_NAME, NEW.PROJECTION_TYPE,
      NEW.SPHEROID_CODE, NEW.PROJECTION_UOM, NEW.ZONE_CODE,
      CURRENT_DATE, DATE '9999-12-31')

CREATE TRIGGER DELETE_P
AFTER DELETE ON PROJECTIONS FOR EACH ROW
  UPDATE P_TT
  SET Stop_Date = CURRENT_DATE
  WHERE P_TT.PROJECTION_ID = OLD.PROJECTION_ID
    AND P_TT.Stop_Date = DATE '9999-12-31'

CREATE TRIGGER UPDATE_P
AFTER UPDATE ON PROJECTIONS FOR EACH ROW
BEGIN ATOMIC
  UPDATE P_TT
  SET Stop_Date = CURRENT_DATE
  WHERE P_TT.PROJECTION_ID = OLD.PROJECTION_ID
    AND P_TT.Stop_Date = DATE '9999-12-31';
```

```
    INSERT INTO P_TT(PROJECTION_NAME, PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
      PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
    VALUES (NEW.PROJECTION_NAME, NEW.PROJECTION_ID,
        NEW.PROJECTION_NAME, NEW.PROJECTION_TYPE,
        NEW.SPHEROID_CODE, NEW.PROJECTION_UOM, NEW.ZONE_CODE,
        CURRENT_DATE, DATE '9999-12-31')
END
```

After the transactions listed in Section 8.2.1, the state table (in this chapter, mention of a state table will imply a transaction-time state table) will contain the rows shown in Table 9.1. As before, we only show two columns, PROJECTION_ID and PROJECTION_TYPE, as well as the timestamp columns.

Triggers on the monitored table can be used to maintain a transaction-time state table.

It is useful to compare Table 9.1 with the state table converted from the tracking log, Table 8.4. The only difference is the value of the Stop_Date column. In the previous table, for some rows the value is 1998-01-16 (the date the conversion was run, "now"), whereas here the value for those rows is 9999-12-31, or "forever." While "now" is in fact more accurate, in that only the past and the current state can be stored in a transaction-time state table (since we cannot predict the future), we use "forever" in this chapter only because constantly updating the Stop_Date value to "now" is impractical.

The state table is somewhat redundant, in two senses. It contains all of the monitored table, as rows with a Stop_Date of "forever." And many of the other stop dates are not strictly necessary, as they are paired with identical Start_Date values; projections 2 and 3 are examples. The Stop_Date value of 1996-03-20 for the first row of projection 2 matches the Start_Date value of the following row of projection 2; indeed, both of these rows were impacted by a single transaction, an UPDATE that

Table 9.1 A transaction-time state table, P_TT.

PROJECTION_ID	PROJECTION_TYPE	Start_Date	Stop_Date
1	12	1996-01-01	9999-12-31
2	10	1996-01-01	1996-03-20
3	15	1996-01-01	1996-05-28
4	17	1996-01-01	1996-07-12
5	18	1996-01-01	1996-02-03
2	13	1996-03-20	1996-06-17
3	11	1996-05-28	9999-12-31
2	14	1996-06-17	9999-12-31

changed the type of projection 2 from 10 to 13. This shows that the tracking log or-ganizations, which use only one timestamp column, are more space efficient than the period-stamped state table organization. However, the state table approach is much easier to query, especially for sequenced queries.

9.2.2   Directly via Rewritten Modifications

A transaction-time state table may also be maintained directly.

The second approach is to *replace* the monitored table with the state table, then define the monitored table as a view on the state table, with the associated space savings over the regular monitored table of the previous section.

Code Fragment 9.3   **Reconstruct the** PROJECTIONS **table as of now, as a view.**

```
CREATE VIEW PROJECTIONS (PROJECTION_ID, PROJECTION_NAME,
      PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
      ZONE_CODE)
AS (SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
    FROM P_TT
    WHERE Stop_Date = DATE '9999-12-31')
```

Applications composed only of queries on the PROJECTIONS table will work fine with this view, oblivious to the fact that the evolution of that table is being retained.

Applications that modify the PROJECTIONS table must be changed to in-stead modify the state table. Chapter 7 discussed 12 (!) categories of modifications on valid-time state tables: INSERT, DELETE, and UPDATE coupled with current, sequenced, and nonse-quenced semantics, along with current modifications in the restricted case (only current modifications ever allowed). The situation with transaction-time state tables is much simpler, as there are only current modifications. We cannot modify the past, as we cannot change the bits that were stored on disks in the past. Similarly, we cannot modify the future, as we cannot ac-curately predict what will be stored on the disk in the future. We can only modify the current state. So, we are left with only three kinds of modifications to consider: cur-rent INSERT, current DELETE, and current

---

Pendulum

The foliot was weighted to slow its oscillation, with the weights movable so that the clock could be adjusted. Unfortunately, the friction of the clock's mechanism and its exact mechanical arrangement kept its accuracy to only about 15 minutes a day. What was needed was a periodic device whose fre-quency was dependent only on the device itself and not on the details of its manufacture.

The Dutch scientist Christian Huygens was the first to apply the pendulum, whose frequency is dependent only on its length, to regulate a clock. Huygens's clock of 1656 was accurate to 10 *seconds* a day, a vast improvement over the foliot clock.

UPDATE. The resulting code is very similar to that given for current modifications to valid-time state tables in Section 7.1.

An INSERT on a transaction-time state table requires only appending the `Start_Date` and `Stop_Date` values, as "now" (`CURRENT_DATE`) and "forever" (`DATE '9999-12-31'`), respectively.

Code Fragment 9.4   **Insert a projection with an ID of 6.**

```
INSERT INTO P_TT (PROJECTION_ID, PROJECTION_NAME,
      PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
       ZONE_CODE,
      Start_Date, Stop_Date)
VALUES (6, 'New Projection', 22, 14, 93, 4,
      CURRENT_DATE, DATE '9999-12-31')
```

The fourth and sixth lines of this fragment were added to capture the change behavior.

A DELETE is mapped into an UPDATE, changing the `Stop_Date` of the deleted row(s) to "now."

Code Fragment 9.5   **Delete projection 2.**

```
UPDATE P_TT
SET Stop_Date = CURRENT_DATE
WHERE PROJECTION_ID = 2
  AND Stop_Date = DATE '9999-12-31'
```

An UPDATE is logically a deletion followed by an insertion, and so is implemented with an UPDATE and an INSERT. However, the INSERT must come first.

Code Fragment 9.6   **Change the type of projection 1 to 43.**

```
INSERT INTO P_TT (PROJECTION_ID, PROJECTION_NAME,
      PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
      ZONE_CODE, Start_Date, Stop_Date)
SELECT PROJECTION_ID, PROJECTION_NAME, 43,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      CURRENT_DATE, DATE '9999-12-31'
FROM P_TT
WHERE PROJECTION_ID = 1
  AND Stop_Date = DATE '9999-12-31'

UPDATE P_TT
SET Stop_Date = CURRENT_DATE
WHERE PROJECTION_ID = 1
  AND PROJECTION_TYPE <> 43
  AND Stop_Date = DATE '9999-12-31'
```

Maintaining the state table via direct modifications obviates the need for a materialized monitored table, decreasing the space overhead for capturing changes over time.

Here we first ensure that there is a row to update, then insert the new value (via the INSERT), and terminate the old value at "now" (via the UPDATE). The `PROJECTION_TYPE <> 43` predicate in the UPDATE is required to avoid changing the row just inserted.

We emphasize that in this approach `P_TT` is a *replacement* for the `PROJECTIONS` table. We maintain the `P_TT` table directly through rewritten modification statements, rather than indirectly through triggers defined on the monitored table.

9.3 QUERIES

As previously mentioned, the advantage of a transaction-time state table over a tracking log (with its single timestamp) is its ease in querying.

Reconstruction queries on state tables are termed *time-slice* queries. The current state view (CF-9.3) was one example. Of course, it is possible to reconstruct the state of the monitored table at any point in the past.

Code Fragment 9.7   **Reconstruct the** PROJECTIONS **table as of April 1, 1996.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
       SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_TT
WHERE Start_Date <= DATE '1996-04-01' AND DATE '1996-04-01' < Stop_Date
```

This is shorter than any of the reconstruction algorithms given for the tracking log organizations (compare with CF-8.3, CF-8.13, CF-8.14, and CF-8.16).

Reconstruction is easy to express on a state table, though only states in the past or present should be requested.

This code also seems to work for dates in the future (try replacing 1996 with 2009), but in fact we have no idea what rows the `PROJECTIONS` table will contain in 2009. Reconstructions should only be done on the current or past dates; using future dates is meaningless. (This doesn't apply to *valid-time* state tables, where future time-slices may be quite meaningful.)

Current queries are even easier: just apply to the monitored table or view.

State tables really shine when sequenced queries are desired. The strategies for such queries on valid-time state tables apply in their entirety on transaction-time state tables, with two important provisions. A sequenced query on a valid-time state table means "give the history of"; a sequenced query on a transaction-time state table means "when was it recorded that" or perhaps "give the change history for."

Sequenced queries over transaction-time state tables are expressed identically to such queries over valid-time state tables. However, their semantics is in terms of "when was it recorded that."

The second difference is that while a sequenced query on a valid-time state table returns another valid-time state table, a sequenced query on a transaction-time state table does *not* return another transaction-time state table. Recall once again that a transaction-time state table specifies the magnetic patterns recorded on the disk at times in the past. The result of a sequenced query itself was emphatically *not* stored on the disk in the past; it has just been calculated now. It does indicate what was recorded, but it itself was not in existence until the query was performed. To emphasize this distinction, we will use the column names `Recorded_Start` and `Recorded_Stop` for these query results.

With those critical distinctions in mind, we now try out some sequenced queries.

As before, selections (the WHERE clause) and projections (the SELECT clause) are easy to render as sequenced (compare with CF-6.7).

Code Fragment 9.8   When was it recorded that a projection had a type of 17?

```
SELECT PROJECTION_ID, PROJECTION_TYPE,
  Start_Date AS Recorded_Start, Stop_Date AS Recorded_Stop
FROM P_TT
WHERE PROJECTION_TYPE = 17
```

Selections (the WHERE clause) are unchanged in sequenced queries; projections (the SELECT clause) require adding the timestamp columns.

UNION is also easy to convert (compare with CF-6.10).

Code Fragment 9.9   Give the change history for projections having a type of 12 or 18.

```
SELECT PROJECTION_ID,
  Start_Date AS Recorded_Start, Stop_Date AS Recorded_Stop
FROM P_TT
WHERE PROJECTION_TYPE = 12
UNION
SELECT PROJECTION_ID, Start_Date, Stop_Date
FROM P_TT
WHERE PROJECTION_TYPE = 18
```

This will tell us when such projections were added or removed, or when the type was changed to or from 12 or 18.

Joins are more challenging. Here we use `first_instant` and `last_instant` PSM functions (CF-6.13). Compare the following query with CF-6.14.

Code Fragment 9.10   When was it recorded that two projections had the same type?

```
SELECT P1.PROJECTION_ID, P2.PROJECTION_ID, P1.PROJECTION_TYPE,
    last_instant(P1.Start_Date, P2.Start_Date) AS Recorded_Start,
```

```
            first_instant(P1.Stop_Date, P2.Stop_Date) AS Recorded_Stop
FROM P_TT AS P1, P_TT AS P2
WHERE P1.PROJECTION_ID <> P2.PROJECTION_ID
  AND P1.PROJECTION_TYPE = P2.PROJECTION_TYPE
  AND last_instant(P1.Start_Date, P2.Start_Date) <
        first_instant(P1.Stop_Date, P2.Stop_Date)
```

Auditing queries are often nonsequenced queries. Once an error is found, queries examining the changes made to the monitored table attempt to determine how and why the error was made. Changes appear in the state table as two periods that meet.

Code Fragment 9.11   **When was the type of a projection erroneously changed to be identical to that of an existing projection?**

```
SELECT P1.PROJECTION_ID, P2.PROJECTION_ID,
    P1.PROJECTION_TYPE AS Identical_TYPE,
    P3.PROJECTION_TYPE AS Prior_TYPE,
    P2.Start_Date AS When_Changed
FROM P_TT AS P1, P_TT AS P2, P_TT AS P3
WHERE P1.PROJECTION_ID <> P2.PROJECTION_ID
  AND P2.PROJECTION_ID = P3.PROJECTION_ID
  AND P1.PROJECTION_TYPE = P2.PROJECTION_TYPE
  AND P2.PROJECTION_TYPE <> P3.PROJECTION_TYPE
  AND P3.Stop_Date = P2.Start_Date
```

Here, P1 is the existing projection; P3 has the old type, which was changed erroneously to P2. This query is arduous when attempted on a tracking log. While the old or the new type is readily available (depending on whether before- or after-images are recorded in the tracking log), determining the type of a projection existing at the same time is difficult with instant-stamped tables.

Another kind of nonsequenced query is the conversion to a tracking log. Before-images and after-images can be easily extracted from a state table.

Code Fragment 9.12   **Extract before-images from a transaction-time state table.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    Stop_Date AS When_Changed
FROM P_TT
WHERE Stop_Date <> DATE '9999-12-31'
```

Code Fragment 9.13   **Extract after-images from a transaction-time state table.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    Start_Date AS When_Changed
FROM P_TT
```

Converting a state table to a backlog is a little more ambitious, as we have to distinguish insertions, deletions, and updates. However, doing so emphasizes that the information content of tracking logs (possibly along with the monitored table), backlogs, and transaction-time state tables is identical.

Code Fragment 9.14 **Extract a backlog from a transaction-time state table.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    Start_Date AS When_Changed, 'I' AS Operation
FROM P_TT AS P1
WHERE NOT EXISTS ( SELECT *
                    FROM P_TT AS P2
                    WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
                      AND P2.Stop_Date = P1.Start_Date)
UNION
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
    Stop_Date AS When_Changed, 'D' AS Operation
FROM P_TT AS P1
WHERE P1.Stop_Date <> DATE '9999-12-31'
  AND NOT EXISTS ( SELECT *
                    FROM P_TT AS P2
                    WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
                      AND P1.Stop_Date = P2.Start_Date)
UNION
SELECT P1.PROJECTION_ID, P1.PROJECTION_NAME,
    P1.PROJECTION_TYPE, P1.SPHEROID_CODE,
    P1.PROJECTION_UOM, P1.ZONE_CODE,
    P2.Start_Date AS When_Changed, 'U' AS Operation
FROM P_TT AS P1, P_TT AS P2
WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
  AND P1.Stop_Date = P2.Start_Date
```

Tracking logs, backlogs, and transaction-time state tables have identical information content.

Here we do a case analysis. INSERTs are indicated by the absence of an immediately preceding state with the same key value, DELETEs by the absence of an immediately following state, and UPDATEs by the presence of an immediately preceding state.

9.4 TEMPORAL PARTITIONING*

As we mentioned, having a Stop_Date of "forever" is awkward, as we cannot accurately predict the future. As discussed in Section 7.5 in the context of valid time,

a temporally partitioned organization finesses this awkwardness, while also achieving a slight space savings, at the expense of requiring more effort to express some queries. This organization uses two (or more) tables to represent a single state table.

9.4.1 Current and Archival Stores

To illustrate, we use two tables, P_TT_PAST, consisting of the rows that have been corrected, and P_TT_CURRENT, consisting of the rows that haven't been corrected, that is, whose period of presence includes "now." Since all rows of P_TT_CURRENT have a Stop_Date of "now," we'll simply omit that column.

Code Fragment 9.15   **Create a temporally partitioned transaction-time state table.**

```
CREATE TABLE P_TT_PAST (PROJECTION_ID INT,
  PROJECTION_NAME CHAR(10),
  PROJECTION_TYPE INT,
  ...,
  Start_Date DATE,
  Stop_Date DATE,
  PRIMARY KEY (PROJECTION_ID, Start_Date))

CREATE TABLE P_TT_CURRENT (PROJECTION_ID INT,
  PROJECTION_NAME CHAR(10),
  PROJECTION_TYPE INT,
  ...,
  Start_Date DATE DEFAULT CURRENT_DATE,
  PRIMARY KEY (PROJECTION_ID))
```

There are three differences between the two tables. The first is that P_TT_CURRENT does not have a Stop_Date column. Its implicit stop date is "now." The second is that P_TT_CURRENT has a default value of "now" for the Start_Date; we will see the utility of this shortly. The final difference is that the primary key of P_TT_CURRENT is that of the original monitored table.

We term P_TT_PAST the *archival store*: it contains rows that have been corrected, and "store" is used rather than "table" to differentiate the representation (the "store") from the logical structure (the state table). P_TT_CURRENT is likewise termed the *current store*.

Another way to think about this is that P_TT_CURRENT is the monitored table, PROJECTIONS, with an additional Start_Date column. This enables us to define PROJECTIONS as a view and also justifies P_TT_CURRENT having the same primary key.

Code Fragment 9.16   **Reconstruct the** PROJECTIONS **table as of now, as a view on a tempo-
rally partitioned table.**

```
CREATE VIEW PROJECTIONS (PROJECTION_ID, PROJECTION_NAME,
      PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
      ZONE_CODE)
AS (SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
        SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
     FROM P_TT_CURRENT)
```

A transaction-time state table may be represented with two tables, a current and an archival store.

Comparing this code fragment with CF-9.3, we notice that here every row is selected; in that earlier view definition, we had to explicitly test the Stop_Date.

The best way to maintain this temporally partitioned state table is via triggers on the current store.

Code Fragment 9.17   **Triggers for maintaining the** P_TT **table.**

```
CREATE TRIGGER DELETE_P
BEFORE DELETE ON P_TT_CURRENT FOR EACH ROW
  INSERT INTO P_TT_PAST(PROJECTION_ID,
    PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
    PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
  SELECT PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      Start_Date, CURRENT_DATE
  FROM P_TT_CURRENT AS P
  WHERE P.PROJECTION_ID = OLD.PROJECTION_ID

CREATE TRIGGER UPDATE_P
BEGIN ATOMIC
BEFORE UPDATE ON P_TT_CURRENT FOR EACH ROW
  INSERT INTO P_TT_PAST(PROJECTION_ID,
    PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
    PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
  SELECT PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      Start_Date, CURRENT_DATE
  FROM P_TT_CURRENT AS C
  WHERE C.PROJECTION_ID = OLD.PROJECTION_ID

  UPDATE P_TT_CURRENT
  SET Start_Date = CURRENT_DATE
  WHERE P_TT_CURRENT.PROJECTION_ID = OLD.PROJECTION_ID
END
```

Table 9.2 The P_TT_PAST **table.**

PROJECTION_ID	PROJECTION_TYPE	Start_Date	Stop_Date
2	10	1996-01-01	1996-03-20
3	15	1996-01-01	1996-05-28
4	17	1996-01-01	1996-07-12
5	18	1996-01-01	1996-02-03
2	13	1996-03-20	1996-06-17

Table 9.3 The P_TT_CURRENT **table.**

PROJECTION_ID	PROJECTION_TYPE	Start_Date
1	12	1996-01-01
3	11	1996-05-28
2	14	1996-06-17

The two-table representation requires a few replacements in the legacy code.

Note that there is no trigger for INSERT. An insertion just affects the current store. However, for a deletion or update, we capture the before-image in the archival store, with a period of presence from the date the row was originally inserted (or updated) to "now." Finally, P_TT_CURRENT.Start_Date of the updated row needs to be changed to "now"; this requires an after-trigger.

Defining the triggers on the current store requires that the following replacements be made to the legacy code:

● INSERT INTO PROJECTIONS replaced with INSERT INTO P_TT_PAST
● DELETE FROM PROJECTIONS replaced with DELETE FROM P_TT_PAST
● UPDATE PROJECTIONS replaced with UPDATE P_TT_PAST

Within the modification statement, and within SELECT statements, references to PROJECTIONS can remain; these will now refer instead to the view defined on P_TT_CURRENT.

These triggers will produce the archival and current stores shown in Tables 9.2 and 9.3, which should be compared with Table 9.1. There were eight rows in that previous table, partitioned here into five corrected rows and three current rows.

9.4.2   Queries

We now turn to queries. Queries on the current state can be applied to the PROJECTIONS view as before. Past states can be reconstructed, using the archival store.

Code Fragment 9.18   **Reconstruct the** PROJECTIONS **table as of April 1, 1996.**

```
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_TT_PAST
WHERE Start_Date <= DATE '1996-04-01' AND DATE '1996-04-01' < Stop_Date
UNION
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
    SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM P_TT_CURRENT
WHERE Start_Date <= DATE '1996-04-01'
```

This is an extension of CF-9.7. An important difference is that future states cannot be reconstructed from the archival store, as the Stop_Date is always before "now." This makes the archival store slightly safer to use.

Both the monitored table and the transaction-time state table can be defined as views on the two stores.

Sequenced queries are a little more difficult because the information is spread across two tables. The easiest approach is to define another view, P_TT, which is the original state table being implemented by the two stores.

Code Fragment 9.19   **Define the state table as a view.**

```
CREATE VIEW P_TT (PROJECTION_ID, PROJECTION_NAME,
    PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
    ZONE_CODE, Start_Date, Stop_Date)
AS (SELECT * FROM P_TT_PAST
    UNION
    SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
        SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
        Start_Date, CURRENT_DATE AS Stop_Date
    FROM P_TT_CURRENT)
```

Having the state table available as a view, we can then query it exactly as discussed in Section 9.3, using it for both sequenced and nonsequenced queries.

9.4.3   Utilizing the Primary Key

One objection to the partitioned organization just presented is that it requires the legacy application to be modified. Granted, the modification is slight—just replace the mention of the PROJECTIONS table in the first line of modification statements—but *any* change to a complex application should not be approached lightly.

In a tripartitioned table, the current store consists of just the primary key and the start date.

The PROJECTIONS table was defined as a view on the current store, just projecting out the Start_Date column. Instead, we can define *three* tables: the original monitored table, the archival

store, and a horizontally truncated current store, containing only the primary key of the monitored table and the start time, when that row was inserted into the monitored table. The current store and the archival store can be maintained via triggers on the monitored table, thereby obviating any changes to legacy code.

Code Fragment 9.20   **Define a truncated current store.**

```
CREATE TABLE P_TT_CURRENT (
  PROJECTION_ID INT
  Start_Date DATE,
  PRIMARY KEY (PROJECTION_ID))
```

The triggers are now defined on the monitored table itself.

Code Fragment 9.21   **Triggers for maintaining a tripartitioned state table.**

```
CREATE TRIGGER INSERT_P
AFTER INSERT ON PROJECTIONS FOR EACH ROW
  INSERT INTO P_TT_CURRENT(PROJECTION_ID, Start_Date)
  VALUES (NEW.PROJECTION_ID, CURRENT_DATE)

CREATE TRIGGER DELETE_P
BEFORE DELETE ON PROJECTIONS FOR EACH ROW
BEGIN ATOMIC
  INSERT INTO P_TT_PAST(PROJECTION_ID,
    PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
    PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
  SELECT OLD.PROJECTION_ID,
      OLD.PROJECTION_NAME, OLD.PROJECTION_TYPE,
      OLD.SPHEROID_CODE, OLD.PROJECTION_UOM, OLD.ZONE_CODE,
      Start_Date, CURRENT_DATE
  FROM P_TT_CURRENT AS PC
  WHERE PC.PROJECTION_ID = OLD.PROJECTION_ID;

  DELETE FROM P_TT_CURRENT
  WHERE PROJECTION_ID = OLD.PROJECTION_ID;
END

CREATE TRIGGER UPDATE_P
AFTER UPDATE ON PROJECTIONS FOR EACH ROW
BEGIN ATOMIC
  INSERT INTO P_TT_PAST(PROJECTION_ID,
    PROJECTION_NAME, PROJECTION_TYPE, SPHEROID_CODE,
    PROJECTION_UOM, ZONE_CODE, Start_Date, Stop_Date)
```

```
SELECT OLD.PROJECTION_ID,
    OLD.PROJECTION_NAME, OLD.PROJECTION_TYPE,
    OLD.SPHEROID_CODE, OLD.PROJECTION_UOM, OLD.ZONE_CODE,
    Start_Date, CURRENT_DATE
FROM P_TT_CURRENT AS PC
WHERE PC.PROJECTION_ID = OLD.PROJECTION_ID;

UPDATE P_TT_CURRENT
SET Start_Date = CURRENT_DATE
WHERE PROJECTION_ID = NEW.PROJECTION_ID
END
```

Here we have the P_TT_CURRENT table mirror the changes to the monitored table, with before-images retained in the archive store.

A view reconstitutes the state table from the three underlying tables. Current queries are expressed against the monitored table, which is now materialized, rather than being defined as a view as in some of the other organizations. Such queries will generally be faster when evaluated against a table than when evaluated against a view of a larger underlying table.

For sequenced and nonsequenced queries, we define a transaction-time state view over the three constituent tables.

Code Fragment 9.22   **Define the state table as a view.**

```
CREATE VIEW P_TT (PROJECTION_ID, PROJECTION_NAME,
    PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
    ZONE_CODE, Start_Date, Stop_Date)
AS (SELECT * FROM P_TT_PAST
    UNION
    SELECT PROJECTIONS.PROJECTION_ID, PROJECTION_NAME,
        PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
        ZONE_CODE, Start_Date, CURRENT_DATE AS Stop_Date
    FROM PROJECTIONS, P_TT_CURRENT AS C
    WHERE PROJECTIONS.PROJECTION_ID = C.PROJECTION_ID)
```

9.5   VACUUMING*

The current store consists of the monitored table plus possibly an additional column, indicating when the row entered the table. As such it will contain the same number of rows as the original monitored table. The archival store contains old rows of the monitored table that have since been corrected; the two additional

columns denote the period of presence of these rows. The archival store is useful for auditing the changes that have been applied to the monitored table, particularly when the changes themselves were incorrect or malicious.

In comparison with a table associated with no temporal support, the database administrator needs to be aware of the implications of adding transaction-time support to that table. First, either the monitored table will be replaced with the current store with an additional DATE column, or there will be an additional table defined with the primary key and the DATE column. Either approach incurs a slight space penalty. Second, the archival store is created. Initially this table is empty, so the immediate cost is one of execution time for deletions and updates, which trigger insertions into the archival store. Depending on the volatility of the monitored table, the archival store will grow slowly or quickly; in any case, the growth will be monotonic. *Every* deletion or update of a row of the monitored table will insert a row into the archival store.

For highly volatile tables, the archival store can become quite large. In one sense, this is desirable because that store is capturing the time-varying behavior of the monitored table, permitting later analysis, which would be difficult to impossible without the archival store. However, eventually the space required by the archival store may become excessive. Transactions on the monitored table get progressively slower as insertions into the archival store take longer; queries on the virtual transaction-time state table also bog down due to the sheer size of the archival store.

The archival store can be reduced in size by purging information on invalid entities.

At some point the DBA may want to *vacuum* the archival store to remove less desired rows, thereby improving the space and time efficiency of the state table, with the drawback of reducing the querying capability of the state table.

There are several kinds of vacuuming operations. Two major classes are *entity vacuuming* and *temporal vacuuming*, distinguished by what is removed. In the former, entities that are judged to be less interesting are removed from the archival store. In the latter, information is removed based on it having been logically deleted at some specified instant in the past. The further back this instant is, the less information discarded.

As an example, we can purge all projections that are no longer currently valid.

Code Fragment 9.23   **Entity vacuum the archival store.**

```
DELETE FROM P_TT_PAST
WHERE NOT EXISTS ( SELECT *
                   FROM P_TT_CURRENT AS C
                   WHERE P_TT_PAST.PROJECTION_ID = C.PROJECTION_ID)
```

This works with either partitioned store organization described in Section 9.4. The danger here is that a projection might have been erroneously deleted only yesterday, yet we have just vacuumed away all evidence of that projection.

Once a transaction-time table has been vacuumed, subsequent queries should be interpreted with that in mind. The query of CF-9.8 needs to be rephrased: "When was it recorded that a *currently present* projection had a type of 17?"

Vacuuming is a dangerous operation because it violates the underlying semantics of the transaction-time state table. The state table allows all prior states of the monitored table to be reconstructed. After the state table has been vacuumed, it is no longer possible to fully reconstruct some past states. For entity vacuuming, the past states will not contain the entities that have been vacuumed.

> Although vacuuming a transaction-time table helps contend with the unchecked growth of the table, it violates the underlying semantics of the table, revising the meaning of subsequent queries.

On page 250, we argued that a view reconstructing a past state of a transaction-time table was inviolate; it will return the same rows independent of when the view was evaluated. Such is not the case in the presence of vacuuming. Before CF-9.23 was run by the DBA to entity vacuum the archival store, projection 1 was in the April 1, 1996, reconstructed state. After the archival store was vacuumed, projection 1 mysteriously disappeared from that state. Although this is probably what the DBA intended, users of the transaction-time table need to be aware that such vacuuming has occurred, and that the information in the transaction-time table is incomplete.

Entity vacuuming is one way to reduce the size of the archival store. Alternatively, we can temporally vacuum the archival store, removing all information older than, say, two years. Many countries have laws that require that certain records be retained only for a fixed length of time. Business policies may also pose similar requirements. Such situations indicate temporal vacuuming.

Code Fragment 9.24   **Temporally vacuum the archival store of data older than two years.**

```
DELETE FROM P_TT_PAST
WHERE (CURRENT_DATE - Stop_Date DAY) > INTERVAL '731' DAY
```

Again, after temporally vacuuming the archival store, queries need to be reinterpreted. The query of CF-9.8 needs to be rephrased: "When was it recorded *over the past two years* that a projection had a type of 17?"

In most settings, a combination makes sense: delete those entities having only old information, which is a more conservative stance than either entity vacuuming or temporal vacuuming alone.

Code Fragment 9.25   **Temporally vacuum old unused entities from the archival store.**

```
DELETE FROM P_TT_PAST
WHERE (CURRENT_DATE - Stop_Date DAY) > INTERVAL '731' DAY
```

```
AND NOT EXISTS ( SELECT *
                 FROM P_TT_CURRENT AS C
                 WHERE PROJECTION_ID = C.PROJECTION_ID)
```

Sometimes additional criteria may be appropriate; these tests can be added to the WHERE clause.

Code Fragment 9.26 **Temporally vacuum old unused entities from the archival store that had a USGS spheroid code of 2.**

```
DELETE FROM P_TT_PAST
WHERE (CURRENT_DATE - Stop_Date DAY) > INTERVAL '731' DAY
  AND NOT EXISTS ( SELECT *
                   FROM P_TT_CURRENT AS C
                   WHERE PROJECTION_ID = C.PROJECTION_ID)
  AND SPHEROID_CODE = 2
```

The DBA may wish to vacuum the archival store based on a combination of criteria.

The DBA can adjust the vacuuming criteria, trading off the realized space savings with the reduction in querying ability. A record should be kept, probably in a separate vacuuming log, of the cleaning that has taken place.

Code Fragment 9.27 **Log the vacuuming operations.**

```
CREATE TABLE Vacuum_Log (
  Table_Name CHAR(40) NOT NULL,
  When_Vacuumed DATE NOT NULL,
  Who CHAR(40) NOT NULL,
  Entity_Vacuuming CHAR(1) NOT NULL,
  Vacuum_Interval INTERVAL,
  Vacuum_Criteria CHAR(256)
  PRIMARY KEY (Table_Name, When_Vacuumed))
```

In this table, the `Entity_Vacuuming` column indicates whether old entities were purged, and the `Vacuum_Interval` indicates whether temporal vacuuming took place, with a NULL value indicating not. Other criteria used in the vacuuming, such as a USGS spheroid code of 2, can be indicated as a prose comment in `Vacuum_Criteria`.

The vacuum log, which indicates the meaning of queries on the state table, should be maintained automatically.

This log should be updated each time a state table is vacuumed. For this reason, it is best to define stored procedures that vacuum a table, given criteria such as whether to do entity vacuuming and the vacuuming interval, while also ensuring that the vacuuming is logged appropriately.

Vacuuming specifications should be monotonic to avoid time-dependent assumptions.

Vacuuming needs to be done in a way that makes sense to users. While complex predicates can be specified in the WHERE

clause of the DELETE statement effecting the vacuuming, the vacuuming crite-
ria should be carefully chosen. One desirable property of a vacuuming specifica-
tion is that it be *monotonic*—once it is satisfied, it will continue to be satisfied, so
that repeated application does not violate the specification. Consider the following
request:

Code Fragment 9.28   **Temporally vacuum the archival store between one and two years
old.**

```
DELETE FROM P_TT_PAST
WHERE (CURRENT_DATE - Stop_Date DAY) > INTERVAL '365' DAY
  AND (CURRENT_DATE - Stop_Date DAY) < INTERVAL '730' DAY
```

This deletion will repeatedly be applied, to keep the size of the archival store in
check. The problem is that data that is 15 months old, and hence is subject to dele-
tion, will eventually become two years old and should then be retained. However,
that data is gone; it cannot be later reconstituted. In contrast, the vacuuming spec-
ification of CF-9.24, "data older than two years," *is* monotonic: once data satisfies
this predicate, it will always satisfy the predicate. Similarly, the vacuuming specifi-
cations of CF-9.25 and CF-9.26 are also monotonic. We note in passing that a non-
monotonic vacuuming specification is an instance of a time-dependent assumption
(see page 66), to be avoided if at all possible.

9.6   IMPLEMENTATION CONSIDERATIONS

The code fragments were implemented in Microsoft SQL Server.

9.6.1   Microsoft SQL Server

Other than the proviso mentioned in Section 8.9.2 concerning using the `inserted`
and `deleted` tables rather than the `OLD` and `NEW` correlation names, all the code
fragments worked as stated on Microsoft SQL Server 6.5 and 7.0.

9.6.2   CD-ROM Materials

All of the code discussed in this chapter has been implemented in Microsoft SQL
Server and is provided on the CD-ROM.

9.7 SUMMARY

A transaction-time state table maintains a history of the changes that have been applied to a monitored table by associating with each row a period of presence, indicating when that row was present in the monitored table.

Several organizations for such state tables were considered:

- *As a period-stamped table augmenting the monitored table (CF-9.1), with triggers defined on the monitored table (CF-9.2).* Here there is some duplication, as rows of the monitored table will also be present in the state table. This approach does not require any changes to legacy code.
- *As a single period-stamped table, with the monitored table defined as a view on the state table, as discussed in the previous chapter.* Modifications must be rephrased to apply to the state table; queries can still reference the view.
- *As a current and an archival store (CF-9.15), with the monitored table defined as a view on the current store (CF-9.16).* The first line of modification statements must be changed to refer to the current store; triggers on this store automatically maintain the archival store. The current store is narrower, as the superfluous stop date is not stored. Sequenced and nonsequenced queries are applied to a view reconstituting the state table.
- *As a tripartitioned store: an archival store as before, a current store containing only the key and the start date (CF-9.20), and the monitored table, with triggers on the monitored table maintaining both stores (CF-9.21).* This organization requires no changes to the legacy code and materializes the monitored table for efficient current queries.

Transaction-time state tables have the same information content as the tracking logs considered in the previous chapter. We demonstrated this with views that extracted the before-images, after-images, and rows of a backlog. The previous chapter provided views that went the other direction, to a state table from a tracking log. A tracking log can be considered to be a temporally partitioned transaction-time table in which the archival store is timestamped with an instant, rather than a period.

To choose between the organizations that support transaction time presented in the last two chapters, the following factors should be considered:

- Are all insertions performed when the table is first created, or can entities come and go?
- Can an entity be deleted and later inserted?
- How critical is the space overhead?
- How critical is modification performance?
- How volatile is the monitored table?
- Which is most prevalent, current, sequenced, or nonsequenced queries?
- Is the legacy code available? If so, how hard is it to modify?

As a rough guide to selecting the manner in which transaction-time support is implemented, the following is suggested:

- Use a tracking log only if sequenced and nonsequenced queries are rare.
- Use before-images only if all insertions are performed when the table is created.
- Use after-images only if no insertions follow deletions.
- Use a backlog only if sequenced and nonsequenced queries are rare and if after-images are not indicated.
- Use a single or bipartitioned transaction-time state table if legacy code can be changed and space is tight.
- Use a state table along with the monitored table if changes to legacy code are not permitted and space isn't too tight.
- Use a tripartitioned state table if the legacy code cannot be changed and space is tight.

Vacuuming can be used to ameliorate the space overhead of maintaining all previous states of the monitored table. The DBA can configure the extent and timing of the purge process via vacuuming specifications; the DBA should ensure that these specifications are monotonic.

9.8 READINGS

Jacob Ben-Zvi originated the concept of a partitioned store in his Ph.D. dissertation [5]. Several others subsequently studied partitioned stores in the context of valid-time, transaction-time, and bitemporal tables [1, 26, 60, 67, 68].

Christian S. Jensen and Leo Mark have developed a comprehensive theory concerning vacuuming that includes the kinds of vacuuming discussed in this chapter [50]. Their approach goes further and considers the impact on query semantics and query evaluation, and discusses how to perform the vacuuming operation. Jensen later applied these concepts to a language extension to TSQL2 to support rudimentary vacuuming [47] and later still extended his theory of vacuuming [85]. These facilities have yet to be proposed for SQL3.

OVERVIEW

A bitemporal table is a glorious structure. It simultaneously records the history of the enterprise, while also capturing the sequence of changes to the record of that history. Bitemporal tables permit queries on the history as best known (over valid time, with a transaction time of "now"), queries on the change history of a stored data item (over transaction time, with a fixed valid time), and queries on the interaction of valid time and transaction time (for example, finding that information stored retroactively, after the fact). It is this range of queries that makes bitemporal tables so versatile and useful.

Bitemporal tables, in capturing both valid time and transaction time, require care in maintaining. Modifications must not disturb previously recorded information; they must be append-only. As such, modifications generally require longer sequences of more complex SQL statements.

Bitemporal tables admit a wide variety of temporal queries and integrity constraints. They also admit a variety of representational schemes, which speed up some common queries at the expense of some rarer queries.

Bitemporal Tables

I
nformation is *the* key asset of many companies. Nykredit, a major Danish mort-
gage bank, is a good example. In 1989, the Danish legislature changed the
Mortgage Credit Act to allow mortgage providers to market loans directly to
customers and through real estate agents; before such loans had to be funneled in-
directly through banks, with the mortgage providers separated from the consumer
by these middlemen.

This change in the law had a dramatic impact on the loan market, with the num-
ber of mortgage credit suppliers in Denmark doubling since the law's amendment.
This new environment represented both an opportunity to Nykredit to enter into
direct marketing, as well as a challenge to fend off its now-burgeoning competitors.

One of the challenges was achieving high data quality on the customers and
their loans, while expanding the traditional focus to also include customer sup-
port. Managers needed access to up-to-date data to set benchmarks and identify
problems in various areas of the business. The sheer volume of the data, nine mil-
lion loans to eight million customers concerning seven million properties, demands
that eliminating errors in the data must be highly efficient.

Jens Gadgaard is a seasoned senior architect at Nykredit Data, Nykredit's in-house
information technology provider, located in bucolic Aalborg. Aalborg is in Jutland,
to the north on the mainland, right on the Lim Fjord and only a few miles from the
North Sea coast. The company offices occupy a modern building several kilometers
outside of the city proper, fronted by a beautiful pond and surrounded by farmland;
I've seen sheep and rabbits contentedly grazing on adjacent fields and lawns. Next
door is Aalborg University, with its newly inaugurated Nykredit Center for Database
Research. Jens has long pushed for better management of temporal data; the Center
and the internal structure of property ownership tables reflect this emphasis.

As Jens explains it, a customer service person reports an error to the IT person-
nel; errors are also discovered by batch jobs producing quarterly reports. The more
information the IT personnel have access to, the better able they are to analyze and
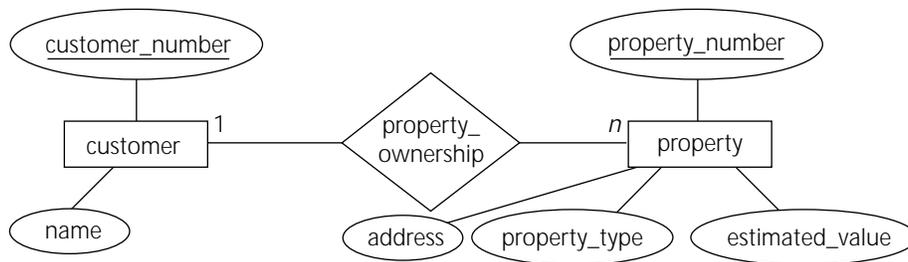
Figure 10.1 The property ownership relationship.

correct these errors. For this reason, Jens mandated that changes to critical tables be tracked. This implies that the tables have transaction-time support. As these tables also model changes in reality, they require valid-time support. The result is termed a *bitemporal table*, reflecting these two aspects of underlying temporal support.

   With such tables, IT personnel can first determine when the erroneous data was stored (a transaction time), roll back the table to that point, and look at the valid-time history. They can then determine what the correct valid-time history should be. At that point, they can tell the customer service person what needs to be changed, or if the error was in the processing of a user transaction, they may update the database manually. Because transaction-time support is included, these changes would be logged as well, enabling someone later to see what happened when the change itself was in error. Although bitemporal tables can be challenging to implement, their support for both valid time and transaction time permits a sophisticated analysis of the evolution of the table, with all the data directly at hand. The alternatives of going back through paper records to reconstruct the sequence of changes that were made, or attempting to extract that sequence from backup tapes or other secondary data sources, are simply not practical in such a dramatically changing environment.

   Nykredit must be doing something right. Despite the arrival of additional competitors, Nykredit was able to increase its market share to become the largest mortgage bank in Denmark by 1997 and is currently Europe's fifth largest provider of real estate loans.

10.1   DEFINITION

One of the central tables of the Nykredit database is the property_ownership table, which we will abbreviate as `Prop_Owner`. This table specifies the relationship between the customer and property entities, as shown in Figure 10.1 as an entity-relationship diagram.

The customer entity has a key of customer_number; the property entity has a key of property_number. (What kind of key, current, sequenced, or nonsequenced?, you may ask. The entity-relationship diagram does not say. We address this critical question only after we decide which aspects of time we wish to capture here.) The customer and property entities have additional attributes. The property_ownership relationship is a one-to-many relationship between customers and properties: a customer can own many properties, but a property is owned by exactly one customer. This relationship is itself associated with other attributes, not shown here.

In mapping this conceptual entity-relationship schema to a logical, relational schema, the customer entity induces a `Customer` table, with primary key `customer_number`, and the property entity induces a `Property` table, with primary key `property_number`. For the property_ownership relationship, we have two alternatives, given that this relationship is a one-to-many relationship. One alternative is to extend the `Property` table with a `customer_number` foreign key, as well as the relationship's attributes. However, since the relationship is not total/mandatory (a property can exist without being owned by a customer present in the Nykredit database), that implies that all of the property_ownership attributes must be nullable, including the `customer_number`. The second alternative, which is more attractive in this case, is to create a separate table, the `Prop_Owner` table, with foreign keys `customer_number` and `property_number`. None of these columns need be nullable.

It turns out that the `Customer` and `Property` tables are temporal tables, but we will defer consideration of that aspect. For the `Prop_Owner` table, Jens wanted to capture both the history in reality of the owner(s) of a property over time, as well as the

A bitemporal state table contains four timestamp columns, two specifying the period of validity and two specifying the period of presence.

sequence of database states, capturing the transactions applied to this table. This requirement, originating from a need for high data quality, meant that both valid time and transaction time were relevant for this table.

A bitemporal state table captures valid-time states via a period timestamp, here as two instant timestamps, `VT_Begin` and `VT_End`, and transaction-time states also via a period timestamp, here as `TT_Start` and `TT_Stop`. Along with the foreign keys, the table is comprised of six columns.

Code Fragment 10.1   **Create the** `Prop_Owner` **table.**

```
CREATE TABLE Prop_Owner (
  customer_number  INT,
  property_number  INT,
  VT_Begin  DATE,
  VT_End  DATE,
  TT_Start  TIMESTAMP,
  TT_Stop  TIMESTAMP)
```

We give the valid timestamps a granularity of day (as a property cannot change owners multiple times in a single day), and the transaction timestamps a granularity of microsecond (to differentiate transactions).

We also specify the `Customer` and `Property` tables as bitemporal tables.

```
CREATE TABLE Customer (
    name            CHAR,
    VT_Begin        DATE,
    VT_End          DATE,
    TT_Start        TIMESTAMP,
    TT_Stop         TIMESTAMP)

CREATE TABLE Property (
    property_number  INT,
    address          CHAR,
    property_type    INT,
    estimated_value  INT,
    VT_Begin         DATE,
    VT_End           DATE,
    TT_Start         TIMESTAMP,
    TT_Stop          TIMESTAMP)
```

Since the property_ownership relationship is one-to-many, the primary key of the `Prop_Owner` table should consist of only the many side, that is, the property foreign key: `property_number`. As has been emphasized many times, the sequenced semantics is generally the natural choice; what is desired here is to specify `property_number` to be a primary key sequenced in both valid time and transaction time. The state of the table at any day in valid time, as stored at any instant in transaction time, should include at most one row in the table for any particular property, meaning that property has one owner at that valid time, as recorded at that transaction time.

So, how do we convert a `property_number` PRIMARY KEY constraint to be sequenced in both valid time and transaction time? Since only current modifications are permitted on tables with transaction-time support, including the `TT_Start` column is sufficient (see the explanations on pages 178 and 221). However, arbitrary modifications (including sequenced and nonsequenced) are generally permitted on tables with valid-time support; that is true here specifically as well. As we saw in Section 5.3 on page 117, it is not sufficient to use either the begin date, the end date, or a combination of the two; instead, an assertion is needed. Leaving the valid timestamps out of the primary key doesn't work either, because the foreign keys plus the transaction start time does not differentiate multiple entries inserted by a single transaction specifying different valid times. Hence, a PRIMARY KEY constraint simply doesn't work in this case.

Components of Every Clock

Every periodic clock, save sundials and clepsydrae, has five primary components. The first is a device that will be periodic, termed a *resonator*. The twisting and untwisting of a string, the back-and-forth of a pendulum, and the vibrating of a crystal are all resonators. The second component is a means of supplying energy to the resonator, so that it doesn't wind down. The escapement provides energy from a falling weight to the twisting of the string and the swinging of a pendulum; a small battery provides energy to a crystal. The energy supply together with the resonator is called an *oscillator*. The third component is a counting device, such as an escapement or solid-state circuit. The fourth is a transmission, getting the count to the fifth component, which is a display, such as the clock's hands.

Stating that a key on a bitemporal table is valid-time sequenced requires an assertion.

To specify that the `property_number` column constitutes a transaction-time sequenced, valid-time sequenced primary key, we need an assertion. We will apply this assertion at the current transaction time; that only current modifications are permitted in transaction time will ensure that it holds over all transaction-time states. This is a slight modification of CF-5.14 on page 124.

Code Fragment 10.2   `property_number` **is a (valid-time sequenced, transaction-time sequenced) primary key for** `Prop_Owner`**.**

```
CREATE ASSERTION P_O_seq_primary_key
CHECK (NOT EXISTS (SELECT *
          FROM Prop_Owner AS P1
          WHERE property_number IS NULL
          OR 1 < (SELECT COUNT(customer_number)
                  FROM Prop_Owner AS P2
                  WHERE P1.property_number = P2.property_number
                    AND P1.VT_Begin < P2.VT_End
                    AND P2.VT_Begin < P1.VT_End
                    AND P1.TT_Stop = DATE '9999-12-31'
                    AND P2.TT_Stop = DATE '9999-12-31'))
      )
```

While we're at it, we also include a nonsequenced valid-time assertion: that there are no gaps in the valid-time history. Specifically, once a property is acquired by a customer, it remains associated with an owner (or sequence of owners) over its existence. As before, we use current transaction time. This is a slight modification of CF-5.22 on page 129.

Code Fragment 10.3   `Prop_Owner.property_number` **defines a contiguous valid-time history.**

```
CREATE ASSERTION P_O_Contiguous_History
CHECK (NOT EXISTS (SELECT *
        FROM Prop_Owner AS P, Prop_Owner AS P2
       WHERE P.VT_End < P2.VT_Begin
         AND P.property_number = P2.property_number
         AND P.TT_Stop = DATE '9999-12-31'
         AND P2.TT_Stop = DATE '9999-12-31'
         AND NOT EXISTS (
             SELECT *
             FROM Prop_Owner AS P3
             WHERE P3.property_number = P.property_number
               AND (((P3.VT_Begin <= P.VT_End)
                     AND (P.VT_End < P3.VT_End))
                   OR ((P3.VT_Begin < P2.VT_Begin)
                     AND (P2.VT_Begin <= P3.VT_End)))
                 AND P3.TT_Stop = DATE '9999-12-31'))
       )
```

The change made here was to apply the assertion at the current transaction time, which, due to the append-only nature of transaction time, renders it a sequenced transaction-time constraint.

10.2   MODIFICATIONS

In Chapter 7 we saw that valid-time state tables admit nine kinds of modifications: current, sequenced, and nonsequenced versions of INSERT, DELETE, and UPDATE. Chapter 9 showed that transaction-time state tables are much simpler: only current versions of INSERT, DELETE, and UPDATE are relevant. So, what is the situation with bitemporal tables? It turns out that here again only nine kinds of modifications apply, all current in transaction time: valid-time current, valid-time sequenced, and valid-time nonsequenced versions of INSERT, DELETE, and UPDATE.

Translating modifications on bitemporal tables into SQL parallels the translation on valid-time tables given in Chapter 7. In fact, we advocate a two-stage transformation. The first stage applies those transformations given for valid-time tables. Only then will we consider transaction time, applying a second set of transformations to render a final sequence of SQL statements that respect the coupled semantics of valid time and transaction time. The mappings themselves are straightforward, albeit somewhat tedious.

Let's follow the history, over both valid time and transaction time, of a flat in Aalborg, at Skovvej 30 for the month of January 1998. This history is quite interesting for illustrating the translation for various kinds of modifications.

10.2.1 Current Modifications

We first consider current insertion, then current updates, and end with current deletions.

Insertions

On the 10th of January, this flat was purchased by Eva Nielsen. We record this information as a current valid-time, current transaction-time insertion.

Code Fragment 10.4   **Eva Nielsen buys the flat at Skovvej 30 in Aalborg on January 10, 1998.**

```
INSERT INTO Prop_Owner (customer_number, property_number, VT_Begin,
    VT_End, TT_Start, TT_Stop)
VALUES (145, 7797, CURRENT_DATE,
    DATE '9999-12-31', CURRENT_TIMESTAMP, DATE '9999-12-31')
```

This information is valid starting now and was inserted now. We will see that the transaction-time extent of *all* modifications is "now" to "until changed," which we encode as "forever."

The interplay between valid time and transaction time can be confusing, so it is useful to have a visualization of the information content of a bitemporal table. Figure 10.2 shows the *bitemporal time diagram*, or simply time diagram, corresponding to the above insertion.

Current insertions require only that the valid and transaction timestamps be appropriately specified.

In this figure, the horizontal axis tracks transaction time and the vertical axis tracks valid time. Information about a row, or about multiple rows associated with a primary key value, is depicted as two-dimensional polygonal regions in the diagram. Arrows extending rightward denote "until changed" in transaction time; arrows extending upward denote "forever" in valid time. Here we have but one region, associated with Eva Nielsen, that starts at time 10 (January 10, 1998) in transaction time and extends to "until changed," and that begins also at time 10 in valid time and extends to "forever." The arrow pointing upward extends to the largest valid-time value ("forever"); the arrow pointing to the right extends to "now," that is, it advances day by day to the right (a transaction time in the future is meaningless).
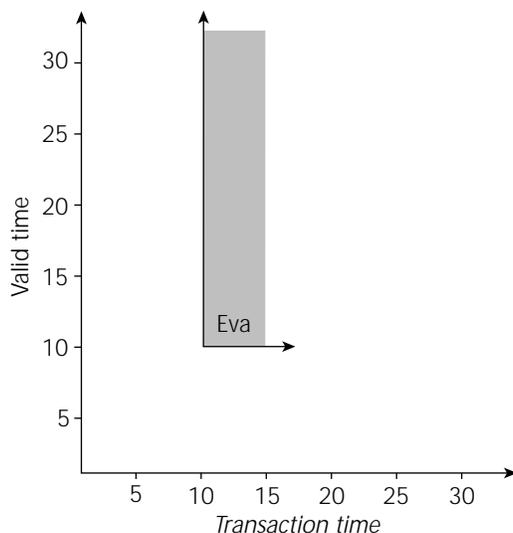
Figure 10.2 A bitemporal time diagram corresponding to Eva purchasing the flat, performed on January 10.

Updates

On the 15th Peter Olsen buys this flat; this legal transaction transfers ownership from Eva to him. The nontemporal expression of this modification is a simple UPDATE.

Code Fragment 10.5   **Peter Olsen buys the flat on January 15, 1998.**

```
UPDATE Prop_Owner
SET customer_number = 827
WHERE property_number = 7797
```

Figure 10.3 illustrates how this update impacts the time diagram. The valid-time extent of a current modification is always "now" to "forever," so from time 15 on, the property is owned by Peter; at the rest of the time, from time 10 to 15, the property was owned by Eva. Both regions extend to the right to "until changed." This time diagram captures two facts: Eva owning the flat and Peter owning the flat, each associated with a bitemporal region.

This figure captures the evolving information content of the `Prop_Owner` table quite effectively. Consider a transaction time-slice, which returns the valid-time history at a given transaction time. Such a time-slice can be visualized as a vertical line intersecting the *x*-axis at the given time. At transaction time 5 (January 5), the table has no record of the flat being owned by anyone. At transaction time 12, the table records that the flat was owned by Eva from January 10 to "forever." If we
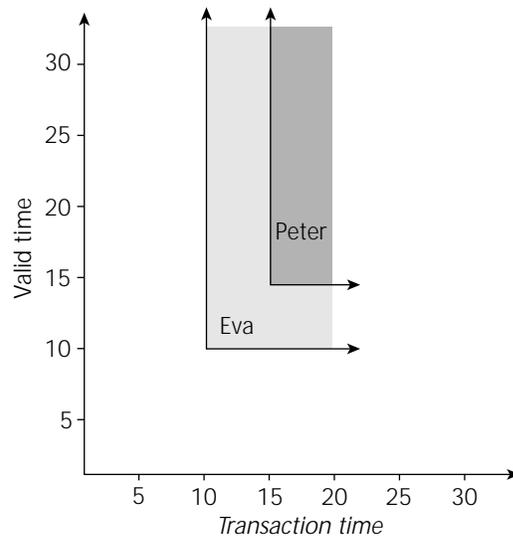
Figure 10.3 A current update: Peter buys the flat, performed on January 15.

time-traveled back to January 12 and asked for the history of the flat, that would be the response. We *thought* then that Eva owns the flat, and that is what the `Prop_Owner` table recorded then. At transaction time 17 the table records that the flat was owned by Eva from January 10 to 15, at which time ownership transferred to Peter, who now owns it to "forever." And that is the history as best known (denoted by the right-pointing arrows); it is what we think is true about the valid-time history.

The current update must effect the change in the `Prop_Owner` table from Figure 10.2 to Figure 10.3, via SQL statements that manipulate the explicit columns as well as the timestamp columns. While CF-10.5 consists of but 3 lines, the translation into SQL-92 is involved, ultimately requiring over 30 lines of SQL! However, the translation is mechanical, so with some patience, all will be clear.

For modifications on bitemporal tables, the first stage contends with valid time, resulting in a series of SQL statements.

CF-7.11 on page 186 translated a valid-time current update into three SQL statements, the first to insert new information valid from "now" until the row ended, the second to terminate the current row at "now," and the third to update any rows that start in the future with the new values. I urge you to revisit that code fragment, read the associated commentary, and ensure that you understand it well because the bitemporal version will build on it.

In addition to contending with valid time, we also must ensure that the transaction-time extent of the modification is from "now" to "until changed." One important property of tables with transaction-time support is that they are *append-only* (cf. Section 8.10). As such tables capture the state of the stored table over time,

once we have recorded that the state was such and such at a particular time, we can't go back and change that later because we can't change the bits stored on the disk at that prior time. The changes always accumulate in the table with transaction-time support (the one exception is when the table is vacuumed, which as we emphasized on page 270 violates the semantics of the table).

Only two kinds of modifications are permitted on bitemporal state tables: insertions with a transaction time of "now" to "forever," and updates that set the transaction-stop time to "now."

The practical ramification is that we never physically delete a row from such a table; the only physical modifications allowed are to insert rows into the table and to change the transaction-stop time of a row from "until changed" to "now," thereby logically deleting the row. All nine types of modifications allowed on bitemporal tables *must* be implemented as a combination of INSERTs with a transaction time of "now" to "until changed" (which is represented with "forever") and UPDATEs that set the transaction-stop time to "now." Any other modification to a bitemporal table will violate its semantics.

As mentioned, we utilize a two-stage procedure for translating a temporal modification on a bitemporal table into a series of SQL statements. The first stage pretends that the table is a valid-time table and uses the mappings elaborated in Chapter 7 for handling modifications of such tables. In the second stage, we identify the statements that violate the semantics of transaction time, which basically are all DELETEs and UPDATEs, and further map these statements into combinations of UPDATEs on the transaction-stop time and INSERTs. Finally, we must add a predicate for each correlation name in the statement that selects the most recent transaction-time version; this can be done by simply checking that the transaction-stop time is "until changed."

A modification on a bitemporal table

A nontemporal modification is mapped into a modification on a bitemporal table in two stages:

> The first transformation assumes that the table only has valid-time support.

> The second transformation then converts updates and deletions into stylized INSERT and UPDATE statements. A WHERE predicate is added for each correlation name selecting the row(s) current in transaction time.

For our first modification, a current insertion, shown in CF-10.4, we didn't need to invoke this second transformation stage, but most of the other temporal modifications will require both stages.

We now examine the three statements of CF-7.11 on page 186, which effects a current update on a valid-time state table. The first stage maps the update of CF-10.5 into the following three statements, which parallel the three statements of CF-7.11.

Code Fragment 10.6   **Peter Olsen buys the flat on January 15, 1998, a current update (partial solution, considering only valid time).**

```
INSERT INTO Prop_Owner
SELECT 827, property_number, CURRENT_DATE, VT_End
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin <= CURRENT_DATE
  AND VT_End > CURRENT_DATE

UPDATE Prop_Owner
SET VT_End = CURRENT_DATE
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE

UPDATE Prop_Owner
SET customer_number = 827
WHERE property_number = 7797
  AND VT_Begin > CURRENT_DATE
```

The first statement in this fragment is an insertion, which is fine, as it upholds the semantics of transaction time. The second statement, which terminates the valid time of the current row at "now," is an update of a column other than the transaction-stop time. So it must be subsequently mapped (as we'll show shortly) into two SQL statements, one to logically delete the entire row, by setting the transaction-stop time to "now," and one to insert the row, with a new valid-time stop time of "now." The third statement, to update any rows that start in the future with new values, must also be subsequently mapped into two SQL statements, an UPDATE and an INSERT. So the second and third statements of CF-10.6 must each be expanded into two statements, yielding a total of five SQL statements expressing the sequenced version of CF-7.11.

There is a further complication in the interplay of two time dimensions, valid time and transaction time. When we were concerned only with valid time, we contended with valid-time periods, lengthening and shortening these periods by altering their beginning and ending instants. Two time dimensions generalize periods to *regions* in the time diagram, which are considerably more involved than simple one-dimensional periods. Quite intricate shapes can result from a series of bitemporal modifications, as we shall see.

Hairspring

A pendulum must be hung vertically, making it fine for grandfather clocks but impractical for wrist watches. Huygens also had the insight that a spring has the same characteristics as a pendulum, but is unaffected by its spatial orientation. His spiral spring has been refined into the hairspring that regulates the motion of the balance wheel in today's mechanical watches.

In terms of the time diagram, a row with two valid-time instants, VT_Begin and VT_End, and two transaction-time instants, TT_Start and TT_Stop, encodes a *rectangle* in bitemporal space. Hence, the region in Figure 10.2, being a single rectangle, requires but one row to encode—the row inserted in CF-10.4. The region associated with Eva in Figure 10.3 requires two rectangles; the region associated with Peter needs but one (see Figure 10.4). Due to the semantics of transaction time, regions are often split with vertical lines in time diagrams. The implication is that in this case two new rows will have to be inserted, and the existing row modified, to effect this temporal modification. (This is the visual analog of the requirement, stated earlier, that an UPDATE of a table with transaction-time support must be mapped into an UPDATE only of the transaction-stop time to "now," and an INSERT with a transaction-start time of "now.")

Returning to the three statements needed to realize a current deletion of a valid-time table, we note that the third statement, to update any rows that start in the future, is not strictly required here, as there are no rows concerning this flat that start in the future. However, for generality, because your application may indeed
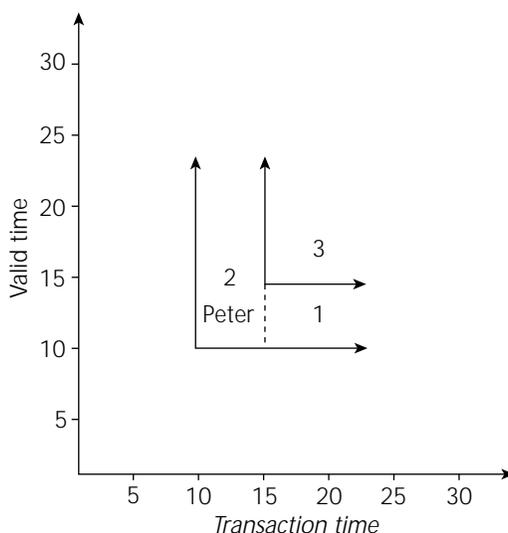
Figure 10.4 Splitting a polygonal region into rectangles.
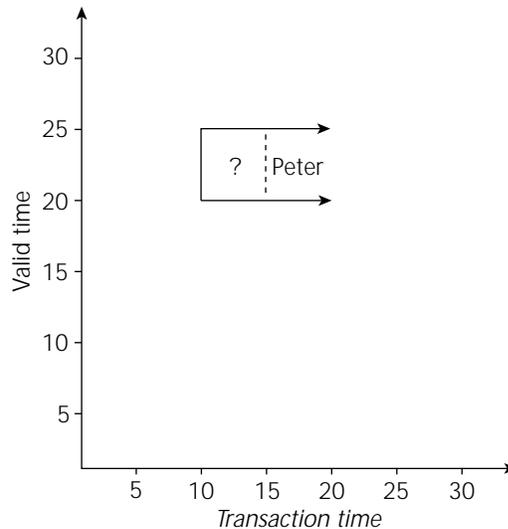
Figure 10.5 A current update of a future row.

have (valid-time) future rows, we include this statement in our discussion here. Say there was a future row, valid from 20 to 25, with a transaction-stop time of "until changed" (see Figure 10.5). Here the flat was owned during these five days (from January 20 to January 24; recall that we're using an open-ended period) by someone else. A current update, "Peter Olsen buys the flat," has a valid-time extent of "now" to "forever"; this extent includes all of January. So we must logically delete the old row and insert a new row, indicating associating that valid time with Peter. The third statement thus maps into two SQL statements, an INSERT and an UPDATE.

In summary, the first stage maps CF-10.5 to CF-10.6, resulting in three SQL statements. The second stage retains the INSERT statement, but maps each of the two UPDATE statements into an INSERT-UPDATE pair. Five SQL statements result, shown as CF-10.7. You should verify that these statements do not violate the transaction-time semantics: all INSERTs have transaction-time extent of "now" to "until changed," and all UPDATEs change only the transaction-stop time from "until changed" to "now," with no other kinds of modifications allowed.

Code Fragment 10.7   Peter Olsen buys the flat on January 15, 1998, a current update.

```
INSERT INTO Prop_Owner
SELECT 827, property_number, CURRENT_DATE, VT_End,
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
```

```
WHERE property_number = 7797
  AND VT_Begin <= CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, CURRENT_DATE,
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT 827, property_number, VT_Begin, VT_End,
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'
```

The first UPDATE must occur after the first two INSERTs; the last UPDATE must occur after the last INSERT. The resulting Prop_Owner table (shown in Table 10.1) contains three rows, corresponding to the three rectangles in Figure 10.4. A careful matching of the dates in this table to the time diagram will aid in understanding how a bitemporal state table encodes the regions found in the time diagram.

Table 10.1 Result of the current insertion.

customer_number	property_number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	*1998-01-15*
145	*7797*	*1998-01-10*	*1998-01-15*	*1998-01-15*	*9999-12-31*
827	*7797*	*1998-01-15*	*9999-12-31*	*1998-01-15*	*9999-12-31*

Deletions

We perform a current deletion on January 20 against Table 10.1. Specifically, we find out that Peter has sold the property to someone else, with the mortgage handled by another mortgage company. From Nykredit's point of view, the property no longer exists as of (a valid time of) January 20.

Code Fragment 10.8   Peter Olsen sells the flat on January 20, 1998.

```
DELETE FROM Prop_Owner
WHERE property_number = 7797
```

Figure 10.6 shows the resulting time diagram. If we now request the valid-time history as best known, we will learn that Eva owned the flat from January 10 to January 15, and Peter owned the flat from January 15 to January 20. Note that all prior states are retained. We can still time-travel back to January 18 and request
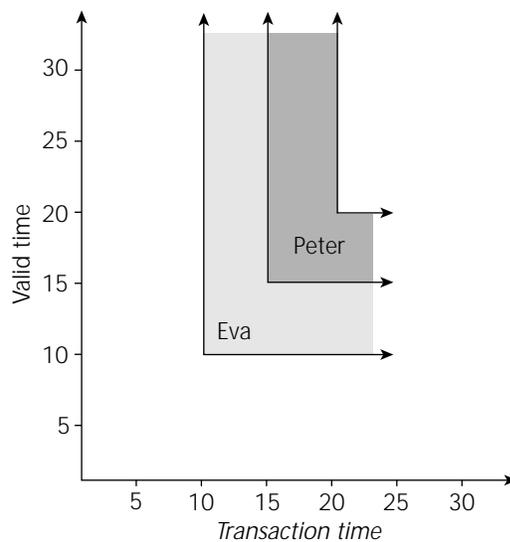
Figure 10.6 A current deletion: Peter sells the flat, performed on January 20.
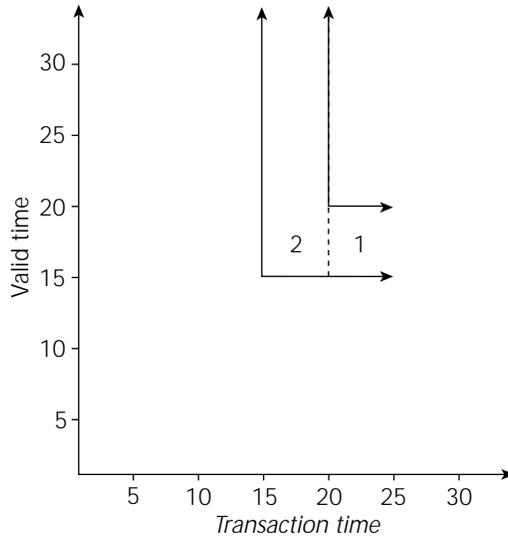
Figure 10.7 A current deletion: splitting into rectangles.

the valid-time history, which will state that on that day we thought that Peter still owned the flat. In Figure 10.3, Peter's region was a rectangle. The current deletion has chopped off the top-right corner, so that the region is now L-shaped.

The row associated with Peter in Figure 10.3 denotes a single rectangle. That rectangle must be converted into the two rectangles shown in Figure 10.7. We do so by terminating the existing row (by setting its transaction-stop time to "now") and by inserting the portion still present, with a valid-end time of "now."

As before, we proceed in two stages. The first stage pretends that the table is a valid-time state table, mapping the temporal modification to SQL. As in CF-7.8 on page 184, two statements are required.

Code Fragment 10.9   **Peter Olsen sells the flat on January 20, 1998, a current deletion (partial version, considering only valid time).**

```
UPDATE Prop_Owner
SET VT_End = CURRENT_DATE
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE

DELETE FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin >= CURRENT_DATE
```

Deletions on bitemporal tables follow these same two stages: first consider valid time, then transaction time.

In the first statement, applied to rows that started in the past and end in the future, the valid-time end date is set to "now." The second deletes those rows that start now or in the future.

The second stage further maps those statements that violate the transaction-time semantics into particular forms of UP-DATEs and INSERTs. The first statement (an UPDATE) is mapped into two, an INSERT and an UPDATE; the second (a DELETE) is mapped into an UPDATE. We also add a predicate for each correlation name that checks for the current transaction-time version. Again, you should be convinced that these statements do not violate the semantics of transaction time.

Code Fragment 10.10   **Peter Olsen sells the flat on January 20, 1998, a current deletion.**

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, CURRENT_DATE,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin >= CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'
```

We can simplify this by combining the two UPDATEs into one.

Code Fragment 10.11   **Peter Olsen sells the flat on January 20, 1998, a current deletion, simplified version.**

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, CURRENT_DATE,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
```

Table 10.2 Result of the current deletion.

customer_number	property_number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	9999-12-31
827	7797	1998-01-15	9999-12-31	1998-01-15	*1998-01-20*
827	*7797*	*1998-01-15*	*1998-01-20*	*1998-01-20*	*9999-12-31*

```
WHERE property_number = 7797
  AND VT_Begin < CURRENT_DATE
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_End > CURRENT_DATE
  AND TT_Stop = DATE '9999-12-31'
```

The UPDATE handles both current and future rows. The resulting table (Table 10.2) contains four rows. The third row was terminated at "now," with the fourth row newly inserted. The modified rows and columns are highlighted with an *italic* font.

In current modifications, valid time and transaction time are coupled: the valid time at which the modification takes effect is "now." Similarly, the transaction time at which the modification is recorded is "now." Sequenced modifications decouple the valid time from the transaction time, allowing the former to be supplied by the user.

10.2.2   Sequenced Modifications

As we saw in Chapter 7, sequenced modifications generalize current modifications to apply over a specified period of applicability. For bitemporal tables, the modification is sequenced only on valid time; the modification is *always* a current modification on transaction time, from "now" to "until changed."

As before, we apply a two-stage transformation, first considering only the valid-time component of the bitemporal table, then further transforming the SQL statements so that they do not violate the semantics of transaction time. The result will be a series of stylized UPDATE and INSERT statements.
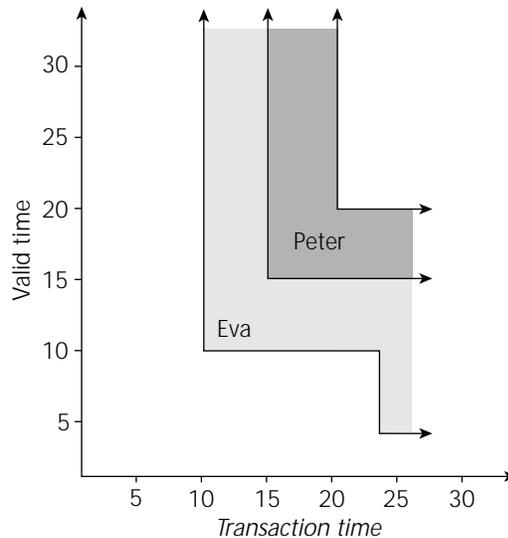
Figure 10.8 A sequenced insertion performed on January 23: Eva actually purchased the flat on January 3.

Insertions

Continuing with the `Prop_Owner` table as depicted in Figure 10.6, we consider a sequenced insertion. On January 23, we find out that Eva had purchased the flat not on January 10, but on January 3, a week earlier. So we insert those additional days, to obtain the time diagram shown in Figure 10.8.

This insertion is termed a *retroactive* modification, as the period of applicability (here, January 3 through 10) is before the modification date (here, January 23). Sequenced (and nonsequenced) modifications can also be *postactive*, an example being a promotion that will occur in the future (in valid time). (A valid-end time of "forever" is generally not considered a postactive modification; only the valid-start time is considered.) A sequenced modification might even be simultaneously retroactive, postactive, and current, when its period of applicability starts in the past and extends into the future (e.g., a fixed-term assignment that started in the past and ends at a designated date in the future).

There are two ways to effect the insertion illustrated in Figure 10.8. The easiest is to use a single SQL INSERT statement.

Code Fragment 10.12   **Eva actually purchased the flat on January 3, performed on January 23.**

```
INSERT INTO Prop_Owner (customer_number, property_number, VT_Begin,
    VT_End, TT_Start, TT_Stop)
VALUES (145, 7797, DATE '1998-01-03',
    DATE '1998-01-10', CURRENT_TIMESTAMP, DATE '9999-12-31')
```
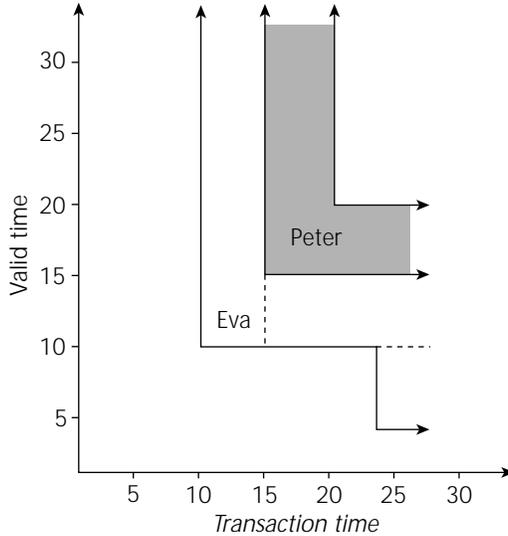
Figure 10.9 One splitting into rectangles.

Table 10.3 Result of the sequenced insertion.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	9999-12-31
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	9999-12-31
145	*7797*	*1998-01-03*	*1998-01-10*	*1998-01-23*	*9999-12-31*

One approach to a sequenced insertion is to simply insert the new period of validity, without regard to how it interacts with the period of validity of the existing rows.

The period of applicability for the insertion appears as the values of the VT_Begin and VT_End columns. The region associated with Eva's ownership consists of the three rectangles shown in Figure 10.9, and the first, second, and fifth rows in Table 10.3.

A second approach is to always split with vertical lines, as shown in Figure 10.10. This is termed *transaction-time splitting* because the regions are split into bands of transaction time. (The first approach is then termed *valid-time splitting*.) In transaction-time splitting, we terminate the current row and insert a new row, with a new period of validity being the union of the original period of validity and the period of applicability, that is, from time 5 to time 15. For this new period, there are four cases, shown in Figure 10.11. In all four cases, the original period of
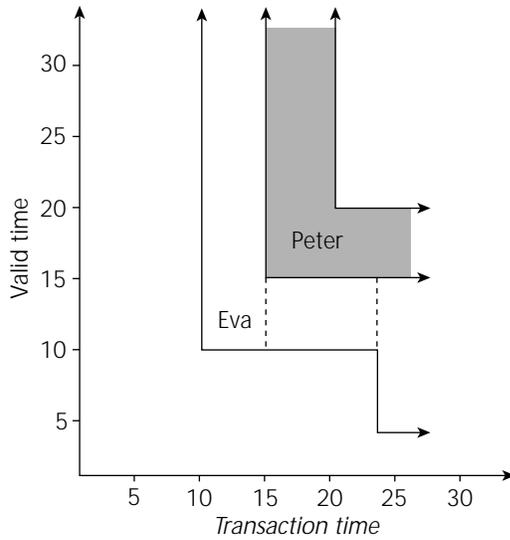
Figure 10.10 An alternate splitting into rectangles.
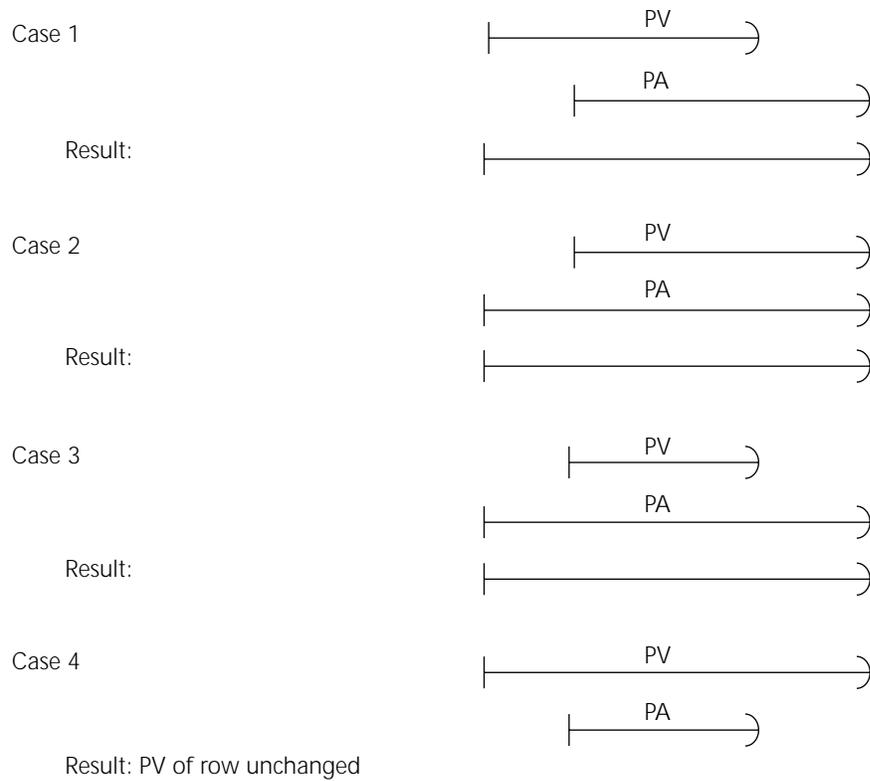
Figure 10.11 Sequenced insertion cases.

validity (PV) overlaps the period of applicability of the insertion (PA). In Case 1, PV starts before PA, with the new period of validity starting from the start of PV to the end of PA. This case applies to the Eva row in this example. In Case 2, the PV starts after PA. In Case 3, the PV is contained in PA, and the new PV is PA. In Case 4, the PA is contained in PV, and the PV need not be changed.

In the following SQL code, we use a CASE statement to compute the new period of validity. Rows are affected if their PV overlaps the PA but does not contain the PA. Such rows are logically deleted by setting the transaction-stop time to "now," then inserted with the new PV. If there are no rows whose PV overlaps the PA, the insertion is performed as before.

Code Fragment 10.13   **Eva actually purchased the flat on January 3, with transaction-time splitting.**

```
-- Do normal insert if there are no overlapping rows that
-- do not contain the period of applicability
INSERT INTO Prop_Owner
SELECT 145, 7797, DATE '1998-01-03', DATE '1998-01-10',
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM DUAL
WHERE NOT EXISTS (SELECT *
                    FROM Prop_Owner
                      WHERE customer_number = 145
                        AND property_number = 7797
                        AND DATE '1998-01-03' < VT_End
                        AND VT_Begin < DATE '1998-01-10'
                        AND NOT (VT_Begin < DATE '1998-01-03'
                            AND DATE '1998-01-10' < VT_End)
                        AND TT_Stop = DATE '9999-12-31')
-- If there is an overlap, extend it, unless PA is contained in PV
INSERT INTO Prop_Owner
SELECT customer_number, property_number,
  CASE WHEN DATE '1998-01-03 < VT_Begin
    THEN DATE '1998-01-03'
    ELSE VT_Begin END,
  CASE WHEN DATE '1998-01-10 < VT_End
    THEN VT_End
    ELSE DATE '1998-01-10' END,
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE customer_number = 145
  AND property_number = 7797
  AND DATE '1998-01-03' < VT_End
  AND VT_Begin < DATE '1998-01-10'
  AND NOT (VT_Begin < DATE '1998-01-03'
```

```
      AND DATE '1998-01-10' < VT_End)
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE customer_number = 145
  AND property_number = 7797
  AND DATE '1998-01-03' < VT_End
  AND VT_Begin < DATE '1998-01-10'
  AND NOT (VT_Begin < DATE '1998-01-03'
    AND DATE '1998-01-10' < VT_End)
  AND TT_Stop = DATE '9999-12-31'
```

A second approach to a sequenced insertion on a bitemporal table computes a new period of validity.

(Note the use of a `DUAL` table, as discussed on page 138.) The three statements can be in any order, as they are disjoint.

This approach minimizes the representation, that is, the number of rows in the `Prop_Owner` table (though in this particular example, there is no difference). As shown in Table 10.4, this table still has five rows (cf. Table 10.3), but here the second row was logically deleted. The drawback of this optimized approach is substantial complexity and execution cost for the temporal insertion.

For the remainder of the modifications, we will not attempt to minimize the representation, but note here that doing so is always an option to be considered.

Deletions

We learn on January 26 that Eva bought the flat not on January 10, as initially thought, nor on January 3, as later corrected, but on January 5. This requires a sequenced version of the following deletion:

Table 10.4 Result of a second approach to the sequenced insertion.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	*1998-01-23*
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	9999-12-31
145	*7797*	*1998-01-03*	*1998-01-15*	*1998-01-23*	*9999-12-31*
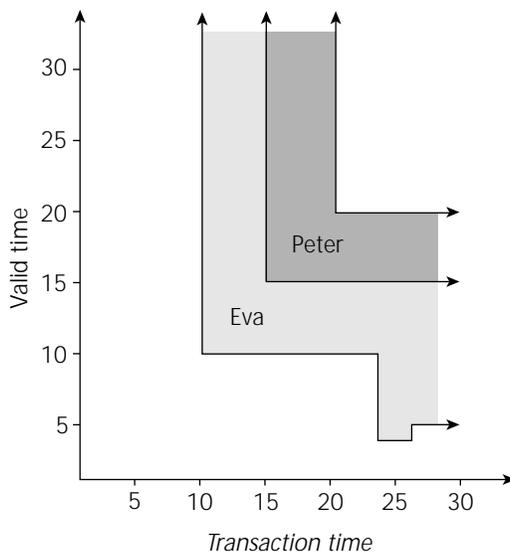
Figure 10.12 A sequenced deletion performed on January 26: Eva actually purchased the flat on January 5.

Code Fragment 10.14   **Eva actually purchased the flat on January 5 (nontemporal version).**

```
DELETE FROM Prop_Owner
WHERE property_number = 7977
```

We specify a period of applicability of January 3 through 5, with the result shown in the time diagram in Figure 10.12.

We need to terminate the current row and insert a new row, with a smaller period of validity. As in the valid-time sequenced deletion of CF-7.16 on page 191, there are four cases, depicted in Figure 7.2. The valid-time sequenced deletion was transformed into four SQL statements.

Valid-time sequenced deletion

Insert the old values from the end of the period of applicability to the end of the period of validity of the original row.

Update the end date to end at the beginning of the period of applicability.

Update the start date to begin at the end of the period of applicability.

Delete entirely rows that are covered by the period of applicability.

When transaction time is considered (recall that all modifications are current in transaction time), updates turn into a combination of terminating the row with a transaction-stop time of "now" and inserting a row with a transaction-start time of "now" and with the new valid-time date. This concerns both the second and the third statements. Logically deleting a row turns into an UPDATE, when transaction time is considered.

Sequenced deletions on valid-time tables require some four SQL statements; when mapped to bitemporal tables, a total of six statements are required.

In summary, we took CF-7.16, applied it to the Prop_Owner table, being careful to utilize a period of applicability of [1998-01-02 - 1998-01-05), then applied the second stage of the transformation, to obtain six SQL statements, all consistent with transaction-time semantics. We admit that in this particular situation, the first two statements suffice to effect the deletion. However, our previous explanation and provided code cover *all* situations.

Code Fragment 10.15   **Eva actually purchased the flat on January 5.**

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, DATE '1998-01-05', VT_End,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin < DATE '1998-01-02'
  AND VT_End > DATE '1998-01-05'
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, DATE '1998-01-02',
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin < DATE '1998-01-02'
  AND VT_End > DATE '1998-01-02'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin < DATE '1998-01-02'
  AND VT_End > DATE '1998-01-02'
  AND TT_Stop = DATE '9999-12-31'
```

Table 10.5 Result of the sequenced deletion.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	9999-12-31
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	9999-12-31
145	7797	1998-01-03	1998-01-10	1998-01-23	*1998-01-26*
145	*7797*	*1998-01-05*	*1998-01-10*	*1998-01-26*	*9999-12-31*

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, DATE '1998-01-05', VT_End,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin < DATE '1998-01-05'
  AND VT_End >= DATE '1998-01-05'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin < DATE '1998-01-05'
  AND VT_End >= DATE '1998-01-05'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797
  AND VT_Begin >= DATE '1998-01-02'
  AND VT_End <= DATE '1998-01-05'
  AND TT_Stop = DATE '9999-12-31'
```

Starting from Table 10.3, the sequenced deletion of CF-10.15 results in Table 10.5.

Updates

We learn on January 28 that Peter bought the flat on January 12, not January 15 as previously thought. This requires a sequenced version of the following update.
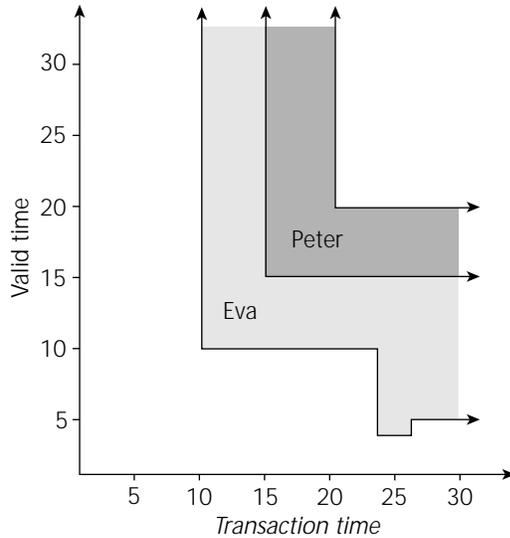
Figure 10.13 A sequenced update performed on January 28: Peter actually purchased the flat on January 12.

Code Fragment 10.16   **Peter actually purchased the flat on January 12 (nontemporal version).**

```
UPDATE Prop_Owner
SET customer_number = 145
WHERE  property_number = 7797
  AND customer_number <> 145
```

Table 10.6 Result of the sequenced update.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	*1998-01-28*
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	*1998-01-28*
145	7797	1998-01-03	1998-01-10	1998-01-23	1998-01-26
145	7797	1998-01-05	1998-01-10	1998-01-26	*1998-01-28*
145	*7797*	*1998-01-05*	*1998-01-12*	*1998-01-28*	*9999-12-31*
827	*7797*	*1998-01-12*	*1998-01-20*	*1998-01-28*	*9999-12-31*

This update requires a period of applicability of January 12 through 15, setting the customer_number to 145, which results in the time diagram in Figure 10.13. Effectively, the ownership must be transferred from Eva to Peter for those three days, resulting in Table 10.6.

The two-stage transformation, first valid time and then transaction time, applies here as well. We modify CF-7.18 on page 194 to apply to Prop_Owner and to utilize a period of applicability of [1998-01-12 - 1998-01-15), then contend with transaction time in the second stage. INSERTs remain; UPDATEs are mapped to a pair of INSERT and UPDATE.

Code Fragment 10.17   **Peter actually purchased the flat on January 12.**

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, DATE '1998-01-12',
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-12'
  AND VT_End > DATE '1998-01-12'
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT customer_number, property_number, DATE '1998-01-15', VT_End,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-15'
  AND VT_End > DATE '1998-01-15'
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT 145, property_number, VT_Begin, VT_End,
    CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-15'
  AND VT_End > DATE '1998-01-12'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998'01-15'
  AND VT_End > DATE '1998-01-12'
  AND TT_Stop = DATE '9999-12-31'
```

```
INSERT INTO Prop_Owner
SELECT customer_number, property_number, DATE '1998-01-12', VT_End,
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-12'
  AND VT_End > DATE '1998-01-12'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-12'
  AND VT_End > DATE '1998-01-12'
  AND TT_Stop = DATE '9999-12-31'

INSERT INTO Prop_Owner
SELECT customer_number, property_number, VT_Begin, DATE '1998-01-15',
  CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM Prop_Owner
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-15'
  AND VT_End > DATE '1998-01-15'
  AND TT_Stop = DATE '9999-12-31'

UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE property_number = 7797 AND customer_number <> 145
  AND VT_Begin < DATE '1998-01-15'
  AND VT_End > DATE '1998-01-15'
  AND TT_Stop = DATE '9999-12-31'
```

Sequenced updates require applying the same two-stage transformation process, resulting in some eight SQL statements to implement a single sequenced update.

While this series of statements is quite intimidating, the two stages employed—that of mapping from the initial sequenced update to the first set of five statements, taking the valid-time component into consideration, as discussed in Chapter 7, then mapping into the eight statements shown here, taking the transaction-time component into consideration—are largely mechanical and more tedious than conceptually challenging.

10.2.3   Nonsequenced Modifications

We saw before that no mapping was required for nonsequenced modifications on valid-time state tables; such statements treat the (valid) timestamps identically to the other columns. When considering the transaction timestamps, we just perform the second-stage mapping discussed above.
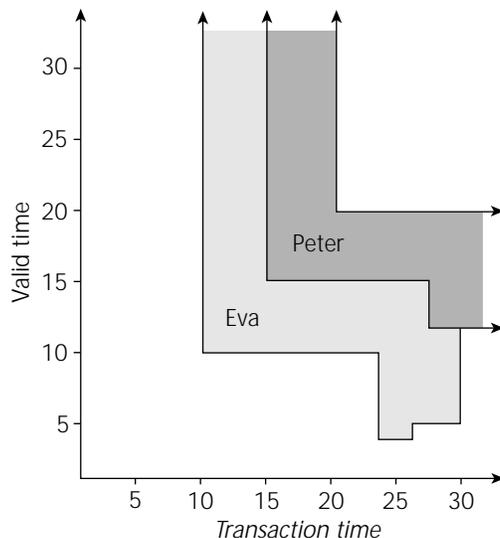
Figure 10.14 A nonsequenced deletion performed on January 30: Delete all records of exactly one-week duration.

As an example, consider the modification "Delete all records with a valid-time duration of exactly one week." This modification is clearly (valid-time) nonsequenced: (1) it depends heavily on the representation, looking for rows with a particular kind of valid timestamp, (2) it does not apply on a per instant basis, and (3) it mentions "records," that is, the recorded information, rather than "reality." The result of this deletion, evaluated on the 30th, is shown in Figure 10.14.

Nonsequenced modifications are initially complex to write, but require no subsequent transformations.

DELETEs are mapped in the second stage into an UPDATE on the transaction-stop time, so one statement suffices for nonsequenced deletions on bitemporal tables.

Code Fragment 10.18   Delete all records with a valid-time duration of exactly one week.

```
UPDATE Prop_Owner
SET TT_Stop = CURRENT_TIMESTAMP
WHERE (VT_End - VT_Begin DAY) = INTERVAL '7' DAY
  AND TT_Stop = DATE '9999-12-31'
```

The result is Table 10.7.

Now that we have a populated bitemporal table, we can discuss bitemporal queries.

Table 10.7 **After a nonsequenced deletion.**

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	1998-01-28
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	1998-01-28
145	7797	1998-01-03	1998-01-10	1998-01-23	1998-01-26
145	7797	1998-01-05	1998-01-10	1998-01-26	1998-01-28
145	7797	1998-01-05	1998-01-12	1998-01-28	*1998-01-30*
827	*7797*	*1998-01-12*	*1998-01-20*	*1998-01-28*	*9999-12-31*

10.3   QUERIES

We first consider a restricted class of queries, time-slice queries, then move on to the full spectrum of possible bitemporal queries.

10.3.1   Time-Slice Queries

A common query or view over the valid-time state table of Chapter 6 was to capture the state of the enterprise at some point in the past (or future). This query was termed a valid-time time-slice. For the tracking log of Chapter 8, we sought to reconstruct the state of the monitored table as of a date in the past; that query was termed a transaction time-slice. As a bitemporal table captures valid and transaction time, both time-slice variants are appropriate on such tables.

A transaction time-slice query corresponds to a vertical slice in the time diagram.

Time-slices are useful also in understanding the information content of a bitemporal table. A *transaction time-slice* of a bitemporal table takes as input a transaction-time instant and results in a *valid-time state table* that was present in the database at that specified time.

Code Fragment 10.19   **Give the history of owners of the flat at Skovvej 30 in Aalborg as of January 1, 1998.**

```
SELECT customer_number, VT_Begin, VT_End
FROM Prop_Owner
WHERE property_number = 7797
  AND TT_Start <= DATE '1998-01-01'
  AND DATE '1998-01-01' < TT_Stop
```

Applying this time-slice to Table 10.7, whose time diagram appears in Figure 10.14, results in an empty table, as no history was yet known about that property.

Taking a transaction time-slice as of January 14 results in a history with one entry:

customer_number	VT_Begin	VT_End
145	1998-01-10	9999-12-31

On January 14, we thought that Eva was the current owner of that property. We now know that Peter purchased the property on January 12, and that Eva never owned the property at all on January 14, but that is 20-20 hindsight. The information we had on January 14 indicated that Eva bought the property on the 10th, and still owns it.

The time-slice as of January 18 tells a different story:

customer_number	VT_Begin	VT_End
145	1998-01-10	1998-01-15
827	1998-01-15	9999-12-31

On January 18 we thought that Eva had purchased the flat on January 10 and sold it to Peter, who now owns it. A transaction time-slice can be visualized on the time diagram as a vertical line situated at the specified date. This line gives the valid-time history of the enterprise that was stored in the table on that date. Figure 10.15 illustrates this transaction time-slice.

Continuing, we take a transaction time-slice as of January 29:

customer_number	VT_Begin	VT_End
145	1998-01-05	1998-01-12
827	1998-01-12	1998-01-20

On January 29, we thought that Eva had purchased the flat on January 5 and sold it to Peter on January 12, who sold the property to someone else on January 20.

Finally, taking the current transaction time-slice,

Code Fragment 10.20   **Give the history of owners of the flat at Skovvej 30 in Aalborg as best known.**

```
SELECT customer_number, VT_Begin, VT_End
FROM Prop_Owner
WHERE property_number = 7797
  AND TT_Stop = DATE '9999-12-31'
```

yields the following result:

customer_number	VT_Begin	VT_End
827	1998-01-12	1998-01-20

Only Peter ever had ownership of the property, since all records with a valid-time duration of exactly one week were deleted. Peter's ownership was for all of eight days, January 12 to January 20.

A valid time-slice query corresponds to a horizontal slice in the time diagram, resulting in a transaction-time state table.

We can also cut the pie (or, more accurately, the time diagram) horizontally. A *valid time-slice* of a bitemporal table takes as input a valid-time instant and results in a *transaction-time state table* capturing when information concerning that specified valid time was recorded in the database. A valid time-slice is expressed in SQL similarly to the transaction time-slice.

Code Fragment 10.21   **When was information about the owners of the flat at Skovvej 30 in Aalborg on January 4, 1998, recorded in the Prop_Owner table?**

```
SELECT customer_number, TT_Start, TT_Stop
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin <= DATE '1998-01-04'
  AND DATE '1998-01-04' < VT_End
```
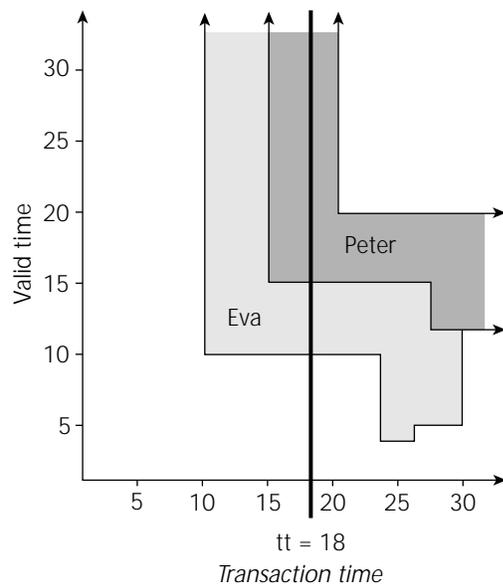

Figure 10.15 A transaction time-slice as of January 18.
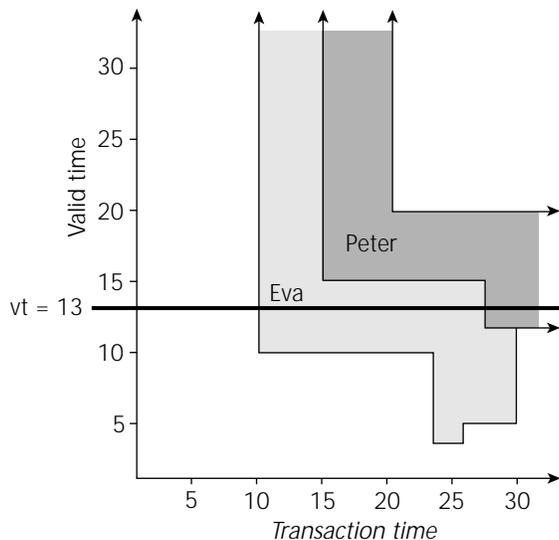
Figure 10.16 A valid time-slice on January 13.

Table 10.8 The valid time-slice on January 13.

customer_number	TT_Start	TT_Stop
145	1998-01-10	1998-01-15
145	1998-01-15	1998-01-28
827	1998-01-28	9999-12-31

Applying this time-slice to Table 10.7, whose time diagram appears in Figure 10.14, results in one row,

customer_number	TT_Start	TT_Stop
145	1998-01-23	1998-01-26

indicating that this information—that the property was owned by Eva on January 4—was inserted into the table on January 26 and subsequently deleted, as it was found to be incorrect, on January 26.

The valid time-slice on January 13 is more interesting. Such a time-slice can be visualized as the horizontal line shown in Figure 10.16. This time-slice results in Table 10.8.

While the horizontal line in Figure 10.16 intersects two regions, *three* rows result from the time-slice. This has to do with the way that the regions in the time diagram are sliced up into rectangles, each associated with a row in the Prop_Owner table.

Figure 10.17 The underlying rectangles encoding the bitemporal regions.

Examine the rectangles in the time diagram in Figure 10.17, which indicates the rectangle associated with each of the eight rows of Table 10.7. This figure makes it clear why the time-slice on January 13 returned three rows: rows 1, 2, and 8.

A *bitemporal time-slice* takes as input *two* instants, a valid-time and a transaction-time instant, and results in a *snapshot* state of the information regarding the enterprise at that valid time, as recorded in the database at that transaction time. This query, CF-10.22, is illustrated in Figure 10.18. The result is the facts located at the intersection of the two lines, in this case, Eva.

A bitemporal time-slice query extracts a single point from a time diagram, resulting in a snapshot table.

customer_number
145

Code Fragment 10.22 **Give the owner of the flat at Skovvej 30 in Aalborg on January 13 as stored in the** Prop_Owner **table on January 18.**

```
SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin <= DATE '1998-01-13'
  AND DATE '1998-01-13' < VT_End
  AND TT_Start <= DATE '1998-01-18'
  AND DATE '1998-01-18' < TT_Stop
```
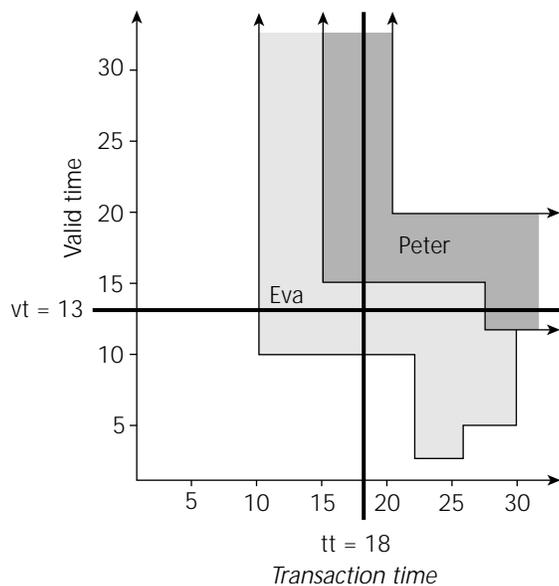
Figure 10.18 A bitemporal time-slice on a valid time of January 13 and as of a transaction time of January 18.

The current bitemporal time-slice uses "now" for both input instants.

Code Fragment 10.23   **Give the owner of the flat at Skovvej 30 in Aalborg today as best known.**

```
SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
  AND VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < VT_End
  AND TT_Stop = DATE '9999-12-31'
```

10.3.2   The Spectrum of Bitemporal Queries

Chapter 6 discussed the three major kinds of queries on valid-time state tables: current ("valid now"), sequenced ("history of"), and nonsequenced ("at some time"). Chapter 8 showed that there were three analogous kinds of queries on transaction-time state tables: current ("as best known"), sequenced ("when was it recorded"), and nonsequenced (e.g., "when was ... erroneously changed"). As a bitemporal table includes both valid-time and transaction-time support, and as these two types of time are orthogonal, it turns out that all nine combinations are possible on such tables.
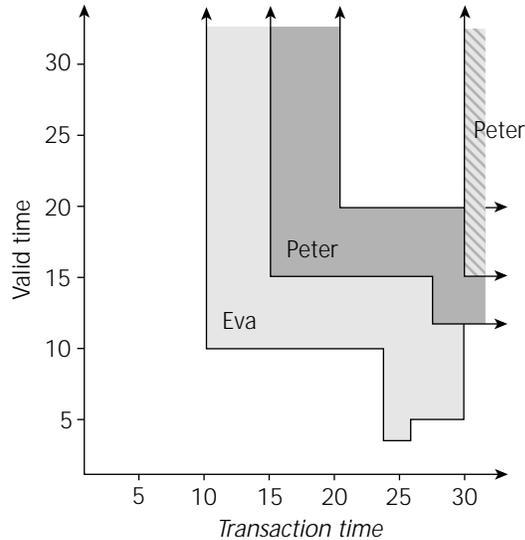
Figure 10.19 A sequenced insertion, performed on January 31, 1998: Peter bought another flat on January 15.

   To illustrate, we will take a nontemporal query and provide all the variations of that query. Before doing that, we add one more row to the `Prop_Owner` table.

Code Fragment 10.24   **Peter Olsen bought another flat, at Bygaden 4 in Aalborg on January 15, 1998; this was recorded on January 31, 1998.**

```
INSERT INTO Prop_Owner (customer_number, property_number,
    VT_Begin, VT_End, TT_Start, TT_Stop)
VALUES (827, 3621, DATE '1998-01-15', DATE '9999-12-31',
  CURRENT_TIMESTAMP, DATE '9999-12-31')
```

Overlaying this information on the time diagram, shown in Figure 10.19, we see that for five days Peter owned two properties, at Bygaden and Skovvej; he sold the Skovvej property on January 20, but retains the Bygaden property.

   We start with a nontemporal query, a simple equijoin, pretending that the `Prop_Owner` table is a snapshot table.

Code Fragment 10.25   **What properties are owned by the customer who owns property 7797?**

```
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
```

Table 10.9 The bitemporal state illustrated in Figure 10.19.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	1998-01-28
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	1998-01-28
145	7797	1998-01-03	1998-01-10	1998-01-23	1998-01-26
145	7797	1998-01-05	1998-01-10	1998-01-26	1998-01-28
145	7797	1998-01-05	1998-01-12	1998-01-28	1998-01-30
827	7797	1998-01-12	1998-01-20	1998-01-28	9999-12-31
827	3621	1998-01-15	9999-12-31	1998-01-31	9999-12-31

We now enumerate the nine kinds of bitemporal queries that are analogous to this nontemporal query, applying each on the state illustrated in Figure 10.19 and given in tabular form in Table 10.9.

Case 1   Valid-time current and transaction-time current

Code Fragment 10.26   What properties *are* owned by the customer who *owns* property 7797, *as best known*?

```
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P1.VT_End
  AND P1.TT_Stop = DATE '9999-12-31'
  AND P2.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P2.VT_End
  AND P2.TT_Stop = DATE '9999-12-31'
```

Current in valid time is implemented by requiring that the period of validity overlap "now"; current in transaction time is implemented by requiring a transaction-stop time of "until changed." The result, a snapshot table, is in this case the empty table because now, as best known, no one owns property 7797. (Peter owned it for some nine days in January, but doesn't own it now.)

Case 2   **Valid-time sequenced and transaction-time current**

Code Fragment 10.27   **What properties are *or were* owned by the customer who owned *at the same time* property 7797, as best known?**

```
SELECT P2.property_number,
  CASE WHEN DATE P1.VT_Begin < P2.VT_Begin
    THEN P2.VT_Begin ELSE P1.VT_Begin END AS VT_Begin,
  CASE WHEN DATE P1.VT_End < P2.VT_End
    THEN P1.VT_End ELSE P2.VT_End END AS VT_End,
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin < P2.VT_End AND P2.VT_Begin < P1.VT_End
  AND P1.TT_Stop = DATE '9999-12-31'
  AND P2.TT_Stop = DATE '9999-12-31'
```

Sequenced in valid time is implemented by selecting the overlap of the periods of validity, when the underlying rows were *both* valid (compare with CF-6.12 on page 152). The result, a valid-time state table, is the following:

property_number	VT_Begin	VT_End
3621	1998-01-15	1998-01-20

For those five days in January, Peter owned both properties.

Case 3   **Valid-time nonsequenced and transaction-time current**

Code Fragment 10.28   **What properties were owned by the customer who owned *at any time* property 7797, as best known?**

```
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.TT_Stop = DATE '9999-12-31'
  AND P2.TT_Stop = DATE '9999-12-31'
```

Nonsequenced in valid time is implemented by ignoring the valid timestamps. The result, a snapshot table, is the following:

property_number
3621

Peter owned both properties. While in this case there was a time when Peter owned both properties simultaneously, the query does not require that. Even if Peter had bought the second property on a valid time of January 31, that property would still be returned by this query.

Case 4   Valid-time current and transaction-time sequenced

Code Fragment 10.29   What properties *did we think* are owned by the customer who owns property 7797?

```
SELECT P2.property_number,
  CASE WHEN DATE P1.TT_Start < P2.TT_Start
    THEN P2.TT_Start ELSE P1.TT_Start END AS Recorded_Start,
  CASE WHEN DATE P1.TT_Stop < P2.TT_Stop
    THEN P1.TT_Stop ELSE P2.TT_Stop END AS Recorded_Stop
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P1.VT_End
  AND P2.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P2.VT_End
  AND P1.TT_Start < P2.TT_Stop AND P2.TT_Start < P1.TT_Stop
```

Sequenced in transaction time is implemented identically to sequenced in valid time: by selecting the overlap of the periods of presence, when the underlying rows were *both* present. As emphasized on page 260, the result of a transaction-time sequenced query is *not* a transaction-time state table. While the result does indicate what was recorded in the Prop_Owner table, it itself was not in existence until the query was performed. We thus use Recorded_Start and Recorded_Stop column names to highlight this distinction. The result, a snapshot table with two additional timestamp columns, is the empty table because there was no time in which we thought that Peter currently owns both properties.

Case 5   Valid-time sequenced and transaction-time sequenced

Code Fragment 10.30   When did we think that some property, *at some time*, was owned by the customer who owned *at the same time* property 7797?

```
SELECT P2.property_number,
  CASE WHEN DATE P1.VT_Begin < P2.VT_Begin
    THEN P2.VT_Begin ELSE P1.VT_Begin END AS VT_Begin,
  CASE WHEN DATE P1.VT_End < P2.VT_End
    THEN P1.VT_End ELSE P2.VT_End END AS VT_End,
```
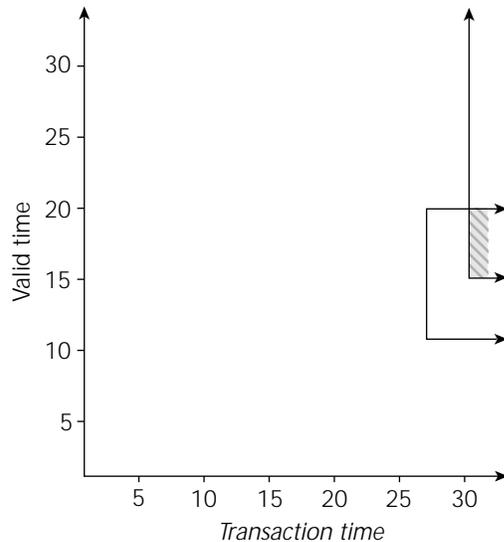
Figure 10.20 A query sequenced in both valid time and transaction time, computing the intersection of two rectangles.

```
  CASE WHEN DATE P1.TT_Start < P2.TT_Start
    THEN P2.TT_Start ELSE P1.TT_Start END AS Recorded_Start,
  CASE WHEN DATE P1.TT_Stop < P2.TT_Stop
    THEN P1.TT_Stop ELSE P2.TT_Stop END AS Recorded_Stop
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin < P2.VT_End AND P2.VT_Begin < P1.VT_End
  AND P1.TT_Start < P2.TT_Stop AND P2.TT_Start < P1.TT_Stop
```

Here we have sequenced in both valid time and transaction time. This is the most involved of all the queries, but the parallel between valid time and transaction time should be apparent in the above query. We must compute the overlap of the underlying rectangles, with the result being a valid-time state table with additional `Recorded` timestamp columns. Figure 10.20 shows the two rectangles that are involved (the last row of Table 10.7 and the row inserted by CF-10.24), and the overlap that is computed. One row results:

property_number	VT_Begin	VT_End	Recorded_Start	Recorded_Stop
3621	1998-01-15	1998-01-20	1998-01-31	9999-12-31

For those five days in January, Peter owned both properties. That information was recorded on January 31 and is still thought to be true (a transaction-stop time of "until changed").

Case 6   **Valid-time nonsequenced and transaction-time sequenced**

Code Fragment 10.31   **When did we think that some property, at some time, was owned by the customer who owned *at any time* property 7797?**

```
SELECT P2.property_number,
  CASE WHEN DATE P1.TT_Start < P2.TT_Start
    THEN P2.TT_Start ELSE P1.TT_Start END AS Recorded_Start,
  CASE WHEN DATE P1.TT_Stop < P2.TT_Stop
    THEN P1.TT_Stop ELSE P2.TT_Stop END AS Recorded_Stop
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.TT_Start < P2.TT_Stop AND P2.TT_Start < P1.TT_Stop
```

As before, nonsequenced in valid time is implemented by ignoring the valid time-stamps. The result, a snapshot table with additional timestamp columns, is the following:

property_number	Recorded_Start	Recorded_Stop
3621	1998-01-31	9999-12-31

From January 31 on, we thought that Peter had owned those two properties, perhaps not simultaneously.

Case 7   **Valid-time current and transaction-time nonsequenced**

Code Fragment 10.32   **When was it recorded that a property is owned by the customer who owns property 7797?**

```
SELECT P2.property_number, P2.TT_Start AS Recorded_Start
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P1.VT_End
  AND P2.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P2.VT_End
  AND P1.TT_Start <= P2.TT_Start AND P2.TT_Start < P1.TT_Stop
```

Nonsequenced in transaction time is implemented by not testing for full overlap in transaction time (sequenced) and by not testing the transaction-stop time for "until changed" (current). The result, a snapshot table, is empty because we never thought that Peter currently owns two properties.

Case 8   **Valid-time sequenced and transaction-time nonsequenced**

Code Fragment 10.33   **When was it recorded that a property is *or was* owned by the customer who owned *at the same time* property 7797?**

```
SELECT P2.property_number,
  CASE WHEN DATE P1.VT_Begin < P2.VT_Begin
    THEN P2.VT_Begin ELSE P1.VT_Begin END AS VT_Begin,
  CASE WHEN DATE P1.VT_End < P2.VT_End
    THEN P1.VT_End ELSE P2.VT_End END AS VT_End,
  P2.TT_Start AS Recorded_Start
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.VT_Begin < P2.VT_End AND P2.VT_Begin < P1.VT_End
  AND P1.TT_Start <= P2.TT_Start AND P2.TT_Start < P1.TT_Stop
```

This query is similar to valid-time sequenced/transaction-time current (CF-10.27), with a different predicate for transaction time. The result, a valid-time state table with an additional timestamp column, is the following:

property_number	VT_Begin	VT_End	Recorded_Start
3621	1998-01-15	1998-01-20	1998-01-31

For those five days in January, Peter owned both properties; this information was recorded on January 31.

Case 9   **Valid-time nonsequenced and transaction-time nonsequenced**

Code Fragment 10.34   **When was it recorded that a property was owned by the customer who owned *at some time* property 7797?**

```
SELECT P2.property_number, P2.TT_Start AS Recorded_Start
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
  AND P1.TT_Start <= P2.TT_Start AND P2.TT_Start < P1.TT_Stop
```

Note how short the FROM list and WHERE clause are. The result, a snapshot table with an additional timestamp column, is the following:

property_number	Recorded_Start
3621	1998-01-31

The two main points of this exercise are that all combinations *do* make sense, and all can be composed by considering valid time and transaction time separately.

- For current queries, just add a predicate to the WHERE clause restricting overlap with "now"; for transaction time, this is easiest done by requiring the transaction-stop time to be "until changed."
- For sequenced queries, the target list computes a new timestamp by taking the overlap of the two underlying timestamps (for the temporal join considered here); the WHERE clause must ensure that the overlap exists. Other kinds of sequenced queries require different approaches, as discussed in Section 6.3.
- For nonsequenced queries, nothing in the WHERE clause is needed. Depending on the query, the target list may or may not need to include one or both of the timestamps.

All combinations of current, sequenced, and nonsequenced over valid time and transaction time are possible and sensible.

Current/current queries are common and can be easily stated in SQL via currency predicates.

Current in valid time translates in English to "at now"; sequenced translates to "at the same time"; and nonsequenced translates to "at any time." Current in transaction time translates to "as best known"; sequenced translates to "when did we think"; and nonsequenced translates to "when was it recorded" or "when was it corrected."

Of these nine types of queries, a few are more prevalent. The most common is the current/current queries, "now, as best known." These queries correspond to queries on the nontemporal version of the table. (The following queries also utilize the `Customer` and `Property` tables, corresponding to the customer and property entities of the entity-relationship diagram in Figure 10.1; we assume that these two tables are also bitemporal.)

Code Fragment 10.35   What is the estimated value of the property at Bygaden 4?

```
SELECT estimated_value
FROM Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.VT_Begin <= CURRENT_DATE AND CURRENT_DATE < P.VT_End
  AND P.TT_Stop = DATE '9999-12-31'
```

Minutes, Seconds, and Jiffies

We've seen (page 22) that the night was partitioned into 12 hours corresponding to the 12 signs of the zodiac, and then the day was similarly ascribed to 12 hours, and hence an hour was eventually defined as 1/24 of a day. But why are there 60 minutes in an hour and 60 seconds in a minute?

King Alfonso X (the Wise) of Castille in the 13th century gathered together Arabic, Jewish, and Christian scholars to publish scientific works, including the *Alphonsine Tables*, which were arguably the most important astonomical charts of the late Middle Ages. These tables consistently use the sex-

agesimal division, in which hours (and days, and degrees of arcs) are divided into minutes, seconds, and "terciae." We can detect the etymology of "minute" from Latin *minutus*, or small, and "second" from Latin *secundus*. A sixtieth of a second should then be properly called a tercia, but instead Unix programmers have dubbed this unit a "jiffy," based on the fact that in the United States and Canada, computer clocks in the 1970s were incremented by 60-cycle power; in most other countries there are 50 jiffies to a second, due to their 50-cycle power.

Current/current queries return a snapshot result. The last two lines of the WHERE clause select the current state in both valid time and transaction time.

Code Fragment 10.36   **Who owns the property at Bygaden 4?**

```
SELECT name
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
  AND PO.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < PO.VT_End
  AND PO.TT_Stop = DATE '9999-12-31'
  AND C.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < C.VT_End
  AND C.TT_Stop = DATE '9999-12-31'
  AND P.VT_Begin <= CURRENT_DATE
  AND CURRENT_DATE < P.VT_End
  AND P.TT_Stop = DATE '9999-12-31'
```

Sequenced/current queries allow you to probe the history as best known.

Although this is a three-way join between bitemporal tables, the fact that it is a current/current query means that only the WHERE clause is affected.

Perhaps the next most common kind of query is a sequenced/current query, "history, as best known." These queries ignore transaction time and return a valid-time state table. Sequenced/current queries over one table are simple to specify.

Code Fragment 10.37   **How has the estimated value of the property at Bygaden 4 varied over time?**

```
SELECT estimated_value, VT_Begin, VT_End
FROM Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.TT_Stop = DATE '9999-12-31'
```

Sequenced joins require more work.

Code Fragment 10.38   **Who has owned the property at Bygaden 4?**

```
SELECT name, GREATEST(PO.VT_Begin, C.VT_Begin, P.VT_Begin),
  LEAST(PO.VT_End, C.VT_End, P.VT_End)
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
  AND GREATEST(PO.VT_Begin, C.VT_Begin, P.VT_Begin) <
    LEAST(PO.VT_End, C.VT_End, P.VT_End)
  AND PO.TT_Stop = DATE '9999-12-31'
  AND C.TT_Stop = DATE '9999-12-31'
  AND P.TT_Stop = DATE '9999-12-31'
```

Current/nonsequenced queries concern incorrectly stored information about now.

Here we use Oracle's GREATEST and LEAST functions to compute the intersection of the periods of validity of the three underlying rows. A (somewhat complex) CASE expression could be substituted for each.

Transaction time is supported in the `Prop_Owner` table to track the changes and to correct errors. A common query searches for the transaction that stored the current information in valid time. This is a current/nonsequenced query.

Code Fragment 10.39   **When was the estimated value for the property at Bygaden 4 stored?**

```
SELECT estimated_value, TT_Start AS Recorded_Start
FROM Property
WHERE address = 'Bygaden 4'
  AND VT_Begin <= CURRENT_DATE AND CURRENT_DATE < VT_End
```

This query will return a snapshot table giving one or more estimated values, along with the date of the transaction recording that value.

Sequenced/nonsequenced queries allow you to determine when invalid information about the history was recorded.

Code Fragment 10.40 **Who has owned the property at Bygaden 4, and when was this information recorded?**

```
SELECT name, GREATEST(PO.VT_Begin, C.VT_Begin, P.VT_Begin) AS VT_Begin,
  LEAST(PO.VT_End, C.VT_End, P.VT_End) AS VT_End,
  PO.TT_Start AS PO_Recorded, C.TT_Start AS C_Recorded,
  P.TT_Start AS P_Recorded_Start
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
  AND GREATEST(PO.VT_Begin, C.VT_Begin, P.VT_Begin) <
    LEAST(PO.VT_End, C.VT_End, P.VT_End)
  AND GREATEST(PO.TT_Start, C.TT_Start, P.TT_Start) <
    LEAST(PO.TT_Stop, C.TT_Stop, P.TT_Stop)
```

Nonsequenced/nonsequenced queries tease out the interaction between valid time and transaction time.

This returns a valid-time state table, with three additional columns stating when that information was recorded in the underlying tables. Subsequent queries could then isolate the identified problem.

Finally, nonsequenced/nonsequenced queries can probe the interaction between valid time and transaction time, identifying, for example, retroactive changes (where the change concerned the past) and postactive changes (where the change concerned the future).

Code Fragment 10.41 **List all retroactive changes made to the** Prop_Owner **table.**

```
SELECT customer_number, property_number, VT_Begin, VT_End,
  TT_Start AS Recorded_Start
FROM Prop_Owner
WHERE VT_Begin < TT_Start
```

This returns the valid-time state table in Table 10.10, indicating that many of the modifications were retroactive.

10.4 INTEGRITY CONSTRAINTS

An integrity constraint can be implemented by first writing a SELECT statement, then embedding it in a CHECK constraint.

There is a strong connection between queries and integrity constraints (here, we are considering general integrity constraints, not just the particular ones, such as uniqueness and key constraints, accorded specific language constructs in SQL). An integrity constraint of the form "it must be the case that …" can be transformed into a query of the form "select those rows such that … is false"; this query can then be inserted into a check constraint of the form CHECK NOT EXISTS.

Implementing an integrity constraint

State the constraint as a SELECT that returns offending rows.
Create a CHECK constraint that requires that those rows not exist.

As a (nontemporal) example, consider the integrity constraint "a customer who owns property 7797 shall own no other property." (Admittedly, this is a rather unusual integrity constraint, but it will be clear momentarily why we use this particular constraint.) We first transform this into a query.

Code Fragment 10.42   **Select those customers who own property 7797 and another property.**

```
SELECT P2.customer_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.customer_number = P2.customer_number
```

We then insert this into a NOT EXISTS.

Code Fragment 10.43   **A customer who owns property 7797 shall own no other property.**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
        SELECT P2.customer_number
        FROM Prop_Owner AS P1, Prop_Owner AS P2
        WHERE P1.property_number = 7797
          AND P2.property_number <> P1.property_number
          AND P1.customer_number = P2.customer_number))
```

Table 10.10 All retroactive changes made to the Prop_Owner **table.**

customer_number	property_number	VT_Begin	VT_End	Recorded_Start
145	7797	1998-01-10	1998-01-15	1998-01-15
827	7797	1998-01-15	1998-01-20	1998-01-20
145	7797	1998-01-03	1998-01-10	1998-01-23
145	7797	1998-01-05	1998-01-10	1998-01-26
145	7797	1998-01-05	1998-01-12	1998-01-28
827	7797	1998-01-12	1998-01-20	1998-01-28
827	3621	1998-01-15	9999-12-31	1998-01-31

What does this have to do with bitemporal tables? Well, just as there are many kinds of bitemporal queries (nine, to be exact), there are many kinds of bitemporal integrity constraints. We give three to illustrate the correspondence. The first is a current/current constraint.

Code Fragment 10.44   **A customer who *owns* property 7797 shall *own* no other property.**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
            SELECT P2.property_number
            FROM Prop_Owner AS P1, Prop_Owner AS P2
            WHERE P1.property_number = 7797
              AND P2.property_number <> P1.property_number
              AND P1.customer_number = P2.customer_number
              AND P1.VT_Begin <= CURRENT_DATE
              AND CURRENT_DATE < P1.VT_End
              AND P1.TT_Stop = DATE '9999-12-31'
              AND P2.VT_Begin <= CURRENT_DATE
              AND CURRENT_DATE < P2.VT_End
              AND P2.TT_Stop = DATE '9999-12-31' ))
```

You may have noticed that all but the first and last lines were copied directly from CF-10.26.

The above constraint doesn't accommodate the past. Often we wish the constraint to hold over all valid time. This can be accomplished with a sequenced/current constraint.

Code Fragment 10.45   **A customer who owned property 7797 shall *concurrently own* no other property.**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
            SELECT *
            FROM Prop_Owner AS P1, Prop_Owner AS P2
            WHERE P1.property_number = 7797
              AND P2.property_number <> P1.property_number
              AND P1.customer_number = P2.customer_number
              AND P1.VT_Begin < P2.VT_End
              AND P2.VT_Begin < P1.VT_End
              AND P1.TT_Stop = DATE '9999-12-31'
              AND P2.TT_Stop = DATE '9999-12-31' ))
```

This is just CF-10.27, with the target list replaced with *, as the NOT EXISTS could care less whether there is one column or many in the row returned by the subquery.

To check for ownership of another property, at some possibly different time, we use a nonsequenced/current constraint.

Code Fragment 10.46   **A customer who owned property 7797 shall own no other prop-
erty,** *even at a different time.*

```
CREATE ASSERTION CHECK ( NOT EXISTS (
          SELECT P2.property_number
          FROM Prop_Owner AS P1, Prop_Owner AS P2
          WHERE P1.property_number = 7797
            AND P2.property_number <> P1.property_number
            AND P1.customer_number = P2.customer_number
            AND P1.TT_Stop = DATE '9999-12-31'
            AND P2.TT_Stop = DATE '9999-12-31'))
```

This is just CF-10.28, placed in a NOT EXISTS.

Integrity constraints are generally applied after modification statements, or at the end of transactions containing modification statements. As such, current in transaction time, as in the above constraints, is appropriate. The rows modified by the transaction will all have a transaction-stop time of "until changed." There are nine variants corresponding to each nontemporal integrity constraint, paralleling the situation with queries. However, transaction-time sequenced integrity constraints should be specified as current in transaction time, reducing the number of usable bitemporal integrity constraints to six: current/current, sequenced/current, nonsequenced/current, current/nonsequenced, sequenced/nonsequenced, and nonsequenced/nonsequenced.

> There are six variants corresponding to each nontemporal integrity constraint.

We now turn to a particularly important constraint, that of referential integrity, and focus on `Prop_Owner.customer_number` as a sequenced/current foreign key to the `Customer` table. The brute-force approach, which always works, is to use the approach above to get a nontemporal constraint, then use the process discussed in Section 10.3.2 to obtain the appropriate bitemporal constraint.

For referential integrity, we first construct the following (nontemporal) constraint:

Code Fragment 10.47   **The customer number in** `Prop_Owner` **is a foreign key referencing
the** `Customer` **table (nontemporal version).**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
          SELECT *
          FROM Prop_Owner
          WHERE customer_number NOT IN (SELECT customer_number
                      FROM Customer))
          )
```

We now need to map this into a sequenced/current constraint, which is done by mapping the SELECT into a sequenced/current query. As noted on page 155, the NOT IN is actually relational difference and can be expressed using that construct.

Code Fragment 10.48   **The customer number in** Prop_Owner **is a foreign key referencing the** Customer **table, using EXCEPT (nontemporal version).**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
            SELECT customer_number
            FROM Prop_Owner
            EXCEPT
            SELECT customer_number
            FROM Customer)
    )
```

A sequenced/current foreign key constraint can be expressed as an embedded sequenced/current query.

We can then map this into a sequenced/current query using the approach illustrated in CF-6.18 on page 155. To do so, we made the following changes to that code fragment:

- Replaced INCUMBENTS with Prop_Owner for I1 and with Customer for the other correlation names, I2, I3, and I4.
- Removed the mentions of specific PCNs.
- Replaced SSN with customer_number.
- Replaced each of the target lists with *.
- Added the transaction currency check to the WHERE clauses for all correlation names.

Code Fragment 10.49   **The customer number in** Prop_Owner **is a foreign key referencing the** Customer **table (valid-time sequenced/transaction-time current version).**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
      SELECT I1.customer_number
      FROM Prop_Owner AS I1, Customer AS I3
      WHERE I1.customer_number = I3.customer_number
        AND I1.TT_Stop = DATE '9999-12-31'
        AND I3.TT_Stop = DATE '9999-12-31'
        AND NOT EXISTS (SELECT *
            FROM Customer AS I4
            WHERE I4.customer_number = I1.customer_number
              AND I4.TT_Stop = DATE '9999-12-31'
              AND I1.VT_Begin < I4.VT_End
              AND I4.VT_Begin < I3.VT_Begin)
      UNION
      SELECT I1.customer_number
      FROM Prop_Owner AS I1, Customer AS I2
      WHERE I1.customer_number = I2.customer_number
```

```
              AND I1.TT_Stop = DATE '9999-12-31'
              AND I2.TT_Stop = DATE '9999-12-31'
              AND NOT EXISTS (SELECT *
                  FROM Customer AS I4
                  WHERE I4.customer_number = I1.customer_number
                    AND I4.TT_Stop = DATE '9999-12-31'
                    AND I2.VT_End < I4.VT_End
                    AND I4.VT_Begin < I1.VT_End)
          UNION
          SELECT I1.customer_number
          FROM Prop_Owner AS I1, Customer AS I2, Customer AS I3
          WHERE I1.customer_number = I2.customer_number
            AND I1.customer_number = I3.customer_number
            AND I1.TT_Stop = DATE '9999-12-31'
            AND I2.TT_Stop = DATE '9999-12-31'
            AND I3.TT_Stop = DATE '9999-12-31'
            AND NOT EXISTS (SELECT *
                  FROM Customer AS I4
                  WHERE I4.customer_number = I1.customer_number
                    AND I4.TT_Stop = DATE '9999-12-31'
                    AND I2.VT_End < I4.VT_End
                    AND I4.VT_Begin < I3.VT_Begin)
          UNION
          SELECT I1.customer_number
          FROM Prop_Owner AS I1
          WHERE I1.TT_Stop = DATE '9999-12-31'
            AND NOT EXISTS (SELECT *
                  FROM Prop_Owner AS I4
                  WHERE I4.customer_number = I1.customer_number
                    AND I4.TT_Stop = DATE '9999-12-31'
                    AND I1.VT_Begin < I4.VT_End
                    AND I4.VT_Begin < I1.VT_End))
      )
```

By studying the particulars of the desired temporal integrity constraint, often a much simpler expression is possible.

Another way to proceed is to look more closely at what the sequenced/current referential integrity is doing and to attempt to come up with a more streamlined version. We already did that analysis for valid-time state tables, developing CF-5.23. This can be generalized to a sequenced/current constraint on bitemporal tables by simply adding a transaction currency predicate for each correlation name.

Code Fragment 10.50   **The customer number in** Prop_Owner **is a foreign key referencing the** Customer **table (valid-time sequenced/transaction-time current, version 2).**

```
CREATE ASSERTION CHECK (NOT EXISTS (
      SELECT *
      FROM Prop_Owner AS P
      WHERE P.TT_Stop = DATE '9999-12-31'
        AND NOT EXISTS (
            SELECT *
            FROM Customer AS C
            WHERE P.PCN = C.PCN
              AND C.TT_Stop = DATE '9999-12-31'
              AND C.VT_Begin <= P.VT_Begin
              AND P.VT_Begin < C.VT_End)
         OR NOT EXISTS (
            SELECT *
            FROM Customer AS C
            WHERE P.PCN = C.PCN
              AND C.TT_Stop = DATE '9999-12-31'
              AND C.VT_Begin < P.VT_End
              AND P.VT_End <= C.VT_End)))
```

This approach yields a much shorter assertion, but requires more analysis; the first approach is largely mechanical.

> For referential integrity constraints between tables supporting differing aspects of time, use a current constraint if the time support is missing.

While the temporal analog of a nontemporal foreign key integrity constraint is normally a sequenced/sequenced constraint, there are exceptions. First, as already noted, transaction-time sequenced can be replaced with transaction-time current. Second, for valid time, if the referencing table does not support valid time, then we should only use the current state of the referenced table, a valid-time current constraint.

10.5   TEMPORAL PARTITIONING*

While a bitemporal table with valid and transaction period timestamps is convenient for many types of queries, as illustrated in Section 10.3, keeping both the valid-time history and the history of the changes in a single table often greatly decreases query performance. To perform a current/current query ("at now as best known") may require scanning the entire bitemporal table, rejecting the records relating to prior dates in valid time and the erroneous records that were subsequently corrected.

Table 10.11 The bitemporal table corresponding to the time diagram of Figure 10.19.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	1998-01-28
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	1998-01-28
145	7797	1998-01-03	1998-01-10	1998-01-23	1998-01-26
145	7797	1998-01-05	1998-01-10	1998-01-26	1998-01-28
145	7797	1998-01-05	1998-01-12	1998-01-28	1998-01-30
827	7797	1998-01-12	1998-01-20	1998-01-28	9999-12-31
827	3621	1998-01-15	9999-12-31	1998-01-31	9999-12-31

Temporal partitioning was previously considered in the context of valid time in Section 7.5 and in the context of transaction time in Section 9.4. The alternatives, and their potential impact on performance, are even greater for bitemporal tables. We will briefly examine a collection of temporal partitioning schemes for bitemporal tables here, continuing with the Prop_Owner table. This table will be represented by two or more tables, each holding a subset of the bitemporal regions. While the general approach is termed "temporal partitioning," often the constituent tables do not actually partition the data; information may be replicated in multiple tables, with the application held responsible for ensuring that the information is consistent.

We illustrate the partitioning on the state of the Prop_Owner table (Table 10.11) corresponding to the time diagram of Figure 10.19 on page 313.

10.5.1   VT Current/TT Current + Bitemporal State

If current/current queries are prevalent, it is advantageous to materialize the current/current state (valid time = transaction time = "now"), in addition to the bitemporal state, resulting in two tables: the current store (PO_Current) and the bitemporal store (PO_Bitemp). The current store in the example will contain but a single row, indicating that Peter owns the flat at Bygaden 4, as best known:

customer_number	property_number
827	3621

As an aside, this arrangement is not a strict partitioning, as the rows in the current store are also in the bitemporal state table. The other variants discussed below are true partitions.

Current/current queries are easy: just apply them to the current store; no additional predicates are required. Of course, all the other kinds of queries can be applied to the bitemporal state table, which is still available.

A current partition has the advantage that a valid-end time of "forever" need not be stored; the valid-end timestamp is implicit in the current store.

The current store must be changed as a side effect of modifications applied to the bitemporal store. Current modifications are simple: transform the modification as discussed in Section 10.2.1 to modify the bitemporal store, and also apply the modification as is to the current store. Sequenced modifications would be also applied to the current store if the period of applicability overlapped "now." Nonsequenced modifications must be handled on a case-by-case basis.

The current store has the disadvantage that the passage of time alone can cause rows to enter and exit this store; managing this movement is awkward.

Interestingly, the current store can change just with the passage of time. Consider the modification that inserts Peter buying the Bygaden flat on January 15, illustrated in Figure 10.19 on page 313. Say that this was instead a postactive modification—that it was performed on January 5. This row was not present in the current store on that date. Indeed, it remains absent until January 15 rolls around, at which point it must be inserted into the current store.

One way to handle these future events is to execute the following action to reestablish consistency once a day, say, in the early morning:

Code Fragment 10.51 **Bring the** PO_Current **table up-to-date.**

```
INSERT INTO PO_Current
SELECT customer_number, property_number
FROM PO_Bitemp
WHERE TT_Stop = DATE '9999-12-31'
  AND VT_Begin = CURRENT_DATE

DELETE FROM PO_Current
WHERE EXISTS ( SELECT *
               FROM PO_Bitemp AS B
               WHERE PO_Current.customer_number = B.customer_number
                 AND PO_Current.property_number = B.property_number
                 AND TT_Stop = DATE '9999-12-31'
                 AND VT_End = CURRENT_DATE)
```

In the deletion, we need to delete those rows of the current store associated with rows in the bitemporal store with the same primary key.

If the granularity of valid time was smaller, say, a minute, then it might make sense to record the valid begin and end times in a separate table, PO_Future, with a single column, When. Each modification to the bitemporal store would insert times into PO_Future and would also set some variable indicating when the action should be evaluated. This action would remove the earliest time from the table and would reschedule itself for the next begin or end date. Alternatively, the above action could be run whenever a query was applied to the current store, to ensure that this state was up-to-date.

Current/current queries are particularly easy and efficient: simply query the current store. Other queries can be applied to the bitemporal state table, which has all the information it did before.

10.5.2   VT State/TT Current + State/State Archive

To avoid the complexity of having to modify the current store as a side effect of the passage of time, we can put the entire valid-time history, as best known, in what is termed a *history store* (PO_History), with the corrected rows—those with a transaction-stop time other than "until changed"—residing in an archival store (PO_Archive). The history store now has more information than the current store of the previous section. Note that the history store here has three timestamp columns: the delimiting times of the valid-time period, and the transaction-start time. The transaction-stop time is implicit; it is "until changed." Making this implicit means that we don't have to come up with an encoding of "until changed" (that value was encoded as "forever" in the bitemporal state table). The history store contains all the rectangles with right pointers in Figure 10.19:

customer_number	property_number	VT_Begin	VT_End	TT_Start
827	7797	1998-01-12	1998-01-20	1998-01-28
827	3621	1998-01-15	9999-12-31	1998-01-31

Correspondingly, the archival store (Table 10.12) has fewer rows than the original bitemporal state table, as the current history is stored elsewhere.

A history store is much easier to maintain than a current store and retains many of its advantages. The archival store can be maintained automatically through triggers defined on the history store.

While not illustrated here, it is possible for future valid times to also be present in the history store, and in the archival store as well. We don't have to be concerned with moving such rows in or moving rows that terminate in the future out, as we did with the current store.

Modifications—whether current, sequenced, or nonsequenced in valid time (recall that all modifications are current in transaction time)—are applied directly to the history store. The archival store can be maintained entirely with triggers,

Table 10.12 Archival store.

customer_ number	property_ number	VT_Begin	VT_End	TT_Start	TT_Stop
145	7797	1998-01-10	9999-12-31	1998-01-10	1998-01-15
145	7797	1998-01-10	1998-01-15	1998-01-15	1998-01-28
827	7797	1998-01-15	9999-12-31	1998-01-15	1998-01-20
827	7797	1998-01-15	1998-01-20	1998-01-20	1998-01-28
145	7797	1998-01-03	1998-01-10	1998-01-23	1998-01-26
145	7797	1998-01-05	1998-01-10	1998-01-26	1998-01-28
145	7797	1998-01-05	1998-01-12	1998-01-28	1998-01-30

generalizing the approach used in Chapter 8 for maintaining a tracking log, specifically CF-8.2 on page 221. These triggers retain a before-image in the archival store.

Code Fragment 10.52   **Triggers for maintaining the** PO_Archive **table.**

```
CREATE TRIGGER Delete_PO
AFTER DELETE ON PO_History FOR EACH ROW
BEGIN
  INSERT INTO PO_Archive VALUES (OLD.customer_number,
      OLD.property_number, OLD.VT_Begin, OLD.VT_End,
      OLD.TT_Start, CURRENT_DATE)
END

CREATE TRIGGER Update_PO
AFTER UPDATE ON PO_History FOR EACH ROW
BEGIN
  INSERT INTO PO_Archive VALUES (OLD.customer_number,
      OLD.property_number, OLD.VT_Begin, OLD.VT_End,
      OLD.TT_Start, CURRENT_DATE)
END
```

In Section 10.2, we outlined a two-stage approach to implement modifications on a bitemporal table: first deal with valid time, then further transform the sequence of SQL statements to deal with transaction time. Under the history/archival store scheme, only the first stage is required; the triggers effect the second stage, thus greatly simplifying the application code. (There is one remnant of the second stage: the transaction-start time must be set/updated to "now" in the application code, in modifications applied to the history store.)

The full bitemporal table can be expressed as a view.

Table 10.13 Another version of the archival store.

customer_number	property_number	VT_Begin	VT_End	TT_Start
145	7797	1998-01-10	9999-12-31	1998-01-10
145	7797	1998-01-10	1998-01-15	1998-01-15
827	7797	1998-01-15	9999-12-31	1998-01-15
827	7797	1998-01-15	1998-01-20	1998-01-20
145	7797	1998-01-03	1998-01-10	1998-01-23
145	7797	1998-01-05	1998-01-10	1998-01-26
145	7797	1998-01-05	1998-01-12	1998-01-28

Transaction-time current queries are applied to the history store. Transaction-time sequenced and nonsequenced queries should be applied to the following view:

Code Fragment 10.53 **Reinstate the bitemporal state table as a view.**

```
CREATE VIEW Prop_Owner (customer_number, property_number, VT_Begin,
   VT_End, TT_Start, TT_Stop) AS
(SELECT customer_number, property_number, VT_Begin,
   VT_End, TT_Start, CURRENT_DATE
FROM PO_History
UNION
SELECT *
FROM PO_Archive)
```

10.5.3 VT State/TT Current + State/Event Archive

This organization differs from that of the previous section in that we use instant timestamping for transaction time, with the accompanying space savings. (In fact, this is the scheme that Nykredit utilizes for its property ownership table. The customer and property entities are maintained in transaction-time tables, each temporally partitioned into a current store, with the transaction-start timestamp—the transaction-stop time is implicitly "until changed"—and an archival store, instant-stamped also with the transaction-start date.) Here the history and archival schemes are identical. The stores themselves will have the same number of rows as the scheme of the previous section; in fact, the history store is identical in the two schemes. The archival store (Table 10.13) omits the transaction-stop time.

The reason that the transaction-stop time is not needed is that it can be found in another row, as the transaction-start time. The transaction-stop time of the first row is found in the second row: January 15. This is because the (sequenced/sequenced) primary key, customer_number and property_number, match, and because the valid times overlap, implying that the second row invalidated the valid-time period of the first row. Following owner 145 (Eva), the period of validity of the fifth row does

not overlap that of the second row, so it can be viewed as additional information. However, both the second row and the fifth row are invalidated by the sixth row. Finally, the fourth row invalidates, and provides the transaction-stop time for, the third row.

As an aside, the ZPOS_COMPENSATION_HISTORY table of Chapter 5 includes a CHRONOLOGY_KEY column, serving as a transaction timestamp, along with HIS-TORY_START_DATE and HISTORY_END_DATE columns, indicating when the information in the record applied (the period of validity). This table can also be considered to be a state/event bitemporal table, supporting both valid time (with period timestamping) and transaction time (with instant timestamping).

The archival store can once again be maintained via triggers defined on the history store. This ensures that the transaction-time semantics is maintained.

As before, to evaluate transaction sequenced or nonsequenced queries, we need to constitute the bitemporal state table, which is rather more involved than before.

Code Fragment 10.54   **Reconstitute the bitemporal state table as a view.**

```
CREATE VIEW Prop_Owner (customer_number, property_number, VT_Begin,
  VT_End, TT_Start, TT_Stop) AS
(SELECT customer_number, property_number, VT_Begin,
  VT_End, TT_Start, CURRENT_DATE
FROM PO_History
UNION
SELECT P1.customer_number, P1.property_number, P1.VT_Begin, P1.VT_End,
  P1.TT_Start, P2.TT_Start
FROM PO_Archive AS P1, PO_Archive AS P2
WHERE P1.customer_number = P2.customer_number
  AND P1.property_number = P2.property_number
  AND P1.TT_Start < P2.TT_Start
  AND NOT EXISTS (SELECT *
          FROM PO_Archive AS P3
          WHERE P1.customer_number = P3.customer_number
            AND P1.property_number = P3.property_number
            AND P3.TT_Start BETWEEN P1.TT_Start AND P2.TT_Start)
UNION
SELECT PA.customer_number, PA.property_number, PA.VT_Begin, PA.VT_End,
  PA.TT_Start, PH.TT_Start
FROM PO_Archive AS PA, PO_History AS PH
WHERE PA.customer_number = PH.customer_number
  AND PA.property_number = PH.property_number
  AND PA.TT_Start < PH.TT_Start
```

```
AND NOT EXISTS (SELECT *
        FROM PO_Archive AS P3
       WHERE PA.customer_number = P3.customer_number
         AND PA.property_number = P3.property_number
         AND PA.TT_Start < P3.TT_Start)
)
```

The archival store can be narrowed by storing only one transaction timestamp: transaction start. The drawback is a more complex view reconstituting the bitemporal state table.

For those rows that were invalidated, we need to locate the row that invalidated it (from either the PO_History or the PO_Archive table) to locate the transaction-stop time. Those that haven't been invalidated (i.e., those in the current store) have a transaction-stop time of "now."

There is one complication with the structure. If the entire history of an owner-property pair can be deleted—say, with a valid-time sequenced deletion with a period of applicability of all of time—there is no place to store the transaction time of that deletion. If this is indeed possible, it is best to create a PO_Deleted table, with columns owner_number, property_number, and TT_Deleted. The view definition would then require another case.

Code Fragment 10.55   **Reinstate the bitemporal state table as a view, when history deletions are allowed.**

```
CREATE VIEW Prop_Owner (customer_number, property_number, VT_Begin,
   VT_End, TT_Start, TT_Stop) AS
SELECT customer_number, property_number, VT_Begin, VT_End,
  TT_Start, CURRENT_DATE
FROM PO_History
UNION
SELECT P1.customer_number, P1.property_number, P1.VT_Begin, P1.VT_End,
  P1.TT_Start, P2.TT_Start
FROM PO_Archive AS P1, PO_Archive AS P2
WHERE P1.customer_number = P2.customer_number
  AND P1.property_number = P2.property_number
  AND P1.TT_Start < P2.TT_Start
  AND NOT EXISTS (SELECT *
        FROM PO_Archive AS P3
       WHERE P1.customer_number = P3.customer_number
         AND P1.property_number = P3.property_number
         AND P3.TT_Start BETWEEN P1.TT_Start AND P2.TT_Start)
  AND NOT EXISTS (SELECT *
        FROM PO_Deleted AS P3
       WHERE P1.customer_number = P3.customer_number
```

```
                AND P1.property_number = P3.property_number
                AND P3.TT_Start BETWEEN P1.TT_Start AND P2.TT_Start)
UNION
SELECT PA.customer_number, PA.property_number, PA.VT_Begin, PA.VT_End,
  PA.TT_Start, PH.TT_Start
FROM PO_Archive AS PA, PO_History AS PH
WHERE PA.customer_number = PH.customer_number
  AND PA.property_number = PH.property_number
  AND PA.TT_Start < PH.TT_Start
  AND NOT EXISTS (SELECT *
        FROM PO_Archive AS P3
        WHERE PA.customer_number = P3.customer_number
          AND PA.property_number = P3.property_number
          AND PA.TT_Start < P3.TT_Start)
  AND NOT EXISTS (SELECT *
        FROM PO_Deleted AS P3
        WHERE P1.customer_number = P3.customer_number
          AND P1.property_number = P3.property_number
          AND P3.TT_Start BETWEEN P1.TT_Start AND P2.TT_Start)
UNION
SELECT PA.customer_number, PA.property_number, PA.VT_Begin, PA.VT_End,
  PA.TT_Start, PH.TT_Start
FROM PO_Archive AS PA, PO_Deleted AS PD
WHERE PA.customer_number = PD.customer_number
  AND PA.property_number = PD.property_number
  AND NOT EXISTS (SELECT *
        FROM PO_Archive AS P3
        WHERE PA.customer_number = P3.customer_number
          AND PA.property_number = P3.property_number
          AND P3.TT_Start BETWEEN PA.TT_Start AND PD.TT_Start)
```

The additional NOT EXISTS in the second and third cases are required if an owner-property pair's history is removed, then later inserted back.

10.6   VACUUMING*

Vacuuming was first introduced in Chapter 9, in the context of transaction-time state tables. Bitemporal tables present an even greater opportunity, and need, for vacuuming. Recording the history of the enterprise can result in many rows; also retaining the change history of the table can dramatically increase the size of the table.

The relative sizes of the components of a bitemporal table can be surprising. In our example, Table 10.9, corresponding to the time diagram in Figure 10.19,

captures three relationships, two involving Peter and one involving Eva. The current/current state ("now as best known") comprises just one row, the last one. The current transaction-time state ("history as best known") comprises two rows, the last plus the next-to-last, both corresponding to Peter. The noncurrent transaction-time states, those with a transaction-stop time other than "forever," comprise the other seven rows; these are usually placed in an archival store. This is in accord with our intuition: the current/current state should be the smallest; the transaction-time current should be larger, as it contains the current/current state; and the archival store should be the largest.

However, often these relative sizes are not found in actual applications. In particular, it is often the case that the archival store is not large at all. The rows in the archival store originate from two separate processes. The first is when a row is inserted with an unknown valid-time end time; in such cases we simply use "forever." Later, the end time becomes known and is stored in the bitemporal table, with the previous row moved to the archival store. Two of the rows of Table 10.9, the first and third rows, thus came into being.

The second situation is when erroneous data was discovered and corrected; the data in error is retained in the archival store. This was the source of five of the rows in the example table.

Often the history store is the largest component of a bitemporal state table, implying that vacuuming the archival store may not be effective in substantially reducing the size of such a table.

The first source of archival rows is not invoked when the period of validity is known a priori. Insurance policies, car loans, and fixed-term appointments are examples in which archival rows need not be generated. And the second source of archival rows is thankfully not that common; a good percentage of data is in fact correct and will not later be changed. (Alternatively, the data may be incorrect, but we may never discover that.) For these reasons, in many applications the current transaction state ("history as best known") dominates the bitemporal table, with the archival store being a fraction of the rows.

We can apply all of the techniques discussed in Section 9.5 to the archival portion of a bitemporal state table. Because valid time is present in such tables, it can be used to further refine the vacuuming.

Code Fragment 10.56   **Temporally vacuum old unused entities from the archival store for which no recent history exists.**

```
DELETE FROM Prop_Owner
WHERE (CURRENT_DATE - TT_Stop DAY) > INTERVAL '731' DAY
  AND NOT EXISTS (SELECT *
      FROM Prop_Owner AS R
      WHERE TT_Stop = DATE '9999-12-31'
        AND (CURRENT_DATE - VT_End DAY) > INTERVAL '731' DAY
        AND customer_number = C.customer_number
        AND property_number = C.property_number)
```

The first predicate identifies archival rows that are old in transaction time (the correction was made more than two years ago); the second predicate ensures that no recent history (less than two years) exists in valid time.

10.7 IMPLEMENTATION CONSIDERATIONS

The CD-ROM contains all the code fragments in this chapter in Oracle8 Server, which run without change after transforming assertions to triggers.

10.8 SUMMARY

A bitemporal table combines both valid time and transaction time into a single structure. It contains four timestamps, two denoting the valid-time period of validity and two denoting the transaction-time period of presence. Such a table can also be represented by several tables, such as a current store, a history store, and an archival store.

Primary key constraints on such tables are generally valid-time sequenced and transaction-time current. Expressing such a constraint requires an SQL PRIMARY KEY constraint and an assertion. Often you will also want to constrain the valid-time history for an entity (or relationship) to not have any gaps; this requires another assertion.

Modifications on bitemporal tables can be challenging. We examined several variants, all current in transaction time: valid-time current insertions, deletions, and updates; valid-time sequenced insertions, deletions, and updates; and valid-time nonsequenced deletions. We saw that valid-time current modifications were tedious, and valid-time sequenced modifications even more so. For the latter, we advocated a two-stage conversion process: first deal with valid time, with each such modification resulting in a series of SQL statements, then deal with transaction time, mapping each SQL statement into one or more statements that maintain the transaction-time semantics. The mapping was mechanical, but resulted in a long series of SQL statements. The worst case was that of sequenced update, in which a nontemporal update of only a few lines expanded to some 60 lines of SQL.

We first considered time-slice queries, in transaction, valid, and bitemporal varieties, then showed that there are nine versions of any nontemporal query. Some of these are more useful than others, but the benefit of a bitemporal table is that it admits the full generality of temporal queries.

Temporal integrity constraints parallel temporal queries; in fact, any temporal integrity constraint can be expressed as a temporal query embedded in a NOT

EXISTS assertion. This mechanical translation allows you to map a temporal constraint into SQL; further analysis may enable the assertion to be simplified.

We examined several ways to represent a bitemporal state table via several tables, each holding a portion of the temporal extent of the table. There are two somewhat orthogonal decisions to be made: how to divide up the time diagram among the representational tables, and for each table, whether to use instant or period timestamping.

Partitioning a bitemporal table renders some queries more efficient, at the expense of other queries. Transaction-time sequenced and nonsequenced queries are negatively affected; transaction-time current queries, especially the valid-time current subset, are made faster. If the performance of current/current queries is critical, having a separate materialized current store can be quite effective, though this arrangement introduces difficulties in maintaining the current store. A nice compromise is a history/archival pair, with the former containing the transaction-current rows and the latter containing those rows with a transaction-stop time before "now." The history store is period-stamped in valid time and instant-stamped in transaction time (the transaction-start time); the transaction-stop time is implicit and is equal to "until changed." The archival store is period-stamped in both valid and transaction time. It is convenient to maintain the archival store via triggers defined on the history store.

If space is at a premium, then it is best to use instant stamping in transaction time in the archival store, at a cost in query time for noncurrent transaction queries.

The archival store component of bitemporal tables can be vacuumed; having valid time around allows more specific vacuuming specifications.

10.9   READINGS

The bitemporal time diagram was introduced by Christian S. Jensen and the author [53]. A variation of this diagram was independently explored by James Clifford and Tomás Isakowitz [24]; the diagram has also been extended to incorporate reference time [23]. Heidi Gregersen and Christian S. Jensen discuss implementing integrity constraints on tables with various kinds of temporal support [39].

Blaha and Premerlani differentiate, for portfolio databases, "the time when a transaction occurs" (in our terminology, the valid time of the transaction), "the time when the user records a transaction" (the transaction time of the transaction), "the time of valuation of an asset" (the valid time of the valuation), "the time a portfolio value is computed" (the valid time of the computed value), and "the time interval (starting time and ending time) for computing ROI" (the valid time of the ROI fact) [9, p. 195].

Table 10.14 Böhlen's classes of temporal integrity constraints.

Böhlen's terminology	This book's terminology	Example
nontemporal	nontemporal	Every employee must be assigned to at least one project.
intrastate	valid-time sequenced	At any point in his employment an employee must be assigned to at least one project.
interstate	valid-time nonsequenced	An employee who was assigned to the KAP92 project may not be assigned to the PMT project.
static	transaction-time sequenced	At every database state, credit entries are limited to $20 million.
dynamic	transaction-time nonsequenced	A project credit may not be deleted and subsequently (i.e., at a later database state) reasserted with an increased credit value.
transition	transaction-time nonsequenced[1]	A project credit may not be increased by updating the credit value.
intraelement	valid-time sequenced/ transaction-time sequenced	At every database state, it must hold that, at any point of his employment, an employee is assigned to at least one project.
interelement	valid-time nonsequenced/ transaction-time nonsequenced	If a project gets credits over a period of six years, which is extended to a period of ten years, then the number of employees assigned to that project must be decreased by two within one year.

[1]The predicate will be of the form of *meets*, that is, `TT_Stop = TT_Start`.

Michael Böhlen has investigated bitemporal integrity constraints in some detail. He developed a taxonomy of such constraints [15]. His terms can be mapped into our terminology, as illustrated in Table 10.14 with the constraints listed in that paper.

The theory behind oscillators is presented in a delightfully approachable manner by James Jespersen and Jane Fitz-Randolph in their concise and highly readable book, *From Sundials to Atomic Clocks* [58]. Also approachable, though at a more advanced level, is Philip Woodward's book, *My Own Right Time* [104].

Gerhard Rossum's book *History of the Hour: Clocks and Modern Temporal Orders* [81] is an expansive history of the mechanical clock and its impact on European society, from the Middle Ages to the industrial revolution. The definition of "jiffy" may be found in the *Hacker's Dictionary* [98].
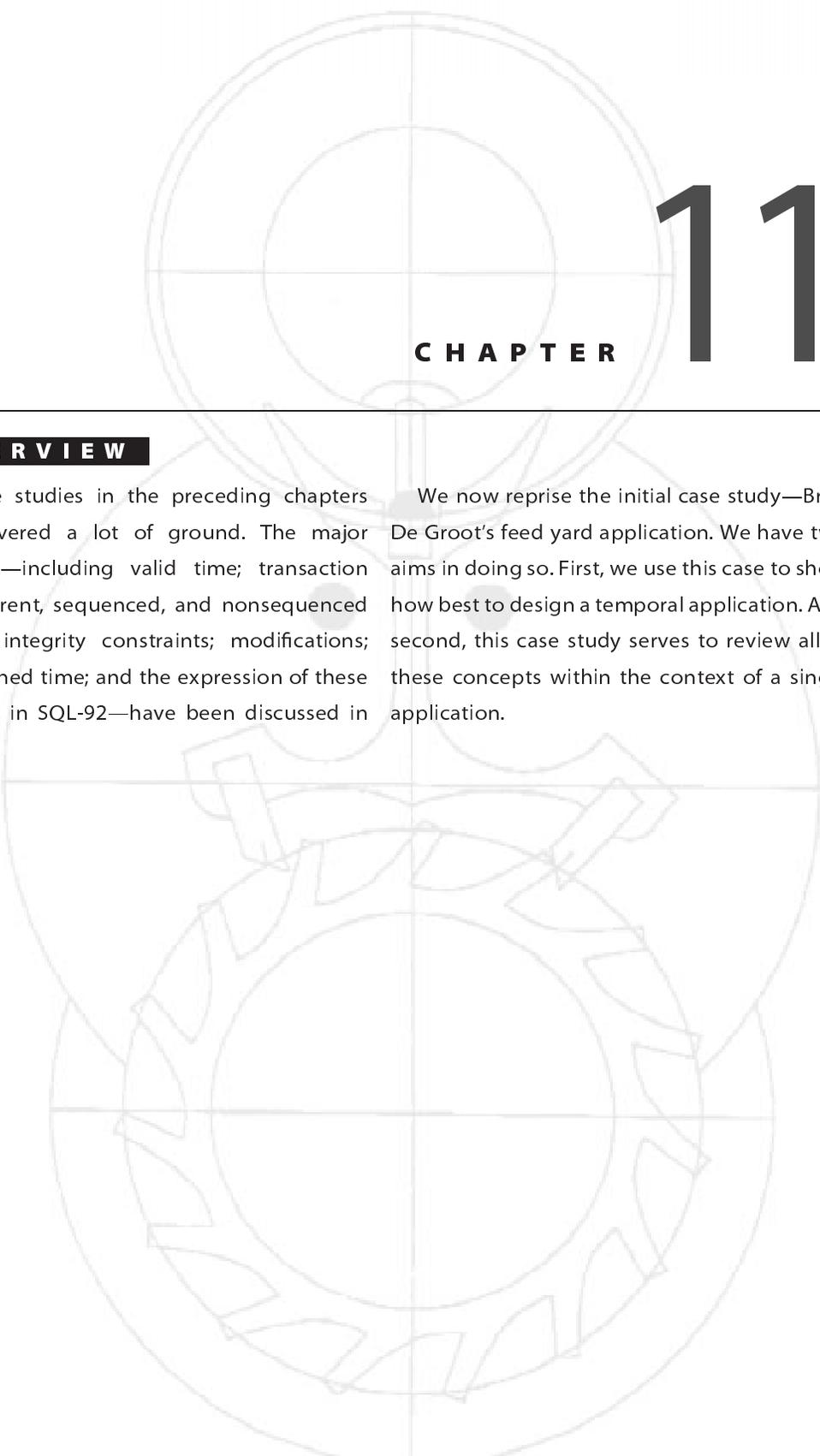
The case studies in the preceding chapters have covered a lot of ground. The major concepts—including valid time; transaction time; current, sequenced, and nonsequenced queries; integrity constraints; modifications; user-defined time; and the expression of these concepts in SQL-92—have been discussed in detail.

We now reprise the initial case study—Brad De Groot's feed yard application. We have two aims in doing so. First, we use this case to show how best to design a temporal application. And second, this case study serves to review all of these concepts within the context of a single application.

Temporal Database Design

The first case study we studied was that of Brad De Groot's feed yard application, discussed informally in Chapter 2. We now return to that case study, this time with the benefit of a deep understanding of temporal semantics conveyed in the intervening chapters.

Brad started his investigation into the temporal relationships between putative risk factor exposure and subsequent health events by understanding the structure of the data files maintained by the feed yards as they track the movement of cattle between pens. He carefully merged these data definitions into a global schema. After months of work, Brad had constructed an entity-relationship (ER) schema with some 40 entity types and relationships and over 150 roles. Even when printed in a small font, this ER diagram required a large poster to see it in its entirety. The relational schema generated from this conceptual model contains 55 tables and about 850 columns. In sum, the schema is typical: large and complex and somewhat overwhelming.

After Brad had finished the schema, he read an early draft of this book and realized that he had a bitemporal database on his hands. He worked with the author to understand his application in the framework presented in this book. This analysis pointed out semantic problems with the schema, which were solved by applying the methodology presented here.

11.1 PROPERLY SEQUENCING THE DESIGN

The case studies appearing in the preceding chapters are inextricably entwined with the design decisions relevant to their particular needs. Should a new application match those needs closely, then perhaps the design decisions would be appropriate

for that application. The purpose of the present chapter is to take a more general view, to outline precisely when each time-related decision should be considered.

Application design and implementation consist fundamentally of making a series of decisions, each impacting subsequent trade-offs, often in subtle and unexpected ways. Over the past two decades, experience and research has converged on a sequence of three basic steps: (1) conceptual design using the ER model, (2) logical design using the relational model, and finally (3) physical design to ensure adequate performance.

Unfortunately, because both the ER and the relational model do not themselves adequately support time-varying information, current practice using these models is actually counterproductive in places. Among the most egregious is temporal partitioning, which is generally considered very early in conceptual design, when in fact it should be one of the *last* considerations in physical design. Overly complex ER schemas and relational schemas have resulted from considering time earlier rather than later.

The temporal aspects of the application should be initially ignored when developing the conceptual schema.

In the approach we espouse here, conceptual design initially ignores the time-varying nature of the application. We focus on capturing the *current reality* and temporarily ignore any history that may be useful to capture. This selective amnesia somewhat simplifies what is often a highly complex task of capturing the full semantics of the application. An added benefit is that existing conceptual design metholodologies apply in full.

Only after the full design is complete do we augment the ER schema with the time-varying semantics of the application. We consider each component of the ER schema in turn, annotating that component with its temporal semantics. Entity types, relationship types, attributes, and keys are each individually addressed. These annotations are expressed in prose, so that they do not clutter the ER schema.

Similarly, logical design proceeds in two stages. First, the nontemporal ER schema is mapped to a nontemporal relational schema, a collection of tables. Here again we ignore the temporal aspects of the application, and thus can apply existing mapping strategies, unencumbered by considering how to capture history.

In the second stage of logical design, each of the annotations is applied to the logical schema, modifying the tables or integrity constraints to accommodate that temporal aspect. We proceed in a disciplined fashion, dealing with each annotation in turn.

In the following, we start with a nontemporal ER schema and proceed to a fully elaborated SQL schema, taking into account the time-varying nature of the application. This serves as yet another flight through the core temporal concepts, and also serves to emphasize the utility of thinking of time-varying data in these terms.

CONCEPTUAL DESIGN

Contrary to current practice, all temporal aspects should be *ignored* during most of conceptual design. An ER schema should be constructed that has no temporal features. All design considerations should be visited at this time, including keys, integrity constraints, composite attributes, multivalued attributes, relationship participation (one-to-one, one-to-many, many-to-many, optional, or mandatory), weak and strong entity types, subclasses, superclasses, categories, specialization, generalization, and attribute inheritance.

Once a carefully worked-out ER schema has been constructed, only then should the temporal aspects be considered.

11.2.1 **Nontemporal ER Schema**

Figure 11.1 shows a fairly small portion (about a sixth) of the initial ER schema for Brad's application, in the traditional notation of rectangles for entities and diamonds for relationships. Note that in describing this diagram, we use present tense, as time is not yet involved. Entity and relationship type names are in all caps, attribute names are capitalized, and individual entities (such as a particular lot) and individual relationships are not capitalized.

Strong Entity Types
There are two (strong) entity types present: FEEDYARD and APPLICATION. Brad has data from five feed yards.

Weak Entity Types
There are four weak entity types, indicated by a double rectangle; a double diamond indicates the identifying relationship type, which relates a strong entity type to the weak entity type. By necessity, this relationship is *total*, indicated with a double line. A pen is a fenced-in plot of land that can hold cattle. It is related to the FEEDYARD strong entity type via the IN_PEN relationship type. Since the relationship is total, every PEN entity must be in a particular FEEDYARD entity.

A lot is a specific group of cattle, resident in one pen or divided into multiple pens. The analysis of treatments and putative factors is carried out on lots of cattle. LOT is related to FEEDYARD through the IN_LOT relationship type.

The feed yards use FoxPro to collect this information; FoxPro stores each table in a dbf file. Brad then periodically copies these files onto a floppy disk and loads the information into his SQL database, running on Sybase SQLAnywhere. He terms each such transfer and loading, which can involve coordinating the data from multiple dbf files, a *backup*, for which a BACKUP weak entity is created.
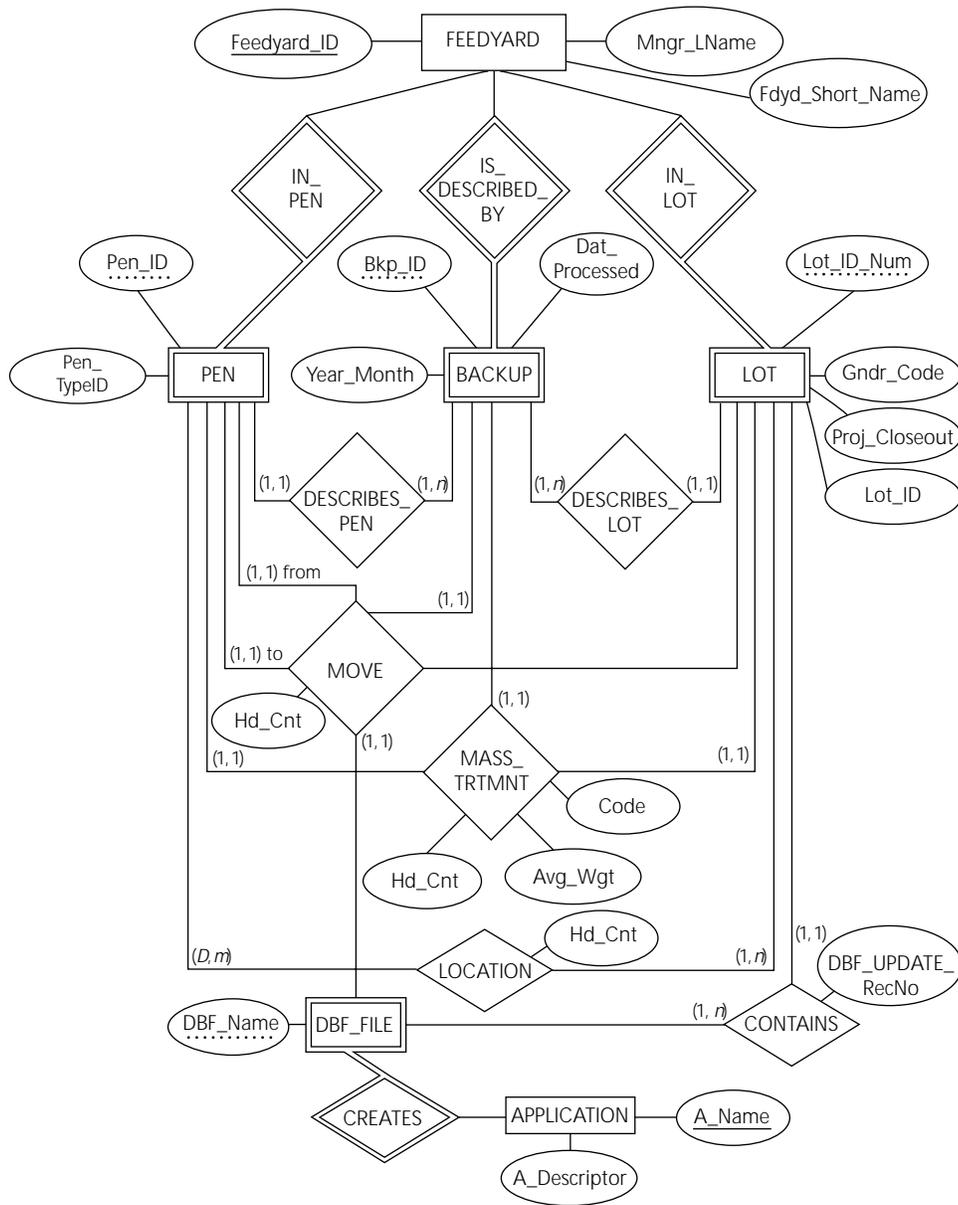
Figure 11.1 A nontemporal entity-relationship schema.

Entity Type Key Attributes

In the ER model, every entity must have an attribute that identifies that entity. Strong entity types have a key, which is underlined in the ER schema; weak entity types have a partial key, which is indicated with a dotted line. The key for FEEDYARD is Feedyard_ID; for APPLICATION it is A_Name.

A combination of the key for the strong entity type and the partial key for the weak entity type identifies each weak entity. Hence, a particular (Feedyard_ID, Pen_ID) pair will identify an individual pen.

Attributes

A few other attributes are included. You might ask, Why does the LOT entity type have both Lot_ID and Lot_ID_Num attributes? It turns out that the former is a character string used by the feed yard; the latter is an assigned integer and is designated the partial key.

The most interesting attributes in this context are the user-defined time attributes: Proj_Closeout in LOT (the projected date that the lot will leave the feed yard), and Year_Month and Date_Processed in BACKUP, indicating when the backup was processed, and which month the data was associated with. These attributes are independent of the other attributes and of the relationships that the entity type participates in. In particular, they do *not* indicate the valid-time or transaction-time extent of the entities or relationships. Rather, they approximate when the backup was taken and when it was processed. Data within the backup is used to calculate a more precise valid time (to the granularity of day). Because of various consistency checking and cleansing activities, the actual transaction time may be different than the Date_Processed.

LOT.Valid is a Boolean attribute indicating whether the information in the entity has passed various validation tests.

Note that we do *not* include timestamp attributes. Those attributes will be added when we map to the relational model (that is, to tables).

Relationship Types

There are five relationship types, in addition to the identifying relationship types discussed earlier. LOCATION identifies the pen(s) each lot is in. Cattle in a lot can reside in multiple pens, and a pen can hold cattle from multiple lots (thereby complicating the analysis of disease propagation). DESCRIBES relates lots to backups. A dbf file contains one or more lots. MASS_TRTMNT is the administration of some drug regime to the cattle of a lot resident in one or more pens. This is a ternary relationship type, as we associate each individual relationship with the backup during which it was loaded.

The MOVE relationship type is complex: it is quintary, capturing the movement of a (perhaps partial) lot of cattle from one pen (in the "from" role) to another pen

(in the "to" role). This movement was loaded during a backup, from a particular dbf file.

Several of the relationship types have associated attributes. Both the MOVE and LOCATION relationship types include a Hd_Cnt attribute; the MASS_TRTMNT relationship type includes Avg_Wgt and Trtmnt_Code attributes.

Participation Constraints

Participation constraints must be specified for each entity type participating in a relationship type. Each constraint is denoted in the ER schema with a pair of integers, denoting the minimum and maximum participation.

The minimum participation constraint (the first component of the pair) differentiates optional from mandatory participation. A minimum participation of 0 is termed optional; a minimum participation greater than zero is termed total, or mandatory. All but the LOCATION relationship type are total; a particular pen may be empty.

The maximum participation constraint (the second component of the pair) identifies the relationship as one-to-one, one-to-many, or many-to-many. For example, the DESCRIBES_LOT relationship is many-to-one. A backup describes many lots, but a lot is described by one backup.

In a nontemporal ER schema, these constraints are considered to hold at any point in time. When the schema is later interpreted to be a temporal ER schema, the constraints will then be considered as sequenced, holding at each isolated point in time.

In the feed yard schema, IN_PEN, IS_DESCRIBED_BY, CREATES, DESCRIBES_PEN, and DESCRIBES_LOT are all one-to-many relationship types. Each location relationship denotes cattle from one lot residing in one or more pens. Each move relationship denotes the movement of cattle from one lot from one pen to another pen, as recorded by one dbf file and one backup. Each mass_trtmnt relationship captures a treatment being applied to one lot in one pen, as recorded by one backup.

11.2.2 Adding Temporal Annotations

Once the nontemporal conceptual schema is complete, it is annotated with the time-varying semantics of each component. You can either indicate such semantics in the ER schema with icons, or you can list the annotations separately, in prose, as we do here.

Entity Lifespans

Entities have a *lifespan* denoting when they existed. Entities are instantaneous (with an extremely short lifespan) or have a lifespan with a duration. The lifespan of an entity can be an instant, a single period, or a set of periods. An application may

Puncta and Ostenta

One 10th-century table divided the day into hours (*horae*), which were further divided into points (*puncta*, five points to the hour) and *ostenta* (12 osts to the punct); each ost is equivalent to our minute. Other divisions, which could not have been measurable, included the 12th-century division of an hour into 4 points, 10 minutes, 15 parts, 40 movements, 60 marks, and 22,500 atoms (a second is exactly 6 1/4 atoms).

choose to model the lifespan of a person entity as starting at birth and terminating at death, while the lifespan of an employee entity may include several non-contiguous periods if she resigned and was later rehired.

The designer may choose whether to record the lifespan. If the entities of an entity type exist for all of (modeled) time, there may be no need to record the lifespan explicitly. Entity types with an implied lifespan are termed *nontemporal*. If the lifespan of entities is a subset of the modeled time, then the designer must decide whether it is relevant to the application to record the lifespan explicitly. If so, the designer should also specify the granularity of the lifespan.

Interestingly, only one entity type, LOT, has a lifespan that we wish to record in the database; that entity type has an associated granularity of DAY. The entities of the other five entity types exist for all of the modeled time (though we note in passing that two have a relevant transaction-time extent that should be stored).

Relationship Valid Time

A relationship type can either model instantaneous events, or it can model relationships that have a duration. The valid time for any specific relationship must be a subset of the intersection of the lifespans of the associated entities. A relationship is always associated with the same entities, regardless of whether it has a temporal duration and whether its attributes vary over time. The designer must choose whether to record the valid time of the relationship. If the valid time of the relationship type is recorded, the granularity should also be noted.

For each entity and relationship type, decide whether the valid time should be recorded, and if so, its granularity.

Table 11.1 summarizes the valid times of the 10 relationship types. The valid time of most relationships is not recorded; such types are considered *nontemporal relationship types*. The granularities of the MOVE and LOCATION lifespans are indicated as finer than a DAY. It turns out that several moves are possible in a day, so the granularity is expressed as DATE along with a Move_Order; multiple moves in a day are sequenced by the Move_Order value. (As noted above, we do not include timestamp attributes in the ER schema, so the Move_Order attribute will appear explicitly only when we map to tables.)

Table 11.1 Valid time of relationship types.

Relationship Type	Valid-Time Granularity	Comments
IN_PEN	Nontemporal	This relationship type is between two nontemporal entity types, and is itself nontemporal.
IS_DESCRIBED_BY	Nontemporal	
CREATES	Nontemporal	
CONTAINS	Nontemporal	This relationship type is between a nontemporal entity type, DBF_FILE, and a time-varying entity type, LOT. The relationship itself does not vary over valid time.
IN_LOT	Nontemporal	While lots have a lifespan, their entire lifespan is spent in one feed yard.
DESCRIBES_PEN	Nontemporal	This relationship type is between two nontemporal entity types.
DESCRIBES_LOT	Nontemporal	The relationship does not vary over time.
MOVE	Sub-DAY	This relationship type doesn't have a duration: a move is an instantaneous event. Hence, it should be associated with a single valid time indicating the instant when the move event occurred.
LOCATION	Sub-DAY	This is an important component, capturing which pen(s) the lot resides in. The location is definitely a time-varying relationship, so we record the lifespan. The granularity is the same as that of the MOVE relationship type. In fact, each LOCATION period is delimited by two MOVE instants.
MASS_TRTMNT	DAY	A treatment of the cattle of a specified lot in a specified pen is an instantaneous event.

Valid Time of Attributes

The value of an attribute may change over the lifespan of the associated entity or the valid time of the associated relationship, or may not vary over time. The valid time of an attribute's value for any specific entity (or relationship) must be a subset of the lifespan (valid time) of that entity (relationship).

> For each attribute, determine if the valid time of the attribute's value should be captured, and if so, the associated granularity.

We may want to record the valid time of the attribute. If the value is fixed for all time, capturing the valid time is probably not useful. Even if the attribute's value varies over time, we may be interested just in the current value, in which case there is no need to record the valid time. An attribute for which the valid time is not captured

The Fourth Harrison

"H-4" is the laconic name given to perhaps the most famous watch of all time. It was the fourth in a series of clocks that John Harrison (1693–1776) constructed in an effort to win the longitude prize, £20,000 (several million dollars in today's currency), promised by the British Parliament for a solution to the problem of determining a ship's longitudinal position while at sea. H-4, a large pocket watch at 13 centimeters (5 inches) in diameter and weighing 1.5 kilograms (3 pounds), was the first marine chronometer of sufficient precision to be used reliably for navigation. John Harrison ultimately collected the monetary award, though it took over 40 years (from H-1, which he started in 1730, to 1773) for Harrison to refine this exquisite manifestation of his mechanical genius. The watch and its predecessors are on display at the National Maritime Museum in the Old Royal Observatory at Greenwich, naturally.

is termed *nontemporal*. For each such attribute, we associate but a single value with each entity.

Table 11.2 summarizes the valid-time extent of the attributes, including some attributes for which there wasn't room in the ER schema. The attributes not mentioned are nontemporal.

The Hd_Cnt attributes for the MOVE and LOCATION relationship types differ in their valid time, emphasizing the need to consider the valid time of attributes separately from their associated entity or relationship type. An individual location relationship indicates that cattle from the specified lot resided in the specified pen. The valid time of this relationship extends from the first day that at least one head was moved to that pen, to the day the last head of cattle from that lot was transferred out of the pen.

During that period, other transfers may have taken place. As long as some cattle remained in the pen, this time-varying head count is associated with a single location relationship. For this reason, we wish to record the valid time of the Hd_Cnt attribute, to the granularity of DAY.

> A time-varying key uniquely identifies a particular entity at each point in time. A nontemporal key identifies a particular entity over all time.

A particular move relationship captures the transfer of cattle from one pen to another. Recall that this models an instantaneous event. Hence, the head count is fixed for this particular relationship, indicating that the attribute is itself nontemporal. The Hd_Cnt attribute of MASS_TRTMNT is nontemporal for the same reason.

Key attributes are particularly interesting. For such attributes, we may or may not record the valid time, independently of whether we record the lifespan of the entity type. In either case, it is important that the entity be identifiable by its key value, and that this value be unique. Associating

Table 11.2 Valid time of attributes.

Attribute	Valid-Time Granularity	Comments
BACKUP.Quirks	Nontemporal	This is a comment attribute.
BACKUP.VTRC_Last_ Date_Mod	Nontemporal	This is a user-defined time attribute, utilized in consistency checking. As such, it is relevant to transaction time (when the data was cleansed and stored in Brad's database) but not valid time (when the data was valid in the real world). In fact, there is no real-world correspondent to this attribute, and so valid time is not relevant.
BACKUP.BRDR_Last_ Date_Mod	Nontemporal	This is also used during consistency checking.
BACKUP.ARCH_Last_ Date_Mod	Nontemporal	This is a third user-defined time attribute, used during consistency checking.
LOT.Gndr_Code	DAY	See the discussion on page 14.
LOT.Proj_Closeout	DAY	The projected closeout is a user-defined attribute, indicating when the cattle are scheduled to be moved from the feed yard to the slaughterhouse. This scheduled date can change, as the weight and health of the individual animals vary.
LOT.In-Weight	DAY	As cattle are added to the lot, the value of this attribute may change.
LOT.Owner	DAY	
LOT.Comment	DAY	Comments can be tied to particular periods.
LOT.Valid	DAY	Consistency checks are performed on the data from the feed yard as it is moved into Brad's database. If any of these checks fail, the Valid attribute is set to false, until Brad manually checks the data and corrects any problems. The information at each day can be independently valid or invalid.
MOVE.Hd_Cnt	Nontemporal	For a particular MOVE relationship, this attribute is nontemporal.
LOCATION.Hd_Cnt	DAY	The head count for a particular location relationship is time-varying.
MASS_TRTMNT.Hd_ Cnt	Nontemporal	For a particular mass_trtmnt relationship, this attribute is nontemporal.

Table 11.3 Transaction time of entity types.

Entity Type	Transaction-Time Granularity	Comments
APPLICATION	—	
BACKUP	DAY	A backup is a dump of a FoxPro database file at a particular date. The information in that backup is processed and inserted into the database, at which time a BACKUP entity is created.
DBF_FILE	—	
FEEDYARD	—	
LOT	DAY	This is perhaps the most important entity type in the entire schema. The lots track cattle through the feed yard; data on lots must be as clean as possible. For this reason, Brad needs to record the transaction time of changes to the LOT entity, to a granularity of DAY (with the understanding that only a partial picture is captured).
PEN	—	

a time-varying key with an entity type indicates that, while the value changes over time, at any point in time it uniquely identifies an entity. It turns out that for this schema, the valid time of the keys for all six entity types is not recorded.

Transaction Time

Recall that recording transaction time retains the sequence of stored states. This aspect is entirely orthogonal to valid time and should be considered separately for entity types, relationship types, and individual attributes.

We first consider the six entity types (see Table 11.3). Unlike valid time, the transaction time of an entity cannot be instantaneous.

The transaction-time extent may not be relevant if the application doesn't care to track the changes of the database, or it may be explicitly stored. If the transaction time is to be recorded, then we also indicate the granularity.

Transaction time is important in this application because the data from the feed yards is quite dirty (the feed yards themselves are *very* dirty). Inconsistencies abound, which must be corrected manually. By recording the transaction-time extent, Brad is able to track when the changes were made and can thus work backwards to find exactly where the erroneous data originated.

The idiosyncrasies of this application do not permit all changes to be captured. Brad visits each feed yard about once a month and makes copies of the FoxPro files created by the various feed yard programs. He then brings these files back to his

Table 11.4 Transaction time of relationship types.

Relationship Type	Transaction-Time Granularity	Comments
CREATES	—	
CONTAINS	DAY	This relationship type is between an entity type that does not support transaction time (DBF_FILE) and an entity type that supports transaction time (LOT).
IN_LOT	—	
IN_PEN	—	
IS_DESCRIBED_BY	—	
DESCRIBES_PEN	—	
DESCRIBES_LOT	—	
LOCATION	DAY	Brad wishes to track changes to this particularly important relationship for putative risk factor exposure.
MASS_TRTMNT	—	
MOVE	DAY	This relationship directly affects the LOCATION relationship, so changes to MOVE are tracked as well.

For each entity and relationship type, decide whether the transaction-time extent should be captured, and if so, its granularity. For each attribute, determine if the transaction time should be recorded, and if so, its granularity.

database server, where he cleans and loads the data into his data warehouse. As such, he captures most changes to the granularity of a day. However, if some data were entered into a feed yard program, then subsequently modified before Brad had a chance to grab the file, then the older version will be lost. Hence, the record of the changes as reflected in the transaction-time support is necessarily partial. One way to address this loss of information would be to modify all of the feed yard programs to maintain transaction time with their data.

Next, we consider the relationship types (Table 11.4). If transaction time is recorded for none of the participating entity types, recording the transaction time for the relationship type is generally not indicated, but is still possible. In any case, the transaction time of a relationship must be contained by the intersection of the transaction times of the participating entities.

Following the entity and relationship types, we draw our attention to the attributes (Table 11.5); again, we omit attributes whose transaction-time extent is not recorded. The transaction-time support of individual attributes is independent of the associated entity or relationship type, or of other attributes of that entity or relationship type. However, in any case, the transaction-time extent of an attribute's value must be contained in the transaction time of the associated entity or relationship.

Table 11.5 Transaction time of attributes.

Attribute	Transaction-Time Granularity	Comments
BACKUP.Year_Month	DAY	Multiple backups might pertain to a particular year and month. Transaction time is recorded to a granularity of DAY, in order to tease apart the sequence of changes.
BACKUP.Date_Processed	DAY	As noted before, because of various consistency checking and cleansing activities, the actual transaction time may be different than the Date_Processed.
BACKUP.Quirks	DAY	
BACKUP.VTRC_Last_Date_Mod	DAY	
BACKUP.BRDR_Last_Date_Mod	DAY	
BACKUP.ARCH_Last_Date_Mod	DAY	
LOT.Lot_ID_Num	DAY	Changes to the attributes of this critical relationship are tracked by recording the transaction time.
LOT.Lot_ID	DAY	
LOT.Gndr_Code	DAY	
LOT.Proj_Closeout	DAY	
LOT.In-Weight	DAY	
LOT.Owner	DAY	
LOT.Comment	DAY	
LOT.Valid	DAY	
LOT.DBF_Valid	DAY	
LOCATION.Hd_Cnt	DAY	This is also an important attribute to track.
MOVE.Hd_Cnt	DAY	This attribute relates to that of the same name in LOCATION.
CONTAINS.DBF_Update_RecNo	DAY	This attribute specifies the record number utilized in the update.

11.3 LOGICAL DESIGN

At this point, we have a nontemporal entity-relationship conceptual schema (Figure 11.1), augmented with annotations in prose that describe the time-varying aspects of this schema. These annotations concern the lifespan and transaction-time aspects of entity types and the valid- and transaction-time aspects of relationship types, attributes, and integrity constraints.

We now map this schema into a logical data model, here, the relational model, resulting in a collection of tables. This mapping is done in two stages. In the first stage, the temporal annotations are for the most part ignored, thereby arriving at a nontemporal logical schema. This logical schema is then modified to accommodate the time-varying aspects, adding timestamp column(s) and decomposing some tables into multiple constituent tables. Also during this second stage, various SQL assertions and constraints are defined on the tables to capture temporal integrity constraints.

11.3.1   Mapping to Relational Schema

In this first stage, the following mapping actions are performed. We are laconic only because this process doesn't concern time, and so should be familiar to you. Indeed, this stage can be performed automatically by CASE tools, as the steps do not concern temporal aspects.

To keep things straight, we use a Roman font for conceptual constructs (e.g., the LOT entity type) and a `sans-serif` font for logical constructs (e.g., the `LOT` table).

1. Create a table for each regular entity type (FEEDYARD, APPLICATION). Add columns for each attribute. The primary key is the key of the corresponding entity type.
2. Create a table for each weak entity type (PEN, BACKUP, LOT, DBF_FILE). Include the key of the strong entity type as a foreign key. The primary key is a combination of this entity type's partial key and the key of the strong entity type.
3. For each one-to-one relationship type, extend a table corresponding to one of the entity types with the key of the other entity type as a foreign key. We have no such relationship types here.
4. For each one-to-many binary relationship type, either extend a table or create a new table for the relationship type. For the DESCRIBES_PEN, DESCRIBES_LOT, and CONTAINS relationships, we extend the table on the "one" side (`PEN`, `LOT`, and `LOT`, respectively) with the key of the other entity type (of BACKUP, of BACKUP, and of DBF_FILE, respectively) as a foreign key, as well as the attributes of the relationship.
5. Create a table for each remaining relationship type (MOVE, MASS_TRTMNT, LOCATION). Add foreign keys for all participating entity types. The primary key is in general a subset of these foreign keys, but is usually (and in these cases) just the combination of them.
6. Create a table for each multivalued attribute. There are no such attributes here.

The result of this first stage is the nontemporal database schema listed in Figure 11.2, with the primary key columns underlined. We omit some of the `FDYD`

```
FDYD ( FDYD_ID, NAME, FDYD_SHORT_NAME, FDYD_MNGR_LNAME,...,
  UNIQUE (FDYD_SHORT_NAME))

LOT (FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT, IN_WEIGHT,
    VALID, OWNER, COMMENT, BKP_ID, A_NAME, DBF_NAME,
    DBF_UPDATE_RECNO,
  UNIQUE (FDYD_ID, LOT_ID_NUM, LOT_ID),
  FOREIGN KEY (FDYD_ID) REFERENCES FDYD,
  FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP,
  FOREIGN KEY (A_NAME, DBF_NAME) REFERENCES DBF_FILE
)

PEN (FDYD_ID, PEN_ID, PEN_TYPE_CODE, BUNK_LENGTH, APRON_WIDTH,
    PEN_AREA, WATER_SPACE, BKP_ID,
  FOREIGN KEY (FDYD_ID) REFERENCES FDYD,
  FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP
)

APPLICATION (A_NAME, A_DESCRIPTION, A_DATA_DIRECTORY)

DBF_FILE (A_NAME, DBF_NAME, DBF_DESCRIPTION, DBF_USED,
  FOREIGN KEY (A_NAME) REFERENCES APPLICATION
)

BKP (FDYD_ID, BKP_ID, YEAR_MONTH,  DATE_PROCESSED, QUIRKS,
    VTRC_LAST_DATE_MOD, BRDR_LAST_DATE_MOD, ARCH_LAST_DATE_MOD,
  FOREIGN KEY (FDYD_ID) REFERENCES FDYD
)
```

Figure 11.2 Initial nontemporal logical schema (continued on page 358).

columns to save space. Finally, we've changed a few table and column names to reflect the actual names that Brad used.

In Figure 11.3, the same (nontemporal) schema is shown using the crow's-feet notation employed by the PowerDesigner tool that Brad used to create the schema. The diamonds represent foreign keys, the crow's feet represent relationships that are many participation, the small circles denote optional participation, and the small slashes across links denote mandatory participation.

11.3.2 Applying Temporal Annotations

In this second stage, we apply the temporal annotations, elaborated in Section 11.2.2. As we are revisiting the ground traversed in previous chapters, back point-

```
LOT_MOVE (FDYD_ID, LOT_ID_NUM, FROM_PEN_ID, TO_PEN_ID, HD_CNT, BKP_ID,
    A_NAME, DBF_NAME,
  FOREIGN KEY (FDYD_ID, FROM_PEN_ID) REFERENCES PEN(FDYD_ID, PEN_ID),
  FOREIGN KEY (FDYD_ID, TO_PEN_ID) REFERENCES PEN(FDYD_ID, PEN_ID),
  FOREIGN KEY (A_NAME, DBF_NAME) REFERENCES DBF_FILE,
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP
)

LOT_LOC (FDYD_ID, LOT_ID_NUM, PEN_ID, HD_CNT, YEAR_MONTH,
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, PEN_ID) REFERENCES PEN
)

MASS_TRTMNT (FDYD_ID, LOT_ID_NUM, PEN_ID, M_TRTMNT_AVG_WGT,
    M_TRTMNT_CODE, M_TRTMNT_HD, BKP_ID,
  FOREIGN KEY (FDYD_ID, PEN_ID) REFERENCES PEN,
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP
)
```

Figure 11.2 (continued)

ers (in the form of page numbers in parentheses) are provided to the pages that introduce these approaches via the case studies, providing elaboration and other alternatives.

User-Defined Time Attributes

Each attribute is mapped to a column in the associated table. Attributes that record user-defined time values are differentiated by type: an instant (page 26), an interval (page 30), or a period (page 89). All temporal values have a granularity (page 74), though some DBMSs fix the granularity of their temporal type(s). For anchored values (instants and periods), SQL-92 and some DBMSs allow a specified time zone to be stored (page 29). For the rest, the time zone is either implicit or fixed. "Now" should be represented with "forever," or a close approximation (page 120).

For each attribute that is a user-defined time, choose a granularity supported by SQL-92.

Periods may be represented as a pair of instants (page 89), whether the representation of each of the delimiting instants is closed (contained in the period) or open (just outside the period). A closed-open period representation, in which the end timestamp specifies the granule *after* the last granule of the period, is preferable (page 91). The time zone, if any, may be stored with the first instant (page 90).

Figure 11.3 A nontemporal crow's-feet schema.

There are six user-defined time columns: `LOT.PROJ_CLOSEOUT`, of type DATE; `BKP.YEAR_MONTH`, of type CHAR (since SQL-92 has no instant type of a MONTH granularity); `BKP.PROCESSED`, of type DATE; and the last three `LAST_DATE_MOD` columns in `BKP`, of type DATE.

Entity Lifespans

To each table corresponding to an entity type for which the lifespan, or valid time of an associated attribute, is captured (page 114), there are two alternatives for timestamps. One possibility is a single time specifying (1) when the entity occurred, if it is instantaneous, (2) when the entity's lifespan began, if the entity has duration, or (3) when an attribute became valid. Another possibility is a period of validity (page 116), generally represented with two instants, start and end. If the entity's lifespan consists of several disjoint periods, or if attributes change values over time, a single entity will be mapped to several rows in the corresponding table.

One entity type has a recorded lifespan: LOT. Hence, we add `FROM_DATE` and `TO_DATE` columns to record the individual periods of that lifespan.

Code Fragment 11.1   `LOT` **is a valid-time state table.**

```
ALTER TABLE LOT ADD COLUMN FROM_DATE DATE
ALTER TABLE LOT ADD COLUMN TO_DATE DATE
```

Relationship Valid Time

To each table corresponding to a relationship type with a recorded valid-time extent or having attribute(s) whose valid time is recorded, we add either instant or period timestamps.

For tables corresponding to entity and relationship types for which valid time is to be recorded, add either a single instant timestamp column or a period timestamp, represented with two instant timestamp columns.

As indicated in Table 11.1, the valid time of three relationship types, LOCATION, MOVE, and MASS_TRTMNT, is recorded. The LOCATION relationships have a temporal duration; relationships of the other two relationship types are instantaneous.

The `LOT_LOC` table has an interesting timestamp, consisting of four (!) columns: `FROM_DATE`, `FROM_MOVE_ORDER`, `TO_DATE`, and `TO_MOVE_ORDER`. Recall that cattle can be moved twice (or more) in a day; the move order differentiates these moves.

Code Fragment 11.2   `LOT_LOC` **is a valid-time state table.**

```
ALTER TABLE LOT_LOC ADD COLUMN FROM_DATE DATE
ALTER TABLE LOT_LOC ADD COLUMN FROM_MOVE_ORDER INT
ALTER TABLE LOT_LOC ADD COLUMN TO_DATE DATE
ALTER TABLE LOT_LOC ADD COLUMN TO_MOVE_ORDER INT
```

For instantaneous relationship types, an instant timestamp is added to the associated table. For the `LOT_MOVE` table, we add `AT_DATE` and `AT_MOVE_ORDER`.

Code Fragment 11.3   LOT_MOVE **is a valid-time event table.**

```
ALTER TABLE LOT_MOVE ADD COLUMN AT_DATE DATE
ALTER TABLE LOT_MOVE ADD COLUMN AT_MOVE_ORDER INT
```

Code Fragment 11.4   MASS_TRTMNT **is a valid-time event table.**

```
ALTER TABLE MASS_TRTMNT ADD COLUMN AT_DATE DATE
```

Valid Time of Attributes

All attributes have a valid time; sometimes this time is recorded. If so, it is associated with a granularity. If the lifespan of the associated entity or the valid time of the associated relationship is *not* recorded, the time-varying columns should be placed in a separate table, along with the primary key of the original table, which also serves as a foreign key to that table. This task is termed *temporal support decomposition*. Fortunately, the only time-varying attributes listed in Table 11.2 are those associated with the LOT and LOCATION entity types, both of which have a recorded valid time.

Symmetrically, if this time *is* recorded, but the valid time of an attribute is not recorded, it may be useful to put that column in a separate table, with a foreign key referencing the original, time-varying table.

Decompose tables so that all attributes of a table have an identical temporal support and precision.

Another rub occurs when the granularity of the attribute is finer than that of the entity or relationship type to which the attribute is attached. In such situations, there are two possible paths. Either we can change the granularity of the associated table to that of the column (say, from YEAR to DAY), or we can break off those columns(s) into a separate table, termed *precision decomposition*. The attributes of the LOT entity type all have the same granularity (DAY) as LOT itself; the Hd_Cnt attribute of LOCATION has a granularity of DAY, which is *coarser* than the granularity (Sub-DAY) of the associated relationship type.

Our analysis thus indicates that neither flavor of decomposition applies to this schema.

Transaction Time

For each table associated with an entity or relationship type for whose transaction-time periods are recorded, or that is associated with attribute(s) whose transaction-time periods are recorded (page 249), there are two basic alternatives: instant-stamped or period-stamped tables.

For an instant-stamped table, the choices are a tracking log, a restricted backlog, or a general backlog. A tracking log is a transaction-time table that retains the original, snapshot table (page 220). Its schema comprises the columns of the

For tables corresponding to entity and relationship types for which transaction time is to be recorded, add either two timestamps denoting the period of presence or a single transaction timestamp, optionally with an additional operation column, if the table is to be a backlog.

monitored table, along with a single transaction timestamp column, indicating when the transaction committed. The value of the timestamp column can never extend past now (page 249). A backlog is a transaction-time table with an additional column, the operation, which may be an insertion, a deletion, or an update (page 233). In some cases, a restricted backlog, with insertions not recorded (page 220), is perfectly fine; in most other cases, it is best to go with the fully general backlog, using after-images consistently (page 235).

The other alternative is a table with a period timestamp indicating the period of presence (page 254), generally represented with two instants, start and end, specifying the period starting when the row was inserted or updated and ending when the period was removed, as a side effect of a deletion or update.

For the LOT table, we use period timestamps, adding two columns of the appropriate granularity to the associated tables. For the LOT_MOVE, LOT_LOC, and BKP tables, we use an instant timestamp, specifically, a backlog containing after-images.

Code Fragment 11.5   LOT**,** LOT_MOVE**,** LOT_LOC**, and** BKP **are transaction-time tables.**

```
ALTER TABLE LOT ADD COLUMN START_DATE DATE
ALTER TABLE LOT ADD COLUMN STOP_DATE DATE

ALTER TABLE LOT_MOVE ADD COLUMN WHEN_CHANGED DATE
ALTER TABLE LOT_MOVE ADD COLUMN OPERATION CHAR(1)

ALTER TABLE LOT_LOC ADD COLUMN WHEN_CHANGED DATE
ALTER TABLE LOT_LOC ADD COLUMN OPERATION CHAR(1)

ALTER TABLE BKP ADD COLUMN WHEN_CHANGED DATE
ALTER TABLE BKP ADD COLUMN OPERATION CHAR(1)
```

As the LOT, LOT_MOVE, and LOT_LOC tables already had valid timestamps, adding transaction-time support renders them bitemporal (page 279).

Transaction-time support may induce additional temporal support decomposition.

As with valid time, we must also consider temporal support decomposition. Basically, the tables should be decomposed until the temporal support (which includes the details of whether valid time is recorded, whether transaction time is recorded, and the granularity of each) of all attributes is identical.

There is one place in this schema where temporal decomposition is indicated. Most of the attributes in LOT are bitemporal, but BKP_ID, A_NAME, DBF_NAME, and DBF_UPDATE_RECNO, because they were obtained from the CONTAINS relationship type, which records only transaction time, do not vary in valid time, and thus differ from

the other attributes in that table in their temporal support. We move those attributes to a separate table, LOT_CONTAINS, include the primary key of LOT, and transfer the foreign keys involving these attributes to LOT_CONTAINS. Finally, we make LOT_CONTAINS a backlog.

Code Fragment 11.6   **Move the** BKP_ID**,** A_NAME**,** DBF_NAME**, and** DBF_UPDATE_RECNO **columns into a separate backlog table.**

```
ALTER TABLE LOT DROP COLUMN BKP_ID
ALTER TABLE LOT DROP COLUMN A_NAME
ALTER TABLE LOT DROP COLUMN DBF_NAME
ALTER TABLE LOT DROP COLUMN DBF_UPDATE_RECNO

CREATE TABLE LOT_CONTAINS (FDYD_ID, LOT_ID_NUM, BKP_ID, A_NAME,
    DBF_NAME, DBF_UPDATE_RECNO,
    WHEN_CHANGED DATE, OPERATION CHAR(1),
  PRIMARY KEY (FDYD_ID, LOT_ID_NUM),
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP,
  FOREIGN KEY (A_NAME, DBF_FILE) REFERENCES DBF_FILE
)
```

Primary Keys

Primary keys are handled differently for valid-time, transaction-time, and bitemporal tables, which we'll consider in order.

If the entity (or relationship) type captures neither valid nor transaction time, the primary key remains as specified in the initial logical schema. This case applies to the FDYD, PEN, APPLICATION, and DBF_FILE tables in Figure 11.2 on page 357.

For entity types, we previously differentiated instantaneous entity types from entity types that have a temporal duration. All of these aspects will come into play when determining precisely what comprises the primary and foreign keys for the tables, as well as what additional assertions are required to correctly reflect the ER schema as tables.

A valid-time sequenced primary key may require an assertion and an additional surrogate identifier column; there are three cases.

Integrity constraints (ICs) for valid-time tables can be classified according to whether they apply to the current state, termed a current IC (page 123); apply to each point in time independently, termed a sequenced IC (page 118); or apply to the table treating the timestamp as just another column(s), termed a nonsequenced IC (page 123).

It is easy to confound keys in the conceptual ER schema with keys in the logical relational schema. To keep them straight, we strictly follow the terminological convention of referring to conceptual keys without the adjectives "primary" or "foreign" (e.g., the key of the LOT entity type)

The Bulova Accutron

This watch advanced chronometry in two ways when it was introduced in 1960: it guaranteed a precision of at least a minute a month (2 seconds a day), and ensured that this guarantee would hold over the life of the watch. Previous watches, even if they started out with high precision, gradually became less accurate due to wear and the changing viscosity of the lubrication between moving parts. The Accutron did not use a pendulum to regulate an uncoiling spring; rather, it used the vibrations of an electrically driven tuning fork to advance the hands. This fork vibrated at 360 Hz, which moved a tiny arm back and forth, turning a miniscule ratchet wheel with 300 microscopic teeth, each separated from the next by a distance of 0.001 inch. Due to the very small forces involved, there is no wear, and since there is no oil at the tuning fork, the watch will hold its rate far better than a lubricated timepiece. The drawback is the difficulty of repair; Bulova provides microscope kits for this purpose.

and to logical keys with "primary" or "foreign" adjectives (e.g., the primary key of the LOT table). Also, logical keys may be current, sequenced, or nonsequenced, but conceptual keys never are.

The primary key of a table associated with an entity or relationship type capturing its valid-time extent is a sequenced primary key.

We have three cases to consider:

Case 1   **The entity or relationship type is instantaneous and the valid-time instant is recorded.**

A particular entity could occur at multiple points in time. A sequenced primary key constraint is required, but is especially easy to state for instantaneous entities and relationships: the timestamp column is added to the primary key of the associated table. This case applies to the MASS_TRTMNT table.

Case 2   **The entity type has lifespan with duration, and the lifespan is recorded.**

An entity whose lifespan has duration raises the question of how to identify that two rows, with different primary key values, are associated with the same entity. The approach is to add a new column, a surrogate identifier column, to the table, which then constitutes a time-invariant key, even as the original key varies over time. (As we'll see on page 378, sometimes the surrogate identifier column is not required.)

The surrogate identifier column should be made the sequenced primary key of the table. Such keys cannot be expressed solely using SQL-92's PRIMARY KEY construct (page 117). Adding the start time of the timestamp of the row, or the end time, or both, does not serve to convert a nontemporal key to a sequenced key (page 118). Instead, a sequenced primary key can be expressed as an SQL assertion

or table constraint (page 118); to keep SQL happy, we also add FROM_DATE to the declared primary key.

Two other changes are required: (a) add an assertion stating that the combination of the surrogate column and the original primary key is sequenced unique, and (b) replace all foreign keys referencing this table with the new surrogate column. This case is not present in the example schema.

Case 3   **The relationship type has a valid-time extent, which is recorded.**

The original primary key on the valid-time state table should be interpreted as a sequenced primary key, which can be expressed as an SQL assertion or table constraint (page 118); we also add FROM_DATE to the declared primary key. This case is not present.

Code Fragment 11.7   MASS_TRTMNT**'s primary key is valid-time sequenced.**

```
ALTER TABLE MASS_TRTMNT DROP PRIMARY KEY

ALTER TABLE MASS_TRTMNT
    ADD PRIMARY KEY (FDYD_ID, LOT_ID_NUM, PEN_ID, AT_DATE)
```

The WHEN_CHANGED column should be added to the primary key of transaction-time tables to effect a current, and thus sequenced, primary key.

For transaction-time tables, things are a little easier because instantaneous tables are not possible. The original primary key on the table should be interpreted as a transaction-time sequenced primary key. As contrasted with valid time, such keys *can* be expressed solely using SQL-92's PRIMARY KEY construct, by simply adding the STOP_DATE column, which serves to convert the original key to a current key, which is equivalent to a sequenced key in the case of a transaction-time table (page 254).

BKP and LOT_CONTAINS are transaction-time tables.

Code Fragment 11.8   **The primary key of** BKP **and** LOT_CONTAINS **is transaction-time sequenced.**

```
ALTER TABLE BKP DROP PRIMARY KEY

ALTER TABLE BKP ADD PRIMARY KEY (FDYD_ID, BKP_ID, WHEN_CHANGED)

ALTER TABLE LOT_CONTAINS DROP PRIMARY KEY

ALTER TABLE LOT_CONTAINS PRIMARY KEY (FDYD_ID, LOT_ID_NUM,
    WHEN_CHANGED)
```

For bitemporal tables, things get more interesting because the two kinds of time must be considered simultaneously. Fortunately, the alternatives combine in the predictable way.

This schema includes three bitemporal tables, LOT, LOT_LOC, and LOT_MOVE. The first is associated with an entity type, and the other two are associated with relationship types. The first two have a valid-time extent, and the third is instantaneous.

There are three cases for the primary key of bitemporal tables, mirroring those for valid-time tables.

The LOT entity type has a recorded lifespan, so we use the second case for valid time to state that the LOT primary key is valid-time sequenced/transaction-time sequenced. Again, we implement transaction-time sequenced with transaction-time current. As will be discussed on page 378, for LOT it turns out that the surrogate identifier column is not required, so we don't include it here.

Code Fragment 11.9   LOT **has a valid-time sequenced/transaction-time sequenced primary key of (**FDYD_ID**,** LOT_ID_NUM**).**

```
ALTER TABLE LOT DROP PRIMARY KEY (FDYD_ID, LOT_ID_NUM)

ALTER TABLE LOT ADD PRIMARY KEY (FDYD_ID, LOT_ID_NUM,
    FROM_DATE, STOP_DATE)

CREATE ASSERTION LOT_seq_seq_primary_key
CHECK (NOT EXISTS ( SELECT *
                    FROM LOT AS L1
                    WHERE L1.STOP_DATE = DATE '9999-12-31'
                      AND 1 < (SELECT COUNT(*)
                               FROM LOT AS L2
                               WHERE L1.FDYD_ID = L2.FDYD_ID
                                 AND L1.LOT_ID_NUM = L2.LOT_ID_NUM
                                 AND L1.FROM_DATE < L2.TO_DATE
                                 AND L2.FROM_DATE < L1.TO_DATE
                                 AND L2.STOP_DATE = DATE '9999-12-31'))
        )
```

MOVE is an instantaneous relationship type, and thus utilizes the first case for valid time. We need only add the valid and transaction timestamps to the primary key.

Code Fragment 11.10   LOT_MOVE **has a valid-time sequenced/transaction-time sequenced primary key of (**FDYD_ID**,** LOT_ID_NUM**,** FROM_PEN_ID**,** TO_PEN_ID**).**

```
ALTER TABLE LOT_MOVE DROP PRIMARY KEY

ALTER TABLE LOT_MOVE ADD PRIMARY KEY (FDYD_ID, LOT_ID_NUM,
    FROM_PEN_ID, TO_PEN_ID, AT_DATE, AT_MOVE_ORDER,
    WHEN_CHANGED)
```

Finally, the key for LOT_LOC, being associated with a noninstantaneous relation-ship type, utilizes the last case for valid time, resulting in a valid-time sequenced/transaction-time sequenced primary key. How-ever, because LOT_LOC is implemented as a backlog, rather than as a transaction-time state table, we can't simply check for a STOP_DATE of "forever." Instead, we must use the last relevant entry in the backlog that is an insert or an update entry.

For transaction-time tables implemented as backlogs, only the last insert or update entry is relevant.

Code Fragment 11.11   LOT_LOC **has a valid-time sequenced/transaction-time sequenced primary key of (**FDYD_ID**,** LOT_ID_NUM**,** PEN_ID**).**

```
ALTER TABLE LOT_LOC DROP PRIMARY KEY (FDYD_ID, LOT_ID_NUM, PEN_ID)

ALTER TABLE LOT_LOC ADD PRIMARY KEY (FDYD_ID, LOT_ID_NUM,
    PEN_ID, FROM_DATE, WHEN_CHANGED)

CREATE ASSERTION LOT_LOC_seq_seq_primary_key
CHECK (NOT EXISTS ( SELECT *
      FROM LOT_LOC AS L1
      -- L1 is the last insert or update entry
      WHERE (L1.OPERATION = 'I' OR L1.OPERATION = 'U')
        AND NOT EXISTS (
                SELECT *
                FROM LOT_LOC AS L3
                WHERE L3.FDYD_ID = L1.FDYD_ID
                  AND L3.LOT_ID_NUM = L1.LOT_ID_NUM
                  AND L3.PEN_ID = L1.PEN_ID
                  AND L3.WHEN_CHANGED > L1.WHEN_CHANGED)
        AND 1 < (SELECT COUNT(*)
          FROM LOT_LOC AS L2
          WHERE L1.FDYD_ID = L2.FDYD_ID
            AND L1.LOT_ID_NUM = L2.LOT_ID_NUM
            AND L1.PEN_ID = L2.PEN_ID
            AND L1.FROM_DATE < L2.TO_DATE
            AND L2.FROM_DATE < L1.TO_DATE
            -- L2 is the last insert or update entry
            AND (L2.OPERATION = 'I' OR L2.OPERATION = 'U')
            AND NOT EXISTS (
                    SELECT *
                    FROM LOT_LOC AS L3
                    WHERE L3.FDYD_ID = L2.FDYD_ID
                      AND L3.LOT_ID_NUM = L2.LOT_ID_NUM
                      AND L3.PEN_ID = L2.PEN_ID
                      AND L3.WHEN_CHANGED > L2.WHEN_CHANGED)))
      )
```

Referential Integrity

There are 17 foreign keys specified in Figure 11.2: 7 for the one-to-many binary relationship types, 2 for the many-to-many LOCATION relationship type, 3 for the MASS_TRTMNT ternary relationship type, and 5 for the quintary MOVE relationship type. The LOT_CONTAINS table also comes with a foreign key (CF-11.6).

If the referenced table is nontemporal, the original foreign key constraints may be retained.

The original foreign key constraints work fine if the referenced table is nontemporal (assuming that a nontemporal table contains time-invariant data (page 126), which is the common assumption). Most relationship types here are nontemporal, and so the associated referential integrity constraints specified before can remain. Similarly, foreign keys referencing nontemporal tables are fine as is. We're thus satisfied with LOT → FDYD, LOT_CONTAINS → DBF_FILE, PEN → FDYD, DBF_FILE → APPLICATION, BKP → FDYD, LOT_MOVE → PEN (two referential integrity constraints), LOT_MOVE → DBF_FILE, LOT_LOC → PEN, and MASS_TRTMNT → PEN.

Referential integrity on valid-time tables should be expressed as a sequenced constraint.

There are six variants of temporal integrity constraints (page 326). However, for referential integrity, the valid-time sequenced variety is generally indicated. If both tables are valid-time tables, then an assertion is required, either on the referencing table individually (page 128), or on the referenced table stating that the histories are contiguous and on the referenced table stating that the history is contained in that of the referenced table (page 129). Ensuring the histories are contiguous can be done by filling the gaps in the referenced table (page 180).

No foreign key constraints here apply to valid-time tables.

If both tables are transaction-time tables, transaction-time current constraints suffice, which can be implemented by adding the STOP_DATE to the foreign key (page 254). This also works for LOT_MOVE → BKP; since LOT_MOVE is valid timestamped with an instant, we can ignore the valid time in the referential integrity constraint.

Code Fragment 11.12   **(**LOT_MOVE.FDYD_ID**,** LOT_MOVE.BKP_ID**) is a nonsequenced/current foreign key for** BKP**.**

```
ALTER TABLE LOT_MOVE DROP FOREIGN KEY (FDYD_ID, BKP_ID)
    REFERENCES BKP

ALTER TABLE LOT_MOVE ADD FOREIGN KEY
    (FDYD_ID, BKP_ID, WHEN_CHANGED) REFERENCES BKP
```

If the referencing table is a nontemporal table (e.g., PEN → BKP), then the most recent transaction-time state should be used. As BKP is a backlog, we employ the trick of CF-11.11 of using the last entry (or entries) in the backlog that is an insert or an update entry.

Code Fragment 11.13   (`PEN.FDYD_ID`, `PEN.BKP_ID`) **is a transaction-time current foreign key for** `BKP`**.**

```
ALTER TABLE PEN DROP FOREIGN KEY (FDYD_ID, BKP_ID)
    REFERENCES BKP

CREATE ASSERTION PEN__Current_Referential_Integrity
CHECK (NOT EXISTS (
        SELECT *
        FROM PEN AS P
        WHERE NOT EXISTS (
          SELECT *
          FROM BKP AS B
          WHERE P.FDYD_ID = B.FDYD_ID
            AND P.BKP_ID = B.BKP_ID
            AND (B.OPERATION = 'I' OR B.OPERATION = 'U')
            AND NOT EXISTS (
                    SELECT *
                    FROM BKP AS B2
                    WHERE B2.FDYD_ID = B.FDYD_ID
                      AND B2.BKP_ID = BKP._ID
                      AND B2.WHEN_CHANGED > B.WHEN_CHANGED)))
    )
```

For a valid-time table referencing a transaction-time table, a current/current constraint is appropriate. In the case of MASS_TRTMNT → BKP, the referencing table is an instant-stamped table, so we use a valid-time nonsequenced constraint, simply ignoring the valid time.

Code Fragment 11.14   (`MASS_TRTMNT.FDYD_ID`, `MASS_TRTMNT.BKP_ID`) **is a nonsequenced/current foreign key for** `BKP`**.**

```
ALTER TABLE MASS_TRTMNT DROP FOREIGN KEY (FDYD_ID, BKP_ID)
    REFERENCES BKP

CREATE ASSERTION MASS_TRTMNT__Curr_Curr_Referential_Integrity
CHECK (NOT EXISTS (
        SELECT *
        FROM MASS_TRTMNT AS M
        WHERE NOT EXISTS (
                SELECT *
                FROM BKP AS B
                WHERE M.FDYD_ID = B.FDYD_ID
                  AND M.BKP_ID = B.BKP_ID
```

```
                    AND (B.OPERATION = 'I' OR B.OPERATION = 'U')
                    AND NOT EXISTS (
                        SELECT *
                        FROM BKP AS B2
                        WHERE B2.FDYD_ID = B.FDYD_ID
                          AND B2.BKP_ID = B.BKP_ID
                          AND B2.WHEN_CHANGED > B.WHEN_CHANGED)))
            )
```

For foreign keys referencing bitemporal tables, if the referencing table supports transaction time, use a transaction-time current constraint. If the referencing table is instant-stamped in valid time, simply ignore the valid time.

For foreign keys involving bitemporal tables, we use an assertion to ensure that the foreign key is valid-time sequenced, on the current state of the two tables (page 329), with variations if one of the two tables doesn't have valid-time or transaction-time support, or if one or both of the tables is associated with an instantaneous entity or relationship type.

For this schema, we need to consider the referential integrity constraints in Table 11.1 that arise from the time-varying MOVE, LOCATION, and MASS_TRTMNT relationship types. However, only the LOT participating entity type has a recorded lifespan, so we are left with five foreign key constraints to consider further: LOT_CONTAINS → BKP, LOT_MOVE → LOT, LOT_LOC → LOT, MASS_TRTMNT → LOT, and LOT_CONTAINS → LOT.

To specify a sequenced/current foreign key constraint between two bitemporal tables, we augment the valid-time sequenced referential integrity assertion with a predicate selecting the currently valid information. For LOT, this translates to checking for a STOP_DATE of "forever." For LOT_LOC, which is a backlog containing after-images, we use the last entry trick.

Code Fragment 11.15   **(LOT_LOC.FDYD_ID, LOT_LOC.LOT_ID_NUM) is a sequenced/current foreign key for LOT.**

```
ALTER TABLE LOT_LOC DROP FOREIGN KEY (FDYD_ID, LOT_ID_NUM)
    REFERENCES LOT

CREATE ASSERTION LOT_LOC_Seq_Current_Referential_Integrity
CHECK (NOT EXISTS (
        SELECT *
        FROM LOT_LOC AS LL
          -- LL is one of the last entries
        WHERE (LL.OPERATION = 'I' OR LL.OPERATION = 'U')
          AND NOT EXISTS (
                SELECT *
                FROM LOT_LOC AS L3
```

```
                WHERE L3.FDYD_ID = LL.FDYD_ID
                  AND L3.LOT_ID_NUM = LL.LOT_ID_NUM
                  AND L3.PEN_ID = LL.PEN_ID
                  AND L3.WHEN_CHANGED > LL.WHEN_CHANGED)
          AND (NOT EXISTS (
                SELECT *
                FROM LOT AS L
                WHERE LL.FDYD_ID = L.FDYD_ID
                  AND LL.LOT_ID_NUM = L.LOT_ID_NUM
                  AND L.STOP_DATE = DATE '9999-12-31'
                  AND L.FROM_DATE <= LL.FROM_DATE
                  AND LL.FROM_DATE < L.TO_DATE)
          OR NOT EXISTS (
                SELECT *
                FROM LOT AS L
                WHERE LL.FDYD_ID = L.FDYD_ID
                  AND LL.LOT_ID_NUM = L.LOT_ID_NUM
                  AND L.STOP_DATE = DATE '9999-12-31'
                  AND L.FROM_DATE < LL.TO_DATE
                  AND LL.TO_DATE <= L.TO_DATE)
          OR EXISTS (SELECT *
                FROM LOT AS L
                WHERE LL.FDYD_ID = L.FDYD_ID
                  AND LL.LOT_ID_NUM = L.LOT_ID_NUM
                  AND L.STOP_DATE = DATE '9999-12-31'
                  AND LL.FROM_DATE < L.TO_DATE
                  AND L.TO_DATE < LL.TO_DATE
                  AND NOT EXISTS (
                      SELECT *
                      FROM LOT AS L2
                      WHERE L2.FDYD_ID = L.FDYD_ID
                        AND L2.LOT_ID_NUM = L.LOT_ID_NUM
                        AND L2.STOP_DATE = DATE '9999-12-31'
                        AND L2.FROM_DATE <= L.TO_DATE
                        AND L.TO_DATE < L2.TO_DATE)))
      ))
```

The LOT_CONTAINS table is a transaction-time table with a foreign key to LOT. For transaction time we need only consider the most recent additions to the table, so this can be implemented as a transaction-time current foreign key. Since LOT also records valid time, we utilize the current valid-time state.

Code Fragment 11.16 (LOT_CONTAINS.FDYD_ID, LOT_CONTAINS.LOT_ID_NUM) **is a current/ current foreign key for** LOT.

```
ALTER TABLE LOT_CONTAINS DROP FOREIGN KEY (FDYD_ID, LOT_ID_NUM)
    REFERENCES LOT

CREATE ASSERTION LOT_CONTAINS_Current_Current_Referential_Integrity
CHECK (NOT EXISTS (
    SELECT *
    FROM LOT_CONTAINS AS C
    WHERE (C.OPERATION = 'I' OR C.OPERATION = 'U')
      AND NOT EXISTS (
                      SELECT *
                      FROM LOT_CONTAINS AS C2
                      WHERE C2.FDYD_ID = C.FDYD_ID
                        AND C2.LOT_ID_NUM = C.LOT_ID_NUM
                        AND C2.WHEN_CHANGED > C.WHEN_CHANGED)
      AND NOT EXISTS (
                      SELECT *
                      FROM LOT AS L
                      WHERE C.FDYD_ID = L.FDYD_ID
                        AND C.LOT_ID_NUM = L.LOT_ID_NUM
                        AND L.STOP_DATE = DATE '9999-12-31'
                        AND L.FROM_DATE <= CURRENT_DATE
                        AND CURRENT_DATE < L.TO_DATE))
    )
```

For LOT_CONTAINS → BKP, both the referencing and referenced tables are transaction-time tables. This translates into a transaction-time current integrity constraint. As both tables are backlogs, we need to get the most recent entry from each.

Code Fragment 11.17 (LOT_CONTAINS.FDYD_ID, LOT_CONTAINS.BKP_ID) **is a transaction-time current foreign key for** BKP.

```
ALTER TABLE LOT_CONTAINS DROP FOREIGN KEY (FDYD_ID, BKP_ID)
    REFERENCES BKP

CREATE ASSERTION LOT_CONTAINS_Current_Referential_Integrity
CHECK (NOT EXISTS (
    SELECT *
    FROM LOT_CONTAINS AS C
    -- C is the last relevant entry
    WHERE (C.OPERATION = 'I' OR C.OPERATION = 'U')
      AND NOT EXISTS (
            SELECT *
```

```
                       FROM LOT_CONTAINS AS C2
                       WHERE C2.FDYD_ID = C.FDYD_ID
                         AND C2.LOT_ID_NUM = C.LOT_ID_NUM
                         AND C2.WHEN_CHANGED > C.WHEN_CHANGED)
                  -- There is not a match for C in BKP
                AND NOT EXISTS (
                       SELECT *
                       FROM BKP AS B
                       WHERE B.FDYD_ID = C.FDYD_ID
                         AND B.BKP_ID = C.BKP_ID
                         -- B is the last relevant entry
                         AND (B.OPERATION = 'I' OR B.OPERATION = 'U')
                         AND NOT EXISTS (
                                      SELECT *
                                      FROM BKP AS B2
                                      WHERE B2.FDYD_ID = B.FDYD_ID
                                        AND B2.BKP_ID = B.BKP_ID
                                        AND B2.WHEN_CHANGED > B.WHEN_CHANGED)))
          )
```

For a sequenced referential integrity constraint from an event table to a state table, each instant timestamp in the *referencing* table must be contained in a period timestamp of the referenced table.

Code Fragment 11.18   (`LOT_MOVE.FDYD_ID`, `LOT_MOVE.LOT_ID_NUM`) **is a sequenced/current foreign key for** `LOT`.

```
ALTER TABLE LOT_MOVE DROP FOREIGN KEY (FDYD_ID, LOT_ID_NUM)
     REFERENCES LOT

CREATE ASSERTION LOT_MOVE_Seq_Current_Referential_Integrity
CHECK (NOT EXISTS (
       SELECT *
       FROM LOT_MOVE AS M
       WHERE (M.OPERATION = 'I' OR M.OPERATION = 'U')
         AND NOT EXISTS (
                SELECT *
                FROM LOT_MOVE AS M2
                WHERE M2.FDYD_ID = M.FDYD_ID
                  AND M2.LOT_ID_NUM = M.LOT_ID_NUM
                  AND M2.FROM_PEN_ID = M.FROM_PEN_ID
                  AND M2.TO_PEN_ID = M.TO_PEN_ID
                  AND M2.WHEN_CHANGED > M.WHEN_CHANGED)
```

```
            AND NOT EXISTS (
                            SELECT *
                            FROM LOT AS L
                            WHERE M.FDYD_ID = L.FDYD_ID
                              AND M.LOT_ID_NUM = L.LOT_ID_NUM
                              AND L.STOP_DATE = DATE '9999-12-31'
                              AND L.FROM_DATE <= M.AT_DATE
                              AND M.AT_DATE < L.TO_DATE))
        )
```

For foreign keys over bitemporal tables, use sequenced/current constraints.

This is shorter than CF-11.15 because for each row of LOT_MOVE, there need be only one row of LOT whose period of validity contains the (instant) lifespan. In contrast, for each row of LOT_LOC, many rows from LOT may coordinate to contain LOT_LOC's period of validity. In general, for foreign keys over bitemporal tables, use sequenced/current constraints, unless one of the participating tables doesn't include valid- or transaction-time support or is associated with an instantaneous entity or relationship type, in which case the assertion is simplified.

The final foreign key to consider, MASS_TRTMNT → LOT, is even simpler because MASS_TRTMNT does not record transaction time.

Code Fragment 11.19   (MASS_TRTMNT.FDYD_ID, MASS_TRTMNT.LOT_ID_NUM) **is a valid-time sequenced foreign key for** LOT**.**

```
ALTER TABLE MASS_TRTMNT DROP FOREIGN KEY (FDYD_ID, LOT_ID_NUM)
    REFERENCES LOT

CREATE ASSERTION MASS_TRTMNT_Seq_Referential_Integrity
CHECK (NOT EXISTS (
          SELECT *
          FROM MASS_TRTMNT AS M
          WHERE NOT EXISTS (
                    SELECT *
                    FROM LOT AS L
                    WHERE M.FDYD_ID = L.FDYD_ID
                      AND M.LOT_ID_NUM = L.LOT_ID_NUM
                      AND L.STOP_DATE = DATE '9999-12-31'
                      AND L.FROM_DATE <= M.AT_DATE
                      AND M.AT_DATE < L.TO_DATE))
      )
```

We have now considered the implications of time-varying tables on all the foreign key constraints.

Uniqueness Constraints

The temporal analog of UNIQUE is sequenced uniqueness, which requires a table constraint (page 124). Other forms (current, value-equivalent, nonsequenced) are less useful, as they are more representational than semantic (page 139).

FDYD_SHORT_NAME is unique. Since FDYD does not record valid time, the original UNIQUE statement may be retained.

Concerning the LOT table, (FDYD_ID, LOT_ID_NUM, LOT_ID) is also unique. Since this relation is bitemporal, this corresponds to sequenced/current uniqueness, which requires an assertion.

Code Fragment 11.20　(LOT.FDYD_ID, LOT.LOT_ID) **is sequenced/current unique.**

```
ALTER TABLE LOT DROP UNIQUE (FDYD_ID, LOT_ID)

ALTER TABLE LOT ADD ASSERTION LOT_ID_Seq_Curr_UNIQUE
CHECK (NOT EXISTS ( SELECT *
          FROM LOT AS L1
          WHERE L1.STOP_DATE = DATE '9999-12-31'
            AND 1 < (SELECT COUNT(*)
                     FROM LOT AS L2
                     WHERE L1.FDYD_ID = L2.FDYD_ID
                     WHERE L1.LOT_ID = L2.LOT_ID
                     WHERE L1.LOT_ID_NUM = L2.LOT_ID_NUM
                       AND L2.STOP_DATE = DATE '9999-12-31'
                       AND L1.FROM_DATE < L2.TO_DATE
                       AND L2.FROM_DATE < L1.TO_DATE))
      )
```

11.4　PHYSICAL DESIGN

There is a sharp line between logical and physical design: logical design concerns preserving the semantics of the application as expressed in the ER schema, and physical design concerns ensuring efficient execution of application queries and modifications. Physical design should properly be done after logical design, so that efficiency considerations do not muddy the semantics.

Conventional physical design involves specifying indexes and storage structures, and perhaps decomposing tables or merging tables. That some tables are time-varying adds temporal partitioning to this phase of the design.

Tables with a valid-time extent can be temporally partitioned into a current store containing only current information and a history store storing data that became invalid before "now" (page 206).

Crystal Clocks

The longer the pendulum, or the longer the coiled spring, the more accurate the clock. However, physical constraints bound the achievable precision. The next advance exploited the "piezoelectric effect" of a quartz crystal, which vibrates when an electronic current of the correct frequency is applied to it. A quartz crystal can thereby replace the pendulum or hairspring as the resonator with a battery and simple electronic circuit providing the energy. Although a quartz oscillator will drift with temperature and with age, it is accurate to *tenths* of a second a day, due to its frequency of 32,768 Hz versus 360 Hz in an Accutron. A crystal clock is also much easier to manufacture and, hence, has sounded the death knell to mechanical watches.

While there are still mechanical watches being manufactured, the attraction is more nostalgic than economic or desire for quality. *Consumer Reports* did a small experiment, comparing a $3 kid's quartz watch against a $1200 Rolex Oyster Perpetual, a superb mechanical watch. Over 6 days, the kid's watch lost 11 seconds, and the Rolex gained 22 seconds [41]. Mechanical clocks, and perhaps even analog displays, will soon be museum curiosities, right next to slide rules and mercury thermometers.

The only valid-time table in this schema is MASS_TRTMNT, which is associated with an instantaneous relationship, and thus cannot be temporally partitioned.

A transaction-time state table may be temporally partitioned into a current store, an archival store, and possibly the monitored table itself (page 264). Sometimes the monitored table is best defined as a view, generally on the current store; often the state table itself is defined as a view (page 268). If a full backlog is used, the monitored table can be defined as a view on the tracking log (page 235).

The only transaction-time tables are BKP and LOT_CONTAINS, which we retain as audit logs.

A bitemporal state table can be temporally partitioned into a current store (current/current), a history store (current in transaction time), and an archival store (not current in transaction time) (page 329). A current store improves the performance of current/current queries, but is awkward to maintain (page 331). A history store will also result in good current/current retrieval performance and is much easier to maintain (page 332). To reduce the space requirements of the archival store, it may be advantageous to store just one transaction timestamp (page 334).

We have three bitemporal tables: LOT (a valid-time state/transaction-time state table), LOT_LOC (a valid-time state/audit log table), and LOT_MOVE (a valid-time event/ audit log table). Since most of the queries will be over the valid-time history of cattle coresiding in pens, an appropriate partitioning of the LOT table is a valid-time state/ transaction-time current history store and a valid-time state/transaction-time state archive, since disk space is not at a premium. We'll call the history table LOT and create a new archive table, LOT_Archive.

Code Fragment 11.21   **Partition** LOT **into a history store and an archival store.**

```
ALTER TABLE LOT DROP COLUMN STOP_DATE

CREATE TABLE LOT_Archive (FDYD_ID, LOT_ID_NUM, LOT_ID,
    GNDR_CODE, PROJ_CLOSEOUT, IN_WEIGHT, VALID, OWNER,
    COMMENT, FROM_DATE, TO_DATE, START_DATE, STOP_DATE,
  PRIMARY KEY (FDYD_ID, LOT_ID_NUM, FROM_DATE, STOP_DATE)
)

CREATE ASSERTION LOT_Archive_seq_seq_primary_key
CHECK (NOT EXISTS ( SELECT *
            FROM LOT_Archive AS L1
            WHERE L1.STOP_DATE = DATE '9999-12-31'
              AND 1 < (SELECT COUNT(*)
                      FROM LOT_Archive AS L2
                      WHERE L1.FDYD_ID = L2.FDYD_ID
                        AND L1.LOT_ID_NUM = L2.LOT_ID_NUM
                        AND L1.FROM_DATE < L2.TO_DATE
                        AND L2.FROM_DATE < L1.TO_DATE
                        AND L2.STOP_DATE = DATE '9999-12-31'))
      )
```

Foreign key assertions that reference LOT (CF-11.15, CF-11.16, CF-11.18, CF-11.19) substantially remain as they are, except that the WHERE condition

```
LOT.STOP_DATE = DATE '9999-12-31'
```

is no longer needed.

Since LOT_LOC and LOT_MOVE both have instant transaction timestamps, partitioning either into a history store and an archival audit log is possible, but would not have a dramatic impact on integrity constraints or queries.

11.5   ADVANCED DESIGN ASPECTS*

The time-varying nature adds a few further wrinkles that can be expressed as temporal annotations to the conceptual schema and mapped to changes to SQL tables.

11.5.1   Additional Temporal Annotations

We first list additional annotations that might be useful to include with the conceptual schema.

Time-Invariant Keys

In the nontemporal ER schema, the key of an entity type is assumed to identify an entity at each point in time. We may want to express a stronger integrity constraint, such as that the (Feedyard_ID, Pen_ID) value identifies a particular pen *over all time*. This is termed a *time-invariant key*. As all the keys in this schema are nontemporal, they are automatically valid time-invariant.

Time-Invariant Uniqueness Constraints

In the nontemporal ER schema, the integrity constraints on individual attributes are assumed to hold at each point in time. We may want to express a stronger integrity constraint, such as that the Fdyd_Short_name for a particular feed yard is unique *over all time*. This is termed a *time-invariant uniqueness constraint*, as it applies over the entire lifespan of the entity.

As another example, the LOT.Lot_ID attribute is also time-invariant unique. Lots enter the feed yard with a Lot_ID value. Once that lot has left the feedyard, a subsequently arriving lot could be assigned the same Lot_ID value; Lot_IDs are reused. So Brad generates a nonreusable Lot_ID_Num attribute to uniquely identify the lot; this attribute is the partial key for the LOT entity type.

While Lot_IDs may be reused, for any given lot entity, the Lot_ID is unique over all time, termed *time-invariant unique*.

Such constraints are not applicable to attributes associated with instantaneous entity and relationship types.

Time-Invariant Participation Constraints

This notion of a time-invariant constraint also applies to participation constraints. Conventional participation constraints are assumed to hold at any point in time. So, for example, each in_pen relationship denotes a pen located in at most one (actually, exactly one) feed yard at any point in time, and each location relationship denotes cattle from one lot residing in one or more pens, at any point in time.

We may want to express a stronger participation constraint, such as that each pen is located in exactly one feed yard over all time. If a particular pen is located in a particular feed yard at one point in time, that pen will be located in that same feed yard at all other points of time during the lifespans of the pen and feed yard. This is termed a *time-invariant participation constraint*. Such constraints are not applicable to instantaneous relationship types.

Of the relationships in this schema, the LOCATION relationship type has a many-to-many participation constraint applied over its valid-time extent, so this participation constraint is time-invariant. IN_PEN, IN_LOT, IS_DESCRIBED_BY, CREATES, CONTAINS, DESCRIBES_PEN, and

Some key, uniqueness, and participation constraints may hold over the entire lifespan (or valid time) of the associated entity (or relationship) type, and are thus designated as time-invariant.

DESCRIBES_LOT are nontemporal relationship types, and so their nontemporal participation constraints are also valid across their lifespan. The MOVE and MASS_TRTMNT relationship types model instantaneous events.

Transaction Time-Invariant Constraints

As with valid time, we also consider whether the integrity constraints hold over the entire transaction-time period.

Two of the entity types, LOT and BACKUP, have transaction-time support, so we must consider whether we wish to record the values of their key attributes as they vary.

The (partial) key of the LOT entity, Lot_ID_Num, is generated when the data is loaded. As such, it normally doesn't change, unless some data is later discovered to be dirty and subsequently corrected. Hence, the Lot_ID_Num can possibly vary over transaction time.

The partial key for BACKUP, BKP_ID, is generated when the backup is taken, and so is invariant in transaction time.

No attribute listed in Table 11.5 is transaction time-invariant unique. While DESCRIBES_PEN and DESCRIBES_LOT are many-to-one relationship types, they are both transaction time-invariant many-to-*many* because a particular pen or lot entity can be described by several backup entities at different transaction times. Similarly, the LOCATION, MOVE, and MASS_TRTMNT relationship types all have transaction time-invariant participation constraints of many-to-many.

> Also consider whether the key, uniqueness, and participation constraints hold over the entire transaction-time extent.

Temporal Specialization

Temporal specialization indicates whether the valid timestamp (or lifespan) and transaction timestamp of an entity, relationship, or attribute are coupled. One entity type, LOT, is bitemporal; two relationship types, MOVE and LOCATION, are also bitemporal.

LOCATION is a *fully general* bitemporal relationship type (also termed *nonspecialized*). A transaction changing the database—specifically, the location of a lot of cattle in a pen—can mention a valid time in the past, if some information is being corrected, or in the future, if moves in the future are anticipated. There is no a priori coupling between valid and transaction time for this relationship.

> Classify each bitemporal entity and relationship type as fully general, retroactive, degenerate, or postactive.

MOVE is a *retroactive* relationship type, denoting that the valid time is always *before* (or equal to) the transaction time. Move records come directly from the feed yard FoxPro database, and thus concern the past; no updates are possible to a particular move relationship once it has been recorded.

A *degenerate* entity type denotes that the beginning of the entity's lifespan exactly corresponds to the beginning of the transaction time. Here, any change in the modeled reality is immediately recorded in the database, so that the valid and transaction times exactly correspond.

A final temporal specialization is a *postactive* entity or relationship type, in which the valid time is always *after* (or equal to) the transaction time: modifications all concern the future, recording something that will later be true.

11.5.2   Applying Temporal Annotations

We map these annotations to SQL constructs.

Time-Invariant Keys

If the key is valid time-invariant, a surrogate identifier column, as discussed on page 364, is not needed, as for example with the `LOT` table.

A transaction time-invariant sequenced primary key requires an assertion stating that the periods associated with any particular key value are contiguous.

To ensure that the key is transaction time-invariant, we add an assertion stating that the periods associated with any particular key value are contiguous, which is a nonsequenced constraint (page 129).

That the primary key for `BKP` is transaction time-invariant requires checking for contiguity, which fortunately is straightforward on a backlog: the most recent entry cannot be an insert entry that followed a previous delete entry.

Code Fragment 11.22   `BKP`**'s primary key is contiguous.**

```
CREATE ASSERTION BKP_Contiguous_History
CHECK (NOT EXISTS (
    SELECT *
    FROM BKP AS B, BKP AS B2
    WHERE B.WHEN_CHANGED < B2.WHEN_CHANGED
      AND B.FDYD_ID = B2.FDYD_ID AND B.BKP_ID = B2.BKP_ID
      AND B.OPERATION = 'D' AND B2.OPERATION = 'I'
        -- I is the last operation
      AND NOT EXISTS (
          SELECT *
          FROM BKP AS B3
          WHERE B3.FDYD_ID = B.FDYD_ID AND B3.BKP_ID = B.BKP_ID
            AND B2.WHEN_CHANGED < B3.WHEN_CHANGED)
      -- There are no operations between the delete and the insert
      AND NOT EXISTS (
          SELECT *
          FROM BKP AS B3
          WHERE B3.FDYD_ID = B.FDYD_ID AND B3.BKP_ID = B.BKP_ID
```

Table 11.6 An excerpt of the LOT table.

...	LOT_ID	...	FROM_DATE	TO_DATE
...	17	...	1998-01-01	1998-03-23
...	19	...	1998-03-23	1998-04-01
...	19	...	1998-05-12	9999-12-31

```
            AND B.WHEN_CHANGED < B3.WHEN_CHANGED
            AND B3.WHEN_CHANGED < B2.WHEN_CHANGED))
    )
```

Time-Invariant Uniqueness Constraints

Time-invariant uniqueness constraints can be specified only if there is a time-invariant primary key available. Consider the excerpt of the LOT table shown in Table 11.6. Is LOT_ID time-invariant unique, that is, having only one value over all time for a particular LOT entity?

It is impossible to tell whether the appearance of LOT_ID values of 17 and 19 are problematic without knowing which rows are associated with which lots. So, in Table 11.7, we focus on the primary key of the LOT table, (FDYD_ID, LOT_ID_NUM), which has already been specified as a time-invariant primary key.

Time-invariant uniqueness requires an assertion that the column(s) are unique with respect to the time-invariant primary key.

We can now see that this instance violates the time-invariant uniqueness constraint for LOT_ID, as a single lot, with a LOT_ID_NUM of 101, has two distinct values for the LOT_ID.

For (FDYD_ID, LOT_ID) to be time-invariant unique, it must be unique with respect to the time-invariant primary key, (FDYD_ID, LOT_ID_NUM). So we replace the assertion in CF-11.20 with the following assertion.

Code Fragment 11.23   **(LOT.FDYD_ID, LOT.LOT_ID) is valid time-invariant unique.**

```
ALTER TABLE LOT DROP ASSERTION LOT_ID_Seq_Curr_UNIQUE

ALTER TABLE LOT ADD ASSERTION LOT_ID_VT_Invariant_UNIQUE
CHECK (NOT EXISTS ( SELECT *
        FROM LOT AS L1
        WHERE 1 < (SELECT DISTINCT COUNT(LOT_ID)
                FROM LOT AS L2
                WHERE L1.FDYD_ID = L2.FDYD_ID
                  AND L1.LOT_ID_NUM = L2.LOT_ID_NUM
                  AND L2.STOP_DATE = DATE '9999-12-31'))
    )
```

Table 11.7 Focusing on the primary key.

FDYD_ID	LOT_ID_NUM	LOT_ID	...	FROM_DATE	TO_DATE
1	101	17	...	1998-01-01	1998-03-23
1	101	19	...	1998-03-23	1998-04-01
1	799	19	...	1998-05-12	9999-12-31

Temporal Specialization

The valid-time start column may be dropped from a table corresponding to a degenerate entity type.

A degenerate entity type allows us to simplify the table(s) of that type. Since the beginning of the entity's lifespan (FROM_DATE) exactly corresponds to the beginning of transaction time (START_DATE), we can omit the latter column and replace all mention of it with the former column.

No tables in this schema are degenerate.

11.6 BENEFITS

This case study provides empirical evidence of the efficacy of the design methodology followed in this chapter. Brad spent several months designing an initial logical schema for the 55-odd tables of his applications. He later then worked with the author to design tables following this methodology. The result is a set of two logical schemas drawn from the same requirements, one developed using the traditional approach, in which time is considered from the beginning, and the schema developed here, in which time is considered only in the second stage of conceptual and logical design.

Brad feels that the latter design is preferable in several ways. In particular, when the methodology is followed and temporal aspects are added later, rather than in the initial ER diagram, the following benefits specific to this case study accrue:

- The ER diagram was simplified.
- The semantics of LOT_LOC was cleaned up considerably.
- The valid-time semantics of LOT was highlighted.

Atomic Clocks

The cesium atom has a natural vibration at 9,192,631,770 Hz. If the driving frequency is just a little off of that natural frequency of the atom, it doesn't resonate. Laboratory cesium oscillators keep time to about one second in *370,000 years*, or, alternatively, to about 3 microseconds per year.

Satellites in geosynchronous orbit contain atomic clocks and send out timing signals from them. Small GPS (global positioning system) receivers use triangulation and the speed of light to calculate highly accurate location data; such calculations depends heavily on a highly accurate clock.

- We discovered during this analysis that `LOT_LOC` has a transaction-time component.
- The transaction-time semantics of `BKP` was emphasized.
- Several of the integrity constraints were corrected.
- Some of the nullable columns were rendered not nullable.
- Twenty-five columns were removed. One table and sixteen columns were added, thereby somewhat simplifying the logical model.
- Queries are easier to express on the revised logical model.

Patience, in considering time later, is indeed a virtue.

11.7 APPLICATION DEVELOPMENT

We now briefly revisit the queries and modifications mentioned in Chapter 2, as a review of the approaches discussed at length in the other case studies.

11.7.1 Queries

We can express any nontemporal query in the three variants, current, sequenced, and nonsequenced. Note the correspondence between these variants.

Current queries are the temporal analog of nontemporal queries. They require a simple addition to the WHERE clause (page 143).

Code Fragment 11.24   **How many head of cattle from lot 219 in yard 1 are (currently) in each pen?**

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

Sequenced queries, which are the "and when" analog of nontemporal queries, must be broken down into their algebraic equivalents to be translated into SQL. Selection, projection, sorting, and union are simple (page 145).

Code Fragment 11.25   **Give the history of how many head of cattle from lot 219 in yard 1 were in each pen.**

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

Nonsequenced queries, which treat the timestamps as regular columns, are generally difficult to express in English, but relatively straightforward to express in SQL on state tables.

Code Fragment 11.26   **How many head of cattle from lot 219 in yard 1 were, at some time, in each pen?**

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

We now turn to temporal joins. Consider the following nontemporal join query. The query involves a self-join on the table, along with projection and selection. The first predicate ensures that we don't get identical pairs; the second and third predicates test for coresidency.

Code Fragment 11.27   **Which lots are coresident in a pen (nontemporal version)?**

```
SELECT DISTINCT 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 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.

Code Fragment 11.28   **Which lots are currently coresident in a pen?**

```
SELECT DISTINCT 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'
```

As before, nonsequenced joins are easy to specify: just ignore the timestamp columns.

Code Fragment 11.29   **Which lots were in the same pen, perhaps at different times?**

```
SELECT DISTINCT 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
```

A sequenced join is challenging to express in SQL (page 151). 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.

Code Fragment 11.30   **Give the history of lots being coresident in a pen.**

```
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
```

The SQL-92 CASE expression allows this query to be written as a single SELECT statement (page 152).

Code Fragment 11.31   **Sequenced temporal join using CASE.**

```
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)
```

Reconstructing the monitored table at a point in time is a simple query or view on a transaction-time state table (compare with CF-9.7).

Code Fragment 11.32   **Provide the state of the** `LOT_CONTAINS` **table on January 12, 1998.**

```
SELECT LOT_ID_NUM, BKP_ID, A_NAME, DBF_NAME, DBF_UPDATE_RECNO
FROM LOT_CONTAINS
WHERE START_TIME <= DATE '1998-01-12'
  AND DATE '1998-01-12' < STOP_DATE
```

We end with two queries on the bitemporal table `LOT`, one a sequenced/nonsequenced query and one a nonsequenced/nonsequenced query.

Code Fragment 11.33   **Provide the history as best known on March 15, 1998.**

```
SELECT LOT_ID_NUM, GNDR_CODE, PROJ_CLOSEOUT, IN_WEIGHT,
    VALID, OWNER, COMMENT
FROM LOT
WHERE START_TIME <= DATE '1998-03-15'
  AND DATE '1998-03-15' < STOP_DATE
```

Code Fragment 11.34   **When were steerings scheduled (as opposed to being recorded after the fact)?**

```
SELECT S.LOT_ID_NUM, S.FROM_DATE AS When_Scheduled,
    S.START_DATE AS When_Recorded
FROM LOT AS C, LOT AS S
WHERE C.FDYD_ID = S.FDYD_ID
  AND C.LOT_ID_NUM = S.LOT_ID_NUM
  AND C.GNDR_CODE = 'c' AND S.GNDR_CODE = 's'
```

```
AND C.TO_DATE = S.FROM_DATE
AND S.START_DATE < S.FROM_DATE
```

11.7.2   Modifications

When one or more of the tables managed by a legacy application is rendered temporal, all of the modifications must be converted to current modifications, whose period of applicability is "now" to "forever" (page 216).

Modifications on bitemporal state tables are written in two stages. First, the modification is transformed according to the valid-time semantics. Second, the resulting SQL statements are further transformed according to the transaction-time semantics (page 286).

Current insertions are easy to code in SQL; the second transformation merely requires that the transaction timestamp be included. We illustrate such an insertion on the LOT table. Recall that this table is temporally partitioned into a history store (see CF-11.21), containing those rows with a transaction-stop time of "forever," and an archival store, containing those rows with a transaction-stop time before "now." We need not record a transaction-stop time in the LOT table because it is assumed to be "forever."

Code Fragment 11.35   **Lot 433 arrives today.**

```
INSERT INTO LOT
VALUES (1, 433, 7, 'h', DATE '1998-12-15', 14533, 1, 'Empire', 'B', 2,
  CURRENT_DATE, DATE '9999-12-31', CURRENT_DATE)
```

A logical current deletion in the general scenario (page 183) is implemented as a physical update and a physical delete. But because LOT has transaction-time support, the update and delete cause the previous values to be retained in LOT_Archive. The update is transformed into moving the old value to the archival store, thus retaining the new value in the history store. The delete is transformed into moving the row to be deleted to the archival store.

Code Fragment 11.36   **Lot 101 leaves the feed yard.**

```
-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 101
```

```
      AND TO_DATE >= CURRENT_DATE
      AND FROM_DATE < CURRENT_DATE

UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 101
  AND TO_DATE >= CURRENT_DATE
  AND FROM_DATE < CURRENT_DATE

-- transformed deletion
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 101
  AND FROM_DATE > CURRENT_DATE

DELETE FROM LOT
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 101
  AND FROM_DATE > CURRENT_DATE
```

These two pairs of statements can be done in either order, as the rows they alter are disjoint, but the insertion into the archival store should occur before the second statement of the pair.

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. Compare with CF-7.11; again, we transform a physical update into an update of the history store and an insertion into the archival store, here, twice.

Code Fragment 11.37   **The cattle in lot 799 are being steered today.**

```
-- transformed insertion
INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, 's', PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    CURRENT_DATE, DATE '9999-123-31', CURRENT_DATE
FROM LOT
WHERE FDYD_ID = 1
```

```
  AND LOT_ID_NUM = 799
  AND FROM_DATE <= CURRENT_DATE
  AND TO_DATE > CURRENT_DATE

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 799
  AND GNDR_CODE <> 's'
  AND FROM_DATE < CURRENT_DATE
  AND TO_DATE > CURRENT_DATE

UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 799
  AND GNDR_CODE <> 's'
  AND FROM_DATE < CURRENT_DATE
  AND TO_DATE > CURRENT_DATE

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 799
  AND FROM_DATE >= CURRENT_DATE

UPDATE LOT
SET GNDR_CODE = 's'
WHERE FDYD_ID = 1
  AND LOT_ID_NUM = 799
  AND FROM_DATE >= CURRENT_DATE
```

The last pair can appear anywhere, but the second and third statements must occur after the insertion. This store grows monotonically.

   Note that in a temporally partitioned store, only insertions are applied to the archival store.

A current modification applies from "now" to "forever." A sequenced modi-
fication 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 CUR-
RENT_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
(page 188).

Code Fragment 11.38   **Lot 426, a collection of heifers, was on the feed yard from March
26 to April 14.**

```
INSERT INTO LOT
VALUES (1, 426, 7, 'h', DATE '1998-12-15', 14533, 1, 'Empire', ' ',
  DATE '1998-03-26', DATE '1998-04-14', CURRENT_DATE)
```

Sequenced deletions (page 190) require a little more work because the period of
applicability of the deletion may end before the row ends. Recall that a current dele-
tion in the general scenario is implemented as an update for those currently valid
rows, and a delete for periods starting in the future. A sequenced deletion requires
seven physical modifications. In the following deletion, the period of applicability
is DATE '1998-10-01' to DATE '1998-10-22'.

Code Fragment 11.39   **Lot 234 will be absent from the feed yard for the first three weeks
of October, when the steering will take place (applied on a valid-
time version of LOT).**

```
-- transformed insertion
INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    DATE '1998-10-22', TO_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE <= DATE '1998-10-01'
  AND TO_DATE > DATE '1998-10-22'

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-01'
  AND TO_DATE >= DATE '1998-10-01'
```

```
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'

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-22'
  AND TO_DATE >= DATE '1998-10-22'

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'

-- transformed deletion
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE >= DATE '1998-10-01'
  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'
```

Updates are equivalent to a current deletion followed by a current insertion, and so can be implemented that way, taking care to correlate these two modifications (pages 186 and 194). The first transformation of a sequenced update yields two insertions and three updates; the second transformation explodes this into eight statements. In the following sequenced update, the period of applicability is DATE '1998-03-01' to DATE '1998-04-01'.

Code Fragment 11.40   **The lot was steered only for the month of March.**

```
-- transformed insertion
INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, DATE '1998-03-01', CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-03-01'
  AND TO_DATE > DATE '1998-03-01'

-- transformed insertion
INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE,  PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    DATE '1998-04-01', TO_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-04-01'
  AND TO_DATE > DATE '1998-04-01'

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-04-01'
  AND TO_DATE > DATE '1998-03-01'

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'

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 799
```

```
    AND FROM_DATE < DATE '1998-03-01'
    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'

-- transformed update
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_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 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'
```

Nonsequenced modifications (page 197) are rare. They are generally easy to implement in SQL (since they are representational in nature). As with constraints and queries, a nonsequenced modification treats the timestamps identically to the other columns. A (nonsequenced/current) deletion turns into a pair of statements that move the row to the archival store.

Code Fragment 11.41   **Delete the records of lot 234 that have duration greater than three months.**

```
-- transformed deletion
INSERT INTO LOT_Archive
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, TO_DATE, START_DATE, CURRENT_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
  AND (TO_DATE - FROM_DATE MONTH) > INTERVAL '3' MONTH

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

The current and sequenced deletions 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.

The following is a current update on a transaction-time table implemented as a backlog (compare with CF-9.6). All modifications on backlogs are implemented as insertions, with the operation code indicating the type of modification.

Code Fragment 11.42   **Correct the backup identifier for lot 433 to 37.**

```
-- transformed insertion
INSERT INTO LOT_CONTAINS (FDYD_ID, LOT_ID_NUM, BKP_ID, A_NAME,
    DBF_NAME, DBF_UPDATE_RECNO, WHEN_CHANGED, OPERATION)
SELECT FDYD_ID, LOT_ID, 37, A_NAME, DBF_NAME,
    DBF_UPDATE_RECNO, CURRENT_DATE, 'I'
FROM LOT_CONTAINS AS L
WHERE LOT_ID_NUM = 433
  AND (OPERATION = 'I' OR OPERATION = 'U')
  AND NOT EXISTS (SELECT *
        FROM LOT_CONTAINS AS L2
        WHERE L.FDYD_ID = L2.FDYD_ID
          AND L.LOT_ID_NUM = L2.LOT_ID_NUM
          AND L.WHEN_CHANGED < L2.WHEN_CHANGED)

-- transformed deletion
INSERT INTO LOT_CONTAINS (FDYD_ID, LOT_ID_NUM, BKP_ID, A_NAME,
    DBF_NAME, DBF_UPDATE_RECNO, WHEN_CHANGED, OPERATION)
SELECT FDYD_ID, LOT_ID, BKP_ID, A_NAME, DBF_NAME,
    DBF_UPDATE_RECNO, CURRENT_DATE, 'D'
FROM LOT_CONTAINS AS L
WHERE LOT_ID_NUM = 433
  AND BKP_ID <> 37
  AND (OPERATION = 'I' OR OPERATION = 'U')
  AND NOT EXISTS (SELECT *
        FROM LOT_CONTAINS AS L2
        WHERE L.FDYD_ID = L2.FDYD_ID
          AND L.LOT_ID_NUM = L2.LOT_ID_NUM
          AND L.BKP_ID = L2.BKP_ID
          AND L.WHEN_CHANGED < L2.WHEN_CHANGED)
```

We finish with a sequenced/current deletion on a bitemporal table (compare with CF-10.15).

Code Fragment 11.43 **Lot 234 will be absent from the feed yard for the first three weeks of October, when the steering will take place (applied on the bitemporal version of** LOT**).**

```
INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    DATE '1998-10-22', TO_DATE, CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-01'
  AND TO_DATE > DATE '1998-10-22'
  AND STOP_DATE = DATE '9999-12-31'

INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    FROM_DATE, DATE '1998-10-01', CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-01'
  AND TO_DATE > DATE '1998-10-01'
  AND STOP_DATE = DATE '9999-12-31'

UPDATE LOT
SET STOP_DATE = CURRENT_TIMESTAMP
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-01'
  AND TO_DATE > DATE '1998-10-01'
  AND STOP_DATE = DATE '9999-12-31'

INSERT INTO LOT
SELECT FDYD_ID, LOT_ID_NUM, LOT_ID, GNDR_CODE, PROJ_CLOSEOUT,
    IN_WEIGHT, VALID, OWNER, COMMENT,
    DATE '1998-10-22', TO_DATE, CURRENT_TIMESTAMP, DATE '9999-12-31'
FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-22'
  AND TO_DATE >= DATE '1998-10-22'
  AND STOP_DATE = DATE '9999-12-31'

UPDATE LOT
SET STOP_DATE = CURRENT_TIMESTAMP
```

```
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-22'
  AND TO_DATE >= DATE '1998-10-22'
  AND STOP_DATE = DATE '9999-12-31'

UPDATE LOT
SET STOP_DATE = CURRENT_TIMESTAMP
WHERE LOT_ID_NUM = 234
  AND FROM_DATE >= DATE '1998-10-01'
  AND TO_DATE <= DATE '1998-10-22'
  AND STOP_DATE = DATE '9999-12-31'
```

11.8   IMPLEMENTATION CONSIDERATIONS

The CD-ROM contains all the code fragments in this chapter in Oracle8 Server. As in previous chapters, assertions must be implemented as triggers.

As an example, CF-11.17 can be implemented by the following Oracle8 Server trigger.

Code Fragment 11.44   (`LOT_CONTAINS.FDYD_ID`, `LOT_CONTAINS.BKP_ID`) **is a transaction-time current foreign key for** `BKP`, **in Oracle8.**

```
CREATE TRIGGER LOT_CONTAINS_Cur_Ref_Integrity
BEFORE INSERT OR DELETE OR UPDATE ON LOT
DECLARE
  valid INTEGER;
BEGIN
  SELECT 1
  INTO valid
  FROM DUAL
  WHERE NOT EXISTS (
    SELECT *
    FROM LOT_CONTAINS C
    -- C is the last relevant entry
    WHERE (C.OPERATION = 'I' OR C.OPERATION = 'U')
      AND NOT EXISTS (
        SELECT *
        FROM LOT_CONTAINS C2
        WHERE C2.FDYD_ID = C.FDYD_ID
          AND C2.LOT_ID_NUM = C.LOT_ID_NUM
          AND C2.WHEN_CHANGED > C.WHEN_CHANGED )
          -- There is not a match for C in BKP
```

```
            AND NOT EXISTS (
              SELECT *
              FROM BKP B
              WHERE B.FDYD_ID = C.FDYD_ID
                AND B.BKP_ID = C.BKP_ID
                -- B is the last relevant entry
                AND (B.OPERATION = 'I' OR B.OPERATION = 'U')
                AND NOT EXISTS (
                  SELECT *
                  FROM BKP B2
                  WHERE B2.FDYD_ID = B.FDYD_ID
                  AND B2.BKP_ID = B.BKP_ID
                  AND B2.WHEN_CHANGED > B.WHEN_CHANGED ) ) );

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR( -20007,
        'FDYD_ID and BKP_ID violate current/current foreign key' );
END;
```

11.9 SUMMARY

We followed a five-step methodology in designing the feed yard application: (1) perform conceptual design ignoring time, yielding a conventional ER schema, (2) add temporal annotations in prose, (3) map the conventional ER schema into a logical SQL schema, (4) apply the temporal annotations, modifying the logical schema along the way, and (5) finish with physical design, including temporal partitioning. This approach moves consideration of temporal aspects from early in the design process to much, much later. The methology includes a systematic evaluation of the temporal aspects of each modeling construct, thereby breaking down the design task into answering (many) brief questions about individual constructs. Brad characterizes the process as "effecting Descartes's reductionism without losing sight of the whole." More pragmatically, initially ignoring time results in less convoluted ER schemas, fewer errors introduced during logical design, and a deeper understanding of the application.

11.10 READINGS

This case study was discussed in two articles in the August and September 1998 issues of *Database: Programming and Design*; the series has been collected in a tech-

nical report [88]. These articles focused on sequenced queries and on modifications to the gender attribute.

Heidi Gregersen and Christian S. Jensen have surveyed the many temporal entity-relationship models that have been proposed over the previous two decades [38]. Their model, the Time Extended EER Model (TIMEER [37]), is particularly elegant; I borrowed heavily from their characterization in my presentation in Section 11.2.2. TIMEER goes further, by proposing iconic indicators of the temporal aspects specified in this chapter in prose. Their approach to mapping TimeER schemas to the relational model [39] is the basis for the steps listed in Section 11.3.

Christian S. Jensen and the author have written a comprehensive treatise on temporal logical design [55], as well as a more accessible example of these strategies [57], which informed the presentation of Section 11.3.2.

The term *lifespan* was introduced by Manfred Klopprogge [62], *time-invariant keys* (TIKs) were first mentioned by Shamkant Navathe and Rafi Ahmed [74], and the concept of temporal specialization was advanced by Christian S. Jensen and the author [54].

David Landes discusses the alternative divisions *puncta* and *ostenta* [65]. He also summarizes the history of timing races in sports events, which slowly transitioned from quarter-seconds (1864) to fifths, to tenths (the 1932 Olympics), to hundredths of a second (the Olympics in the 1970s). "There are few people outside the realm of organized religion who are as conservative as sports officials." As an example, instruments capable of resolving hundredths of a second were introduced at the 1924 Paris Olympic games, yet handheld mechanical watches continued to pick the winner until the 1960 games. Handheld watches generally yielded faster races, as human judges had a tendency to jump the finish.

Harrison's marine clocks (H-1 through H-5) provide the central thread of Dava Sobel's masterful account of the solving of what was considered the greatest scientific problem of the time [95]. This short (184-page) and diminutive (13 cm × 20 cm) book, just slightly larger than H-4 itself, has just been reissued as a lavish coffee-table edition [96], with illustrations located and selected by William Andrewes, and has inspired a PBS science series, *Lost at Sea: The Search for Longitude.*

OVERVIEW

The case studies in this book have amply demonstrated that SQL-92 does not look favorably upon time-oriented applications. Even the most simple tasks, such as specifying a primary key or joining two tables, become mired in complexity when time is introduced.

Fortunately, the clouds part at the horizon. A minor language extension proposed for SQL3 dramatically simplifies coding such applications by providing support for periods, valid time, and transaction time. In this chapter we rephrase the applications of previous chapters using these new constructs. That all can be reimplemented in a single chapter is an indication of the reduction in code length and complexity occasioned by these new constructs.

Finally, we examine some public domain and commercial tools that ease the transition until temporal support is directly incorporated into database systems.

Language Directions

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

12.1 SQL-92

Many knotty problems arise when we have to contend with time-varying data in SQL-92.

- Avoiding duplicates in a time-varying table requires an aggregate (CF-5.14) or a complex trigger (e.g., CF-5.26).
- A simple 3-line join when applied to time-varying tables explodes to a 29-line query consisting of four SELECT statements (CF-6.11) or a complex 23-line statement with four CASE expressions (CF-6.12).
- A 3-line UPDATE of a time-varying table translates into five modification statements totaling 27 lines (CF-7.18).
- Maintaining a tracking log requires several triggers comprising some three dozen lines (CF-9.2).
- The same UPDATE of a bitemporal table translates into *eight* modification statements totaling 58 lines (CF-10.17)!

In addition to being long, these statements are often highly convoluted, with multiple levels of correlated subqueries. Clearly something is amiss.

12.2 SQL-92 LIMITATIONS

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

Second

By definition, there are exactly 24 sidereal hours in a sidereal day, 60 sidereal minutes in a sidereal hour, and 60 sidereal seconds in a sidereal minute. However, because the orbit and rotation of the earth vary slightly, the duration of a sidereal second is not constant. So instead, astronomers use an *ephemeris second*, which is a constant duration of time: 1/31,556,925.9747 of the period of the tropical year between the vernal equinoxes of 1899 and 1900. While this may seem an odd definition, the ephemeris second is actually the average value of a second calculated from astronomical observations over the 18th and 19th centuries.

The advent of atomic clocks has provided another unit that is fixed for all practical purposes. In October 1967, the *atomic second* was adopted as the fundamental unit of time (the SI second) by the international standards community, thereby shifting the basis of time from celestial to quantum mechanics. Specifically, the second in the International System of Weights and Measures was defined to be 9,192,631,770 periods of the radiation emitted by the transition between two hyperfine states of the cesium 133 atom in the ground state. On January 1, 1972, the atomic second became the practical unit of time. The UTC clock runs just a little fast with respect to mean solar time, gaining about a second a year. UTC is adjusted by applying leap seconds on January 1 or July 1 to keep UTC within 0.7 seconds of solar time.

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 our terminology, all that SQL supports is nonsequenced operations, which we saw were often the least useful.

12.3   SQL3

> All the time in the world
> Climbs the walls, swells the doors
> It goes flying out the window
> All the time in the world...
> These precious days we live through
> Thrown away like tissue
> I wish that I could give you all the time in the world
>
> —Beth Nielsen Chapman and Bill Lloyd, "All the Time in the World"

At the heart of this book is the profound revelation that time is much more than just a column, and that SQL-92 is abysmally deficient in the constructs it provides to express time-varying applications. Fortunately, there are now specific proposals

for temporal support in SQL3 that are being considered by the standards committees (see Section 12.16) and are being incorporated into products by vendors. This chapter will summarize these new SQL3 constructs and revisit the preceding case studies, 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 proposals or 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. It is doubtful that SQL/Temporal will reach ISO standard status before the next millennium. That said, we examine these constructs as an indication of where things are going. Commercial implementations are already starting to appear; it is likely that one of the prevalent database systems will provide temporal support before such support is officially accepted as part of the SQL3 standard.

We retain the chapter numbers of the case studies as section numbers in this chapter. As an example, the topic of Chapter 6, querying state tables (in SQL-92), parallels Section 6 of this chapter, on querying state tables in SQL3. That we can reimplement in SQL3 in a single chapter all four case studies, which took the bulk of this book to code in SQL-92, is itself testament to the purity and expressive power of the new constructs.

12.4 PERIODS

SQL3 has a period type constructor. Period types can be constructed from datetime and exact numeric element types.

SQL3 adds the `PERIOD( )` constructor. An SQL data type constructor specifies a new type constructed out of a specified type. Examples are SQL sets, multisets (i.e., with duplicates), and lists (i.e., with ordering). In the case of the period type constructor, you can specify period data types of the SQL datetime data types as well as of exact numerics with a scale of 0 (i.e., integers). Hence, the following period data types are available.

- `PERIOD(DATE)`
- `PERIOD(TIME)` and `PERIOD(TIME WITH TIME ZONE)`
- `PERIOD(TIMESTAMP)` and `PERIOD(TIMESTAMP WITH TIME ZONE)`
- `PERIOD(INT)` and `PERIOD(INTEGER)`
- `PERIOD(SMALLINT)`
- `PERIOD(NUMERIC)`
- `PERIOD(DECIMAL)`

All but the first allow a scale (number of fractional digits) to be specified, though for the last four types, this scale must be 0.

12.4.1   Period Literals

SQL3 period literals support all combinations of open and closed delimiting datetimes.

Period literals are quite complex, for several reasons. First, all four variants of closed-closed, closed-open, open-closed, and open-open are supported. The closed variants use square brackets ('[' and ']'); the open variants, parentheses ('(' and ')'). Note, however, that this distinction concerns the period value that the literal denotes; it has nothing to do with the internal representation of a period, which is properly not specified in the standard. Second, the delimiters can each be a date value (denoting a date period literal), a time value (denoting a time period literal), or both (denoting a timestamp period literal). Third, only the ending delimiter can have an (optional) time zone; if present, it applies to both delimiters. Finally, the separator between the delimiting values can be either a minus sign, in which case there must be spaces around it, or a comma, in which case the surrounding spaces are optional.

The following are all valid period literals and denote the same value, of type `PERIOD(DATE)`.

- `PERIOD '[1997-01-01 - 1997-12-31]'`
- `PERIOD '[1997-01-01 - 1998-01-01)'`
- `PERIOD '(1996-12-31 - 1997-12-31]'`
- `PERIOD '(1996-12-31 - 1998-01-01)'`
- `PERIOD '[1997-01-01,1997-12-31]'`

The following are also valid literals, of type `PERIOD(TIMESTAMP WITH TIME ZONE)`. The time zone appears last in the literal.

- `PERIOD '[1997-01-01 00:00:00 - 1997-12-31 23:59:59-07:00]'`
- `PERIOD '[1997-01-01 00:00:00 - 1998-01-01 00:00:00-07:00)'`
- `PERIOD '(1996-12-31 23:59:59 - 1997-12-31 23:59:59-07:00]'`
- `PERIOD '(1996-12-31 23:59:59 - 1998-01-01 00:00:00-07:00)'`

12.4.2   Predicates

SQL3 defines several predicates on periods:

- p `OVERLAPS` q (discussed for SQL-92 on page 35) is extended in SQL3 to allow either operand to be a period value, in addition to the datetime-datetime and datetime-interval pairs supported in SQL-92. p `OVERLAPS` q implements p *overlaps* $q \vee p$ *overlaps*$^{-1}$ $q \vee p$ *starts* $q \vee p$ *starts*$^{-1}$ $q \vee p$ *finishes* $q \vee p$ *finishes*$^{-1}$ $q \vee p$ *during* $q \vee p$ *during*$^{-1}$ $q \vee p$ *equals* q.
- p `PRECEDES` q implements *before*.
- p `SUCCEEDS` q implements *before*$^{-1}$.
- p `MEETS` q implements p *meets* $q \vee p$ *meets*$^{-1}$ q.

- p `CONTAINS` q implements p *during* $q \wedge p \neq q$.

SQL3 raises an exception if the result of a period constructor is not a valid period.

12.4.3 Constructors

SQL3 adds several datetime constructors:

- `BEGIN` implements *beginning*.
- `END` implements *ending*.
- `LAST` implements *last*.
- `PRIOR` implements '`-1`', when applied to a datetime. Hence, *previous*(p) can be expressed in SQL3 as `PRIOR(BEGIN(`p`))`.
- `NEXT` implements '`+1`', when applied to a datetime.

SQL3 adds one interval constructor, `INTERVAL(`p`)`, which implements *duration*. An interval qualifier can also be specified, as in `INTERVAL(`p `DAY)`.

SQL3 adds several period constructors:

- `PERIOD[`a, b`)` yields a period beginning at a and ending at b; closed-closed, open-closed, and open-open variants are also included.
- p `P_UNION` q implements '$\cup$' over periods; an exception is raised if `NOT` p `OVERLAPS` q.
- p `P_EXCEPT` q implements '$-$' over periods; an exception is raised if p `CONTAINS` q `OR` q `CONTAINS` p.
- p `P_INTERSECT` q implements '$\cap$' over periods; an exception is raised if `NOT` p `OVERLAPS` q.
- `CAST(`p `AS` *type*`)` allows you to change the granularity of p.

Table 12.1 summarizes how the period operations can be implemented in SQL/ Temporal. In the first column, p and q denote period values, and i denotes an

interval value. The OVERLAPS in the first column, next-to-last line of the predicates is the SQL-92 OVERLAPS predicate.

12.5   DEFINING VALID-TIME STATE TABLES

The University Information System consists of some 300 tables, 4 of which are listed below:

```
EMPLOYEES(SSN, LAST_NAME, FIRST_NAME, ANNUAL_SALARY)
INCUMBENTS(SSN, PCN)
POSITIONS(PCN, JOB_TITLE_CODE1)
JOB_TITLES(JOB_TITLE_CODE, JOB_TITLE)
```

These tables are snapshot tables, in that they capture the current state of the modeled reality. The EMPLOYEES table specifies each employee's current annual salary, the INCUMBENTS table identifies the position code for each current employee, and the POSITIONS and JOB_TITLES tables in concert provide the job title(s) for each position. SQL3 can express many useful queries on these tables.

Code Fragment 12.1   What is Bob's position?

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
  AND EMPLOYEES.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
```

SQL3 can also express integrity constraints on such tables. The following is an especially useful one.

Code Fragment 12.2   (SSN, PCN) is a sequenced primary key for INCUMBENTS.

```
ALTER TABLE INCUMBENTS ADD PRIMARY KEY (SSN, PCN)
```

Valid-time support is specified in SQL3 with an ADD VALIDTIME clause.

To indicate that the history of the time-changing reality is to be captured in the INCUMBENTS table, *valid-time support* is added to that table, associating with each row a period indicating when that row was valid in reality. SQL3 includes specific constructs to define, query, and modify tables with valid-time support.

Table 12.1 Period operations in SQL3.

Period Operations	SQL3 Equivalent
Types:	
period	PERIOD(*datetime type*)
Predicates:	
p equals q	$p = q$
p before q	p PRECEDES q
p before^{-1} q	p SUCCEEDS q
p meets q	END(p) = BEGIN(q)
p meets^{-1} q	END(q) = BEGIN(p)
p overlaps q	BEGIN(p) < BEGIN(q) AND BEGIN(q) < END(p)
p overlaps^{-1} q	BEGIN(q) < BEGIN(p) AND BEGIN(p) < END(q)
p during q	BEGIN(q) < BEGIN(p) AND END(p) < END(q)
p during^{-1} q	BEGIN(p) < BEGIN(q) AND END(q) < END(p)
p starts q	BEGIN(p) = BEGIN(q) AND END(p) < END(q)
p starts^{-1} q$_2$	BEGIN(p) = BEGIN(q) AND END(q) < END(p)
p finishes q	BEGIN(q) < BEGIN(p) AND END(p) = END(q)
p finishes^{-1} q	BEGIN(p) < BEGIN(q) AND END(p) = END(q)
p OVERLAPS *q*	p OVERLAPS q
p IS NULL	p IS NULL
Datetime Constructors:	
beginning(*p*)	BEGIN(p)
previous(*p*)	PRIOR(BEGIN(p))
last(*p*)	LAST(p)
ending(*p*)	END(p)
Interval Constructors:	
duration(*p*)	INTERVAL(p), INTERVAL(p AS *qual*)

extract_time_zone(*p*)	CAST(EXTRACT(TIMEZONE_HOUR
	        FROM BEGIN(p)) AS HOUR) +
	    CAST(EXTRACT(TIMEZONE_MINUTE
	        FROM BEGIN(p)) AS MINUTE)

Period Constructors:	
p + i	PERIOD[BEGIN(p) + i, END(p) + i)
i + p	PERIOD[BEGIN(p) + i, END(p) + i)
p - i	PERIOD[BEGIN(p) - i, END(p) - i)
p extend q	not possible
p ∩ q	p P_INTERSECT q
p - q	p P_EXCEPT q
p ∪ q	p P_UNION q
p AT TIME ZONE *i*	PERIOD[BEGIN(p) AT TIME ZONE i, END(p) AT TIME ZONE i)
p AT LOCAL	PERIOD[BEGIN(p) AT LOCAL, END(p) AT LOCAL)
Other Operators:	
CAST(*a* AS PERIOD)	PERIOD[*a*, *a*]
CAST(*p* AS CHAR)	CAST(p AS CHAR)

Code Fragment 12.3   **Add valid-time support to** INCUMBENTS**.**

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

The period initially associated with each row has the indicated granularity, here, day, of "now" (that is, CURRENT_DATE) to "forever" (that is, '9999-12-31').

12.5.1   Temporal Keys and Uniqueness

The primary key was specified when INCUMBENTS was a snapshot table. An important property of SQL3 is that such integrity constraints continue to hold after valid-time support is added. This property is termed *temporal upward compatibility*, and requires that each integrity constraint on an associated snapshot database (e.g., the original INCUMBENTS table) be interpreted to hold on the current time-slice of the temporal counterpart of the database (the table with valid-time support).

Constraints expressed on nontemporal tables are interpreted in SQL3 as current constraints when valid-time support is added to the table.

When INCUMBENTS did not have valid-time support, it modeled the current reality. As position assignments changed, the table was modified to reflect the new situation. Integrity constraints such as primary and foreign keys applied to the information currently in the table.

When history is retained, by adding valid-time support to the table, these integrity constraints still apply to the current information. Temporal upward compatibility thus implies that existing constraints on a nontemporal table become *current* constraints when valid-time support is added to the table. Compare the primary key constraint, CF-12.2 above, which still holds, with CF-5.12, which does the same thing in SQL-92, but requires nine lines in a CHECK constraint.

An SQL-92 statement can be converted into a sequenced statement in SQL3 simply by prepending VALIDTIME.

This primary key constraint is perfectly fine if only current modifications are made to the table. If sequenced or nonsequenced modifications are possible, then we need to ensure that the primary key holds on all instants of time, even when the information valid on a particular day may be changed by a later modification. For that, we need a sequenced primary key constraint: "No employee can have the same position more than once simultaneously."

In SQL3, *any* statement can be rendered sequenced by prepending the reserved word VALIDTIME.

Code Fragment 12.4   (SSN**,** PCN) **is a sequenced primary key for** INCUMBENTS**.**

```
ALTER TABLE INCUMBENTS ADD VALIDTIME PRIMARY KEY (SSN, PCN)
```

Compare this with CF-5.8, a 10-line CHECK constraint containing an aggregate and two levels of nested subqueries.

The Minute Hand (Second Major Advance)
After the hour hand was invented around 300 B.C.E., the next transition had to await the passage of two millennia, to Huygens's pendulum clock in 1656 C.E. The dramatic vault in accuracy afforded by the pendulum enabled the addition of a minute hand. It is difficult to imagine now a society in which time was known only to roughly an hour.

As we've seen throughout this book, the sequenced variant is generally the one to use. It is for that reason that the SQL3 syntax is designed to succinctly indicate that a sequenced semantics is desired.

As one further example, consider the uniqueness constraint: "No employee can have two identical positions." On the original, nontemporal INCUMBENTS table, this is expressed as a uniqueness constraint.

Code Fragment 12.5   **Prevent duplicates in** INCUMBENTS**.**

```
ALTER TABLE INCUMBENTS ADD UNIQUE (SSN, PCN)
```

When valid-time support is added to the INCUMBENTS table, this constraint continues to be interpreted as current uniqueness.

Expressing sequenced uniqueness, "At no time can an employee have two identical positions," requires but a single additional reserved word.

Code Fragment 12.6   **Prevent sequenced duplicates in** INCUMBENTS**.**

```
ALTER TABLE INCUMBENTS ADD VALIDTIME UNIQUE (SSN, PCN)
```

(Compare with CF-5.14.)

12.5.2   Referential Integrity

We now revisit the various kinds of referential integrity (RI) and show how they may be expressed with the proposed constructs by examining the four cases from Section 5.6.

Case 1   **Neither table is temporal.**

Assume initially that neither the INCUMBENTS nor the POSITIONS table has temporal support. Referential integrity can then be expressed in SQL-92 as follows:

Code Fragment 12.7   INCUMBENTS.PCN **is a foreign key for** POSITIONS.PCN **(neither table is temporal).**

```
ALTER TABLE INCUMBENTS
ADD FOREIGN KEY (PCN) REFERENCES POSITIONS
```

Case 2   **Only the referencing table is temporal.**

When valid-time support was added to the INCUMBENTS table, via ADD VALID-TIME PERIOD, the foreign key constraint still applies directly. It translates to a current constraint: "For each currently valid row of INCUMBENTS, the PCN is also in POSITIONS." Temporal upward compatibility ensures that applications are not broken when temporal support is added to an underlying table.

Case 3   **Both tables are temporal.**

We now add temporal support to the referenced table, POSITIONS.

Code Fragment 12.8   INCUMBENTS.PCN **is a current foreign key for** POSITIONS.PCN **(both tables are temporal).**

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

Unlike the complex statements required in SQL-92, the sequenced variant in SQL3 is almost identical to the nontemporal analog.

The foreign key specified above (CF-12.7) when applied to referencing and referenced tables with valid-time support continues to be interpreted as a current foreign key. (Compare with the SQL-92 version, CF-5.20, at 12 lines.)

The sequenced RI constraint, "At each point in time, each incumbent's PCN is valid at that time," is the most natural application of the nontemporal RI constraint to time-varying information. This required a complex 32-line assertion (CF-5.21). Using the proposal constructs, only one additional keyword, VALIDTIME, is necessary to obtain a sequenced integrity constraint.

Code Fragment 12.9   INCUMBENTS.PCN **is a sequenced foreign key for** POSITIONS.PCN **(both tables are temporal).**

```
ALTER TABLE INCUMBENTS
ADD VALIDTIME FOREIGN KEY (PCN) REFERENCES POSITIONS
```

An SQL-92 statement can be converted into a nonsequenced statement in SQL3 simply by prepending NONSEQUENCED VALIDTIME.

A nonsequenced RI constraint ("For each value of INCUMBENTS.PCN, there existed at some, possibly different, time that value in POSITIONS.PCN") is equally simple to specify: we need only prepend the reserved word NONSEQUENCED. While nonsequenced constraints (and queries) are notoriously awkward to express in English, their translations to SQL-92 and SQL3 are almost identical, differing only in that single reserved word NONSEQUENCED.

Code Fragment 12.10   `INCUMBENTS.PCN` **is a nonsequenced foreign key for** `POSITIONS.PCN` **(both tables are temporal).**

```
ALTER TABLE INCUMBENTS ADD NONSEQUENCED VALIDTIME
FOREIGN KEY (PCN) REFERENCES POSITIONS
```

Case 4   **Only the referenced table is temporal.**

Here we drop the temporal support on the referencing table.

Code Fragment 12.11   `INCUMBENTS.PCN` **is a current foreign key for** `POSITIONS.PCN` **(only** `POSITIONS` **is temporal).**

```
ALTER TABLE INCUMBENTS DROP VALIDTIME
```

The original RI constraint, CF-12.7, continues to apply and is equivalent to the 10-line assertion in CF-5.24, in which every `PCN` value in `INCUMBENTS` must also occur in the current state of the `POSITIONS` table.

In summary, temporal upward compatibility ensures that existing constraints, such as the referential integrity constraint of CF-12.7, are interpreted as current constraints when applied to tables with valid-time support. Sequenced constraints are specified by prepending VALIDTIME.

12.6   QUERYING STATE TABLES

Although `INCUMBENTS` now has valid-time support, standard SQL queries still apply directly, retrieving (as before valid-time support was added) the current information.

Code Fragment 12.12   **What is Bob's position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
  AND EMPLOYEES.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
```

This query is *identical* to CF-6.2 on the original `INCUMBENTS` table, without valid-time support. This is another example of temporal upward compatibility (TUC). TUC in this context requires that each query will return the same result on an associated snapshot database (e.g., the original `INCUMBENTS` table) as on the temporal counterpart of the database (the table with valid-time support).

TUC applies uniformly to all corners of the language. CF-6.3 requires 10 lines in SQL-92, but is shorter in SQL3 because the timestamp columns need not be mentioned.

Code Fragment 12.13   What is Bob's current position and salary?

```
SELECT JOB_TITLE_CODE1, AMOUNT
FROM EMPLOYEES, INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
    AND SAL_HISTORY.SSN = EMPLOYEES.SSN
```

Code Fragment 12.14   What employees currently have no position?

```
SELECT FIRST_NAME
FROM EMPLOYEES
WHERE NOT EXISTS ( SELECT *
                  FROM INCUMBENTS
                  WHERE EMPLOYEES.SSN = INCUMBENTS.SSN)
```

(Compare with CF-6.4.)

12.6.1   Extracting States

A valid time-slice query is nonsequenced, with the associated timestamp period compared with the specified instant.

A valid time-slice query is nonsequenced. It is not a current query because it involves information in the past or future, and it is not a sequenced query because the result of the query has no timestamp.

Nonsequenced queries are signaled by the NONSEQUENCED reserved word in SQL3. Additionally, the valid time-slice query needs to access the period timestamp to ensure information valid at the specified date is retrieved. To access the valid timestamp of a row, use the VALIDTIME( ) function, which evaluates to a period.

Code Fragment 12.15   What was Bob's position at the beginning of 1997?

```
NONSEQUENCED VALIDTIME SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
    AND VALIDTIME(INCUMBENTS) OVERLAPS DATE '1997-01-01'
```

(Compare with CF-6.5.)

12.6.2   Sequenced Queries

In all cases, a nontemporal query can be rendered sequenced by simply prepending VALIDTIME.

Code Fragment 12.16   **Who makes or has made more than $50,000 annually?**

```
VALIDTIME SELECT *
FROM SAL_HISTORY
WHERE AMOUNT > 50000
```

Code Fragment 12.17   **List the social security numbers of current and past employees.**

```
VALIDTIME SELECT SSN
FROM SAL_HISTORY
```

In CF-6.7, we had to explicitly mention the timestamp columns. Here, SQL3 handles them automatically.

Code Fragment 12.18   **Sequenced sort** INCUMBENTS **on the position code (first version).**

```
VALIDTIME SELECT *
FROM INCUMBENTS
ORDER BY PCN
```

Here again, we needn't be concerned with where the timestamp columns should be placed in the ORDER BY clause (compare with CF-6.8).

Code Fragment 12.19   **Who makes or has made more than $50,000 annually or less than $10,000?**

```
VALIDTIME SELECT *
FROM SAL_HISTORY
WHERE AMOUNT > 50000
UNION ALL
SELECT *
FROM SAL_HISTORY
WHERE AMOUNT < 10000
```

Here, the VALIDTIME applies to the entire query expression, which is SELECT . . . UNION ALL SELECT . . . .

Sequenced joins are quite difficult in SQL-92, but only require adding one reserved word in SQL3. The sequenced variant in SQL3 of a nontemporal query retains the nontemporal query, adding only VALIDTIME. In SQL-92, converting to the sequenced analog requires completely rewriting the query.

Code Fragment 12.20   **Provide the salary and position history for all employees.**

```
VALIDTIME SELECT S.SSN, AMOUNT, PCN
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
```

In SQL-92, this query required some four UNIONs and 29 lines (CF-6.11), or four CASE expressions and 23 lines (CF-6.12), or two SQL/PSM FUNCTIONs (CF-6.14).

The Second Hand (Not a Major Advance)

While the minute hand was invented in 1656, after a period of two millennia, the second hand appeared astonishingly quickly, after but three *decades*, in 1690, for doctors' watches. The second hand was perfectly fine for determining a person's pulse rate, but was of little use in telling the time, for watches weren't, and generally still aren't, of sufficient accuracy to need a second hand: the second hand carries no information as to which instant it is. In 1776, an independent second train, an extraordinarily complex mechanical device, was invented to start and stop the second hand. That most modern watches do not have such a facility is more a testament to marketing (implying that the watch is accurate to the second) than to utility or performance. The second hand on the watch on your wrist is undoubtedly of the same use as on watches of 300 years ago: to provide a visual indication that your watch hadn't stopped.

Nested queries are similarly converted into their sequenced analog.

Code Fragment 12.21   **List the employees who are or were department heads but were not also professors.**

```
VALIDTIME SELECT SSN
FROM INCUMBENTS AS I1
WHERE PCN = 455332
  AND NOT EXISTS (SELECT *
              FROM INCUMBENTS AS I2
              WHERE I2.SSN = I1.SSN
                AND I2.PCN = 821197)
```

The entire query, including the nested portion, is (conceptually) evaluated independently at each instant. This query requires four SELECTs UNIONed together, or 45 lines of SQL-92 (CF-6.18).

This query can also be expressed via EXCEPT.

Code Fragment 12.22   **List the employees who are or were department heads but were not also professors (an equivalent version).**

```
VALIDTIME SELECT SSN
FROM INCUMBENTS
WHERE PCN = 455332
EXCEPT
SELECT SSN
FROM INCUMBENTS
WHERE PCN = 821197
```

12.6.3 Nonsequenced Queries

Nonsequenced queries require (naturally) the NONSEQUENCED adverb in SQL3. As before, the timestamp is available via VALIDTIME( ).

Code Fragment 12.23   **List all the salaries, past and present, of employees who had been a hazardous waste specialist at some time.**

```
NONSEQUENCED VALIDTIME SELECT AMOUNT
FROM INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE INCUMBENTS.SSN = SAL_HISTORY.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
  AND JOB_TITLE_CODE1 = 20730
```

The phrases "past and present" and "at some time" indicate that the query is a nonsequenced one. This query in SQL3 is similar to the SQL-92 variant (CF-6.19); the only difference is the two added keywords.

Code Fragment 12.24   **When did employees receive raises?**

```
NONSEQUENCED VALIDTIME
SELECT S2.SSN, BEGIN(VALIDTIME(S2)) AS RAISE_DATE
FROM SAL_HISTORY AS S1, SAL_HISTORY AS S2
WHERE S2.AMOUNT > S1.AMOUNT
  AND S1.SSN = S2.SSN
  AND VALIDTIME(S1) MEETS VALIDTIME(S2)
```

Here, we access the timestamp in two places, in the SELECT clause, where we grab the beginning date of the timestamp (we could have just as easily used END(VALIDTIME(S1))), and in the WHERE clause. The MEETS predicate is quite useful in such situations.

A nonsequenced query does not view the underlying table with valid-time support as a sequence of states; rather, it views the underlying table as one with an additional timestamp column (of type period) that can be accessed in the query via the function VALIDTIME( ).

12.6.4 Eliminating Duplicates

Duplicate removal of any variant—current, sequenced, or nonsequenced—is specified in SQL3 using DISTINCT. You will notice an appealing consistency to the following three queries:

Code Fragment 12.25   **Remove current duplicates from INCUMBENTS.**

```
SELECT DISTINCT SSN, PCN
FROM INCUMBENTS
```

Code Fragment 12.26   **Remove sequenced duplicates from** INCUMBENTS.

```
VALIDTIME SELECT DISTINCT SSN, PCN
FROM INCUMBENTS
```

Code Fragment 12.27   **Remove nonsequenced duplicates from** INCUMBENTS.

```
NONSEQUENCED VALIDTIME SELECT DISTINCT *
FROM INCUMBENTS
```

The SQL-92 queries (CF-6.23, CF-6.24–6.26, and CF-6.21, respectively) vary dramatically, from 2 lines for nonsequenced to 19–30 lines for sequenced.

<div style="margin-left:2em;">

12.7   **MODIFYING STATE TABLES**

SQL3's explicit support for current, sequenced, and nonsequenced statements applies equally to modification statements.

12.7.1   Current Modifications

Modifications that don't involve the new reserved words when applied to tables with valid-time support are interpreted as current modifications.

Code Fragment 12.28   **Bob joins as associate director of the Computer Center.**

```
INSERT INTO INCUMBENTS
VALUES (111223333, 999071)
```

</div>

Current modifications in SQL3 are the same whether applied to nontemporal tables or to tables with valid-time support.

The default timestamp of PERIOD[CURRENT_DATE, DATE '9999-12-31') is automatically provided when the underlying table has valid-time support. Current uniqueness, primary key, and referential integrity constraints are automatically maintained, so the gymnastics of CF-7.2–CF-7.5 is not necessary.

Code Fragment 12.29   **Bob was just fired as associate director of the Computer Center.**

```
DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
```

Compare with CF-7.7, in which a current deletion in the restricted case is implemented in SQL-92 as an UPDATE, or CF-7.8, in which a current deletion in the general case is implemented as an UPDATE and a DELETE statement.

Code Fragment 12.30   **Today, Bob was promoted to director of the Computer Center.**

```
UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

Compare with CF-7.10 and CF-7.11. In particular, no complex case analysis is required.

12.7.2   Sequenced Modifications

The period of applicability of a sequenced modification is specified in SQL3 immediately after VALIDTIME.

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

In SQL3, the period of applicability is specified immediately after VALIDTIME.

Code Fragment 12.31   **Bob was assigned the position of associate director of the Computer Center for 1997.**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' INSERT INTO INCUMBENTS
VALUES (111223333, 999071)
```

Of course, all (current and sequenced) primary key and referential integrity constraints continue to be checked. In SQL-92, these must be checked within the insertion (CF-7.13 and CF-7.14).

Deletions also allow a period of applicability to be specified.

Code Fragment 12.32   **Bob was removed as associate director of the Computer Center for 1997.**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' DELETE FROM INCUMBENTS
WHERE SSN=111223333
  AND PCN = 999071
```

(Compare with CF-7.16.)

Code Fragment 12.33   **Bob was promoted to director of the Computer Center for 1997.**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

Compare with CF-7.18, which consists of five separate SQL-92 statements. No complex case analysis is required.

A True Second Hand (Third Major Advance)

The hour hand was invented around 300 B.C.E., and the minute hand in 1656. A truly accurate second hand is just now becoming prevalent, in the form of radio watches that enclose Lilliputian radio receivers that tune into signals to synchronize with the standard atomic clocks. The most inexpensive of these watches retails for about $100 at the time of this writing and will certainly fall dramatically in price as production and demand ramp up and as economies of scale come into play. You can marvel that one of the three most momentous transitions in horology over 2300 years occurred in your lifetime.

12.7.3   Nonsequenced Modifications

As with constraints and queries, a nonsequenced modification treats the period timestamp (available via the VALIDTIME( ) function) identically to the other columns.

Code Fragment 12.34   **Delete Bob's records that include 1997 stating that he was associate director of the Computer Center.**

```
NONSEQUENCED VALIDTIME DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
  AND VALIDTIME(INCUMBENTS) CONTAINS DATE '1997-12-31'
```

The CONTAINS predicate is particularly useful here.

Code Fragment 12.35   **Extend Bob's position as associate director of the Computer Center for an additional year.**

```
NONSEQUENCED VALIDTIME UPDATE INCUMBENTS
SET VALIDTIME = PERIOD(BEGIN(VALIDTIME(INCUMBENTS)),
                END(VALIDTIME(INCUMBENTS)) + INTERVAL '1' YEAR]
WHERE SSN = 111223333
  AND PCN = 999071
```

The SQL-92 and SQL3 versions of nonsequenced modifications are quite similar.

In comparison with CF-7.20, this is one of the few situations in which the SQL3 version is longer (in this case, by one line) than the SQL-92 version. Except for details of syntax, though, the two versions are very similar.

12.7.4   Modifications That Mention Other Tables

When one or more tables are rendered temporal, existing constraints, queries, and modifications still apply over the period of applicability of "now" to "forever."

Code Fragment 12.36   **Bob is promoted to director of the Computer Center (current version).**

```
UPDATE INCUMBENTS
SET PCN = (SELECT PCN
           FROM POSITIONS, JOB_TITLES
           WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
           AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER')
WHERE SSN = 111223333
```

Even complex sequenced modifications over several tables can be easily expressed in SQL3 via the VALIDTIME construct.

It is perhaps no longer surprising that this requires some 30 lines of SQL-92 code (CF-7.22).

You will also be able to easily express this update as a sequenced update, over a period of validity of the year 1997.

Code Fragment 12.37   **Bob was promoted to director of the Computer Center for 1997 (sequenced version).**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' UPDATE INCUMBENTS
SET PCN = (SELECT PCN
           FROM POSITIONS, JOB_TITLES
           WHERE POSITIONS.JOB_TITLE_CODE1 = JOB_TITLE_CODE
           AND JOB_TITLE = 'DIRECTOR, COMPUTER CENTER')
WHERE SSN = 111223333
```

(Compare with CF-7.24, requiring eight SQL-92 statements and 77 lines.)

12.7.5   Temporal Partitioning*

Temporal partitioning is an aspect of physical design.

Temporal partitioning is effectively a *physical* design aspect, and as such should not impact the expression of queries or modifications on the partitioned table. Of course, as SQL-92 does not include the notion of tables with valid-time support, the partitioning must be implemented manually by the application programmer. As an analogy, if a DBMS didn't implement indexes, then they might be simulated by the application programmer via additional tables, which would dramatically complicate the SQL code.

In SQL3, temporal partitioning has no impact on queries.

As SQL3 does support tables with valid-time support, it enables the underlying DBMS to include partitioning. A specific syntax for specifying temporal partitioning is not included in SQL3 for the same reason that syntax to specify indexing is not included in SQL-92.

In the following, we assume that temporal partitioning of the INCUMBENTS table has been specified in some DBMS-specific manner.

Code Fragment 12.38   **What is Bob's current position (partitioned)?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
    AND EMPLOYEES.SSN = INCUMBENTS.SSN
    AND INCUMBENTS.PCN = POSITIONS.PCN
```

This is similar to CF-7.25, with the exception that the original table, INCUMBENTS, is mentioned, rather than the current store, INCUMBENTS_CURRENT.

Code Fragment 12.39   **Provide the salary and department history for all employees (partitioned).**

```
VALIDTIME SELECT S.SSN, AMOUNT, PCN
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
```

Incredibly, this requires 50 lines in SQL-92 (CF-7.26).
    Current modifications work as before.

Code Fragment 12.40   **Bob was assigned the position of associate director of the Computer Center (partitioned).**

```
INSERT INTO INCUMBENTS
VALUES (111223333, 6201945234)
```

This is quite similar to the SQL-92 version (CF-7.27). The SQL3 version will also ensure that no current or sequenced integrity constraints are violated.

Code Fragment 12.41   **Bob was fired as associate director of the Computer Center (partitioned).**

```
DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
```

SQL3 really shines in sequenced operations.

Code Fragment 12.42   **Bob was removed as associate director of the Computer Center for 1997 (partitioned).**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' DELETE FROM INCUMBENTS
WHERE SSN = 111223333
  AND PCN = 999071
```

(Compare with the eight statements of CF-7.30.) Note that the core of the modification in SQL3 is unaffected by either the modification being sequenced or an underlying table being temporally partitioned.

Code Fragment 12.43   **Bob was promoted to director of the Computer Center for 1997 (partitioned).**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

(Compare with CF-7.31 at 57 lines.)

12.8   RETAINING A TRACKING LOG

The previous sections concerned tables with valid-Time support, capturing the history of the modeled reality. A tracking log is a quite different animal, as it captures the history of the table itself, allowing prior states to be retrieved. The states of the database at all previous points of time are retained, and modifications are append-only. Changes are not allowed on the past states, as that would prevent secure auditing. Instead, compensating transactions are used to correct errors.

SQL3 supports transaction time in a fashion parallel to valid time, utilizing the TRANSACTIONTIME reserved word.

12.8.1   Defining Transaction-Time Tables

The simplest way to implement a tracking log is to add transaction-time support to the monitored table. Transaction-time support may be specified in SQL3 with a simple ADD TRANSACTIONTIME.

Code Fragment 12.44   **Add transaction-time support to the** PROJECTIONS **table.**

```
ALTER TABLE PROJECTIONS ADD TRANSACTIONTIME
```

Unlike valid-time support, the granularity is not specified, but instead is provided by the DBMS. Previously defined integrity constraints, such as PROJECTION_ID being a primary key, are retained, interpreted as current constraints. Accomplishing this in SQL-92 required a set of triggers (CF-8.2).

The DBMS is now responsible for maintaining transaction time for us. In particular, we don't have to worry about an application inadvertently corrupting past states (say, by incorrectly altering the timestamp columns), or a white-collar criminal intentionally "changing history" to cover up his tracks. The DBMS simply does not permit past states to be modified.

The DBMS controls the representation of a table with transaction-time support. It might utilize a single instant timestamp, termed a *tracking log* on page 220, a pair of instants or

The representation of a table with transaction-time support in SQL3 is specified with physical design statements supported by the DBMS.

a period as a timestamp, an instant timestamp coupled with an operation code, termed a *backlog* on page 233, or a backlog with after-images. The DBMS may provide syntax to choose among several representations; such physical design statements are not included in SQL3.

12.8.2   Queries

A query on the current state of a table with transaction-time support is trivial: simply omit any mention of time.

Code Fragment 12.45   **List the information on projection 5.**

```
SELECT *
FROM PROJECTIONS
WHERE PROJECTION_ID = 5
```

To reconstruct the table as of some point in the past, a transaction-time nonsequenced query is required.

Code Fragment 12.46   **Reconstruct the** PROJECTIONS **table as of April 1, 1996.**

```
NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
    PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM PROJECTIONS AS P
WHERE TRANSACTIONTIME(P) OVERLAPS DATE '1996-04-01'
```

Here we use the TRANSACTIONTIME( ) function to access the transaction time-stamp associated with each row. Compare with CF-8.3.

This can also be defined as a view.

Code Fragment 12.47   **Reconstruct the** PROJECTIONS **table as of April 1, 1996, as a view.**

```
CREATE VIEW April_PROJECTIONS
    ( PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE)
AS (NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
    PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE
FROM PROJECTIONS AS P
WHERE TRANSACTIONTIME(P) OVERLAPS DATE '1996-04-01')
```

CF-8.5 showed how to convert P_Log to a transaction-time state table. Here, PROJECTIONS is already a state table. So perhaps an analogous operation would be to convert the PROJECTIONS table to one containing a When_Changed column.

Code Fragment 12.48   **Convert** PROJECTIONS **to an instant-stamped table.**

```
CREATE VIEW P_Log
  (PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
     SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
     When_Changed)
AS ( NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
     PROJECTION_NAME, PROJECTION_TYPE,
     SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
     END(TRANSACTIONTIME(PROJECTIONS)) )
FROM PROJECTIONS
WHERE END(TRANSACTIONTIME(PROJECTIONS)) < CURRENT_TIMESTAMP
```

Transaction-time sequenced queries are signaled in SQL3 with the TRANSACTIONTIME prefix.

Since all currently active rows have a transaction timestamp ending at "now," we eliminate those in the WHERE clause, since they haven't been previously changed.

Transaction-time sequenced queries ("when was it recorded") are easy on tables with transaction-time support: just prepend TRANSACTIONTIME.

Code Fragment 12.49   **When was it recorded that a projection had a type of 17?**

```
TRANSACTIONTIME SELECT PROJECTION_ID, PROJECTION_TYPE
FROM PROJECTIONS_State
WHERE PROJECTION_TYPE = 17
```

Code Fragment 12.50   **List the projections recorded as having the same USGS zone code as the projection with ID 13447.**

```
TRANSACTIONTIME SELECT S1.PROJECTION_NAME
FROM PROJECTIONS_state AS S1, PROJECTIONS_state AS S2
WHERE S1.ZONE_CODE = S1.ZONE_CODE
  AND S2.PROJECTION_ID = 13447
  AND S1.PROJECTION_ID <> 13447
```

(Compare with CF-8.18.)

Another kind of nonsequenced query is the conversion to a tracking log (compare with CF-9.12–9.14), containing before-images, containing after-images, or as a backlog, respectively.

Code Fragment 12.51   **Extract before-images from a transaction-time state table.**

```
NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      END(TRANSACTIONTIME(PROJECTIONS)) AS When_Changed
FROM PROJECTIONS
WHERE END(TRANSACTIONTIME(PROJECTIONS)) <> CURRENT_TIMESTAMP
```

Code Fragment 12.52   **Extract after-images from a transaction-time state table.**

```
NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      BEGIN(TRANSACTIONTIME(PROJECTIONS)) AS When_Changed
FROM PROJECTIONS
```

Code Fragment 12.53   **Extract a backlog from a transaction-time state table.**

```
NONSEQUENCED TRANSACTIONTIME SELECT PROJECTION_ID,
      PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      BEGIN(TRANSACTIONTIME(P1)) AS When_Changed, 'I' AS Operation
FROM PROJECTIONS AS P1
WHERE NOT EXISTS ( SELECT *
        FROM PROJECTIONS AS P2
        WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
          AND TRANSACTIONTIME(P2) MEETS TRANSACTIONTIME(P1))
UNION
SELECT PROJECTION_ID, PROJECTION_NAME, PROJECTION_TYPE,
      SPHEROID_CODE, PROJECTION_UOM, ZONE_CODE,
      END(TRANSACTIONTIME(P1)) AS When_Changed, 'D' AS Operation
```

```
FROM PROJECTIONS AS P1
WHERE END(TRANSACTIONTIME(P1)) <> CURRENT_TIMESTAMP
  AND NOT EXISTS ( SELECT *
        FROM PROJECTIONS AS P2
        WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
          AND TRANSACTIONTIME(P1) MEETS TRANSACTIONTIME(P2))
UNION
SELECT P1.PROJECTION_ID, P1.PROJECTION_NAME,
       P1.PROJECTION_TYPE, P1.SPHEROID_CODE,
       P1.PROJECTION_UOM, P1.ZONE_CODE,
       BEGIN(TRANSACTION(P2)) AS When_Changed, 'U' AS Operation
FROM PROJECTIONS AS P1, PROJECTIONS AS P2
WHERE P1.PROJECTION_ID = P2.PROJECTION_ID
  AND TRANSACTIONTIME(P1) MEETS TRANSACTIONTIME(P2)
```

Here again MEETS turns out to be quite useful.

12.8.3   Modifications

Sequenced and nonsequenced modifications are not permitted on tables with transaction-time support, as such modifications would violate the semantics of such tables. Instead, only current modifications are allowed. Such modifications in SQL3 are expressed on tables with transaction-time support without mention of time.

Code Fragment 12.54   **Insert a projection with an ID of 6.**

```
INSERT INTO PROJECTIONS (PROJECTION_ID, PROJECTION_NAME,
       PROJECTION_TYPE, SPHEROID_CODE, PROJECTION_UOM,
       ZONE_CODE)
VALUES (6, 'New Projection', 22, 14, 93, 4)
```

(Compare with CF-9.4.)

Code Fragment 12.55   **Delete projection 2.**

```
DELETE FROM PROJECTIONS
WHERE PROJECTION_ID = 2
```

Code Fragment 12.56   **Change the type of projection 1 to 43.**

```
UPDATE PROJECTIONS
SET PROJECTION_TYPE = 43
WHERE PROJECTION_ID = 1
```

Temporal upward compatibility ensures that such modifications, which don't mention time, maintain the prior states.

12.9 **TRANSACTION-TIME STATE TABLES**

As mentioned previously, the representation of a table with temporal support is a physical design concern. The DBMS may provide syntax to choose among several representations; such physical design statements are not included in SQL3. Hence, the queries and modifications presented above are still appropriate for a period-stamped representation. Here, we present the remaining queries.

12.9.1 **Queries**

As with valid time, a sequenced transaction-time prefix applies to the entire query, in the following case, to the UNION.

Code Fragment 12.57 **Give the change history for projections having a type of 12 or 18.**

```
TRANSACTIONTIME SELECT PROJECTION_ID
FROM P_TT
WHERE PROJECTION_TYPE = 12
UNION
SELECT PROJECTION_ID
FROM P_TT
WHERE PROJECTION_TYPE = 18
```

(Compare with CF-9.9.)

Code Fragment 12.58 **When was it recorded that two projections had the same type?**

```
TRANSACTIONTIME SELECT P1.PROJECTION_ID, P2.PROJECTION_ID,
    P1.PROJECTION_TYPE
FROM PROJECTIONS AS P1, PROJECTIONS AS P2
WHERE P1.PROJECTION_ID <> P2.PROJECTION_ID
  AND P1.PROJECTION_TYPE = P2.PROJECTION_TYPE
```

Auditing queries are often transaction-time nonsequenced queries.

Code Fragment 12.59 **When was the type of a projection erroneously changed to be identical to that of an existing projection?**

```
NONSEQUENCED TRANSACTIONTIME SELECT P1.PROJECTION_ID,
    P1.PROJECTION_TYPE AS Identical_TYPE,
    P3.PROJECTION_TYPE AS Prior_TYPE,
    BEGIN(TRANSACTIONTIME(P2)) AS When_Changed
FROM P_TT AS P1, P_TT AS P2, P_TT AS P3
WHERE P1.PROJECTION_ID <> P2.PROJECTION_ID
  AND P2.PROJECTION_ID = P3.PROJECTION_ID
  AND P1.PROJECTION_TYPE = P2.PROJECTION_TYPE
  AND P2.PROJECTION_TYPE <> P3.PROJECTION_TYPE
  AND TRANSACTIONTIME(P3) MEETS TRANSACTIONTIME(P2)
```

The MEETS predicate is quite handy for "change" queries. Compare with CF-9.11.

12.9.2 Temporal Partitioning and Vacuuming

Temporal partitioning is representational, and thus in the domain of physical design. You could envision the DBMS providing an ADD PARTITIONING clause to the ALTER TABLE statement.

As with all physical design decisions, such a statement would not impact the specification of queries or modifications. Hence, the code fragments given above work perfectly fine with a temporally partitioned representation.

Vacuuming is not yet supported in SQL3.

12.10 BITEMPORAL TABLES

Valid-time support and transaction-time support in concert result in a bitemporal table.

Valid-time support and transaction-time support are orthogonal in SQL3. They can be used separately, as above, or together, resulting in a table with both kinds of support, termed a *bitemporal table*.

Code Fragment 12.60 **Create the** `Prop_Owner` **table.**

```
CREATE TABLE Prop_Owner (
  customer_number INT,
  property_number INT)
AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME
```

While the granularity of the valid timestamp is specified by the user (here, to a granularity of day), the granularity of the transaction timestamp is supplied by the DBMS.

As before, nontemporal queries, views, constraints, assertions, and modifications are interpreted as current (in both valid time and transaction time) when applied to a bitemporal table. The concepts of temporal upward compatibility, sequenced semantics, and nonsequenced semantics apply orthogonally to valid time and transaction time.

The semantics is dictated by three simple rules:

- The absence of VALIDTIME (respectively, TRANSACTIONTIME) indicates valid-time (transaction-time) upward compatibility. The result of such a query does not include valid-time (transaction-time) support.
- VALIDTIME (respectively, TRANSACTIONTIME) indicates sequenced valid (transaction) semantics. An optional period expression temporally scopes the result. The result of such a query includes valid-time (transaction-time) support.

- NONSEQUENCED denotes nonsequenced valid (respectively transaction) semantics. An optional period expression after NONSEQUENCED VALIDTIME (TRANSACTIONTIME) provides a valid-time (transaction-time) timestamp, yielding valid-time (transaction-time) support in the result.

The valid-time and transaction-time clauses can be used together in assertions and constraints.

Code Fragment 12.61   `property_number` **is a (valid-time sequenced, transaction-time sequenced) primary key for** `Prop_Owner`**.**

```
CREATE ASSERTION P_O_seq_primary_key
VALIDTIME AND TRANSACTIONTIME PRIMARY KEY (property_number)
```

(Compare with CF-10.2.)

Code Fragment 12.62   **The customer number in** `Prop_Owner` **is a foreign key referencing the** `Customer` **table (valid-time sequenced/transaction-time current version).**

```
ALTER TABLE Prop_Owner ADD
VALIDTIME FOREIGN KEY (customer_number) REFERENCES Customer
```

(Compare with CF-10.49, at 48 lines, and CF-10.50, 18 lines.)

12.10.1 Queries

We start with time-slice queries, which are generally nonsequenced in one dimension. The first is a transaction time-slice query, resulting in a table with valid-time support (as such, it is sequenced in valid time).

Code Fragment 12.63   **Give the history of owners of the flat at Skovvej 30 in Aalborg as of January 1, 1998.**

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
  AND TRANSACTIONTIME(Prop_Owner) OVERLAPS DATE '1998-01-01'
```

The result is a table with one column, `customer_number`, and one (valid) period timestamp. Here we again use the TRANSACTIONTIME function to extract the transaction timestamp. Compare with CF-10.19.

The OVERLAPS predicate is not required for a current transaction time-slice.

Code Fragment 12.64   **Give the history of owners of the flat at Skovvej 30 in Aalborg as best known.**

```
VALIDTIME SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
```

The valid time-slice results in a transaction-time state table; it is nonsequenced in valid time but sequenced in transaction time.

Code Fragment 12.65 **When was the information about the owners of the flat at Skovvej 30 in Aalborg on January 4, 1998, recorded in the** `Prop_Owner` **table?**

```
NONSEQUENCED VALIDTIME AND TRANSACTIONTIME
SELECT customer_number FROM Prop_Owner
WHERE property_number = 7797
  AND VALIDTIME(Prop_Owner) OVERLAPS DATE '1998-01-04'
```

Here, the result has a transaction timestamp.

A bitemporal time-slice takes as input two instants, a valid-time and a transaction-time instant, and results in a snapshot state of the information regarding the enterprise at that valid time, as recorded in the database at that transaction time. It is nonsequenced in both valid and transaction time.

Code Fragment 12.66 **Give the owner of the flat at Skovvej 30 in Aalborg on January 13 as stored in the** `Prop_Owner` **table on January 18.**

```
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
  AND VALIDTIME(Prop_Owner) OVERLAPS DATE '1998-01-13'
  AND TRANSACTIONTIME(Prop_Owner) OVERLAPS DATE '1998-01-18'
```

The result includes no implicit timestamps.

The current bitemporal time-slice is particularly easy to write in SQL3.

Code Fragment 12.67 **Give the owner of the flat at Skovvej 30 in Aalborg today as best known.**

```
SELECT customer_number
FROM Prop_Owner
WHERE property_number = 7797
```

All combinations of current, sequenced, and nonsequenced queries over valid time and transaction time are easily expressed in SQL3.

Again, the result has but one column, and no implicit timestamps.

The orthogonality of valid-time and transaction-time support enables us to combine in arbitrary ways current, sequenced, and nonsequenced semantics for both kinds of time, resulting in nine temporal variants of every nontemporal query. Current in valid time translates in English to "at now"; sequenced translates to "at the same time"; and nonsequenced translates to "at any time." Current in transaction time translates to "as best known"; sequenced

translates to "when did we think"; and nonsequenced translates to "when was it recorded" or "when was it corrected."

We illustrate this by providing all variants of the query "What properties are owned by the customer who owns property 7797?", repeating the nine cases from Section 10.3.2. Unlike SQL-92, in which the variants can differ, sometimes dramatically, in SQL3 all the variants are identical, save their prefixes.

Case 1   **Valid-time current and transaction-time current**

Code Fragment 12.68   **What properties *are* owned by the customer who *owns* property 7797, *as best known*?**

```
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

(Compare with CF-10.26.)

Case 2   **Valid-time sequenced and transaction-time current**

Code Fragment 12.69   **What properties are *or were* owned by the customer who owned *at the same time* property 7797, as best known?**

```
VALIDTIME SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 3   **Valid-time nonsequenced and transaction-time current**

Code Fragment 12.70   **What properties were owned by the customer who owned *at any time* property 7797, as best known?**

```
NONSEQUENCED VALIDTIME SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 4   **Valid-time current and transaction-time sequenced**

Code Fragment 12.71   **What properties *did we think* are owned by the customer who owns property 7797?**

```
TRANSACTIONTIME SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 5   **Valid-time sequenced and transaction-time sequenced**

Code Fragment 12.72   **When did we think that some property, *at some time*, was owned by the customer who owned *at the same time* property 7797?**

```
VALIDTIME AND TRANSACTIONTIME SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 6   **Valid-time nonsequenced and transaction-time sequenced**

Code Fragment 12.73   **When did we think that some property, at some time, was owned by the customer who owned *at any time* property 7797?**

```
NONSEQUENCED VALIDTIME AND TRANSACTIONTIME
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 7   **Valid-time current and transaction-time nonsequenced**

Code Fragment 12.74   **When was it recorded that a property is owned by the customer who owns property 7797, as best known?**

```
NONSEQUENCED TRANSACTIONTIME SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 8   **Valid-time sequenced and transaction-time nonsequenced**

Code Fragment 12.75   **When was it recorded that a property is *or was* owned by the customer who owned *at the same time* property 7797?**

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT P2.property_number
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

Case 9   **Valid-time nonsequenced and transaction-time nonsequenced**

Code Fragment 12.76   **When was it recorded that a property was owned by the customer who owned *at some time* property 7797?**

```
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT P2.property_number, BEGIN(TRANSACTIONTIME(P2)) AS Recorded_Start
FROM Prop_Owner AS P1, Prop_Owner AS P2
WHERE P1.property_number = 7797
  AND P2.property_number <> P1.property_number
  AND P1.property_owner = P2.property_owner
```

We end with the remaining bitemporal queries presented in the Nykredit chapter, providing additional examples of combinations of valid and transaction time with current, sequenced, and nonsequenced queries.

Code Fragment 12.77   **What is the estimated value of the property at Bygaden 4 (current/current)?**

```
SELECT estimated_value
FROM Property AS P
WHERE P.address = 'Bygaden 4'
```

Code Fragment 12.78   **Who owns the property at Bygaden 4 (current/current)?**

```
SELECT name
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
```

(Compare with CF-10.36, at 14 lines.)

Code Fragment 12.79   **How has the estimated value of the property at Bygaden 4 varied over time (sequenced/current)?**

```
VALIDTIME SELECT estimated_value
FROM Property AS P
WHERE P.address = 'Bygaden 4'
```

Code Fragment 12.80   **Who has owned the property at Bygaden 4 (sequenced/current)?**

```
VALIDTIME SELECT name
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
```

Code Fragment 12.81   **When was the estimated value for the property at Bygaden 4 stored (current/nonsequenced)?**

```
NONSEQUENCED TRANSACTIONTIME
SELECT estimated_value, BEGIN(TRANSACTIONTIME(Property))
    AS Recorded_Start
FROM Property
WHERE address = 'Bygaden 4'
```

Code Fragment 12.82   **Who has owned the property at Bygaden 4, and when was this information recorded (sequenced/nonsequenced)?**

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT name, BEGIN(TRANSACTIONTIME(PO)) AS PO_Recorded,
    BEGIN(TRANSACTIONTIME(C)) AS C_Recorded,
    BEGIN(TRANSACTIONTIME(P)) AS P_Recorded_Start
FROM Prop_Owner AS PO, Customer AS C, Property AS P
WHERE P.address = 'Bygaden 4'
  AND P.property_number = PO.property_number
  AND C.customer_number = PO.customer_number
```

Code Fragment 12.83   **List all retroactive changes made to the** `Prop_Owner` **table (nonsequenced/nonsequenced).**

```
NONSEQUENCED VALIDTIME VALIDTIME(Prop_Owner)
    AND NONSEQUENCED TRANSACTIONTIME
SELECT customer_number, property_number,
    BEGIN(TRANSACTIONTIME(Prop_Owner)) AS Recorded_Start
FROM Prop_Owner
WHERE BEGIN(VALIDTIME(Prop_Owner))
        < BEGIN(TRANSACTIONTIME(Prop_Owner))
```

Following NONSEQUENCED VALIDTIME with a period expression in SQL3 renders the result a table with valid-time support.

Here we see an expression appearing after NONSEQUENCED VALIDTIME. This renders the result a table with valid-time support, with the specified period forming the period of validity of the row. The resulting table will have three columns, `customer_number`, `property_number`, and `Recorded_Start`; each row will also be associated with a valid timestamp.

12.10.2   Integrity Constraints

The following assertion is nonsequenced in valid time and current in transaction time:

Code Fragment 12.84   `Prop_Owner.property_number` **defines a contiguous valid-time history.**

```
CREATE ASSERTION P_O_Contiguous_History
NONSEQUENCED VALIDTIME CHECK (NOT EXISTS (SELECT *
  FROM Prop_Owner AS P, Prop_Owner AS P2
  WHERE END(VALIDTIME(P)) < BEGIN(VALIDTIME(P2))
    AND P.property_number = P2.property_number
    AND NOT EXISTS (
      SELECT *
      FROM Prop_Owner AS P3
      WHERE P3.property_number = P.property_number
        AND ((BEGIN(VALIDTIME(P3)) <= END(VALIDTIME(P)))
             AND (END(VALIDTIME(P) < END(VALIDTIME(P3))))
           OR ((BEGIN(VALIDTIME(P3)) < BEGIN(VALIDTIME(P2)))
             AND (BEGIN(VALIDTIME(P2)) <= END(VALIDTIME(P3)))))) )
           ))
```

SQL3 integrity constraints on bitemporal tables can be any combination of current, sequenced, and nonsequenced.

(Compare with CF-10.3.)

As with queries, assertions (and constraints and views) can also include with combinations of valid-time and transaction-time sequenced and nonsequenced prefixes. Compare with CF-10.44–10.46.

Code Fragment 12.85   **A customer who *owns* property 7797 shall *own* no other property (current/current).**

```
CREATE ASSERTION CHECK ( NOT EXISTS (
          SELECT P2.property_number
          FROM Prop_Owner AS P1, Prop_Owner AS P2
          WHERE P1.property_number = 7797
            AND P2.property_number <> P1.property_number
            AND P1.property_owner = P2.property_owner))
```

Code Fragment 12.86　**A customer who owned property 7797 shall *concurrently own* no other property (sequenced/current).**

```
CREATE ASSERTION VALIDTIME CHECK ( NOT EXISTS (
          SELECT *
          FROM Prop_Owner AS P1, Prop_Owner AS P2
          WHERE P1.property_number = 7797
            AND P2.property_number <> P1.property_number
            AND P1.property_owner = P2.property_owner))
```

Code Fragment 12.87　**A customer who owned property 7797 shall own no other property, *even at a different time* (nonsequenced/current).**

```
CREATE ASSERTION NONSEQUENCED VALIDTIME CHECK ( NOT EXISTS (
          SELECT P2.property_number
          FROM Prop_Owner AS P1, Prop_Owner AS P2
          WHERE P1.property_number = 7797
            AND P2.property_number <> P1.property_number
            AND P1.property_owner = P2.property_owner))
```

12.10.3　Modifications

Current modifications in SQL3 on bitemporal tables are identical to their nontemporal counterparts.

During modifications the DBMS provides the transaction time of facts, in contrast with the valid time, which is provided by the user. This derives from the different semantics of transaction time and valid time. Specifically, when a fact is (logically) deleted from a table with transaction-time support, its transaction-stop time is set automatically by the DBMS to the current time, "now." When a fact is inserted into the table, its transaction-start time is set by the DBMS, again to the current time. An update is treated, concerning the transaction timestamps, as a deletion followed by an insertion. The transaction times that a set of modification transactions give to the modified rows must be consistent with the serialization order of those transactions. Most importantly, though, the semantics of transaction time across modification statements is maintained automatically by the DBMS. No user intervention is required, and no mention of transaction time is indicated, or is even possible, in modifications of tables with transaction-time support, including bitemporal tables.

Current valid-time modifications are identical to their nontemporal versions.

Code Fragment 12.88　**Eva Nielsen buys the flat at Skovvej 30 in Aalborg on January 10, 1998.**

```
INSERT INTO Prop_Owner (customer_number, property_number)
VALUES (145, 7797)
```

(Compare with CF-10.4.)

Code Fragment 12.89 **Peter Olsen sells the flat on January 20, 1998.**

```
DELETE FROM Prop_Owner
WHERE property_number = 7797
```

Code Fragment 12.90 **Peter Olsen buys the flat on January 15, 1998, a current update.**

```
UPDATE Prop_Owner
SET customer_number = 827
WHERE property_number = 7797
```

The period of applicability for valid-time sequenced modifications in SQL3 is specified immediately following the VALIDTIME prefix.

A current modification carries with it an implied period of applicability of "now" to "forever." Sequenced modifications have an implied period of applicability of all of time: "beginning" to "forever." In the latter, though, the user can specify a different period of applicability, right after the VALIDTIME reserved word.

Code Fragment 12.91 **Eva actually purchased the flat on January 3, performed on January 23.**

```
VALIDTIME PERIOD '[1998-01-03 - 1998-01-10)'
INSERT INTO Prop_Owner (customer_number, property_number)
VALUES (145, 7797)
```

Code Fragment 12.92 **Eva actually purchased the flat on January 5.**

```
VALIDTIME PERIOD '[1998-01-02 - 1998-01-5)' DELETE FROM Prop_Owner
WHERE property_number = 7977
```

Code Fragment 12.93 **Peter actually purchased the flat on January 12.**

```
VALIDTIME PERIOD '[1998-01-12 - 1998-01-15)' UPDATE Prop_Owner
SET customer_number = 145
WHERE  property_number = 7797
  AND customer_number <> 145
```

Code Fragment 12.94 **Delete all records with a valid-time duration of exactly one week.**

```
NONSEQUENCED VALIDTIME DELETE Prop_Owner
WHERE INTERVAL(PERIOD(Prop_Owner)) = INTERVAL '7' DAY
```

12.10.4 Temporal Partitioning and Vacuuming

As has been emphasized before, temporal partitioning is a physical design decision, and as such has no impact on the expression of SQL3 queries, modifications, integrity constraints, and so on.

Vacuuming hasn't yet been specified for SQL3.

Internet Time	
New requirements inspire new clocks. Consider the netizen, trying deep into the night to connect via a chatroom with a correspondent halfway across the world. Via email, he had suggested a rendezvous time, expressed in her local time, but he was off by an hour, and they miss their opportunity. Swatch, the Swiss watch company, responded with the Webmaster, a watch (retailing at 100. SFr) that displays both the local time and *Internet Time*, based on *Biel Mean Time* (BMT). Internet Time is the same time the world over, so no conversions are necessary. He tells his cyber partner to check in	at 750 BMT, or more colloquially, at 750 Swatch Beats, which corresponds, on this crisp winter day, to 6 P.M. in Biel, Switzerland (Central European Wintertime), 10 A.M. in Tucson, Arizona (Mountain Standard Time), and 4 A.M. (Eastern Summer Time) in Sydney, Australia. Each day contains 1000 Swatch Beats, each 1 minute and 26.4 seconds long. The BMT meridian may be seen on the façade of the Swatch International Headquarters on Jakob-Staempfli Street in Biel.

12.11 CAPSTONE CASE

In this last case, Brad's feed yard application, everything comes together. This is actually the third time we're attacking this application. The first time was in Chapter 2, where we surveyed the major concepts and expressed queries and modifications in prose. In Chapter 11 we showed how to design temporal applications, using this case. Here, we reexpress the schema, queries, and modifications in SQL3, as yet another example of how these constructs simplify development of temporal applications.

12.11.1 Temporal Relational Schema

Section 11.1 emphasized a five-step methodology for database design: (1) perform conceptual design ignoring time, yielding a conventional ER schema, (2) add temporal annotations in prose, (3) map the conventional ER schema into a logical SQL schema, (4) apply the temporal annotations, modifying the logical schema along the way, and (5) finish with physical design, including temporal partitioning.

When using SQL3, we retain the first three steps and greatly simplify the fourth step. The fifth step is DBMS-dependent, as the SQL3 standard does not include constructs for physical schema specification.

We start with the output from the third step: the initial nontemporal logical schema shown in Figure 11.2. We then apply the temporal annotations, paralleling the material in Section 11.3.2.

User-Defined Time Attributes

Attributes whose type is period can use the SQL3 PERIOD type directly, rather than simulating these with two instant columns.

Entity Lifespans

To each table corresponding to an entity type for which the lifespan or valid time of an associated attribute is captured, we add an AS VALIDTIME clause. One entity type has a recorded lifespan: LOT.

Code Fragment 12.95   `LOT` **is a valid-time state table.**

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

Relationship Valid Time

To each table corresponding to a relationship type with a recorded valid-time extent, or having attribute(s) whose valid time is recorded, we add an AS VALIDTIME clause.

For tables corresponding to entity and relationship types for which valid time is to be recorded, use an AS VALIDTIME clause in SQL3.

As indicated in Table 11.1, the valid time of three relationship types, LOCATION, MOVE, and MASS_TRTMNT, is recorded. The LOCATION relationship type has a temporal extent; the other two are instantaneous relationship types. All three must use a period timestamp because SQL3 does not (yet) support event tables, that is, tables with an instant timestamp. Instead, a period of a single granule will be used.

For LOCATION and MOVE, the granularity is denoted as "Sub-DAY" in Table 11.1. The `MOVE_ORDER` column was used to differentiate multiple moves on a single day. As this is not possible in SQL3, we specify a granularity of MINUTE for the associated tables.

Code Fragment 12.96   `LOT_LOC`, `LOT_MOVE`, **and** `MASS_TRTMNT` **have valid-time support.**

```
ALTER TABLE LOT_LOC ADD VALIDTIME PERIOD(TIMESTAMP MINUTE)

ALTER TABLE LOT_MOVE ADD VALIDTIME PERIOD(TIMESTAMP MINUTE)

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

Valid Time of Attributes

Tables should be decomposed so that all attributes in a table have identical temporal support. No temporal decomposition is indicated here.

Transaction Time

TRANSACTIONTIME without a stated granularity may be used in SQL3.

For each table associated with an entity or relationship type for whose transaction-time periods are recorded, or that are associated with attribute(s) whose transaction-time periods are recorded, the AS TRANSACTIONTIME clause should be used. SQL3 provides the granularity for transaction time; it may not be specified by the user.

Code Fragment 12.97   LOT, LOT_MOVE, LOT_LOC, and BKP **are transaction-time tables.**

```
ALTER TABLE LOT ADD TRANSACTIONTIME

ALTER TABLE LOT_MOVE ADD TRANSACTIONTIME

ALTER TABLE LOT_LOC ADD TRANSACTIONTIME

ALTER TABLE BKP ADD TRANSACTIONTIME
```

As the LOT, LOT_MOVE, and LOT_LOC tables already had valid timestamps, adding transaction-time support renders them bitemporal.

Transaction-time support may induce additional temporal support decomposition.

As with valid time, we must also consider temporal support decomposition. Most of the attributes in LOT are bitemporal, but BKP_ID, A_NAME, DBF_NAME, and DBF_UPDATE_RECNO, because they were inherited from the CONTAINS relationship type, which records only transaction time, do not vary in valid time, and thus differ from the other attributes in that table in their temporal support. We move those attributes to a separate transaction-time table, LOT_CONTAINS, and include the primary key of LOT.

Code Fragment 12.98   **Move the** BKP_ID, A_NAME, DBF_NAME, **and** DBF_UPDATE_RECNO **columns into a separate transaction-time table.**

```
ALTER TABLE LOT DROP COLUMN BKP_ID
ALTER TABLE LOT DROP COLUMN A_NAME
ALTER TABLE LOT DROP COLUMN DBF_NAME
ALTER TABLE LOT DROP COLUMN DBF_UPDATE_RECNO

CREATE TABLE LOT_CONTAINS (FDYD_ID, LOT_ID_NUM, BKP_ID, A_NAME,
    DBF_NAME, DBF_UPDATE_RECNO,
  PRIMARY KEY (FDYD_ID, LOT_ID_NUM),
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) REFERENCES LOT
  )
AS TRANSACTIONTIME
```

Primary Keys

For tables with valid-time support, the primary key should be valid-time sequenced; for tables with transaction-time support, the primary key should be transaction-time sequenced.

In SQL3, the primary key should reflect the temporal support accorded the table.

The original primary key for these tables, which is interpreted as current/current, can be dropped if the table has some kind of temporal support. The collected result of these changes is shown in Figure 12.1.

Time-invariant primary keys are inherently nonsequenced.

The primary key for the BACKUP entity type is transaction-time invariant, which is translated into a NONSEQUENCED TRANSACTIONTIME PRIMARY KEY.

Referential Integrity

Referential integrity constraints are inherently sequenced. The prefix for FOREIGN KEY depends on the temporal support of both participating tables:

- *Both tables have valid-time support.* The prefix should be VALIDTIME, indicating a valid-time sequenced foreign key.
- *Only the referencing table has valid-time support.* The prefix should be NON-SEQUENCED VALIDTIME, indicating a valid-time nonsequenced foreign key, ignoring the timestamp of the referencing table.
- *The referencing table does not have valid-time support.* No prefix should be used, indicating a valid-time current foreign key, which applies to the current state of the referenced table, should it have valid-time support, and to the table as it is, should it not have valid-time support.

The same rules hold for transaction-time support. Generally, in SQL3, if both the referencing table and the referenced table have valid-time support, then VALID-TIME is indicated; if only the referencing table has valid-time support, then NON-SEQUENCED VALIDTIME is indicated. Transaction-time support is analogously handled.

Uniqueness Constraints

As with all integrity constraints, the presence of temporal support implies a sequenced constraint. The one exception is time-invariant integrity constraints, which are inherently nonsequenced. This may be seen in the uniqueness clause of the `LOT` table.

In SQL3, integrity constraints should reflect the temporal support accorded the table and be interpreted as sequenced. Time-invariant integrity constraints, which hold over all time, correspond to nonsequenced integrity constraints.

```
FDYD (FDYD_ID, NAME, FDYD_SHORT_NAME, FDYD_MNGR_LNAME, ...,
    PRIMARY KEY (FDYD_ID),
    UNIQUE (FDYD_SHORT_NAME)
)

CREATE TABLE PEN (FDYD_ID, PEN_ID, PEN_TYPE_CODE, BUNK_LENGTH,
        APRON_WIDTH, PEN_AREA, WATER_SPACE, BKP_ID,
    PRIMARY KEY (FDYD_ID, PEN_ID),
    FOREIGN KEY (FDYD_ID) REFERENCES FDYD,
    FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP
)

CREATE TABLE APPLICATION (A_NAME, A_DESCRIPTION, A_DATA_DIRECTORY,
    PRIMARY KEY (A_NAME)
)

CREATE TABLE DBF_FILE (A_NAME, DBF_NAME, DBF_DESCRIPTION, DBF_USED,
    PRIMARY KEY (A_NAME, DBF_NAME),
    FOREIGN KEY (A_NAME) REFERENCES APPLICATION
)

CREATE TABLE LOT (FDYD_ID, NAME, LOT_ID_NUM, LOT_ID, GNDR_CODE,
        PROJ_CLOSEOUT, IN_WEIGHT, VALID, OWNER, COMMENT, BKP_ID,
        A_NAME, DBF_NAME, DBF_UPDATE_RECNO,
    VALIDTIME AND TRANSACTIONTIME PRIMARY KEY (FDYD_ID, LOT_ID_NUM),
    FOREIGN KEY (FDYD_ID) NONSEQUENCED VALIDTIME
        REFERENCES FDYD,
    NONSEQUENCED VALIDTIME AND TRANSACTIONTIME
        UNIQUE (FDYD_ID, LOT_ID_NUM, LOT_ID)
    )
AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME

CREATE TABLE LOT_CONTAINS (FDYD_ID, LOT_ID_NUM, BKP_ID, A_NAME,
        DBF_NAME, DBF_UPDATE_RECNO,
    TRANSACTIONTIME PRIMARY KEY (FDYD_ID, LOT_ID_NUM),
    FOREIGN KEY (FDYD_ID, LOT_ID_NUM) TRANSACTIONTIME REFERENCES LOT,
    FOREIGN KEY (FDYD_ID, BKP_ID) REFERENCES BKP,
    FOREIGN KEY (A_NAME, DBF_FILE) REFERENCES DBF_FILE
    )
AS TRANSACTIONTIME

CREATE TABLE BKP (FDYD_ID, BKP_ID, YEAR_MONTH, DATE_PROCESSED,
        QUIRKS, VTRC_LAST_DATE_MOD, BRDR_LAST_DATE_MOD,
        ARCH_LAST_DATE_MOD,
```

Figure 12.1 SQL3 schema.

```
    NONSEQUENCED TRANSACTIONTIME PRIMARY KEY (FDYD_ID, BKP_ID),
    FOREIGN KEY (FDYD_ID) REFERENCES FDYD
    )
AS TRANSACTIONTIME

CREATE TABLE LOT_MOVE (FDYD_ID, LOT_ID_NUM, FROM_PEN_ID, TO_PEN_ID,
    HD_CNT, BKP_ID, A_NAME, DBF_NAME,
  VALIDTIME AND TRANSACTIONTIME PRIMARY KEY
    (FDYD_ID, LOT_ID_NUM, FROM_PEN_ID, TO_PEN_ID),
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) VALIDTIME AND TRANSACTIONTIME
    REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, FROM_PEN_ID) NONSEQUENCED VALIDTIME
    REFERENCES PEN (FDYD_ID, PEN_ID),
  FOREIGN KEY (FDYD_ID, TO_PEN_ID) NONSEQUENCED VALIDTIME
    REFERENCES PEN (FDYD_ID, PEN_ID),
  FOREIGN KEY (FDYD_ID, BKP_ID) NONSEQUENCED VALIDTIME
    REFERENCES BKP,
  FOREIGN KEY (A_NAME, DBF_FILE) NONSEQUENCED VALIDTIME
    REFERENCES DBF_FILE
    )
VALIDTIME PERIOD(TIMESTAMP MINUTE) AND TRANSACTIONTIME

CREATE TABLE LOT_LOC (FDYD_ID, LOT_ID_NUM, PEN_ID, HD_CNT,
    YEAR_MONTH,
  VALIDTIME AND TRANSACTIONTIME PRIMARY KEY
    (FDYD_ID, LOT_ID_NUM, PEN_ID),
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) VALIDTIME AND TRANSACTIONTIME
    REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, PEN_ID) NONSEQUENCED VALIDTIME
    REFERENCES PEN
    )
AS VALIDTIME PERIOD(TIMESTAMP MINUTE) AND TRANSACTIONTIME

CREATE TABLE MASS_TRTMNT (FDYD_ID, LOT_ID_NUM, PEN_ID,
    M_TRTMNT_AVG_WGT,
  VALIDTIME PRIMARY KEY (FDYD_ID, LOT_ID_NUM, PEN_ID),
  FOREIGN KEY (FDYD_ID, LOT_ID_NUM) VALIDTIME REFERENCES LOT,
  FOREIGN KEY (FDYD_ID, PEN_ID) NONSEQUENCED VALIDTIME
    REFERENCES PEN,
  FOREIGN KEY (FDYD_ID, BKP_ID) NONSEQUENCED VALIDTIME
    REFERENCES BKP
    )
AS VALIDTIME PERIOD(DAY)
```

Figure 12.1 (continued)

12.11.2 Queries

As we've seen from the previous case studies, queries on time-varying tables are *much* easier to express in SQL3 than in SQL-92. We continue this comparison with the queries in Section 11.7.1.

Code Fragment 12.99   **How many head of cattle from lot 219 in yard 1 are (currently) in each pen?**

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

Code Fragment 12.100   **Give the history of how many head of cattle from lot 219 in yard 1 were in each pen.**

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

Code Fragment 12.101   **How many head of cattle from lot 219 in yard 1 were, at some time, in each pen?**

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

Temporal joins are handled in the same way.

Code Fragment 12.102   **Which lots are currently coresident in a pen?**

```
SELECT DISTINCT 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
```

Code Fragment 12.103   **Which lots were in the same pen, perhaps at different times?**

```
NONSEQUENCED VALIDTIME
SELECT DISTINCT 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
```

Code Fragment 12.104   **Give the history of lots being coresident in a pen.**

```
VALIDTIME SELECT DISTINCT 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
```

Code Fragment 12.105   **Provide the state of the** LOT_CONTAINS **table on January 12, 1998.**

```
NONSEQUENCED TRANSACTIONTIME
SELECT LOT_ID_NUM, BKP_ID, A_NAME, DBF_NAME, DBF_UPDATE_RECNO
FROM LOT_CONTAINS AS L
WHERE TRANSACTIONTIME(L) OVERLAPS DATE '1998-01-12'
```

Code Fragment 12.106   **Provide the history of the** LOT **table as best known on March 15, 1998.**

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT LOT_ID_NUM, GNDR_CODE, PROJ_CLOSEOUT, IN_WEIGHT,
    VALID, OWNER, COMMENT
FROM LOT
WHERE TRANSACTIONTIME(LOT) OVERLAPS DATE '1998-03-15'
```

Code Fragment 12.107   **When were steerings scheduled (as opposed to being recorded after the fact)?**

```
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT S.LOT_ID_NUM, BEGIN(VALIDTIME(S)) AS When_Scheduled,
    BEGIN(TRANSACTIONTIME(S)) AS When_Recorded
FROM LOT AS C, LOT AS S
WHERE C.FDYD_ID = S.FDYD_ID
  AND C.LOT_ID_NUM = S.LOT_ID_NUM
  AND C.GNDR_CODE = 'c' AND S.GNDR_CODE = 's'
  AND VALIDTIME(C) MEETS VALIDTIME(S)
  AND BEGIN(TRANSACTIONTIME(S)) < BEGIN(VALIDTIME(S))
```

12.11.3 Modifications

We start with current modifications, which are unchanged from their nontemporal analogs.

Code Fragment 12.108   **Lot 433 arrives today.**

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

Code Fragment 12.109   **Lot 101 leaves the feed yard.**

```
DELETE FROM LOT
WHERE LOT_ID_NUM = 234
```

Code Fragment 12.110   **The cattle in lot 799 are being steered today.**

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

We now consider sequenced modifications.

Code Fragment 12.111   **Lot 426, a collection of heifers, was on the feed yard from March 26 to April 14.**

```
VALIDTIME INSERT INTO LOT
VALUES (426, 'h')
```

The following modification will apply regardless of whether LOT has transaction-time support.

Code Fragment 12.112   **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
```

Code Fragment 12.113   **Lot 799 was steered only for the month of March.**

```
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.

Nonsequenced modifications are also straightforward.

Code Fragment 12.114   **Delete the records of lot 234 that have duration greater than three months.**

```
NONSEQUENCED VALIDTIME DELETE FROM LOT
WHERE LOT_ID_NUM = 234
  AND INTERVAL(VALIDTIME(LOT) AS MONTH) > INTERVAL '3' MONTH
```

Code Fragment 12.115   **Correct the backup identifier for lot 433 to 37.**

```
UPDATE LOT_CONTAINS
SET BKP_ID = 37
WHERE LOT_ID = 433
```

12.12   MIGRATION

The potential users of the extensions to SQL3 just described are enterprises with applications that need to manage potentially large amounts of time-varying information. In the case studies we just covered, we tacitly assumed that we could code these applications in SQL3 from scratch. In reality, it is probable that these enterprises are already managing time-varying data using SQL-92 (more specifically, a particular variant provided by the DBMS vendor) and that the temporal applications are already in place and working. Indeed, the uninterrupted functioning of applications is likely to be of vital importance. The question then becomes how to recast the application to use the new SQL3 constructs. We provide an effective migration path by identifying four successively more general levels of queries and modifications.

In this section, we differentiate the temporal constructs proposed for SQL3 from the (massive) remainder of SQL3. To do so, we use the name of Part 7 of SQL3—SQL/Temporal—to denote the temporal constructs, and we use the name of Part 2—SQL/Foundation—to denote the nontemporal portion.

12.12.1   Upward Compatibility

Perhaps the most important aspect of ensuring a smooth transition is to guarantee that all application code without modification will work with the new system exactly with the same functionality as with the existing system.

To explore the relationship between nontemporal and temporal data and queries, we employ a series of figures that demonstrate increasing query and update functionality. In Figure 12.2, a conventional table is denoted with a rectangle. The current state of this table is the rectangle in the upper-right corner. Whenever a modification is made to this table, the previous state is discarded; hence, at any time only the current state is available. The discarded prior states are denoted with dashed rectangles; the right-pointing arrows denote the modification that took the table from one state to the next state.

When a query q is applied to the current state of a table, a resulting table is computed, shown as the rectangle in the bottom-right corner. While this figure only concerns queries over single tables, the extension to queries over multiple tables is clear.

Time
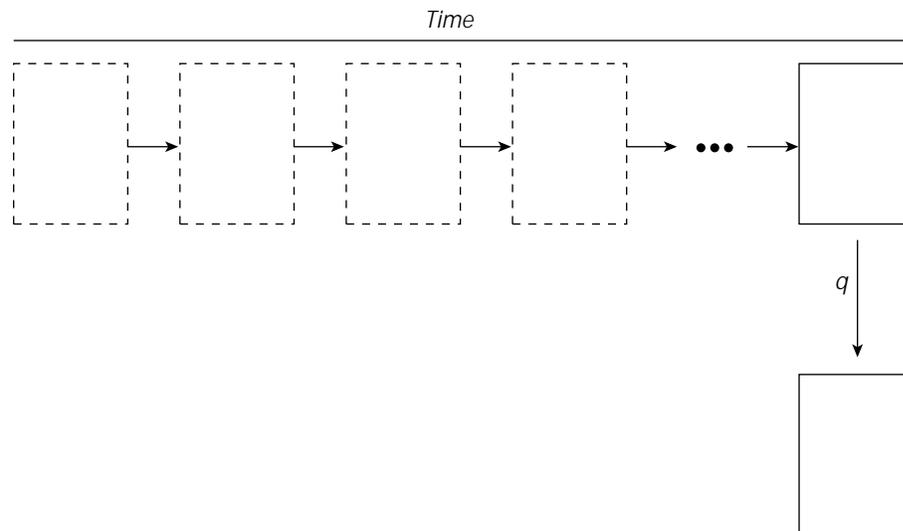
Figure 12.2 Evaluating an SQL/Foundation query over a table without temporal support, to return a table also without temporal support.

Upward compatibility states that (1) all instances of tables in SQL/Foundation are instances of tables in SQL/Temporal, (2) all SQL/Foundation modifications to tables in SQL/Foundation result in the same tables when the modifications are evaluated according to SQL/Temporal semantics, and (3) all SQL/Foundation queries result in the same tables when the queries are evaluated according to SQL/Temporal.

By requiring that the temporal constructs be a strict superset (i.e., only *adding* language constructs), it is relatively easy to ensure that the temporal constructs are upward compatible with SQL/Foundation.

12.12.2 Temporal Upward Compatibility

If an existing or new application needs support for the temporal dimension of the data in one or more tables, the table can be defined with or altered to add temporal support (e.g., by using the CREATE TABLE ... AS VALID or AS TRANSACTION or ALTER ... ADD VALID or ADD TRANSACTION statements). It is undesirable to be forced to change the application code that accesses the table without temporal support, when temporal support is added to that table. Previously we have mentioned a requirement that states that the existing applications on tables without temporal support will continue to work with no changes in functionality when the tables they access are altered to add temporal support. Specifically, *temporal upward compatibility* requires that each query will return the same result on an associated
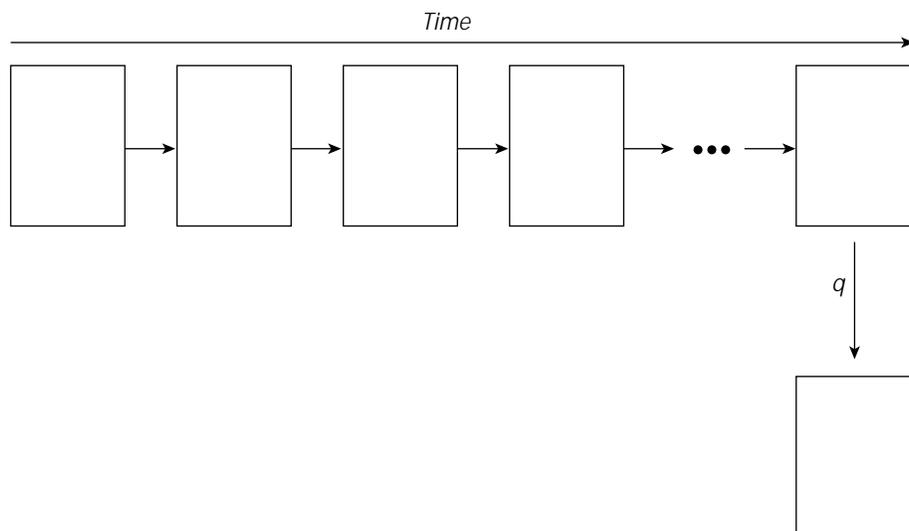
Figure 12.3 Evaluating an SQL/Foundation query over a table with temporal support, to return a table without such support.

snapshot database as on the temporal counterpart of the database. Further, this property applies to modifications to those tables with temporal support.

Temporal upward compatibility is illustrated in Figure 12.3. When temporal support is added to a table, the history is preserved, and modifications over time are retained. In this figure, the state to the far left was the current state when the table was made temporal. All subsequent modifications, denoted by the arrows, result in states that are retained, and thus are solid rectangles. Temporal upward compatibility ensures that the states will have identical contents to those states resulting from modifications of the table without temporal support.

The query q is an SQL/Foundation query. Due to temporal upward compatibility, the semantics of this query must not change if it is applied to a table with temporal support. Hence, the query only applies to the current state, and a table without temporal support results.

As an example from the University Information System, take the following SQL-92 query on a nontemporal INCUMBENTS table:

Code Fragment 12.116   **What is Bob's position?**

```
SELECT JOB_TITLE_CODE1
FROM EMPLOYEES, INCUMBENTS, POSITIONS
WHERE FIRST_NAME = 'Bob'
  AND EMPLOYEES.SSN = INCUMBENTS.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
```

After adding valid-time support to INCUMBENTS, this query is interpreted as a current query, returning Bob's current position (which is, after all, what the query did when evaluated on the nontemporal version of the INCUMBENTS table).

Temporal upward compatibility applies to all SQL/Foundation statements. Consider the following SQL-92 modification on a nontemporal table:

Code Fragment 12.117   **Bob was promoted to director of the Computer Center.**

```
UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

Temporal upward compatibility demands that this modification, when applied to a table with valid-time support, be interpreted as impacting the current state of that table, that is, that it be interpreted as a current modification, and so it is.

The same holds for tables with transaction-time support and for bitemporal tables.

Temporal upward compatibility at its core says this: Take an application that doesn't involve time, that concerns only the current reality. An example is an application storing the current job assignments of employees. Alter one or more of the tables so that they now have temporal support (valid-time, transaction-time, or both). The application should run as before, *without changing a single line of code*. This is an extremely powerful notion.

It is instructive to consider temporal upward compatibility in more detail. When designing information systems, two general approaches have been advocated. In the first approach, the system design is based on the *function* of the enterprise that the system is intended for (the "Yourdon" approach); in the second, the design is based on the *structure* of the reality that the system is about (the "Jackson" approach). It has been argued that the latter approach is superior because structure may remain stable when the function changes, but the opposite is generally not possible. Thus, a more stable system design, needing less maintenance, is achieved when adopting the second design principle. This suggests that the data needs of an enterprise are relatively stable and only change when the actual business of the enterprise changes.

When replacing a nontemporal system with a temporal system, that is, to utilize the constructs in SQL/Temporal, the enterprise is not changing its business, but wants the extra support offered by the temporal system for managing its temporal data. Thus, it is atypical for an enterprise to suddenly desire to record temporal information where it previously recorded only snapshot information. Such a change would be motivated by a change in the business.

The typical situation is rather more complicated. The nontemporal database system is likely to already manage temporal data, which is encoded using tables without temporal support, perhaps using the techniques discussed in early chap-

ters. When adopting the SQL/Temporal constructs, upward compatibility guarantees that it is not necessary to change the database schema or application programs. However, without changes, the benefits of the added temporal support are also limited. Only when defining new tables or modifying existing applications can the new temporal support be exploited. The enterprise then gradually benefits from the temporal support available in the system.

Nevertheless, the concept of temporal upward compatibility is still relevant for several reasons. First, it provides an appealing intuitive notion of a table with temporal support: the semantics of queries and modification are retained from tables without such support; the only difference is that intermediate states are also retained. Second, in those cases where the original table contained no historical information, temporal upward compatibility affords a natural means of migrating to temporal support. In such cases, not a single line of the application need be changed when the table is altered to be temporal. Third, conventional tables that do contain temporal information and for which temporal support has been added can still be queried and modified by conventional SQL/Foundation statements in a consistent manner.

12.12.3   Sequenced Extensions

The requirements covered so far have been aimed at protecting investments in legacy code and at ensuring uninterrupted operation of existing applications when achieving substantially increased temporal support. Upward compatibility guarantees that (nontemporal) legacy application code will continue to work without change when migrating, and temporal upward compatibility in addition allows legacy code to coexist with new temporal applications following the migration.

The requirement introduced in this section aims at protecting investment in programmer training and at ensuring continued efficient, cost-effective application development upon migration. This is achieved by exploiting the fact that programmers are likely to be comfortable with SQL.

Sequenced semantics states that SQL/Temporal must offer, for each query in SQL/Foundation, a temporal query that "naturally" generalizes this query, in a specific technical sense. In addition, we require that the SQL/Temporal query be syntactically similar to the SQL/Foundation query that it generalizes.

With this requirement satisfied, SQL/Foundation queries on tables with temporal support have semantics that are easily ("naturally") understood in terms of the semantics of the SQL/Foundation queries on tables without temporal support. The familiarity of the similar syntax and the corresponding, naturally extended semantics make it possible for programmers to immediately and easily write a wide range of temporal queries, with little need for expensive training.

Figure 12.4 illustrates this property. We have already seen that an SQL/Foundation query q on a table with temporal support applies the standard SQL3 semantics
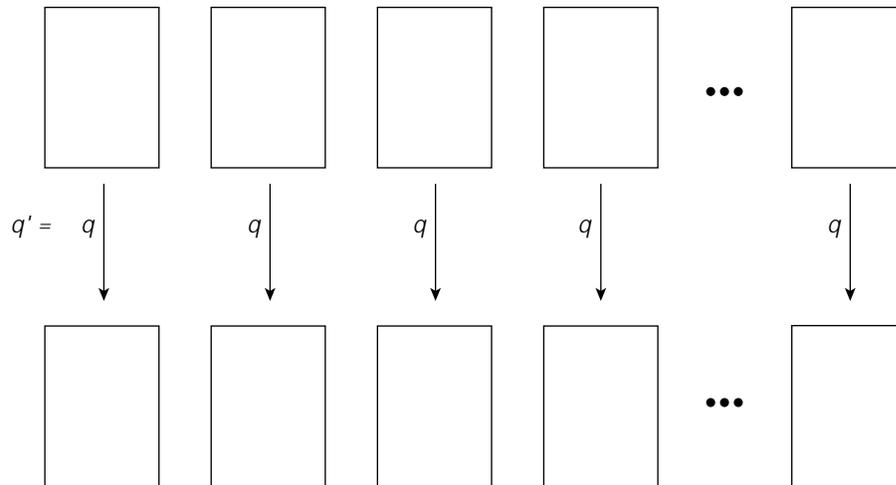
Figure 12.4 Evaluating a sequenced query over a table with valid-time support, to return a table with similar support.

on the current state of that table, resulting in a table without temporal support. This figure illustrates a new query, q', which is an SQL/Temporal sequenced query. Query q' is applied to the table with temporal support (the sequence of states across the top of the figure), and results in a table also with temporal support, which is the sequence of states across the bottom.

We would like the meaning of q' to be easily understood by the SQL/Foundation programmer. Satisfying sequenced semantics along with the syntactical similarity requirement makes this possible. Specifically, the meaning of q' is precisely that of applying SQL3 query q on each state of the input table (which must have temporal support), producing a state of the output table for each such application. And when q' also closely resembles q syntactically, temporal queries are easily formulated and understood. To generate query q', we need only prepend the reserved word VALIDTIME or TRANSACTIONTIME to query q.

As an example, let q be the following nontemporal query,

Code Fragment 12.118   **Provide the salary and department for all employees.**

```
SELECT S.SSN, AMOUNT, PCN
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
```

and q' be the sequenced variant of q.
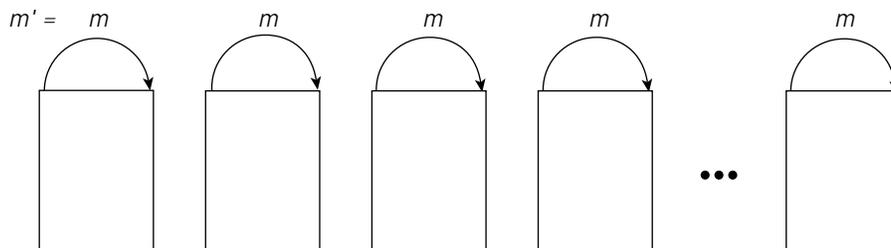
Figure 12.5 Evaluating an SQL/Temporal modification on a table with valid-time support.

Code Fragment 12.119   **Provide the salary and department history for all employees.**

```
VALIDTIME SELECT S.SSN, AMOUNT, PCN
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
```

One small addition was made to the English statement, "history," and one small addition was made to the SQL/Temporal statement, VALIDTIME.

These concepts also apply to sequenced *modifications*, illustrated in Figure 12.5. A valid-time modification destructively modifies states as illustrated by the curved arrows. As with queries, the modification is applied on a state-by-state basis. Hence, the semantics of the SQL/Temporal modification is a natural extension of the SQL/Foundation modification statement that it generalizes.

Elaborating on the example, the following nontemporal modification m, CF-12.120, can modify all times with a sequenced version, $m' = $ VALIDTIME m, CF-12.121, or with a sequenced version with a specified period of applicability, CF-12.122.

Code Fragment 12.120   **Bob was promoted to director of the Computer Center.**

```
UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

Code Fragment 12.121   **Bob was promoted to director of the Computer Center for all time.**

```
VALIDTIME UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```
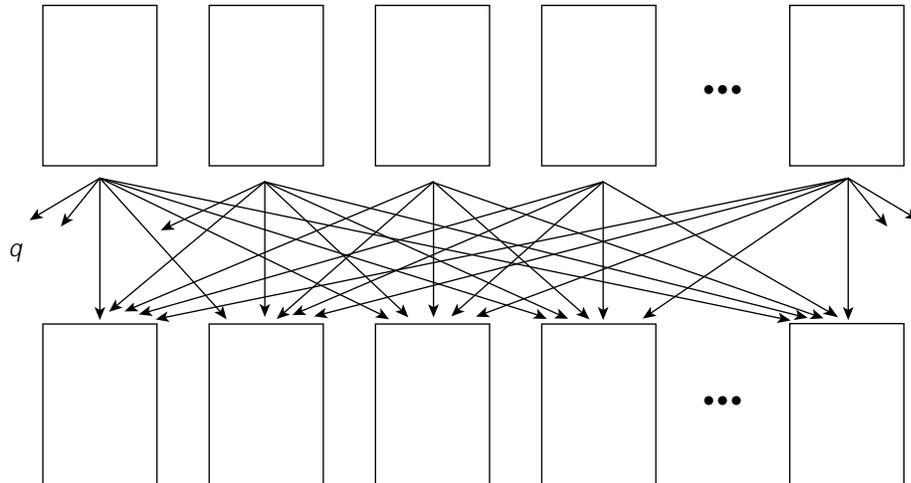
Figure 12.6 Evaluating a nonsequenced query over a table with valid-time support, to return a table with similar support.

Code Fragment 12.122   **Bob was promoted to director of the Computer Center for 1997.**

```
VALIDTIME PERIOD '[1997-01-01 - 1997-12-31]' UPDATE INCUMBENTS
SET PCN = 908739
WHERE SSN = 111223333
```

12.12.4   Nonsequenced Queries and Modifications

In a sequenced query, the information in a particular state of the resulting table with temporal support is derived solely from information in the state at that same time of the source table(s). However, there are many reasonable queries that require other states to be examined. Such queries are illustrated in Figure 12.6, in which each state of the resulting table requires information from possibly all states of the source table.

In this figure, two tables with temporal support are shown, one consisting of the states across the top of the figure, and the other, the result of the query, consisting of the states across the bottom of the figure. A single query q performs the possibly complex computation, with the information usage illustrated by the downward-pointing arrows. Whenever the computation of a single state of the result table may utilize information from a state at a different time, that query is nonsequenced. Such queries are more complex than sequenced queries, and they require a new construct in the query language, specifically, the NONSEQUENCED reserved word.
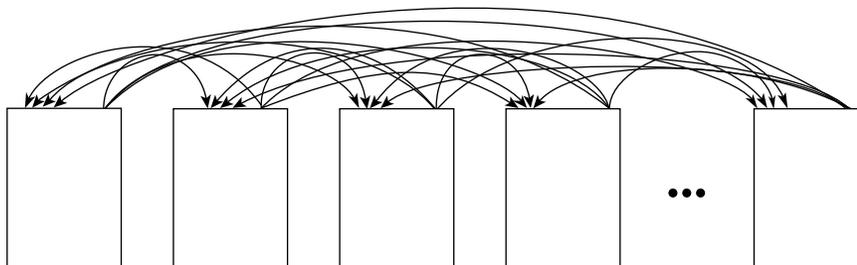
Figure 12.7 Evaluating a nonsequenced modification on a table with valid-time support.

Code Fragment 12.123   **List all the salaries, past and present, of employees who had been hazardous waste specialists at some time.**

```
NONSEQUENCED VALIDTIME SELECT AMOUNT
FROM INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE INCUMBENTS.SSN = SAL_HISTORY.SSN
  AND INCUMBENTS.PCN = POSITIONS.PCN
  AND JOB_TITLE_CODE1 = 20730
```

The phrases "past and present" and "at some time" indicate that the query is a nonsequenced one.

It is important to note that nonsequenced queries are very different from sequenced queries. In the latter, the query language is providing a temporal semantics; in the former, the query language interprets the timestamp as simply another column. For the user, this means that in nonsequenced queries (modifications, assertions, etc.) the period timestamps must be manipulated explicitly. The operations, such as join and relational difference, are performed with respect to the periods themselves, rather than on the individual states of the tables with temporal support. Reserved words are used to syntactically differentiate temporally upward compatible queries, sequenced queries, and nonsequenced queries, each of which applies a distinct semantics.

The concept of nonsequenced queries naturally generalizes to modifications. *Nonsequenced modifications* destructively change states, with information retrieved from possibly all states of the original table. In Figure 12.7, each state of the table with valid-time support is possibly modified, using information from possibly all states of the table before the modification. Nonsequenced modifications include future modifications. (We note in passing that nonsequenced modifications are only permitted on tables with valid-time support; all modifications are current in transaction time.)

In the nonsequenced modification of CF-12.124, the deletion of a particular state is dependent on the PCN being in the state on December 31, 1997.

Result of NORMALIZE:
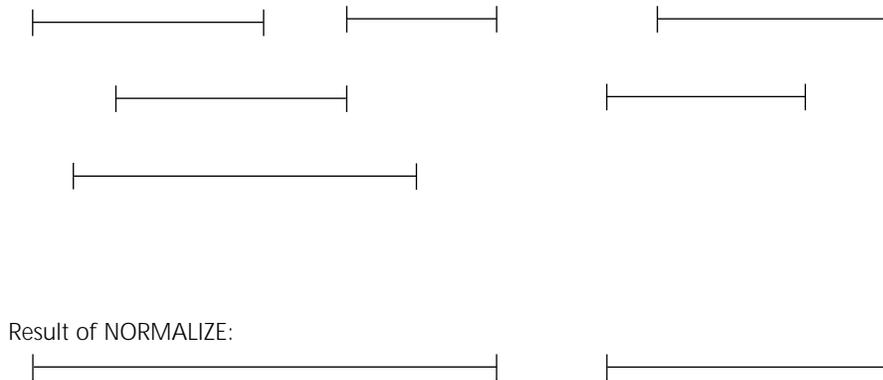
Figure 12.8 The NORMALIZE operator.

Code Fragment 12.124   **Delete Bob's records that include 1997 stating that he was associate director of the Computer Center.**

```
NONSEQUENCED VALIDTIME DELETE FROM INCUMBENTS
WHERE SSN = 111223333
    AND PCN = 999071
    AND VALIDTIME(INCUMBENTS) CONTAINS DATE '1997-12-31'
```

12.13 ADDITIONAL CONSTRUCTS OF SQL3*

A few additional constructs are defined in SQL/Temporal and are mentioned here for completeness.

A set constructor specialized to periods is included. The `NORMALIZE(pset)` operator, where `pset` is a set of periods, yields a potentially smaller set of periods that are disjoint, that is, they do not meet or overlap. Figure 12.8 illustrates the result of this operator on a set of periods. The six periods at the top comprise the input to the operator; the two periods at the bottom comprise the result. Effectively, the periods are coalesced with duplicate elimination. The parallel operation of coalescing while retaining duplicates is not provided.

An EXPAND operator, which converts a period into a SET of the period's element type, is also provided. `EXPAND(PERIOD '[1997-01-01 00:00:00 - 1997-12-31 00:00:00]')` would result in a value of type `SET(TIMESTAMP(0))` (i.e., to the granularity of second), with 31,536,000 elements (!). This set can be converted back to the original period via a new NORMALIZE ON construct of the select statement. This

construct is best explained with an example. The following statement EXPANDs a period column (`ATable.APeriod`), then normalizes it back into the original period.

Code Fragment 12.125 **Expanding each period into a set of granules, then normalizing back to a period.**

```
SELECT A, B, C, PERIOD[E3, E3] AS P
FROM (SELECT A, B, C, EXPAND(APeriod) AS E FROM ATable) AS E1,
    TABLE(E1.E) AS E2(E3)
NORMALIZE ON P
```

This rather complex query expands the `APeriod` column of `ATable`; the resulting table `E1` has a single timestamp column, `E`, of type `SET(TIMESTAMP(0))`. The second expression in the FROM clause makes a table out of each set; the value of the `E3` column for each row of `E2` is a particular timestamp value, such as `TIMESTAMP '1997-10-21 17:05:30'`. The SELECT clause converts each of these times into a period of duration one second. The NORMALIZE ON clause then normalizes the result by collecting the values of `P` for rows with identical values for the remaining columns (`A`, `B`, and `C`), applying NORMALIZE to this set of periods, then associating a copy of the row with each of the resulting periods. (Whew!) As we'll see later, this idiom can be used to implement certain types of temporal queries.

If the original table `ATable` contained duplicate rows, then this statement will normalize the periods associated with those rows, thereby removing sequenced duplicates. Say table `ATable` contained six rows with identical values for `A`, `B`, and `C` and with the periods shown in the top portion of Figure 12.8 as the value of the `APeriod` column. The result of the above statement would contain two rows, associated with the periods in the bottom portion of Figure 12.8.

Finally, an EXPANDING clause is added to UNION, EXCEPT, and INTERSECT, but only if ALL is not specified. This clause provides another way to effect a sequenced query. Thus, the following query, on a table *without temporal support*, rather with an explicit `When` column of type PERIOD, implements a sequenced EXCEPT (compare with CF-12.22, which replaces the `EXPANDING(When)` with the prefix VALIDTIME).

Code Fragment 12.126 **List the employees who are department heads but are not also professors (using EXPANDING).**

```
SELECT SSN
FROM INCUMBENTS
WHERE PCN = 455332
EXCEPT EXPANDING (When)
SELECT SSN
FROM INCUMBENTS
WHERE PCN = 821197
```

This is syntactic sugar for three steps: applying EXPAND to the timestamp, performing the operation, then normalizing the result.

Code Fragment 12.127 **List the employees who are department heads but are not also professors (using EXPAND and NORMALIZE).**

```
SELECT SSN, PERIOD[W2, W2] AS When
FROM (SELECT SSN, W2
        FROM (SELECT SSN, EXPAND(When) AS W
                FROM INCUMBENTS
                WHERE PCN = 455332) AS E1,
          TABLE(E1.W) AS E2(W2)
        EXCEPT
        SELECT SSN, W2
        FROM (SELECT SSN, EXPAND(When) AS W
                FROM INCUMBENTS
                WHERE PCN = 821197) AS E1,
          TABLE(E1.W) AS E2(W2)
        ) AS E3
NORMALIZE ON When
```

While this query, at 14 lines, is certainly shorter than CF-6.18, at 45 lines, it is *highly* inefficient as stated. Each row of INCUMBENTS is expanded into potentially millions of rows of E3, then normalized back to the result. And because NORMALIZE is used, duplicates are automatically removed. There is no parallel to EXCEPT EXPANDING ALL; instead, VALIDTIME ... EXCEPT ALL ... should be used. VALIDTIME lends a sequenced semantics to *any* query, and as such is preferred over the EXPANDING option, which is constrained to some uses of UNION, EXCEPT, and INTERSECT.

In summary, many uses of EXPAND and NORMALIZE are more easily performed by sequenced queries using VALIDTIME or TRANSACTIONTIME, which provides a sequenced semantics to any nontemporal query, view, modification, assertion, and cursor.

12.14 IMPLEMENTATION CONSIDERATIONS

SQL3 is in draft form, and vendors are still adding support for the SQL-92 standard; there is currently available (early 1999) no DBMS compliant at the Full SQL level. So understandably no DBMS yet supports the constructs just described.

An alternative strategy is to implement temporal support *outside* the DBMS, as middleware imposed between the application and the DBMS. The user issues temporal statements, which are translated into the variant of SQL supported by the underlying DBMS. This strategy has been employed in several research prototypes, as well as one commercially available product.

12.14.1 TIMEDB

TIMEDB is middleware that supports the constructs proposed for SQL3, including valid time, transaction time, queries, modifications, views, assertions, and constraints. It also supports user-specified valid-time coalescing. It doesn't support the EXPAND, EXCEPT EXPANDING, and NORMALIZE constructs described in Section 12.13, though as we saw there, those constructs can often be simulated in a more natural fashion with sequenced statements.

There are two quite separate versions of TIMEDB. Both accept the same language, but are distinct code bases.

- TIMEDB 1: This system runs as a front end to Oracle, accepts textual input, and utilizes the Oracle Call Interface (OCI) to evaluate the user requests. The system is implemented in Prolog. Three versions are available, running under SICStus Prolog, Macintosh SICStus Prolog, and SWI Prolog. The TIMEDB 1 source (about 10,000 lines of Prolog) is public domain and is included in the distribution. A main-memory emulator of OCI, also implemented in Prolog, allows the system to be used even if Oracle is not available. As SWI Prolog is available free for noncommercial use, TIMEDB 1 allows exploration of the SQL3 constructs at no cost.
- TIMEDB 2: This system is implemented in Java, provides a graphical user interface, and uses JDBC to communicate with the underlying DBMS. It has been tested with Oracle8 Server, Sybase's Adaptive Server Enterprise (Version 11.5), and Cloudscape's JBMS (Version 1.1). It is a commercial product from TimeConsult.

Both versions of TIMEDB were developed by Andreas Steiner.

12.14.2 TIGER

TIGER is another front end to Oracle supporting the SQL3 temporal constructs. The TIGER source (about 4000 lines of SWI Prolog code) is freely usable for educational and research purposes. The language supported is ATSQL, which is similar to that proposed for SQL3, including valid time, transaction time, queries, modifications, views, assertions, and constraints, with the following differences:

- The reserved words VALID and TRANSACTION replace VALIDTIME and TRANSACTIONTIME, respectively. The functions VTIME( ) and TTIME( ) replace VALIDTIME( ) and TRANSACTIONTIME( ).
- The INTERVAL( ) function is replaced with the DURATION( ) function.
- SEQUENCED is a new reserved word. If VALID is used as a prefix, either SEQUENCED or NONSEQUENCED is required in TIGER.
- There is a new SET VALID clause, permitting a wider range of expressions than that allowed in SQL3's VALIDTIME <expression> clause.

- User-specifiable coalescing, either on valid time, transaction time, or both, in either order, is supported.
- EXPAND, EXCEPT EXPANDING, and NORMALIZE are not supported.

TIGER was developed by a team directed by Michael Böhlen.

12.14.3 Synchrony

Synchrony is a data warehouse query tool developed by the if. . . software company.

A data warehouse is a collection of information, often extracted from multiple production databases, that supports sophisticated dimensional analysis tools. Data warehouses are always temporal: the fact table has a time attribute that is a foreign key to a time dimension table.

As an example, a data warehouse supporting sales information might have time, store, product, and customer dimension tables in a star schema, in which the fact table has a foreign key to each dimension table.

In a conventional data warehouse, the dimensions are themselves nontemporal, which limits their functionality. Suppose an unmarried customer buys a toaster in January. This would cause a row to be inserted into the fact table. If that customer later marries, her marital status would be updated in the customer dimension table. Since the customer table is nontemporal, the previous marital status is lost, and the information about the toaster sale is incorrect, as the warehouse implies that the toaster was bought by a married person.

Synchrony is notable in that it supports in a comprehensive fashion time-varying dimension tables. Synchrony provides powerful data loading and query facilities for such tables. Queries such as "What percentage of customers have lived in the same home for over three years, and what percentage of overall sales do they account for?" can be easily expressed.

Synchrony differs from the other systems described here in that it doesn't offer a temporal extension of SQL based on the relational model. Rather, Synchrony's query language is graphical and based on a star schema data model. However, Synchrony is similar to the other systems in that it extends a query language and data model to make it easier to express temporal queries, and it generates conventional SQL for evaluation by an underlying DBMS, thereby freeing the user from having to write these complex statements manually.

12.14.4 CD-ROM Materials

A wealth of material is available on the CD-ROM:

- Some 25 change proposals and expert contributions that have been submitted to the ANSI or ISO SQL3 committees are included. Of particular interest are the valid-time and transaction-time change proposals [92, 93].

- The full distribution of TIMEDB 1 is included, along with all source code and over 30 demos.
- A demonstration version of TIMEDB 2 is included as Java class files. The features missing in the demo version are views, assertions, constraints, update statements, delete statements, subqueries, and transaction-time support. A comprehensive manual is provided, as are over a dozen demos.
- Information, a thorough manual, and white papers on TIGER are included. TIGER may be run through a web browser at *cs.auc.dk/~tigeradm/*.
- A white paper and a guide containing several demos of Synchrony are available on the CD-ROM.
- Several reports discuss the technical challenges of implementing the SQL3 temporal constructs and provide some solutions.

12.15   SUMMARY

In SQL-92, users must "roll their own" temporal support in the application code. SQL3 instead provides explicit support for periods and valid and transaction time, thereby dramatically simplifying the code that must be written.

We first examined the PERIOD data type, introduced in SQL/Temporal. We then looked at the valid-time support available, specifically the VALIDTIME and NONSEQUENCED reserved words and the VALIDTIME function. Transaction-time support was introduced via the TRANSACTIONTIME reserved word and function. Bitemporal support is afforded by combining the orthogonal support for valid time and transaction time.

These minimal constructs were illustrated by rephrasing the code fragments of the previous case studies in SQL3. Table 12.2 compares the SQL/Temporal code fragments of this chapter with the analogous SQL-92 code fragments for four major case studies: Brad De Groot's feed yard application, Cheryl Bach's University Information System, Nigel Corbin's oil field application, and Jens Gadgaard's property ownership application. The code fragments in SQL-92 for these four applications totaled 1848 lines; only 520 lines of SQL3 were required to do exactly the same thing. These tables show that, over a wide range of data definition, query, and modification fragments, the SQL-92 version is *three* times longer in number of lines than the SQL3 version, and many times more complex. In fact, very few SQL3 fragments were more than 10 lines long; some fragments in SQL-92 comprised literally dozens of lines of highly complex code.

Table 12.2 Application development in SQL-92 and SQL3.

Operation	SQL-92 Fragment(s)	Lines of Code	SQL3 Fragment	Lines of Code
Valid-Time State Tables:				
adding valid-time support	5.4	2	12.3	1
sequenced primary key constraint	5.8	10	12.4	1
current uniqueness constraint	5.12	9	12.5	1
sequenced uniqueness constraints	5.14	7	12.6	1
referential integrity (referencing table is temporal)	5.19	1	12.7	2
current referential integrity (both tables are temporal)	5.20	12	12.7	2
current referential integrity (referenced table is temporal)	5.24	10	12.7	2
sequenced referential integrity (both tables are temporal)	5.21	32	12.9	2
nonsequenced referential integrity (both tables are temporal)	5.19	1	12.10	2
current join queries	6.2, 6.3	17	12.12, 12.13	11
current not exists query	6.4	7	12.14	5
valid time-slice query	6.5	7	12.15	6
sequenced selection query	6.6	3	12.16	3
sequenced projection query	6.7	2	12.17	2
sequenced sort queries	6.8, 6.9	3	12.18	3
sequenced union query	6.10	7	12.19	7
sequenced join query	6.11–*6.14*	7–29	12.20	3
sequenced nested query	6.18	45	12.21	7
sequenced except queries	6.18	45	12.22, 12.126	7
nonsequenced join queries	6.19, 6.20	10	12.23, 12.24	11
remove current duplicates	6.23	3	12.25	2
remove sequenced duplicates	6.24–*6.26*	19–30	12.26	2
remove nonsequenced duplicates	6.21	2	12.27	2
current insertions	7.1–*7.3*, 7.5	2–9	12.28	2
current deletions	7.7, 7.8	5, 10	12.29	3
current updates	7.10, 7.11, 7.22	9–30	12.30, 12.36	3–6
sequenced insertion	7.12–*7.14*	2–28	12.31	2
sequenced deletion	7.16	24	12.32	3
sequenced updates	7.18, 7.24	27–77	12.33, 12.37 12.122	3–6
nonsequenced deletion	7.19	5	12.34	4
nonsequenced update	7.20	4	12.35	5
partitioned current join query	7.25	5	12.38	5

Table 12.2 (continued)

Operation	SQL-92 Fragment(s)	Lines of Code	SQL3 Fragment	Lines of Code
Valid-Time State Tables (continued):				
partitioned sequenced join	7.26	50	12.39	3
partitioned current insertions	7.27, 7.28	2, 6	12.40	2
partitioned current deletion	7.29	8	12.41	3
partitioned sequenced deletion	7.30	44	12.42	3
partitioned sequenced update	7.31	57	12.43	3
Transaction-Time Tables:				
adding transaction-time support	8.1+8.2, 8.12, 8.15, 9.1+9.2, 9.15+9.17, 9.20+9.21	17–44	12.44	1
current query	8.4	3	12.45	3
extracting a prior state	8.3, 8.13, 8.14, 8.16, 9.7, 9.18	4–19	12.46	4
prior state as a view	8.5, 8.17	17–27	12.47	7
current state as a view	8.20, 9.3 9.16, 9.19, 9.22	0–11	—	0
converting to a state table	8.7	38	—	—
converting to an event table	9.12–*9.14*	4–26	12.48, 12.51–*12.53*	6–27
sequenced selection queries	8.8, 9.8	3–4	12.49	3
sequenced union query	9.9	8	12.57	7
sequenced join query	9.10	8	12.58	5
sequenced self-join query	8.18	13	12.50	5
nonsequenced query	9.11	10	12.59	10
current insertions	8.9, 9.4	4–6	12.54	4
current deletions	8.10, 9.5	2–4	12.55	2
current updates	8.11, 9.6	3–14	12.56	3
entity vacuum	9.23	4	—	—
temporally vacuum	9.24–*9.26*, 9.28	2–6	—	—
log the vacuuming operations	9.27	8	—	—
Bitemporal State Tables:				
adding bitemporal support	10.1, 10.52	7–17	12.60	4
sequenced/sequenced primary key	10.2	13	12.61	2
current/current integrity constraint	10.44	12	12.85	6

Table 12.2 (continued)

Operation	SQL-92 Fragment(s)	Lines of Code	SQL3 Fragment	Lines of Code
Bitemporal State Tables (continued):				
sequenced/current integrity constraint	10.45	10	12.86	6
nonsequenced/current integrity constraint	10.46	8	12.87	6
sequenced/current referential integrity	10.49, 10.50	18–48	12.62	2
contiguous history assertion	10.3	16	12.84	14
transaction time-slice	10.19	5	12.63	5
current transaction time-slice	10.20	4	12.64	3
valid time-slice	10.21	5	12.65	5
bitemporal time-slice	10.22	7	12.66	6
current bitemporal time-slice	10.23	6	12.67	3
current/current queries	10.26, 10.35	16	12.68, 12.77	8
sequenced/current queries	10.27, 10.37	16	12.69, 12.79	8
nonsequenced/current query	10.28	7	12.70	5
current/sequenced query	10.29	14	12.71	5
sequenced/sequenced query	10.30	15	12.72	5
nonsequenced/sequenced query	10.31	10	12.73	6
current/nonsequenced queries	10.32, 10.39	13	12.74, 12.81	9
sequenced/nonsequenced query	10.33	12	12.75	6
nonsequenced/nonsequenced queries	10.31, 10.41	9	12.73, 12.83	13
current/current multiway join query	10.36	14	12.78	1
sequenced/current join query	10.38	11	12.80	5
sequenced/nonsequenced multiway join query	10.40	12	12.82	8
current insertion	10.4	4	12.88	2
current deletion	10.10, 10.11	13–19	12.89	2
current update	10.7	34	12.90	3
sequenced insertion	10.12, 10.13	4–36	12.91	3
sequenced deletion	10.15	42	12.92	2
sequenced update	10.17	58	12.93	3
nonsequenced deletion	10.18	4	12.94	2
reconstitute bitemporal state table as a view	10.53–*10.55*	8–51	—	0
temporally vacuum	10.56	8	—	—
Capstone Case:				
schema definition	*Fig.* 11.2, 11.1–*11.20*, 11.22, 11.23	245	*Fig.* 12.1	82

Table 12.2 (continued)

Operation	SQL-92 Fragment(s)	Lines of Code	SQL3 Fragment	Lines of Code
Capstone Case (continued):				
current queries	11.24, 11.28	11	12.99, 12.102	8
sequenced queries	11.25, 11.30 or 11.31	20–36	12.100, 12.104	7
nonsequenced queries	11.26, 11.29	8	12.101, 12.103	8
nonsequenced/nonsequenced query	11.34	9	12.107	9
extracting a prior state	11.32, 11.33	9	12.105, 12.106	15
current modifications	11.35, 11.37, 11.42	68	12.108–*12.110*, 12.115	10
sequenced modifications	11.38–*11.40*, 11.43	149	12.111–*12.113*, 12.112	12
nonsequenced modification	11.41	10	12.114	3

To recap, five requirements 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, are retained. This implies, for example, that a conventional query on tables with temporal support should be interpreted as a current query. Upward compatibility and temporal upward compatibility guarantee that legacy application code needs no modification when migrating and that new temporal applications may coexist with existing applications. They are thus aimed at protecting investments in legacy application code.
4. Sequenced variants should be easy to express for *all* constructs of the language, including queries, modifications, views, assertions and constraints, and cursors. This requirement ensures that the extended query language is easy to use for programmers familiar with the existing query language, thus helping to protect investments in programmer training. 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 SQL/Temporal 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 time-slices will be correct. Either kind of time can be used individually or together, forming a bitemporal table.
2. The temporal constructs of SQL/Temporal are 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 (respectively, modifications, etc.). As an example, when valid-time support was added to the `INCUMBENTS` table, the existing code of this application, perhaps tens of thousands of lines, did not require a single change.
4. An SQL/Foundation query can be converted to a sequenced query in SQL/Temporal 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 the case studies throughout this chapter, these proposed constructs (three new reserved words, VALIDTIME, TRANSACTIONTIME, and NON-SEQUENCED, 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 three or more, while improving the comprehensibility of that code.

Finally, we examined two prototypes (TIMEDB 1 and TIGER) and a commercial product (TIMEDB 2) that support these constructs, as well as another product, Synchrony, which supports time-varying dimensions in a data warehouse.

12.16   READINGS

SQL-86 and SQL-89 had 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 [87].

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 [91].

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 in Ottawa in July 1995 as Part 7 of the SQL3 draft standard. Jim Melton agreed to edit this new part [69]. There is presently a two-year experiment (ending in 2000) to allow drafts to be visible to the public. Hence, this draft may be viewed at *ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sqltmprl.ps*. PDF and ASCII text versions are also available.

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 [92, 93]. 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 32/WG 3). The full story may be found at *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 in 1999 [30]. At that time, the committees will revisit the other parts and move them through the exhaustive process towards standardization.

Michael Böhlen and Robert Marti introduced the notion of *temporal semicompleteness* as a (highly desirable) property of a temporal query language [14]. Translated into our terminology, temporal semicompleteness says that a sequenced query should be a syntactic augmentation of the associated nontemporal query, with the augmentation consisting solely of constructs added before and after the query. SQL3 is thus temporally semicomplete because a nontemporal query can be rendered sequenced by simply prepending VALIDTIME. The initial design of this construct appeared in 1995 [15].

Transitioning from SQL-92 to SQL3 is covered in depth by John Bair et al. [3] and by the author and others [94]. System design based on the function of the enterprise is advocated by Edward Yourdon [106]; system design based on the structure of the reality that the system is about has been elaborated by Michael Jackson [46].

TIMEDB was developed by the TimeConsult software company under the direction of Andreas Steiner. TimeConsult provides consulting in handling temporal data in relational, object-relational, and object-oriented DBMSs; implements temporal database applications; and supports their TIMEDB 2 product. For more information, see their Web site at *www.timeconsult.com* or contact Andreas Steiner at *steiner@timeconsult.com*.

TIGER was developed by a team directed by Michael Böhlen at the Department of Computer Science, Aalborg University, Denmark. He can be reached at *boehlen@cs.auc.dk*; his Web page is *www.cs.auc.dk/~boehlen/*. The TIGER Web page is *cs.auc.dk/~tigeradm/*, which includes an online demo. The Prolog code for both TIGER and TIMEDB1 resembles that developed in the early 1990s to support ChronoLog [10]. TIGER supports the ATSQL language [12].

Synchrony is analytic software developed by the if…software company. if…develops analytic software designed for business analysts and decision makers to examine in-depth relationships across multiple aspects of their businesses. For more information, contact if…at 510-864-3480 or *www.iftime.com*.

The Swatch Webmaster and BMT are described at *www.swatch.ch*, clicking on ".beat".
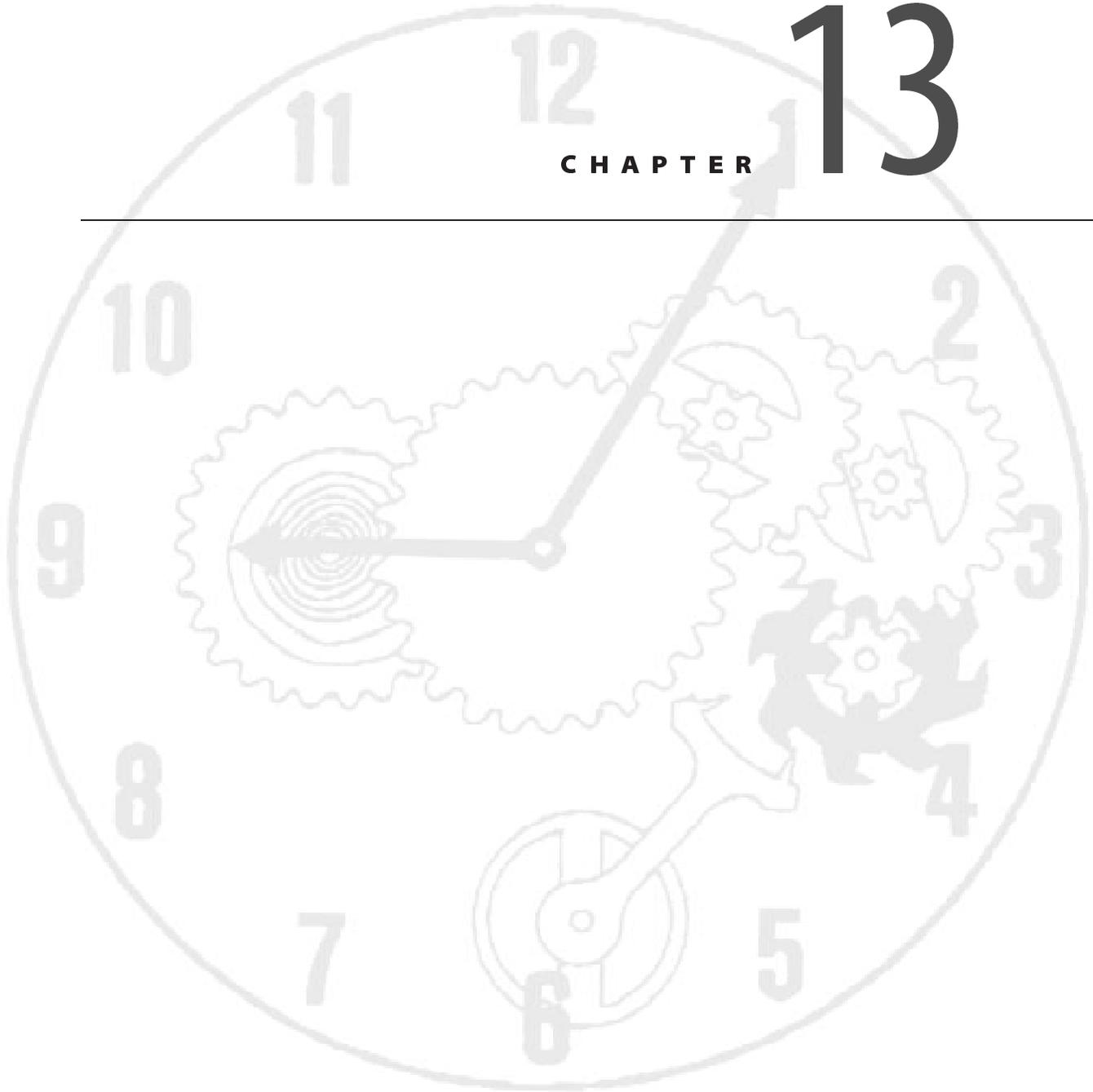
Prospects

So we see that development of time-varying database applications in SQL is possible, but is devilishly difficult. There are proposals before the SQL3 committee for new language constructs that help immensely. How long will we have to wait until such facilities are widely available?

In the near term, developers of time-oriented applications have three choices. The easiest option is to minimize the use of time in the application, reducing the complexity of the SQL needed, as well as the functionality of the delivered application. The second option is to use the concepts and techniques introduced in this book to develop powerful and correct time-oriented applications. Of course, the SQL code required will often be daunting. The last option is to use third-party software to translate temporal queries (expressed in the SQL/Temporal syntax, or a related syntax) into SQL-92. Such translators are already available commercially, and as the market for such middleware matures, their number and sophistication will grow.

In the longer term, vendors will start integrating temporal support into the DBMS itself. Initially, this will be done by simply imposing a middleware translator between the application and the DBMS. While this is architecturally identical to the third option just described, the perception will be that the DBMS itself has temporal support, an important marketing distinction. The vendor can then gradually move the functionality from the middleware component to the internals of the DBMS, retaining the temporal language but gaining in efficiency, with the timing of this transfer from middleware to DBMS internals based on market demand.

Unlike other constructs, such as triggers and object-relational extensions that are now de rigueur for commercial DBMS offerings, a standard for temporal support was proposed *before* such support was implemented in a product. When a middleware or DBMS vendor decides to implement temporal support, the language of choice should be something similar to what has been proposed (and summarized in the previous chapter). The ever-present danger, existing in previous extensions

and to which temporal extensions are susceptible, is that vendors will attempt to impose proprietary language constructs. We've already seen that occur with the basic temporal types, with many vendors basically ignoring the SQL-92 standard data types of DATE, TIME, and TIMESTAMP. We can only hope that when support for valid time and transaction time is added, the vendors utilize the mature language constructs already designed and refined by the SQL3 standards committees.

Temporal extensions will not occur in a vacuum—they must be integrated with other facilities now being added or considered for the future. Fortunately, the requirements of upward compatibility, temporal upward compatibility, and sequenced support ensure that whatever constructs are later added will be usable in current, sequenced, and nonsequenced queries, views, modifications, assertions, and cursors over tables with temporal support.

Predicting when the transition from third-party middleware solutions to integrated temporal DBMS offerings will occur is difficult. I estimate that the first will appear by the year 2000. As the benefits of such support become clear and acknowledged (the readers of this chapter are already aware of these advantages), pressure will mount on the remaining DBMS vendors to follow suit. Very quickly thereafter, a phase shift will occur, with temporal support along the lines proposed for SQL3 appearing in *all* DBMS offerings, as such support becomes a mandatory requirement.

Glossary

This glossary is an extension of the official one, which is in two parts, general concepts [49] and time granularity concepts [7].

The page in which the term is defined is indicated in parentheses.

A.D.   *anno Domini*, "in the year of our Lord".

after-image   The new value of a modified row or column.

A.H.   *anno Hegirae*, the first year of the Hijri (Islamic) calendar.

A.M.   (1) *annus mundi*, or "year of the world". 6000 A.M. started October 27, 1997. (2) *ante meridiem*, or "before noon."

append-only   A property of tables with transaction-time support, in which changes are implemented only as insertions, or changing the transaction-stop time to "now," so that the changes always accummulate in the table.

archival store   The component of a temporally partitioned transaction-time state table containing rows that have been corrected, that is, rows with a transaction-time stop date before "now".

as-of date   The (transaction-time) date for which a prior state of a monitored table is to be reconstructed.

atom   There are 22,500 atoms in an hour, so a second is exactly 6 1/4 atoms.

atomic second   9,192,631,770 periods of the radiation emitted by the transition between two particular hyperfine states of the cesium 133 atom in the ground state.

A.U.C.   *ab urbe condita*, or "from the foundation of the city". A.U.C. was the prevailing year numbering system before the use of B.C.–A.D. 1 A.D. is 754 A.U.C.

backlog   A tracking log with the modification operation explicitly identified.

B.C.   An acronym for "before Christ".

B.C.E.   The preferred term for B.C.; an acroymn for "before the Christian Era" or "before the Common Era".

before-image   The previous value of a modified row or column.

bitemporal table   Having support for both valid time and transaction time.

bitemporal time diagram   A two-dimensional graphical notation of the information content of a bitemporal table.

bitemporal time-slice   A query or view that selects the snapshot state of a bitemporal table at specified valid-time and specified transaction-time instants.

BMT   Biel Mean Time, based on the meridian for Internet Time in Biel, Switzerland.

B.P.   An acronym for "before the present".

C.E.   The preferred term for A.D.; an acronym for "Common Era".

chronology   The science of timekeeping. *See also* horology.

clepsydra   A clock powered by the flow of water.

coalesce   Reduce the number of rows in a table by merging the periods of validity of value-equivalent rows.

comparable   In SQL-92, indicates whether values of two types can be compared.

current duplicate   Two rows that are sequenced duplicates in the current state.

current insertion   A row that became valid now.

current integrity constraint   Must hold only for the current state.

current join   A join over the current states of the two argument tables.

current modification   Concerns a change that occurred now.

current store   The component of a temporally partitioned state table containing rows that are still current, that is, rows with a stop date of "now".

current time-slice query   Expressed on the current state of the table or database.

current update   Changes a fact now.

current valid time-slice query   Expressed on the current valid-time state of the table or database.

DATE   An SQL-92 data type, with a granularity of day. Supported by IBM DB2 UDB; by Informix–Universal Server; by Oracle8 Server, with a granularity of second; and by UniSQL, also with a granularity of second.

datetime   An SQL-92 instant data type or value.

DATETIME   A data type supported by Informix–Universal Server, with a specifiable precision; by Microsoft Access, with a granularity of slightly less than one millisecond; by Microsoft SQL Server, with a granularity of 1/300 of a second; and by Sybase SQLServer, also with a granularity of 1/300 of a second.

day-time interval   An SQL-92 interval data type containing only the day, hour, minute, and second fields, with an optional fractional seconds.

degenerate specialization   A bitemporal entity type, relationship type, or table in which the beginning of the valid-time extent (the entity's lifespan) exactly corresponds to the beginning of the transaction time.

dirty read   An isolation level that allows transactions to read uncommitted values.

entity vacuuming   Purging entities that are judged to be less interesting from the archival store.

ephemeris second   1/31,556,925.9747 of the period of the tropical year between the vernal equinoxes of 1899 and 1900.

equation of time   The difference between mean solar time and true solar time.

equinoctial day   A day for which the daylight is equal in duration to the night, that is, March 21 or September 21.

escapement   The mechanism within a mechanical clock that regulates the advancement of the hands, while subtly imparting energy to the oscillator, to keep the clock running.

event   An instantaneous fact, that is, something occurring at an instant in the modeled reality.

fully general   A bitemporal entity type, relationship type, or table that exhibits no coupling between its valid and transaction timestamps.

gnomon   The pointer portion of a sundial.

gnomonics   The science of sundials. *See also* chronology.

granularity   A partitioning of the time line, for example, years, days, microseconds.

granule   A specific unit element of a granularity.

heliochronometer   A sundial.

hemicyclium   A truncated, and thus lighter, hemispherium.

hemispherium   Berossos's sundial, made by hollowing out a half-sphere in a rectangular block of stone.

history store   The component of a temporally partitioned state table containing rows that were valid in the past, that is, rows with a stop date before "now".

horology   The science of time measurement, and the art of constructing instruments that indicate time.

hour   1/24th of a mean solar day.

instant   A time point on an underlying time axis, or equivalently, an anchored location on that axis.

Internet Time   A day in Internet Time starts at midnight BMT and contains 1000 Swatch Beats.

interstate integrity constraint   A constraint applied across states of the table. *See also* nonsequenced integrity constraint.

interval   An unanchored contiguous portion of the time line.

INTERVAL   An SQL-92 data type, with a user-specifiable granularity. *See also* day-time interval *and* year-month interval. Supported by Informix–Universal Server.

intrastate integrity constraint   A constraint applied independently to each state of the table. *See also* sequenced integrity constraint.

jiffy   In the United States and Canada, 1/60th of a second; in most other places, 1/50th of a second, though 10-millisecond jiffies (1/100th of a second) are becoming more common.

k'o   Roughly a quarter hour, indicated by Chinese clepsydrae: each 14 minutes and 24 seconds long.

labeled duration   An IBM DB2 UDB construct consisting of a numeric expression followed by a time unit, singular or plural.

lifespan   The valid-time extent of an entity or of a relationship in the ER model.

lunar day   The average interval between successive moonrises.

mean time   The time calculated by a fictitious earth traversing at a constant speed around the sun.

millennium bug   A catch phrase for the year 2000 problem.

monitored table   A table for which a tracking log is maintained.

monotonic vacuuming specification   Repeated application of the vacuuming does not violate the specification.

nonsequenced duplicate   Rows with identical values for their timestamp and nontimestamp columns.

nonsequenced integrity constraint   Treats the timestamp as just another column.

nonsequenced modification   Treats the row's timestamp as just another column.

nonspecialized   *See* fully general.

nontemporal entity type   An entity type for which the lifespan is not recorded.

nontemporal relationship type   A relationship type for which the valid time is not captured.

ost   There are 60 osts (*ostenta*) to the hour, so an ost is equivalent to a minute.

period   The time between two instants, or equivalently, an anchored duration of the time line.

PERIOD   An SQL3 data type constructor, with a user-specified granularity.

period of applicability   The period over which a sequenced modification applies.

period of presence   The (transaction-time) period for which a row was logically resident in the table.

period of validity   The period for which a fact represented by a row is valid in the modeled reality.

P.M.   *post meridiem*, or "after noon."

point   There are five points (*puncta*) to the hour.

position   In SQL-92, the number of characters from the character set SQL_TEXT that it would take to represent any value of that type.

postactive modification   A modification in which the period of applicability is after the transaction-start time.

postactive specialization   A bitemporal entity type, relationship type, or table in which the beginning of the valid timestamp is always after or equal to the beginning of the transaction timestamp.

precision   In SQL-92, the number of fractional digits allowed in the value of the associated data type.

precision decomposition   Columns with a finer granularity than that of the table they are associated with are placed in a separate table, with a timestamp of that finer granularity.

proleptic Gregorian calendar   The Gregorian calendar extrapolated backwards to 1 C.E.; used in SQL-92.

reconstruct   Retrieve a prior state of a monitored table, as of a specified date.

retroactive modification   A modification in which the period of applicability is before the transaction-start time.

retroactive specialization   A bitemporal entity type, relationship type, or table in which the beginning of the valid timestamp is equal to or before the beginning of the transaction timestamp.

sequenced deletion   Removes rows that became invalid now.

sequenced duplicate   Rows that are duplicates at some instant.

sequenced integrity constraint   Is applied independently at each point in time.

sequenced join   Join two tables by joining, at each point in time, the corresponding states of the argument tables.

sequenced primary key   A constraint stating that the specified columns constitute a primary key in every state.

sequenced update   Occurred independently in each state, over the period of applicability of the update.

serialization   An isolation level in which only committed values may be read.

sidereal day   The time between two meridian transits of a star, 23 hours and 56 minutes of true solar time.

sidereal second   1/86,400th of a sidereal day.

SMALLDATETIME   A Microsoft SQL Server data type, with a granularity of minute, and a Sybase SQLServer data type, also with a granularity of minute.

snapshot equivalent   Two rows or tables in which all of their snapshots, resulting from a time-slice at an instant of time, are identical.

snapshot of a table   The rows of that table valid at (or within the period of presence of, for a transaction-time table) a specified instant.

snapshot table   A table that is not timestamped with either valid time or transaction time.

solar day   *See* true day.

Sothic cycle   When the lunar New Year coincides with the solar New Year in the Egyptian calendar, once every 1460 years.

Swatch Beat   One-thousandth of a solar day, of length 1 minute 26.4 seconds.

temporal   As a modifier, used to indicate that the modified concept concerns some aspect of time.

temporal constructor   An expression that returns a temporal value.

temporal specialization   Denotes the coupling of the valid and transaction timestamps of a bitemporal entity type, relationship type, or table.

temporal support decomposition   Columns that differ in their temporal support from the enclosing table are placed in a separate table.

temporal table   A table supporting valid time or transaction time.

temporal upward compatibility   An SQL-92 statement (query, modification, view, assertion, constraint) will have the same effect on an associated snapshot database as on the temporal counterpart of the database.

temporal vacuuming   Purging old information from the archival store.

temporally partitioned table   A state table that is represented physically as two or more tables, with the criterion of which underlying table a row resides based on the temporal extent of that row.

TIME   An SQL-92 data type, with a default granularity of second. Supported by IBM DB2 UDB and by UniSQL, with a granularity of second.

time-dependent assumption   An often implicit assumption that will be invalidated purely by the course of time. The year 2000 problem is a specific instance.

time-invariant   Data or a table that doesn't vary over time.

time-invariant key   A key that identifies a particular entity over its entire lifespan.

time-invariant participation constraint   A participation constraint that holds over the entire valid time of the relationship.

time-invariant unique   Attribute(s) that are unique for a particular entity or relationship over its entire lifespan or valid time.

time-sequence   Of an object, the sequence (ordered by time) of pairs of a data object and an instant.

time-slice   Of a row or a table: the value(s) at a specified instant of time.

time-slice query   A query applied to a particular state of a table or database.

timestamp   Of a row, either the period of validity or the period of presence of that row. Of a table, the table's timestamp column(s).

TIMESTAMP   An SQL-92 data type, with a default granularity of microsecond. Supported by IBM DB2 UDB and UniSQL, with a granularity of second.

timestamp column   The column(s) of the table denoting the timestamp.

tracking log   Records the past states of the monitored table.

transaction time   Of a fact, the time when the fact is current in the database and may be retrieved. This time is a period delimited by when the fact was inserted into the database and when it was modified or logically deleted.

transaction time-invariant participation constraint   A participation constraint that holds over the entire transaction-time period of the relationship.

transaction time-invariant unique   Attribute(s) that are unique for a particular entity or relationship over its entire transaction-time period.

transaction time-slice   A query or view that selects the snapshot state of a transaction-time table at a specified transaction time, or a valid-time state of a bitemporal table.

transaction-time splitting   A method of partitioning a region of a bitemporal time diagram into rectangles by inserting vertical splits, resulting in contiguous bands in transaction time.

transaction-time state table   One with facts timestamped with their transaction time.

tropical year   The time interval between two consecutive passages of the earth across a given point of its orbit.

true day   The time between two consecutive noons, 24 hours.

true time   The hour-angle of the sun starting from noon.

user-defined time   An uninterpreted datetime. Contrast with valid time and transaction time.

vacuum   Remove less desired information, for example, from an archival or history store.

vacuuming criteria   Additional predicates that characterize the rows to be purged from a temporal table.

valid time   Of a fact, when the fact was true in the modeled reality.

valid time-slice   A query or view that selects the snapshot state of a valid-time table at a specified valid time, or a transaction-time state of a bitemporal table.

valid-time splitting   A method of partitioning a region of a bitemporal time diagram into rectangles by inserting horizontal splits, resulting in contiguous bands in valid time.

valid-time state table   One that records the history of the modeled reality by timestamping facts with periods denoting when they were valid.

valid-time support   A row with an associated valid time, which is a value of the period data type. An SQL3 table is one in which each row is a row with valid-time support.

value-equivalent   Rows on the same schema with identical values for their nontimestamp columns.

year-month interval   An SQL-92 interval data type containing only the year and month fields.

year 2000 problem   A hardware or software bug arising from using just two digits to record the year. Y2K is the acronym for the year 2000 problem.

Bibliography

[1]   Ahn, I., and R. T. Snodgrass. "Partitioned Storage Structures for Temporal Databases," *Information Systems* 13(4):369–391, 1988.

[2]   Allen, J. F. "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM* 26(11):832–843, 1983.

[3]   Bair, J., M. Böhlen, C. S. Jensen, and R. T. Snodgrass. "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)* 39(1):25–34, 1997.

[4]   Barnert, R., G. F. Knolmayer, and T. Myrach. Rules for Ensuring Referential Integrity in a Time-Extended Relational Model. Technical Report, University of Bern, Switzerland, 1996.

[5]   Ben-Zvi, J. The Time Relational Model. Ph.D. dissertation, Computer Science Department, University of California at Los Angeles, 1982.

[6]   Bernstein, A., V. Hadzilacos, and N. Goodman (eds.). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[7]   Bettini, C., C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang. "A Glossary of Time Granularity Concepts," in [33], pp. 406–413, 1998.

[8]   Bjork, L. A., Jr. "Generalized Audit Trail Requirements and Concepts for Data Base Applications," *IBM Systems Journal* 14(3):229–245, 1975.

[9]   Blaha, M., and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, Upper Saddle River, NJ, 1998.

[10]   Böhlen, M. H. The Temporal Deductive Database System ChronoLog. Ph.D. dissertation, Department Informatik, ETH Zurich, 1994.

[11]   Böhlen, M. H. Valid Time Integrity Constraints. Technical Report 94–30, University of Arizona, Tucson, November, 1994.

[12]   Böhlen, M. H., and C. S. Jensen. Seamless Integration of Time into SQL. Technical Report R-96-2049, Aalborg University, Aalborg, Denmark, December, 1996.

[13]   Böhlen, M. H., C. S. Jensen, and R. T. Snodgrass. "Evaluating the Completeness of TSQL2," in [25], pp. 153–174, 1995.

[14]   Böhlen, M. H., and R. Marti. "On the Completeness of Temporal Database Query Languages," in *Proceedings of the First International Conference on Temporal Logic*, pp. 283–300, July, 1994.

[15]   Böhlen, M. H., R. T. Snodgrass, and M. D. Soo. "Coalescing in Temporal Databases," in *Proceedings of the International Conference on Very Large Databases*, pp. 180–191, Mumbai (Bombay), India, September, 1996.

[16]   Bourne, K. C. *Year 2000 for Dummies*. IDG Books Worldwide, Foster City, CA, 1997.

[17]   Brackin, A. J. *Clocks: Chronicling Time.* Encyclopedia of Discovery and Invention, Lucent Books, 1991.

[18]   *Britannica Online. www.eb.com:180/cig-bin/g?DocF=micro/394/19.html*

[19]   Cannan, S. J. (ed.).   Database Language SQL—Technical Corrigendum 3, ISO/IEC 9075:1992/Cor. 3:1998.

[20]   Celko, J. "Regions, Runs, and Sequences," Chapter 22 of *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, San Francisco, 1995.

[21]   Celko, J. "Tooling Around," *DBMS Magazine* 11(5):20–25, 1998.

[22]   Chamberlin, D. *Using the New DB2*. Morgan Kaufmann, San Francisco, 1996.

[23]   Clifford, J., C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. "On the Semantics of 'Now' in Databases," *ACM Transactions on Database Systems* 22(2):171–214, 1997.

[24]   Clifford, J., and T. Isakowitz. "On the Semantics of (Bi)temporal Variable Databases," in *Proceedings of the Fourth International Conference on Extending Database Technology*, Cambridge, England, pp. 215–230, March, 1994.

[25]   Clifford, J., and A. Tuzhilin (eds.).   *Recent Advances in Temporal Databases*. Springer-Verlag, New York, 1995. Also *Proceedings of the VLDB International Workshop on Temporal Databases*.

[26]   Dadam, P., V. Lum, and H. D. Werner. "Integration of Time Versions into a Relational Database System," in *Proceedings of the Conference on Very Large Databases*, Singapore, pp. 509–522, 1984.

[27]   Dershowitz, N., and E. M. Reingold. *Calendrical Calculations*. Cambridge University Press, Cambridge, 1997.

[28]   Dyreson, C. E., and R. T. Snodgrass. "Timestamp Semantics and Representation," *Information Systems* 18(3):143–166, 1993.

[29]   Eisenberg, A., and J. Melton. "Standards in Practice," *ACM SIGMOD Record* 27(3):53–58, 1998.

[30]   Eisenberg, A., and J. Melton. "SQL:1999, Formerly Known as SQL3," *ACM SIGMOD Record* 28(1):131–138, March, 1999.

[31]   Electronics Industries Association. Recommended Practice for Line 21 Data Service. ANSI/EIA-608-94, September, 1994.

[32]   Ensor, D., and I. Stevenson. *Oracle Design*. O'Reilly & Associates, Sebastopol, CA, 1997.

[33]   Etzion, O., S. Jajodia, and S. M. Sripada (eds.).   *Temporal Databases: Research and Practice*. Springer-Verlag, New York, 1998.

[34]   Fraser, J. *Time: The Familiar Stranger*. Tempus Books, Redmond, WA, 1987.

[35]   Gould, S. J. *Questioning the Millennium: A Rationalist's Guide to a Precisely Arbitrary Countdown*. Harmony Books, New York, 1997.

[36]   Gray, J., and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 1993.

[37]   Gregersen, H., and C. S. Jensen. Conceptual Modeling of Time-Varying Information. TIME-CENTER Technical Report TR-35, September, 1998.

[38]   Gregersen, H., and C. S. Jensen. "Temporal Entity-Relationship Models—A Survey" to appear in *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[39]   Gregersen, H., L. Mark, and C. S. Jensen. Mapping Temporal ER Diagrams to Relational Schemas. TIMECENTER Technical Report TR-39, December, 1998.

[40]   Hofstadter, D. R. *Gödel, Escher, Bach: An External Golden Braid*. Vintage Books, New York, 1979.

[41]   "How to Buy a Wristwatch," *Consumer Reports*, 62(11):6, 1997.

[42]   Howse, D. *Greenwich Time and the Discovery of the Longitude*. Oxford University Press, Oxford, 1980.

[43]   ISO. Data Elements and Interchange Formats—Information Interchange—Representation of Dates and Times. ISO 8601:1988.

[44]   ISO. Database language SQL. ISO/IEC 9075:1992. ANSI X3.135-1992.

[45]   ISO. Database language SQL—Part 4: Persistent Stored Modules (SQL/PSM). ISO/IEC 9075-4:1995, ANSI/ISO/IEC 9075-4:1995.

[46]   Jackson, M. A. *System Development.* Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1983.

[47]   Jensen, C. S. "Vacuuming," Chapter 23 of [91], pp. 451–462, 1995.

[48]   Jensen, C. S., 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, 1994.

[49]   Jensen, C. S., and C. E. Dyreson (eds.), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio, and G. Wiederhold. "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in [33], pp. 367–413, 1998.

[50]   Jensen, C. S., and L. Mark. A Framework for Vacuuming Temporal Databases. Technical Report CS-TR-2516, UMIACS-TR-90-105, Department of Computer Science, University of Maryland, College Park, MD, August, 1990.

[51]   Jensen, C. S., and L. Mark. "Differential Query Processing in Transaction-Time Databases," Chapter 19 of [102], pp. 457–492, 1994.

[52]   Jensen, C. S., L. Mark, and N. Roussopoulos. "Incremental Implementation Model for Relational Databases with Transaction Time," *IEEE Transactions on Knowledge and Data Engineering* 3(4):461–473, 1991.

[53]   Jensen, C. S., and R. T. Snodgrass. "Temporal Specialization," in *Proceedings of the International Conference on Data Engineering*, Tempe, Arizona, pp. 594–603, February, 1992.

[54]   Jensen, C. S., and R. T. Snodgrass. "Temporal Specialization and Generalization," *IEEE Transactions on Knowledge and Data Engineering* 6(6):954–974, 1994.

[55] Jensen, C. S., and R. T. Snodgrass. "Semantics of Time-Varying Information," *Information Systems* 21(4):311–352, 1996.

[56] Jensen, C. S., and R. T. Snodgrass. "Temporal Data Management," *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, January/February, 1999.

[57] Jensen, C. S., and R. T. Snodgrass. "Temporally Enhanced Database Design," to appear in *Object-Oriented Data Modeling*, M. P. Papazoglou, S. Spaccapietra, and Z. Tari (eds.), MIT Press, Cambridge, MA, 1999.

[58] Jespersen, J., and J. Fitz-Randolph. *From Sundials to Atomic Clocks: Understanding Time and Frequency*. Dover, Mineola, NY, 1982.

[59] Jones, C. "Bad Days for Software," *IEEE Spectrum* 35(9):47–52, 1998.

[60] Katz, R. H., and T. Lehman. "Database Support for Versions and Alternatives of Large Design Files," *IEEE Transactions on Software Engineering* 10(2):191–200, 1984.

[61] Kimball, K. A. The DATA System. Master's thesis, University of Pennsylvania, 1978.

[62] Klopprogge, M. R. "TERM: An Approach to Include the Time Dimension in the Entity-Relationship Mode," in *Proceedings of the Second International Conference on the Entity Relationship Approach*, pp. 477–512, October, 1981.

[63] Koch, G., and K. Loney. *Oracle: The Complete Reference*. Osborne McGraw Hill, Berkeley, CA, 1997.

[64] Kuhn, T. S. *The Structure of Scientific Revolutions*. University of Chicago Press, third edition, 1996.

[65] Landes, D. S. *Revolution in Time: Clocks and the Making of the Modern World.* Barnes & Noble Books, New York, 1998.

[66] Leung, T. Y. C., and H. Pirahesh. "Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL," in [25], pp. 315–331, 1995.

[67] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. D. Werner, and J. Woodfill. "Designing DBMS Support for the Temporal Dimension," in *Proceedings of the ACM International Conference on Management of Data*, Boston, MA, pp. 115–130, 1984.

[68] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. D. Werner, and J. Woodfill. "Design of an Integrated DBMS to Support Advanced Applications," in *Proceedings Conference on Foundations of Data Organization,* Kyoto, Japan, 1985.

[69] Melton, J. (ed.). (ISO Working Draft) Temporal (SQL/Temporal). American National Standards Institute X3H2-97-135, International Organization for Standardization DBL:LGW-013, April, 1997.

[70] Melton, J. *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*. Morgan Kaufmann, San Francisco, 1998.

[71] Melton, J., and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco, 1993.

[72] Myrach, T., G. F. Knolmayer, and R. Barnert. "On Ensuring Keys and Referential Integrity in the Temporal Database Language TSQL2," in H.-M. Haav and B. Thalheim (eds.). *Databases and Information Systems, Proceedings of the Second International Baltic Workshop*, Tallinn, June

12–14, Volume I: Research Track, Tampere University of Technology Press, pp. 171–181, 1996.

[73]   National Institute of Standards and Technology. Federal Information Processing Standard 127-2, Database Language (SQL), December 3, 1993. *ftp://speckle.ncsl.nist.gov/vpl/sqlintro.htm*, June 18, 1998.

[74]   Navathe, S. B., and R. Ahmed. "A Temporal Relational Model and a Query Language," *Information Sciences* 49:147–175, 1989.

[75]   Neumann, P. G. "Y2K Update," *Communications of the ACM* 40(9):128, 1998.

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

[77]   Pais, A. *Niels Bohr's Times, in Physics, Philosophy, and Polity*. Clarendon Press, Oxford, 1991.

[78]   Palmer, J. D. "Time, Tide and the Living Clocks of Marine Organisms," *American Scientist* 84(6):570–578, 1996.

[79]   Quinn, T. J. "The BIPM and the Accurate Measurement of Time," *Proceedings of the IEEE,* 79(9):894–906, 1991.

[80]   Rohr, R. R. J. *Sundials: History, Theory, and Practice.* Dover Publications, Mineola, NY, 1970.

[81]   Rossum, G. D.-V. *History of the Hour: Clocks and Modern Temporal Orders.* The University of Chicago Press, Chicago, 1996.

[82]   Rozenshtein, D., A. Abramovich, and E. Birger. "Loop-Free SQL Solutions for Finding Continuous Regions," *SQL Forum* 2(6), November–December, 1993.

[83]   Rudolph, J. S. *Make Your Own Working Paper Clock.* Harper Perennial, New York, 1983.

[84]   Sarda, N. L. "Extensions to SQL for Historical Databases," *IEEE Transactions on Knowledge and Data Engineering* 2(2):220–230, 1990.

[85]   Skyt, J., and C. S. Jensen. Vacuuming Temporal Databases. TIMECENTER Technical Report TR-32, September, 1998.

[86]   Snodgrass, R. T. "The Temporal Query Language TQuel," *ACM Transactions on Database Systems* 12(2):247–298, 1987.

[87]   Snodgrass, R. T. "An Overview of TQuel," Chapter 6 of [102], pp. 141–182, 1994.

[88]   Snodgrass, R. T. "Managing Temporal Data—A Five-Part Series," TEMPIS Technical Report TR-28, September, 1998. *www.cs.arizona.edu/people/rts/DBPD/*.

[89]   Snodgrass, R. T., and I. Ahn. "A Taxonomy of Time in Databases," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Austin, TX, pp. 236–246, May, 1985.

[90]   Snodgrass, R. T., and I. Ahn. "Temporal Databases," *IEEE Computer* 19(9):35–42, 1986.

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

[92]   Snodgrass, R. T., M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. Change proposal, ANSI X3H2-96-501r1, ISO/IEC JTC1/SC21/WG3 DBL MCI-146r2, November, 1996. *ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad146.pdf*.

[93]   Snodgrass, R. T., M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. Change proposal, ANSI X3H2-96-502r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-147r2, November, 1996. *ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad147.pdf*.

[94]   Snodgrass, R. T., M. H. Böhlen, C. S. Jensen, and A. Steiner. "Transitioning Temporal Support in TSQL2 to SQL3," in [33], pp. 150–194, 1998.

[95]   Sobel, D. *Longitude*. Walker and Company, New York, 1995.

[96]   Sobel, D., and W. J. H. Andrewes. *The Illustrated Longitude*. Walker and Company, New York, 1998.

[97]   Soo, M. D., R. T. Snodgrass, and C. S. Jensen. "Efficient Evaluation of the Valid-Time Natural Join," in *Proceedings of the International Conference on Data Engineering*, Houston, TX, pp. 282–292, February, 1994.

[98]   Steele, G. (ed.). *The Hacker's Dictionary*. Harper and Row, New York, 1983.

[99]   Stoneman, M. *Easy-to-Make Wooden Sundials: Instructions and Plans for Five Projects*. Dover, Mineola, NY, 1981.

[100] Stork, D. G. (ed.). *HAL's Legacy: 2001's Computer as Dream and Reality*. MIT Press, Cambridge, MA, 1997.

[101] Sykes, J. M. Time Zones—A Tidying. ISO/IEC JTC 1/SC 21/WG 3 DBL MCI-068, April, 1996.

[102] Tansel, A., 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, Redwood City, CA, 1994.

[103] Tsotras, V. J., and X. S. Wang. "Temporal Databases," in *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons, New York, 1999.

[104] Woodward, P. *My Own Right Time: An Exploration of Clockwork Design*. Oxford University Press, Oxford, 1995.

[105] Wu, Y., S. Jajodia, and X. S. Wang. "Temporal Database Bibliography Update," in [33], pp. 338–366, 1998.

[106] Yourdon, E. *Managing the System Life Cycle*. Yourdon Press, 1982.

[107] Zaniolo, C., S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, San Francisco, 1997.

Author Index

Subject Index

About the Author

Richard Snodgrass is a professor of computer science at the University of Arizona, where he joined the Computer Science Department in 1989. He holds a B.A. degree in physics from Carleton College, and M.S. and Ph.D. degrees in computer science from Carnegie Mellon University. He has written or edited five books, as well as many journal articles and a column on time-oriented SQL for *Database Programming & Design Online*. He is an associate editor of the *ACM Transactions on Database Systems*, the *IEEE Transactions on Knowledge and Data Engineering*, and the *International Journal on Very Large Databases*. He is an ACM fellow.

He chairs the ACM Special Interest Group on Management of Data (SIGMOD) and is the American Program Committee chair for VLDB 2001. He chaired the program committees for the 1994 SIGMOD Conference and the 1993 International Workshop on an Infrastructure for Temporal Databases. In addition, he has served as a vice chair or member of many program committees. He chaired the TSQL2 Language Design Committee and is now working closely with the ISO SQL3 committee to add temporal support to that language. He initiated the SQL/Temporal part of the SQL3 draft standard.

He codirects TIMECENTER, an international center for the support of temporal database applications on traditional and emerging DBMS technologies.

About the CD-ROM

SYSTEM REQUIREMENTS

This CD-ROM may be read by Microsoft Windows (95/98/NT), Apple Macintosh, and Unix (including Ultrix, Solaris, Linux, and other variants). The CD-ROM is a hybrid disk in ISO 9660 format with Rock Ridge and Joliet extensions, allowing users of those operating systems to share the same directory structure and access common files. All file names are eight characters or less, with an extension having no more than three characters.

A web browser, such as Netscape Explorer, Netscape Communicator, or Microsoft Explorer, should be used to view the contents of this CD-ROM. To view the PDF files, you need a version of Adobe Acrobat Reader; included on the CD-ROM are versions of Acrobat Reader for several platforms. For some documents, a PostScript version is also included.

GETTING STARTED

Open "cover.htm" in the top-level directory in your web browser.

DESCRIPTION OF CD-ROM MATERIALS

The accompanying CD-ROM contains nine directories containing web pages, documents in PDF format, and executable files as an archive (tar) file.

The directories "datatype" (Chapters 3 and 4), "vtstbl" (Chapters 5–7), "audit" (Chapter 8), "transtbl" (Chapter 9), "bitmptbl" (Chapter 10), "tddesign" (Chapter 11), and "lang" (Chapter 12) contain web pages and code fragments, often for many different DBMSs. For instance, many of the code fragments in Chapter 5 are given

for Microsoft Access and SQL Server, IBM DB2 Universal Database, Oracle8 Server, Sybase SQLServer, and UniSQL. The files in these directories may be accessed by following links from the top-level web page, "cover.htm." The "images" directory contains some gif files used in the various web pages.

The "doc" directory contains executable versions of Adobe Acrobat Reader for various platforms.

The "lang" directory contains information on the if... Synchrony and TimeConsult TIMEDB 2 products; the Tiger and TIMEDB 1 prototypes; executable code for the TIMEDB 1 and TIMEDB 2 systems; and technical reports on temporal databases, TSQL2, and SQL3.

Finally, the "misc" directory contains the Web URLs that appear in the book and a list of all the principles appearing as pull quotes in the book.

The code fragments on this CD-ROM are for illustrative purposes only. They have been developed to maximize readability over performance. They have not been thoroughly tested.

COPYRIGHT NOTE