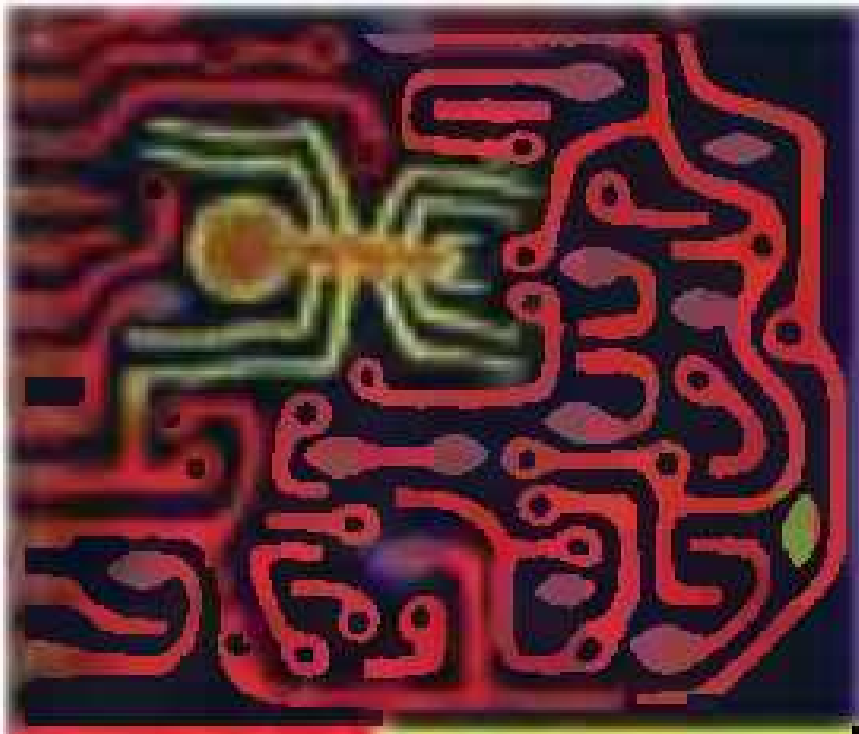


An Introduction to

Database Systems

C.J. Date

Hemy Mumb

EIGHTH EDITION

An Introduction to

Database Systems

The material covered in *An Introduction to Database Systems* is organized into six major parts:

- **Part I** (four chapters) provides a broad introduction to the concepts of database systems in general and relational systems in particular. It also introduces the standard database language, SQL.
- **Part II** (six chapters) consists of a detailed and very careful description of the relational model, which is not only the theoretical foundation underlying relational systems but is, in fact, the theoretical foundation for the entire database field.
- **Part III** (four chapters) discusses the general question of database design. Three chapters are devoted to design theory, and the fourth considers semantic modeling and the entity/relationship model.
- **Part IV** (two chapters) is concerned with transaction management (i.e., recovery and concurrency controls).
- **Part V** (eight chapters) shows how relational concepts are relevant to a variety of further aspects of database technology—security, distributed databases, temporal data, decision support, and so on.
- **Part VI** (three chapters) describes the impact of object technology on database systems. Chapter 25 describes object systems specifically; Chapter 26 considers the possibility of a *rapprochement* between object and relational technologies and discusses object/relational systems; and Chapter 27 addresses the relevance to databases of XML.

About the Author

C. J. DATE is an author, lecturer, researcher, and independent consultant specializing in relational database systems. An active member of the database community for nearly 35 years, C. J. Date devotes the major part of his career to exploring, expanding, and expounding the theory and practice of relational technology. He enjoys a reputation second to none for his ability to explain complex technical material in a clear and understandable fashion.

"[C. J. Date's] book is the flag bearer of relational theory and mathematical treatment in general...as well as the runaway leader in discussing the SQL standards. It exercises much more respect for careful language and the importance of concepts and principles in gaining mastery of the field."

—CARL ECKBERG, *San Diego State University*

"[The] 8th Edition is an excellent and comprehensive presentation of the contemporary database field. In particular, Date's chapters on types, relations, object databases, and object-relational databases together provide an exceptionally clear, self-contained exposition of the object-relational approach to databases."

—MARTIN K. SOLOMON, *Florida Atlantic University*

"Chris Date is the computer industry's most respected expert and thinker on database technology, and his book *An Introduction to Database Systems* continues to be the definitive work for those wanting a comprehensive and current guide to database systems."

—COLIN J. WHITE, *President, Intelligent Business Strategies*

"This is the best explanation of concurrency that I have seen in literature, and it covers the ground quite thoroughly."

—BRUCE O. LARSEN, *Stevens Institute of Technology*

"...both an indispensable read and an indispensable reference. No serious information systems or database practitioner should be without this book."

—DECLAN BRADY, MICS, *Systems Architect and Database Specialist, Fujitsu*

"The author's deep insights into the area, informal treatment of profound topics, open-ended discussions of critical issues, comprehensive and up-to-date contents, as well as rich annotations on bibliography have made the book most popular in the database area for more than two decades."

—QIANG ZHU, *The University of Michigan, Dearborn*

"[The book's] appeal is its comprehensiveness and the fact that it is very up-to-date with research developments. The latter factor is due mainly to [Date's] involvement with these developments, which gives him a unique opportunity to write about them."

—DAVID LIVINGSTONE, *University of Northumbria at Newcastle*

Senior Acquisitions Editor: Maite Suarez-Rivas
Project Editor: Katherine Harutunian
Marketing Manager: Nathan Schultz
Production Supervisor: Marilyn Lloyd
Project Management: Elisabeth Beller
Composition: Nancy Logan
Technical Art: Dartmouth Publishing, Inc.
Copyeditor: Daril Bentley
Proofreader: Jennifer McClain
Design Manager: Joyce Cosentino Wells
Cover Design: Night & Day Design
Cover Image: Lindy Date
Prepress and Manufacturing: Caroline Fell

Access the latest information about Addison-Wesley titles from our World Wide Web site:
http://www.aw.com/cs

ISBN 0-321-18956-6

2 3 4 5 6 7 8 9 10-HAM-06050403

This book is dedicated to my wife Lindy
and to the memory of my mother Rene—

also to the memory of Ted Codd, who, sadly,
passed away as this book was going to press

Those who cannot remember the past
are condemned to repeat it

Usually quoted in the form:

Those who don't know history are
doomed to repeat it

—George Santayana

I would like to see computer science
teaching set deliberately
in a historical framework. . .
Students need to understand
how the present situation has come
about, what was tried,
what worked and what did not, and
how improvements in hardware
made progress possible. The absence
of this element in
their training causes people to
approach every problem from
first principles. They are apt to
propose solutions that
have been found wanting in the past.
Instead of standing
on the shoulders of their precursors,
they try to go it alone.

—Maurice V. Wilkes

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is based in Healdsburg, California.

In 1967, following several years as a mathematical programmer and programming instructor for Leo Computers Ltd. (London, England), Mr. Date moved to the IBM (UK) Development Laboratories, where he worked on the integration of database functionality into PL/I. In 1974 he transferred to the IBM Systems Development Center in California, where he was responsible for the design of a database language known as the Unified Database Language, UDL, and worked on technical planning and externals design for the IBM products SQL/DS and DB2. He left IBM in May, 1983.

Mr. Date has been active in the database field for well over 30 years. He was one of the first people anywhere to recognize the significance of Codd's pioneering work on the relational model. He has lectured widely on technical subjects—principally on database topics, and especially on relational database—throughout North America and also in Europe, Australia, Latin America, and the Far East. In addition to the present book, he is author or coauthor of a number of other database texts, including, from Morgan Kaufmann, *Temporal Data and the Relational Model* (2003) and, from Addison-Wesley, *Foundation for Future Database Systems: The Third Manifesto* (2nd edition, 2000), a detailed proposal for the future direction of the field; *Database: A Primer* (1983), which treats database systems from the nonspecialist's point of view; a series of *Relational Database Writings* books (1986, 1990, 1992, 1995, 1998), which deal with various aspects of relational technology in depth; and another series of books on specific systems and languages—*A Guide to DB2* (4th edition, 1993), *A Guide to SYBASE and SQL Server* (1992), *A Guide to SQL/DS* (1988), *A Guide to INGRES* (1987), and *A Guide to the SQL Standard* (4th edition, 1997). His books have been translated into several languages, including Braille, Chinese, Dutch, French, German, Greek, Italian, Japanese, Korean, Polish, Portuguese, Russian, and Spanish.

Mr. Date has also produced over 300 technical articles and research papers and has made a variety of original contributions to database theory. For several years, he was a regular columnist for the magazine *Database Programming & Design*. He also contributes regularly to the website *http://dbdebunk.com*. His professional seminars on database technology, offered both in North America and overseas, are widely considered to be second to none for the quality of the subject matter and the clarity of the exposition.

Mr. Date holds an Honours Degree in Mathematics from Cambridge University, England (BA 1962, MA 1966) and the honorary degree of Doctor of Technology from De Montfort University, England (1994).

Contents

Preface to the Eighth Edition

This book is a comprehensive introduction to the now very large field of database systems. It provides a solid grounding in the foundations of database technology and gives some idea as to how the field is likely to develop in the future. The book is meant primarily as a textbook, not a work of reference (though I hope and believe it can be useful as a reference also, to some extent). The emphasis throughout is on insight and understanding, not just on formalisms.

PREREQUISITES

The book as a whole is meant for anyone professionally interested in computing in some way who wants to gain an understanding of what database systems are all about. I assume you have at least a basic knowledge of both:

- The storage and file management capabilities (indexing, etc.) of a modern computer system
- The features of at least one high-level programming language (Java, Pascal, PL/I, etc.)

Regarding the first of these prerequisites, however, please note that a detailed tutorial on such matters can be found in the online Appendix D, "Storage Structures and Access Methods."

STRUCTURE

I have to say that I am a little embarrassed at the size of this book. The fact is, however, that database technology has become a very large field, and it is not possible to do it justice in fewer than 1,000 pages or so (indeed, most of the book's competitors are also about 1,000 pages). Be that as it may, the book overall is divided into six major parts:

I. Basic Concepts
II. The Relational Model
III. Database Design
IV. Transaction Management

V. Further Topics

VI. Objects, Relations, and XML

Each part in turn is divided into several chapters:

- Part I (four chapters) provides a broad introduction to the concepts of database systems in general and relational systems in particular. It also introduces the standard database language SQL.
- Part II (six chapters) consists of a detailed and very careful description of the relational model, which is not only the theoretical foundation underlying relational systems, but is in fact the theoretical foundation for the entire database field.
- Part III (four chapters) discusses the general question of database design; three chapters are devoted to design theory, the fourth considers semantic modeling and the entity/relationship model.
- Part IV (two chapters) is concerned with transaction management (i.e., recovery and concurrency controls).
- Part V (eight chapters) is a little bit of a *potpourri*. In general, however, it shows how relational concepts are relevant to a variety of further aspects of database technology—security, distributed databases, temporal data, decision support, and so on.
- Finally, Part VI (three chapters) describes the impact of object technology on database systems. Chapter 25 describes object systems specifically; Chapter 26 considers the possibility of a *rapprochement* between object and relational technologies and discusses object/relational systems; and Chapter 27 addresses the relevance to databases of XML.

There are also four appendixes. The first is an overview of a dramatic new and radically different implementation technology called The TransRelational™ Model; the next gives a BNF grammar for SQL expressions; the third contains a glossary of the more important abbreviations, acronyms, and symbols introduced in the body of the text; and the last is, as already explained, an online tutorial on storage structures and access methods.

ONLINE MATERIALS

The following instructor supplements are available only to qualified instructors. For information on accessing them, please contact your local Addison-Wesley Sales Representative, or send e-mail to *aw.cse@aw.com*.

- An Instructor's Manual provides guidance on how to use the book as a basis for teaching a database course. It consists of a series of notes, hints, and suggestions on each part, chapter, and appendix, as well as other supporting material.
- Answers to exercises (included in Instructor's Manual)
- Lecture slides in PowerPoint format

The following supplements are available to all readers of this book at *http://www.aw.com/cssupport.*

- Appendix D on storage structures and access methods (as already mentioned)
- Answers to a selected subset of the exercises

HOW TO READ THIS BOOK

The book overall is meant to be read in sequence more or less as written, but you can skip later chapters, and later sections within chapters, if you choose. A suggested plan for a first reading would be:

- Read Chapters 1 and 2 "once over lightly."
- Read Chapters 3 and 4 very carefully (except perhaps for Sections 4.6 and 4.7).
- Read Chapter 5 "once over lightly."
- Read Chapters 6, 7, 9, and 10 carefully, but skip Chapter 8 (except perhaps for Section 8.6 on SQL).
- Read Chapter 11 "once over lightly."
- Read Chapters 12 and 14 carefully,[1] but skip Chapter 13.
- Read Chapters 15 and 16 carefully (except perhaps for Section 15.6 on two-phase commit).
- Read subsequent chapters selectively (but in sequence), according to taste and interest.

Each chapter opens with an introduction and closes with a summary: in addition, most chapters include exercises, and the online answers often give additional information about the topic of the exercise. Most chapters also include a set of references, many of which are annotated. This structure allows the subject matter to be treated in a layered fashion, with the most important concepts and results being presented "in line" in the main body of the text and various subsidiary issues and more complex aspects being deferred to the exercises or answers or references (as appropriate). *Note:* References are identified by two-part numbers in square brackets. For example, the reference "[3.1]" refers to the first item in the list of references at the end of Chapter 3: namely, a paper by E. F. Codd published in *CACM 25,* No. 2, in February, 1982. (For an explanation of abbreviations used in references—e.g., "CACM"—see Appendix C.)

COMPARISON WITH EARLIER EDITIONS

We can summarize the major differences between this edition and its immediate predecessor as follows:

- *Part I:* Chapters 1–4 cover roughly the same ground as Chapters 1-4 in the seventh edition, but they have been significantly revised at the detail level. In particular,

[1] You could also read Chapter 14 earlier if you like, possibly right after Chapter 4.

Chapter 4, the introduction to SQL, has been upgraded to the level of the current standard SQL:1999, as indeed has SQL coverage throughout the entire book. (This fact all by itself caused major revisions to more than half the chapters from the seventh edition.) *Note:* Facilities likely to be included in the next version of the standard—which will probably be ratified in late 2003—are also mentioned where appropriate.

- *Part II:* Chapters 5–10, on the relational model, are a totally rewritten, considerably expanded, and very much improved version of Chapters 5–9 from the seventh edition. In particular, the material on types—also known as domains—has been expanded into a chapter of its own (Chapter 5), and the material on integrity (Chapter 9) has been completely restructured and rethought. I hasten to add that the changes in these chapters do not represent changes in the underlying concepts but, rather, changes in how I have chosen to present them, based on my practical experience in teaching this material in live presentations.

  Note: Some further words of explanation are in order here. Earlier editions of the book used SQL as a basis for teaching relational concepts, in the belief that it was easier on the student to show the concrete before the abstract. Unfortunately, however, the gulf between SQL and the relational model grew and continued to grow, ultimately reaching a point where I felt it would be actively misleading to use SQL for such a purpose any longer. The sad truth is that SQL is now so far from being a true embodiment of relational principles—it suffers from so many sins of both omission and commission—that I would frankly prefer not to discuss it at all! However, SQL is obviously important from a commercial point of view; thus, every database professional needs to have some familiarity with it, and it would just not be appropriate to ignore it in a book of this nature. I therefore settled on the strategy of including (a) a chapter on SQL basics in Part I of the book and (b) individual sections in other chapters, as applicable, describing those aspects of SQL that are specific to the subject of the chapter in question. In this way the book still provides comprehensive—indeed, extensive—coverage of SQL material, but puts that coverage into what I feel is the proper context.

- *Part III:* Chapters 10–13 are a mostly cosmetic revision of Chapters 9–12 from the seventh edition. There are detail-level improvements throughout, however.

  Note: Again some further explanation is in order. Some reviewers of earlier editions complained that database design issues were treated too late. But it is my feeling that students are not ready to design databases properly or to appreciate design issues fully until they have some understanding of what databases are and how they are used; in other words, I believe it is important to spend some time on the relational model and related matters before exposing the student to design questions. Thus, I still believe Part III is in the right place. (That said, I do recognize that many instructors prefer to treat the entity/relationship material much earlier. To that end, I have tried to make Chapter 14 more or less self-contained, so that they can bring it in immediately after, say, Chapter 4.)

- *Part IV:* The two chapters of this part, Chapters 15 and 16, are completely rewritten, extended, and improved versions of Chapters 14 and 15 from the seventh edition. In

particular, Chapter 16 now includes a careful analysis of—and some unorthodox conclusions regarding—the so-called ACID properties of transactions.

- *Part V:* Chapter 20 on type inheritance and Chapter 23 on temporal databases have been totally rewritten to reflect recent research and developments in those areas. Revisions to other chapters are mostly cosmetic, though there are improvements in explanations and examples throughout and new material here and there.
- *Part VI:* Chapters 25 and 26 are improved and expanded versions of Chapters 24 and 25 from the seventh edition. Chapter 27 is new.

Finally, Appendix A is also new, while Appendixes B and C are revised versions of Appendixes A and C, respectively, from the seventh edition (the material from Appendix B in that edition has been incorporated into the body of the book). Appendix D is a significantly revised version of Appendix A from the *sixth* edition.

WHAT MAKES THIS BOOK DIFFERENT?

Every database book on the market has its own individual strengths and weaknesses, and every writer has his or her own particular ax to grind. One concentrates on transaction management issues; another stresses entity/relationship modeling; another looks at everything through an SQL lens; yet another takes a pure "object" point of view; still another views the field exclusively in terms of some commercial product; and so on. And, of course, I am no exception to this rule—I too have an ax to grind: what might be called the foundation ax. I believe very firmly that we must get the foundation right, and understand it properly, before we try to build on that foundation. This belief on my part explains the heavy emphasis in this book on the relational model; in particular, it explains the length of Part II—the most important part of the book—where I present my own understanding of the relational model as carefully as I can. I am interested in foundations, not fads and fashions. Products change all the time, but principles endure.

In this regard, I would like to draw your attention to the fact that there are several important ("foundation") topics for which this book, virtually alone among the competition, includes an entire in-depth chapter (or an appendix, in one case). The topics in question include:

- Types
- Integrity
- Views
- Missing information
- Inheritance
- Temporal databases
- The TransRelational™ Model

In connection with that same point (the importance of foundations), I have to admit that the overall tone of the book has changed over the years. The first few editions were mostly descriptive in nature; they described the field as it actually was in practice, "warts and all." Later editions, by contrast, were much more *pre*scriptive; they talked about the way the field *ought* to be and the way it ought to develop in the future, if we did things right. And the present edition is certainly prescriptive in this sense (so it is a text with an attitude!). Since the first part of that "doing things right" is surely educating oneself as to what those right things are, I hope this new edition can help in that endeavor.

Yet another related point: As you might know, I recently published, along with my colleague Hugh Darwen, another "prescriptive" book, *Foundation for Future Database Systems: The Third Manifesto* (reference [3.3] in the present book).[2] That book, which we call *The Third Manifesto* or just the *Manifesto* for short, builds on the relational model to offer a detailed technical proposal for future database systems; it is the result of many years of teaching and thinking about such matters on the part of both Hugh and myself. And, not surprisingly, the ideas of the *Manifesto* permeate the present book. Which is not to say the *Manifesto* is a prerequisite to the present book—it is not; but it *is* directly relevant to much that is in the present book, and further related information is often to be found therein.

Note: Reference [3.3] uses a language called Tutorial D for illustrative purposes, and the present book does the same. Tutorial D syntax and semantics are intended to be more or less self-explanatory (the language might be characterized, loosely, as "Pascal-like"), but individual features are explained when they are first used if such explanation seems necessary.

A CLOSING REMARK

I would like to close these prefatory notes with the following lightly edited extract from another preface—Bertrand Russell's own preface to *The Bertrand Russell Dictionary of Mind, Matter and Morals* (ed., Lester E. Denonn), Citadel Press, 1993, reprinted here by permission:

> *I have been accused of a habit of changing my opinions . . . I am not myself in any degree ashamed of [that habit]. What physicist who was already active in 1900 would dream of boasting that his opinions had not changed during the last half century? . . . [The] kind of philosophy that I value and have endeavoured to pursue is scientific, in the sense that there is some definite knowledge to be obtained and that new discoveries can make the admission of former error inevitable to any candid mind. For what I have said, whether early or late, I do not claim the kind of truth which theologians claim for their creeds. I claim only, at best, that the opinion expressed was a sensible one to hold at the time . . . I should be much surprised if subsequent research did not show that it needed to be modified. [Such opinions were not] intended as pontifical pronouncements, but only as the best I could do at the time towards the promotion of clear and accurate thinking. Clarity, above all, has been my aim.*

[2] There is a website, too: *http://www.thethirdmanifesto.com*. See also *http://www.dbdebunk.com* for much related material.

If you compare earlier editions of this book with this eighth edition, you will find that I too have changed my opinions on many matters (and no doubt will continue to do so). I hope you will accept the remarks just quoted as adequate justification for this state of affairs. I share Bertrand Russell's perception of what the field of scientific inquiry is all about, but he expresses that perception far more eloquently than I could.

ACKNOWLEDGMENTS

Once again it is a pleasure to acknowledge my debt to the many people involved, directly or indirectly, in the production of this book:

- First of all, I must thank my friends David McGoveran and Nick Tindall for their major involvement in this edition; David contributed the first draft of Chapter 22 on decision support, and Nick contributed the first draft of Chapter 27 on XML. I must also thank my friend and colleague Hugh Darwen for major help (in a variety of forms) with all SQL portions of the manuscript. Nagraj Alur and Fabian Pascal also provided me with a variety of technical background material. A special vote of thanks goes to Steve Tarin for inventing the technology described in Appendix A and for his help in getting me to understand it fully.
- Second, the text has benefited from the comments of students on the seminars I have been teaching over the past several years. It has also benefited enormously from the comments of, and discussions with, numerous friends and reviewers, including Hugh Darwen, IBM; Guy de Tré, Ghent University; Carl Eckberg, San Diego State University; Cheng Hsu, Rensselaer Polytechnic Institute; Abdul-Rahman Itani. The University of Michigan-Dearborn; Vijay Kanabar, Boston University; Bruce O. Larsen, Stevens Institute of Technology; David Livingstone, University of Northumbria at Newcastle; David McGoveran, Alternative Technologies; Steve Miller, IBM; Fabian Pascal, independent consultant; Martin K. Solomon. Florida Atlantic University; Steve Tarin, Required Technologies; and Nick Tindall, IBM. Each of these people reviewed at least some part of the manuscript or made technical material available or otherwise helped me find answers to my many technical questions, and I am very grateful to all of them.
- I would also like to thank my wife Lindy for contributing the cover art once again and for her support throughout this and all my other database-related projects over the years.
- Finally, I am grateful (as always) to everyone at Addison-Wesley—especially Maite Suarez-Rivas and Katherine Harutunian—for all of their encouragement and support throughout this project, and to my packager Elisabeth Beller for another sterling job.

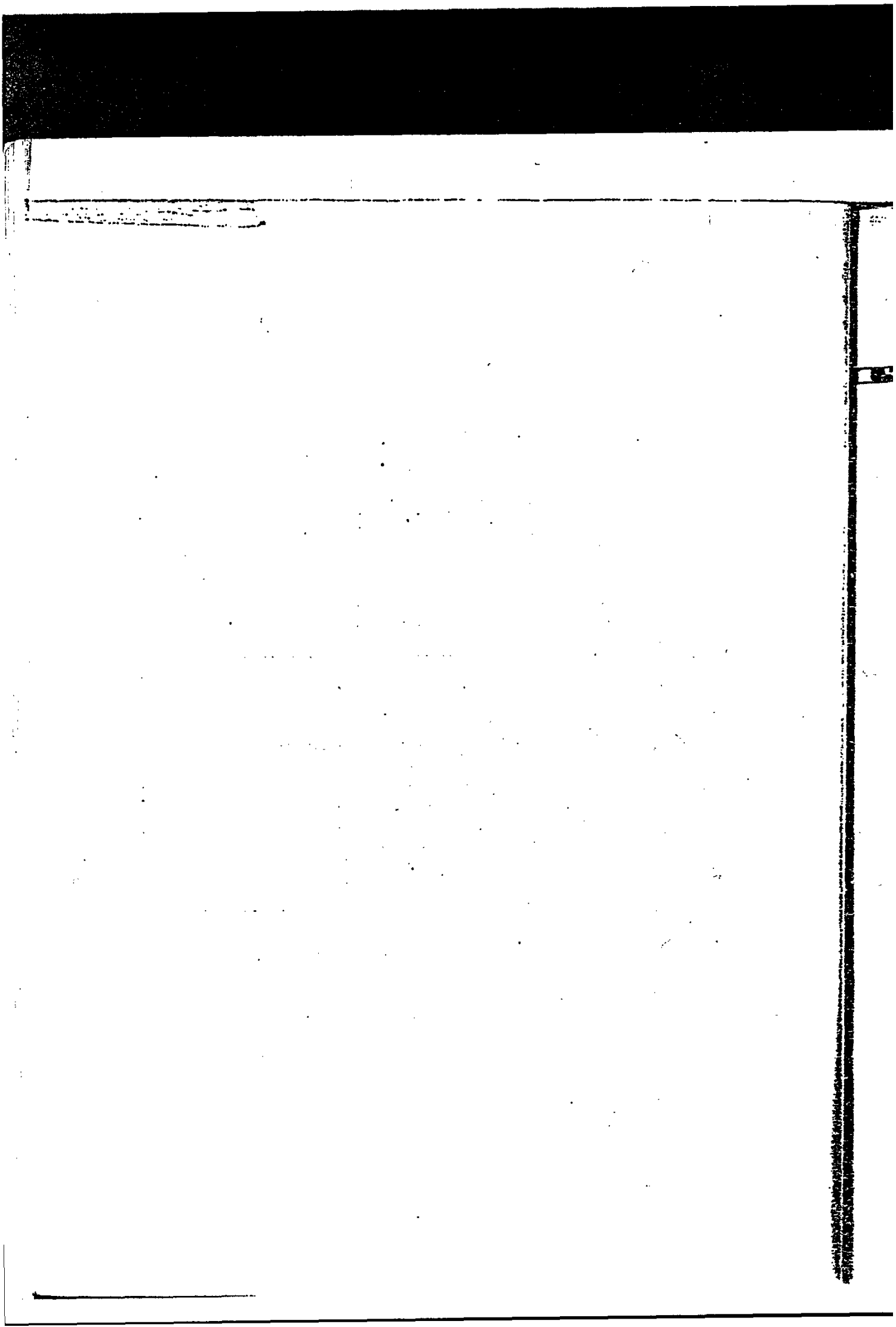
Healdsburg, California C. J. Date
2003

PART I

PRELIMINARIES

Part I consists of four introductory chapters:

- Chapter 1 sets the scene by explaining what a database is and why database systems are desirable. It also briefly discusses the difference between relational systems and others.
- Next, Chapter 2 presents a general architecture for database systems, the so-called ANSI/SPARC architecture. That architecture serves as a framework on which the rest of the book will build.
- Chapter 3 then presents an overview of relational systems (the aim is to serve as a gentle introduction to the much more comprehensive discussions of the same subject in Part II and later parts of the book). It also introduces and explains the running example, the suppliers-and-parts database.
- Finally, Chapter 4 introduces the standard relational language SQL (more precisely, SQL:1999).



CHAPTER 1

An Overview of Database Management

1.1 INTRODUCTION

A database system is basically just a *computerized record-keeping system*. The database itself can be regarded as a kind of electronic filing cabinet; that is, it is a repository or container for a collection of computerized data files. Users of the system can perform (or request the system to perform, rather) a variety of operations involving such files—for example:

- Adding new files to the database
- Inserting data into existing files
- Retrieving data from existing files
- Deleting data from existing files
- Changing data in existing files
- Removing existing files from the database

Fig. 1.1 shows a very small database containing just one file, called CELLAR, which in turn contains data concerning the contents of a wine cellar. Fig. 1.2 shows an example of a retrieval request against that database, together with the data returned by that request. (Throughout this book we show database requests, file names, and other such material in uppercase for clarity. In practice it is often more convenient to enter such material in lowercase. Most systems will accept both.) Fig. 1.3 gives examples, all more or less self-explanatory, of insert, delete, and change requests on the wine cellar database. Examples of adding and removing entire files are given in later chapters.

Several points arise immediately from Figs. 1.1–1.3:

1. First of all, the SELECT, INSERT, DELETE, and UPDATE requests (also called *statements*, *commands*, or *operators*) in Figs. 1.2 and 1.3 are all expressed in a language called SQL. Originally a proprietary language from IBM, SQL is now an international standard that is supported by just about every database product commercially available; in fact, the market is totally dominated by SQL products at the time of writing. Because of its commercial importance, therefore, Chapter 4 presents a general overview of the SQL standard, and most subsequent chapters include a section called "SQL Facilities" that describes those aspects of that standard that are pertinent to the topic of the chapter in question.

   By the way, the name SQL originally stood for *Structured Query Language* and was pronounced *sequel*. Now that it is a standard, however, the name is officially just a name—it is not supposed to be an abbreviation for anything at all—and it is officially pronounced *ess-cue-ell*. We will assume this latter pronunciation throughout this book.

2. Note from Fig. 1.3 that SQL uses the keyword UPDATE to mean "change" specifically. This fact can cause confusion, because the term *update* is also used to refer to the three operators INSERT, DELETE, and UPDATE considered as a group. We will distinguish between the two meanings in this book by using lowercase when the generic meaning is intended and uppercase to refer to the UPDATE operator specifically.

   Incidentally, you might have noticed that we have now used both the term *operator* and the term *operation*. Strictly speaking, there is a difference between the two (the *operation* is what is performed when the *operator* is invoked); in informal discussions, however, the terms tend to be used interchangeably.

3. In SQL, computerized files such as CELLAR in Fig. 1.1 are called tables (for obvious reasons); the rows of such a table can be thought of as the *records* of the file, and the columns can be thought of as the *fields*. In this book, we will use the terminology of files, records, and fields when we are talking about database systems in general (mostly just in the first two chapters); we will use the terminology of tables, rows, and columns when we are talking about SQL systems in particular. (And when we get to our more formal discussions in Chapter 3 and later parts of the book, we will meet yet another set of terms: *relations*, *tuples*, and *attributes* instead of tables, rows, and columns.)

BIN#	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	2001	1	2003
3	Chardonnay	Geyser Peak	2001	5	2003
6	Chardonnay	Simi	2000	4	2002
12	Joh. Riesling	Jekel	2002	1	2003
21	Fumé Blanc	Ch. St. Jean	2001	4	2003
22	Fumé Blanc	Robt. Mondavi	2000	2	2002
30	Gewürztraminer	Ch. St. Jean	2002	3	2003
43	Cab. Sauvignon	Windsor	1995	12	2004
45	Cab. Sauvignon	Geyser Peak	1998	12	2006
48	Cab. Sauvignon	Robt. Mondavi	1997	12	2008
50	Pinot Noir	Gary Farrell	2000	3	2003
51	Pinot Noir	Fetzer	1997	3	2004
52	Pinot Noir	Dehlinger	1999	2	2002
58	Merlot	Clos du Bois	1998	9	2004
64	Zinfandel	Cline	1998	9	2007
72	Zinfandel	Rafanelli	1999	2	2007

Fig. 1.1 The wine cellar database (file CELLAR)

```
Retrieval:
SELECT WINE, BIN#, PRODUCER
FROM   CELLAR
WHERE  READY = 2004 ;
```

Result (as shown on, e.g., a display screen):

WINE	BIN#	PRODUCER
Cab. Sauvignon	43	Windsor
Pinot Noir	51	Fetzer
Merlot	58	Clos du Bois

Fig. 1.2 Retrieval example

```
Inserting new data:
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 53, 'Pinot Noir', 'Saintsbury', 2001, 6, 2005 ) ;

Deleting existing data:
DELETE
FROM   CELLAR
WHERE  BIN# = 2 ;

Changing existing data:
UPDATE CELLAR
SET    BOTTLES = 4
WHERE  BIN# = 3 ;
```

Fig. 1.3 Insert, delete, and change examples

4. With respect to the CELLAR table, we have made a tacit assumption for simplicity that columns WINE and PRODUCER contain character-string data and all other columns contain integer data. In general, however, columns can contain data of arbitrary

complexity. For example, we might extend the CELLAR table to include additional columns as follows:

- LABEL (photograph of the wine bottle label)
- REVIEW (text of a review from some wine magazine)
- MAP (map showing where the wine comes from)
- NOTES (nudio recording containing our own tasting notes)

and many other things. For obvious reasons, the majority of examples in this book involve only very simple kinds of data, but do not lose sight of the fact that more exotic possibilities are always available. We will consider this question of column data types in more detail in later chapters (especially Chapters 5–6 and 26–27).

5. Column BIN# constitutes the primary key for table CELLAR (meaning, loosely, that no two CELLAR rows ever contain the same BIN# value). In figures like Fig. 1.1 we use *double underlining* to indicate primary key columns.

One last point to close this preliminary section: While a full understanding of this chapter and the next is necessary to a proper appreciation of the features and capabilities of a modern database system, it cannot be denied that the material is somewhat abstract and rather dry in places (also, it does tend to involve a large number of concepts and terms that might be new to you). In Chapters 3 and 4 you will find material that is much less abstract and hence more immediately understandable, perhaps. You might therefore prefer just to give these first two chapters a "once over lightly" reading for now, and to reread them more carefully later as they become more directly relevant to the topics at hand.

1.2 WHAT IS A DATABASE SYSTEM?

To repeat from the previous section, a database system is basically a computerized record-keeping system; in other words, it is a computerized system whose overall purpose is to store information and to allow users to retrieve and update that information on demand. The information in question can be anything that is of significance to the individual or organization concerned—anything, in other words, that is needed to assist in the general process of running the business of that individual or organization.

Incidentally, please note that we treat the terms *data* and *information* as synonyms in this book. Some writers prefer to distinguish between the two, using *data* to refer to what is actually stored in the database and *information* to refer to the *meaning* of that data as understood by some user. The distinction is clearly important—so important that it seems preferable to make it explicit, where appropriate, instead of relying on a somewhat arbitrary differentiation between two essentially synonymous terms.

Fig. 1.4 is a simplified picture of a database system. As the figure shows, such a system involves four major components: data, hardware, software, and users. We consider these four components briefly here. Later we will discuss each in much more detail (except for the hardware component, details of which are mostly beyond the scope of this book).

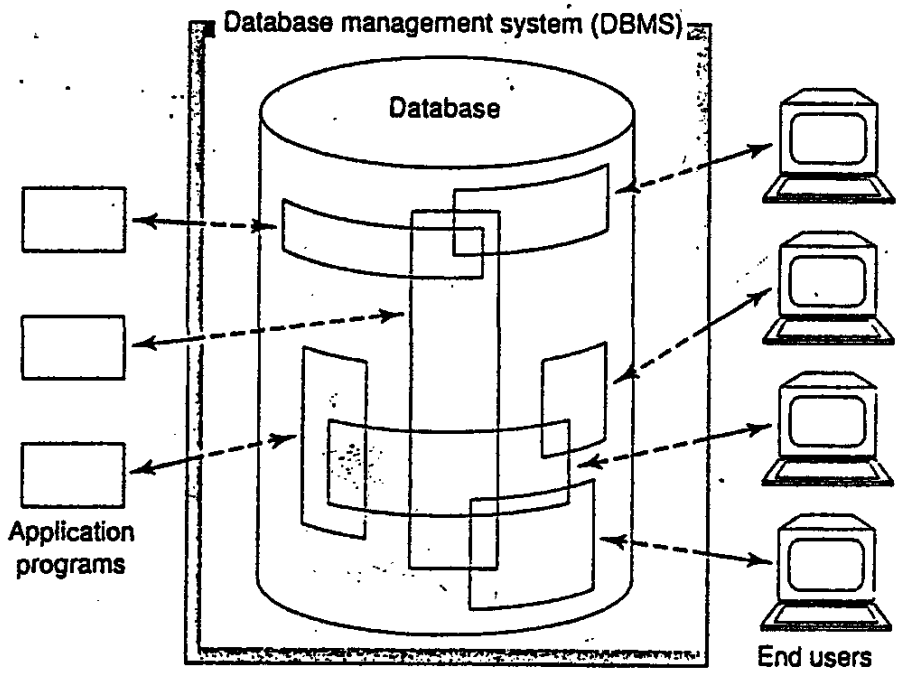


Fig. 1.4 Simplified picture of a database system

Data

Database systems are available on machines that range all the way from the smallest hand-held or personal computers to the largest mainframes or clusters of mainframes. Needless to say, the facilities provided by any given system are determined to some extent by the size and power of the underlying machine. In particular, systems on large machines ("large systems") tend to be *multi-user*, whereas those on smaller machines ("small systems") tend to be *single-user*. A single-user system is a system in which at most one user can access the database at any given time; a multi-user system is a system in which many users can access the database at the same time. As Fig. 1.4 suggests, we will normally assume the latter case in this book, for generality; in fact, however, the distinction is largely irrelevant so far as most users are concerned, because it is precisely an objective of multi-user systems in general to allow each user to behave as if he or she were working with a *single*-user system instead. The special problems of multi-user systems are primarily problems that are internal to the system, not ones that are visible to the user (see Part IV of this book, especially Chapter 16).

Now, it is convenient to assume for the sake of simplicity that the totality of data in the system is all stored in a single database, and we will usually make that assumption in this book, since it does not materially affect any of our other discussions. In practice, however, there might be good reasons, even in a small system, why the data should be split across several distinct databases. We will touch on some of those reasons later, in Chapter 2 and elsewhere.

In general, then, the data in the database—at least in a large system—will be both *integrated* and *shared.* As we will see in Section 1.4, these two aspects, data integration and data sharing, represent a major advantage of database systems in the "large" environment, and data integration, at least, can be significant in the "small" environment as well. Of course, there are many additional advantages also, to be discussed later, even in the small environment. But first let us explain what we mean by the terms *integrated* and *shared:*

- By integrated, we mean the database can be thought of as a unification of several otherwise distinct files, with any redundancy among those files partly or wholly eliminated. For example, a given database might contain both an EMPLOYEE file, giving employee names, addresses, departments,. salaries, and so on, and an ENROLLMENT file, representing the enrollment of employees in training courses (refer to Fig. 1.5). Now suppose that, in order to carry out the process of training course administration, it is necessary to know the department for each enrolled student. Then there is clearly no need to include that information redundantly in the ENROLLMENT file, because it can always be discovered by referring to the EMPLOYEE file instead.
- By shared, we mean the database can be shared among different users, in the sense that different users can have access to the same data, possibly even at the same time ("concurrent access"). Such sharing, concurrent or otherwise, is partly a consequence of the fact that the database is integmted. In the example of Fig. 1.5, for instance, the department information in the EMPLOYEE file would typically be shared by users in the Personnel Department and users in the Education Department. (A database that is not shared in the foregoing sense is sometimes said to be "personal" or "application-specific.")

Another consequence of the foregoing facts—that the database is integrated and shared—is that any given user will typically be concerned only with some small portion of the total database; moreover, different users' portions will overlap in various ways. In other words, a given database will be perceived by different users in many different ways. In fact, even when two users share the same portion of the database, their views of that portion might differ considerably at a detailed level. This latter point is discussed more fully in Section 1.5 and in later chapters (especially Chapter 10).

We will have more to say regarding the nature of the data component of the system in Section 1.3.

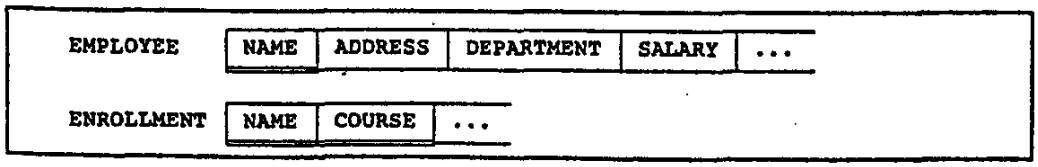


Fig. 1.5 The EMPLOYEE and ENROLLMENT files

Hardware

The hardware components of the system consist of:

- The secondary storage volumes—typically magnetic disks—that are used to hold the stored data, together with the associated I/O devices (disk drives, etc.), device controllers, I/O channels, and so forth
- The hardware processor(s) and associated main memory that are used to support the execution of the database system software (see the next subsection)

This book does not concern itself very much with the hardware aspects of the system, for the following reasons among others: First, those aspects form a major topic in their own right; second, the problems encountered in this area are not peculiar to database systems; and third, those problems have been very thoroughly investigated and documented elsewhere.

Software

Between the physical database itself—that is, the data as physically stored—and the users of the system is a layer of software, known variously as the database manager or database server or, most commonly, the database management system (DBMS). All requests for access to the database are handled by the DBMS; the facilities sketched in Section 1.1 for adding and removing files (or tables), retrieving data from and updating data in such files or tables, and so on, are all facilities provided by the DBMS. One general function provided by the DBMS is thus *the shielding of database users from hardware-level details* (much as programming language systems shield application programmers from hardware-level details). In other words, the DBMS provides users with a perception of the database that is elevated somewhat above the hardware level, and it supports user operations (such as the SQL operations discussed briefly in Section 1.1) that are expressed in terms of that higher-level perception. We will discuss this function, and other functions of the DBMS, in considerably more detail throughout the body of the book.

A couple of further points:

- The DBMS is easily the most important software component in the overall system, but it is not the only one. Others include utilities, application development tools, design aids, report writers, and (most significant) the *transaction manager* or *TP monitor*. See Chapters 2, 3, and especially 15 and 16 for further discussion of these components.
- The term *DBMS* is also used to refer generically to some particular product from some particular vendor—for example, IBM's DB2 Universal Database product. The term *DBMS instance* is then sometimes used to refer to the particular copy of such a product that happens to be running at some particular computer installation. As you will surely appreciate, sometimes it is necessary to distinguish carefully between these two concepts.

That said, you should be aware that people often use the term *database* when they really mean *DBMS* (in either of the foregoing senses). Here is a typical example: "Vendor *X*'s database outperformed vendor *Y*'s database by a factor of two to one." This usage is sloppy, and deprecated, but very, very common. (The problem is: If we call the DBMS the database, what do we call the database? *Caveat lector!*)

Users

We consider three broad (and somewhat overlapping) classes of users:

- First, there are **application programmers**, responsible for writing database application programs in some programming language, such as COBOL, PL/I, C++, Java, or some higher-level "fourth-generation" language (see Chapter 2). Such programs access the database by issuing the appropriate request (typically an SQL statement) to the DBMS. The programs themselves can be traditional batch applications, or they can be online applications, whose purpose is to allow an end user—see the next paragraph—to access the database interactively (e.g., from an online workstation or terminal or a personal computer). Most modern applications are of the online variety.
- Next, there are **end users**, who access the database interactively as just described. A given end user can access the database via one of the online applications mentioned in the previous paragraph, or he or she can use an interface provided as an integral part of the system. Such vendor-provided interfaces are also supported by means of online applications, of course, but those applications are built in, not user-written. Most systems include at least one such built-in application, called a **query language processor**, by which the user can issue database requests such as SELECT and INSERT to the DBMS interactively. SQL is a typical example of a database query language. (As an aside, we remark that the term *query language*, common though it is, is really a misnomer, inasmuch as the verb "to query" suggests *retrieval* only, whereas query languages usually—not always—provide update and other operators as well.)

  Most systems also provide additional built-in interfaces in which end users do not issue explicit database requests such as SELECT and INSERT at all, but instead operate by (e.g.) choosing items from a menu or filling in boxes on a form. Such menu- or forms-driven interfaces tend to be easier to use for people who do not have a formal training in IT (IT = information technology; the abbreviation IS, short for information systems, is also used with much the same meaning). By contrast, command-driven interfaces—that is, query languages—do tend to require a certain amount of professional IT expertise, though perhaps not much (obviously not as much as is needed to write an application program in a language like COBOL). Then again, a command-driven interface is likely to be more flexible than a menu- or forms-driven one, in that query languages typically include certain features that are not supported by those other interfaces.
- The third class of user, not illustrated in Fig. 1.4, is the **database administrator** or DBA. Discussion of the database administration function—and the associated (very

important) data administration function—is deferred to Section 1.4 and Chapter 2 (Section 2.7).

This completes our preliminary description of the major aspects of a database system. We now go on to discuss the ideas in more detail.

1.3 WHAT IS A DATABASE?

Persistent Data

It is customary to refer to the data in a database as "persistent" (though it might not actually persist for very long!). By *persistent*, we mean, intuitively, that database data differs in kind from other more ephemeral data, such as input data, output data, work queues, software control blocks, SQL statements, intermediate results, and more generally any data that is transient in nature. More precisely, we say that data in the database "persists" because, once it has been accepted by the DBMS for entry into the database in the first place, *it can subsequently be removed from the database only by some explicit request to the DBMS*, not as a mere side effect of (e.g.) some program completing execution. This notion of persistence thus allows us to give a slightly more precise definition for the term *database:*

- A database is a collection of persistent data that is used by the application systems of some given enterprise.

The term *enterprise* here is simply a convenient generic term for any reasonably self-contained commercial, scientific, technical, or other organization. An enterprise might be a single individual (with a small personal database), or a complete corporation or similar large body (with a large shared database), or anything in between. Here are some examples:

1. A manufacturing company
2. A bank
3. A hospital
4. A university
5. A government department

Any enterprise must necessarily maintain a lot of data about its operation. That data is the "persistent data" referred to in the definition. The enterprises just mentioned would typically include the following (respectively) among their persistent data:

1. Product data
2. Account data
3. Patient data
4. Student data
5. Planning data

Note: The first six editions of this book used the term *operational data* in place of *persistent data.* That earlier term reflected the original emphasis in database systems on operational or production applications—that is, routine, highly repetitive applications that were executed over and over again to support the day-to-day operation of the enterprise (for example, an application to support the deposit or withdrawal of cash in a banking system). The term ***online transaction processing*** (OLTP) has come to be used to refer to this kind of environment. However, databases are now increasingly used for other kinds of applications as well—that is, decision support applications—and the term *operational data* is thus no longer entirely appropriate. Indeed, enterprises nowadays frequently maintain two separate databases, one containing operational data and one, often called the data warehouse, containing decision support data. The data warehouse often includes *summary information* (e.g., totals, averages), where the summary information in question is extracted from the operational database on a periodic basis—say once a day or once a week. See Chapter 22 for an in-depth treatment of decision support databases and applications.

Entities and Relationships

We now consider the example of a manufacturing company ("KnowWare Inc.") in a little more detail. Such an enterprise will typically wish to record information about the *projects* it has on hand; the *parts* that are used in those projects; the *suppliers* who are under contract to supply those parts; the *warehouses* in which those parts are stored; the *employees* who work on those projects; and so on. Projects, parts, suppliers, and so on, thus constitute the basic entities about which KnowWare Inc. needs to record information (the term *entity* is commonly used in database circles to mean any distinguishable object that is to be represented in the database). Refer to Fig. 1.6.

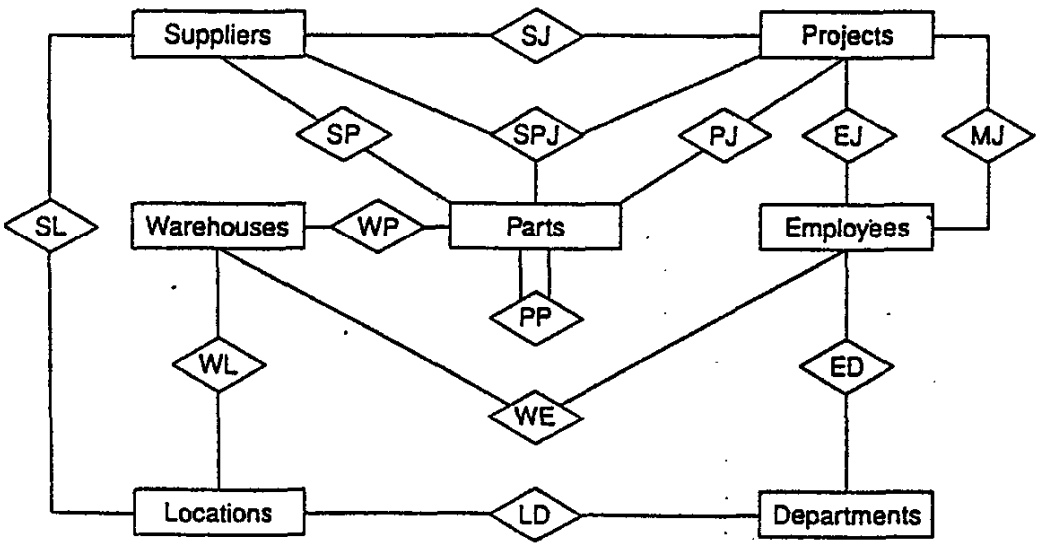


Fig. 1.6 Entity/relationship (E/R) diagram for KnowWare Inc.

In addition to the basic entities themselves (suppliers, parts, and so on, in the example), there will also be relationships linking those basic entities together. Such relationships are represented by diamonds and connecting lines in Fig. 1.6. For example, there is a relationship ("SP" or *shipments*) between suppliers and parts: Each supplier supplies certain parts, and conversely each part is supplied by certain suppliers (more accurately, each supplier supplies certain *kinds* of parts, each *kind* of part is supplied by certain suppliers). Similarly, parts are used in projects, and conversely projects use parts (relationship PJ); parts are stored in warehouses, and warehouses store parts (relationship WP); and so on. Note that these relationships are all *bidirectional*—that is, they can be traversed in either direction. For example, relationship SP between suppliers and parts can be used to answer both of the following queries:

- Given a supplier, get the parts supplied by that supplier.
- Given a part, get the suppliers who supply that part.

The significant point about this relationship (and all of the others illustrated in the figure) is that *they are just as much a part of the data as are the basic entities.* They must therefore be represented in the database, just like the basic entities.[1]

We note in passing that Fig. 1.6 is an example of what is called (for obvious reasons) an entity/relationship diagram (E/R diagram for short). We will consider such diagrams in detail in Chapter 14.

Fig. 1.6 also illustrates a number of other important points:

1. Although most of the relationships in that figure involve two entity types—that is, they are *binary* relationships—it is by no means the case that all relationships are binary in this sense. In the example there is one relationship ("SPJ") involving three entity types (suppliers, parts, and projects): a *ternary* relationship. The intended interpretation is that certain suppliers supply certain parts to certain projects. Note carefully that this ternary relationship ("suppliers supply parts to projects") is *not* equivalent, in general, to the combination of the three binary relationships "suppliers supply parts," "parts are used in projects," and "projects are supplied by suppliers." For example, the statement[2] that

   a. Smith supplies monkey wrenches to the Manhattan project

   tells us *more* than the following three statements do:

   b. Smith supplies monkey wrenches

   c. Monkey wrenches are used in the Manhattan project

   d. The Manhattan project is supplied by Smith

[1] In a relational database specifically (see Section 1.6), the basic entities and the relationships connecting them are both represented by means of *relations*, or in other words by tables like the one shown in Fig. 1.1, loosely speaking. Note carefully, therefore, that the term *relationship* as used in the present section and the term *relation* as used in the context of relational databases do not mean the same thing.

[2] The term *statement* is unfortunately used in the database world to mean two rather different things: It can be used, as here, to mean an *assertion of fact*, or what logicians call a *proposition* (see the subsection "Data and Data Models" later in this section); it can also be used, as we already know from earlier discussions, as a synonym for *command*, as in the expression "SQL statement."

—we cannot (validly!) infer *a* knowing only *b*, *c*, and *d*. More precisely, if we know *b*, *c*, and *d*, then we might be able to infer that Smith supplies monkey wrenches to *some* project (say project *Jz*), that *some* supplier (say supplier *Sx*) supplies monkey wrenches to the Manhattan project, and that Smith supplies *some* part (say part *Py*) to the Manhattan project—but we cannot validly infer that *Sx* is Smith or *Py* is monkey wrenches or *Jz* is the Manhattan project. False inferences such as these are examples of what is sometimes called the **connection trap**.

2. The figure also shows one relationship (PP) involving just *one* entity type (parts). The relationship here is that certain parts include other parts as immediate components (the so-called **bill-of-materials** relationship); for example, a screw is a component of a hinge assembly, which is also a part and might in turn be a component of some higher-level part such as a lid. Note that this relationship is still binary; it is just that the two entity types involved, parts and parts, happen to be one and the same.
3. In general, a given set of entity types might be involved in any number of distinct relationships. In the example in Fig. 1.6, there are two distinct relationships involving projects and employees: One (EJ) represents the fact that employees are assigned to projects; the other (MJ) represents the fact that employees manage projects.

We now observe that *a relationship can be regarded as an entity in its own right.* If we take as our definition of entity "any object about which we wish to record information," then a relationship certainly fits the definition. For instance, "part P4 is stored in warehouse W8" is an entity about which we might well wish to record information—for example, the corresponding quantity. Moreover, there are definite advantages (beyond the scope of the present chapter) to be obtained by not making any unnecessary distinctions between entities and relationships. In this book, therefore, we will tend to treat relationships as just a special kind of entity.

Properties

As just indicated, an entity is any object about which we wish to record information. It follows that entities (relationships included) can be regarded as having properties, corresponding to the information we wish to record about them. For example, suppliers have *locations;* parts have *weights;* projects have *priorities;* assignments (of employees to projects) have *start dates;* and so on. Such properties must therefore be represented in the database. For example, an SQL database might include a table called S representing suppliers, and that table might include a column called CITY representing supplier locations.

Properties in general can be as simple or as complex as we please. For example, the "supplier location" property is presumably quite simple, consisting as it does of just a city name, and can be represented in the database by a simple character string. By contrast, a warehouse might have a "floor plan" property, and that property might be quite complex, consisting perhaps of an entire architectural drawing and associated descriptive text. As noted in Section 1.1, in other words, the kinds of data we might want to keep in (for example) columns of SQL tables can be arbitrarily complex. As also noted in that same section, we will return to this topic later (principally in Chapters 5–6 and 26–27); until

then, we will mostly assume, where it makes any difference, that properties are "simple" and can be represented by "simple" data types. Examples of such "simple" types include numbers, character strings, dates, times, and so forth.

Data and Data Models

There is another (and important) way of thinking about what data and databases really are. The word *data* derives from the Latin for "to give"; thus, data is really *given facts*, from which additional facts can be inferred. (Inferring additional facts from given facts is exactly what the DBMS does when it responds to a user query.) A "given fact" in turn corresponds to what logicians call a *true proposition;* for example, the statement "Supplier S1 is located in London" might be such a true proposition. (A proposition in logic is something that is either true or false, unequivocally. For instance, "William Shakespeare wrote *Pride and Prejudice*" is a proposition—a false one, as it happens.) It follows that a database is really a collection of true propositions.

Now, we have already said that SQL products have come to dominate the marketplace. One reason for this state of affairs is that SQL products are based on a formal theory called the relational model of data, and that theory in turn supports the foregoing interpretation of data and databases very directly—almost trivially, in fact. To be specific, in the relational model:

1. Data is represented by means of rows in tables,[3] and such rows can be directly interpreted as true propositions. For example, the row for BIN# 72 in Fig. 1.1 can be interpreted in an obvious way as the following true proposition:

   Bin number 72 contains two bottles of 1999 Rafanelli Zinfandel, which will be ready to drink in 2007

2. Operators are provided for operating on rows in tables, and those operators directly support the process of inferring additional true propositions from the given ones. As a simple example, the relational *project* operator (see Section 1.6) allows us to infer, from the true proposition just quoted, the following additional true proposition among others:

   Some bottles of Zinfandel will be ready to drink in 2007

   (More precisely: "Some bottles of Zinfandel in some bin, produced by some producer in some year, will be ready to drink in 2007.")

The relational model is not the only data model, however; others do exist (see Section 1.6)—though most of them differ from the relational model in being *ad hoc* to a degree, instead of being firmly based as the relational model is on formal logic. Be that as it may, the question arises: What in general *is* a data model? Following reference [1.1] (but paraphrasing considerably), we can define the concept thus:

- A data model is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users

[3] More precisely, by *tuples* in *relations* (see Chapter 3).

interact. The objects allow us to model the *structure* of data. The operators allow us to model its *behavior.*

We can then draw a useful (and very important!) distinction between the model and its *implementation:*

- An implementation of a given data model is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

In a nutshell: The model is what users have to know about; the implementation is what users do not have to know about.

As you can see from the foregoing, the distinction between model and implementation is really just a special case—a very important special case—of the familiar distinction between *logical* and *physical.* Sadly, however, many of today's database systems, even ones that profess to be relational, do not make these distinctions as clearly as they should. Indeed, there seems to be a fairly widespread lack of understanding of these distinctions and the importance of making them. As a consequence, there is all too often a gap between database *principles* (the way database systems ought to be) and database *practice* (the way they actually are). In this book we are concerned primarily with principles, but it is only fair to warn you that you might therefore be in for a few surprises, mostly of an unpleasant nature, if and when you start using a commercial product.

In closing this section, we should mention the fact that *data model* is another term that is used in the literature with two quite different meanings. The first meaning is as already explained. The second is as follows: A data model (second sense) is *a model of the persistent data of some particular enterprise* (e.g., the manufacturing company KnowWare Inc. discussed earlier in this section). The difference between the two meanings can be characterized thus:

- A data model in the first sense is like a *programming language*—albeit one that is somewhat abstract—whose constructs can be used to solve a wide variety of specific problems, but in and of themselves have no direct connection with any such specific problem.
- A data model in the second sense is like a *specific program* written in that language. In other words, a data model in the second sense takes the facilities provided by some model in the first sense and applies them to some specific problem. It can be regarded as *a specific application* of some model in the first sense.

In this book, the term *data model* will be used only in the first sense from this point forward, barring explicit statements to the contrary.

1.4 WHY DATABASE?

Why use a database system? What are the advantages? To some extent the answer to these questions depends on whether the system in question is single- or multi-user (or rather, to be more accurate, there are numerous *additional* advantages in the multi-user case). We consider the single-user case first.

Refer back to the wine cellar example once again (Fig. 1.1), which we can regard as illustrative of the single-user case. Now, that particular database is so small and so simple that the advantages might not be all that obvious. But imagine a similar database for a large restaurant, with a stock of perhaps thousands of bottles and very frequent changes to that stock; or think of a liquor store, with again a very large stock and high turnover on that stock. The advantages of a database system over traditional, paper-based methods of record-keeping are perhaps easier to see in these cases. Here are some of them:

- *Compactness:* There is no need for possibly voluminous paper files.
- *Speed:* The machine can retrieve and update data far faster than a human can. In particular, *ad hoc*, spur-of-the-moment queries (e.g., "Do we have more Zinfandel than Pinot Noir?") can be answered quickly without any need for time-consuming manual or visual searches.
- *Less drudgery:* Much of the sheer tedium of maintaining files by hand is eliminated. Mechanical tasks are always better done by machines.
- *Currency:* Accurate, up-to-date information is available on demand at any time.
- *Protection:* The data can be better protected against unintentional loss and unlawful access.

The foregoing benefits apply with even more force in a multi-user environment, where the database is likely to be much larger and more complex than in the single-user case. However, there is one overriding additional advantage in such an environment: *The database system provides the enterprise with centralized control of its data* (which, as you should realize by now, is one of its most valuable assets). Such a situation contrasts sharply with that found in an enterprise without a database system, where typically each application has its own private files—quite often its own private tapes and disks, too—so that the data is widely dispersed and difficult to control in any systematic way.

Data Administration and Database Administration

We elaborate briefly on this concept of centralized control. The concept implies that there will be some identifiable person in the enterprise who has this central responsibility for the data, and that person is the **data administrator** (DA for short) mentioned briefly at the end of Section 1.2. Given that, to repeat, the data is one of the enterprise's most valuable assets, it is imperative that there should be some person who understands that data, and the needs of the enterprise with respect to that data, *at a senior management level.* The data administrator is that person. Thus, it is the data administrator's job to decide what data should be stored in the database in the first place, and to establish policies for maintaining and dealing with that data once it has been stored. An example of such a policy might be one that dictates who can perform what operations on what data in what circumstances—in other words, a *data security* policy (see the next subsection).

Note carefully that the data administrator is a manager, not a technician (although he or she certainly does need to have some broad appreciation of the capabilities of database systems at a technical level). The *technical* person responsible for implementing the data

administrator's decisions is the data*base* administrator (DBA for short). The DBA, unlike the data administrator, is thus an information technology (IT) professional. The job of the DBA is to create the actual database and to put in place the technical controls needed to enforce the various policy decisions made by the data administrator. The DBA is also responsible for ensuring that the system operates with adequate performance and for providing a variety of other technical services. The DBA will typically have a staff of system programmers and other technical assistants (i.e., the DBA function will typically be performed in practice by a team of people, not just by one person); for simplicity, however, it is convenient to assume that the DBA is indeed a single individual. We will discuss the DBA function in more detail in Chapter 2.

Benefits of the Database Approach

In this subsection we identify some more specific advantages that accrue from the foregoing notion of centralized control.

- *The data can be shared.*

  We discussed this point in Section 1.2, but for completeness we mention it again here. Sharing means not only that existing applications can share the data in the database, but also that new applications can be developed to operate against that same data. In other words, it might be possible to satisfy the data requirements of new applications without having to add any new data to the database.

- *Redundancy can be reduced.*

  In nondatabase systems each application has its own private files. This fact can often lead to considerable redundancy in stored data, with resultant waste in storage space. For example, a personnel application and an education records application might both own a file that includes department information for employees. As suggested in Section 1.2, however, those two files can be integrated, and the redundancy eliminated, as long as the data administrator is aware of the data requirements for both applications—that is, as long as the enterprise has the necessary overall control.

  Note: We do not mean to say that *all* redundancy can or necessarily should be eliminated. Sometimes there are sound business or technical reasons for maintaining several distinct copies of the same data. However, we do mean that any such redundancy should be carefully controlled; that is, the DBMS should be aware of it, if it exists, and should assume responsibility for "propagating updates" (see the point immediately following).

- *Inconsistency can be avoided (to some extent).*

  This is really a corollary of the previous point. Suppose a given fact about the real world—say the fact that employee E3 works in department D8—is represented by two distinct entries in the database. Suppose also that the DBMS is not aware of this duplication (i.e., the redundancy is not controlled). Then there will necessarily be occasions on which the two entries will not agree: namely, when one of the two has been updated and the other not. At such times the database is said to be *inconsistent*. Clearly, a data-

base that is in an inconsistent state is capable of supplying incorrect or contradictory information to its users.

Of course, if the given fact is represented by a single entry (i.e., if the redundancy is removed), then such an inconsistency cannot occur. Alternatively, if the redundancy is not removed but is *controlled* (by making it known to the DBMS), then the DBMS can guarantee that the database is never inconsistent *as seen by the user*, by ensuring that any change made to either of the two entries is automatically applied to the other one as well. This process is known as propagating updates.

- *Transaction support can be provided.*

  A transaction is a logical unit of work (more precisely, a logical unit of *database* work), typically involving several database operations—in particular, several update operations. The standard example involves the transfer of a cash amount from one account *A* to another account *B*. Clearly two updates are required here, one to withdraw the cash from account *A* and the other to deposit it to account *B*. If the user has made the two updates part of the same transaction, then the system can effectively guarantee that either both of them are done or neither is—even if, for example, the system fails (say because of a power outage) halfway through the process.

  Note: The transaction *atomicity* feature just illustrated is not the only benefit of transaction support, but unlike some of the others it is one that applies, at least in principle, even in the single-user case. (On the other hand, single-user systems often do not provide any transaction support at all but simply leave the problem to the user.) A full description of all of the various advantages of transaction support and how they can be achieved appears in Chapters 15 and 16.

- *Integrity can be maintained.*

  The problem of integrity is the problem of ensuring (as far as possible) that the data in the database is correct. Inconsistency between two entries that purport to represent the same fact is an example of lack of integrity (see the discussion of this point earlier in this subsection); of course, this particular problem can arise only if redundancy exists in the stored data. Even if there is no redundancy, however, the database might still contain incorrect information. For example, an employee might be shown as having worked 400 hours in the week instead of 40, or as belonging to a department that does not exist. Centralized control of the database can help in avoiding such problems—insofar as they can be avoided—by permitting the data administrator to define, and the DBA to implement, integrity constraints to be checked when update operations are performed.

  It is worth pointing out that data integrity is even more important in a database system than it is in a "private files" environment, precisely because the data is shared. For without appropriate controls it would be possible for one user to update the database incorrectly, thereby generating bad data and so "infecting" other innocent users of that data. It should also be mentioned that most database products are still quite weak in their support for integrity constraints (though there have been some recent improvements in this area). This fact is unfortunate, given that (as we will see in Chapter 9) integrity constraints are both fundamental and crucially important—much more so than is usually realized, in fact.

- *Security can be enforced.*

  Having complete jurisdiction over the database, the DBA (under appropriate direction from the data administrator) can ensure that the only means of access to the database is through the proper channels, and hence can define security constraints to be checked whenever access is attempted to sensitive data. Different constraints can be established for each type of access (retrieve, insert, delete, etc.) to each piece of information in the database. Note, however, that without such constraints the security of the data might actually be more at risk than in a traditional (dispersed) filing system: that is, the centralized nature of a database system in a sense *requires* that a good security system be in place also.

- *Conflicting requirements can be balanced.*

  Knowing the overall requirements of the enterprise, as opposed to the requirements of individual users, the DBA (again under the data administrator's direction) can so structure the system as to provide an overall service that is "best for the enterprise." For example, a physical representation can be chosen for the data in storage that gives fast access for the most important applications (possibly at the cost of slower access for certain other applications).

- *Standards can be enforced.*

  With central control of the database, the DBA (under the direction of the data administrator once again) can ensure that all applicable standards are observed in the representation of the data, including any or all of the following: departmental, installation, corporate, industry, national, and international standards. Standardizing data representation is particularly desirable as an aid to *data interchange*, or movement of data between systems (this consideration is becoming particularly important with the advent of distributed systems—see Chapters 2, 21, and 27). Likewise, data naming and documentation standards are also very desirable as an aid to data sharing and understandability.

Now, most of the advantages listed so far are probably fairly obvious. However, one further point—which might not be as obvious, although it is in fact implied by several of the others—needs to be added to the list: *the provision of data independence*. (Strictly speaking, this is an *objective* for database systems, rather than an advantage as such.) The concept of data independence is so important that we devote a separate section to it.

1.5 DATA INDEPENDENCE

We begin by observing that there are two kinds of data independence, physical and logical [1.3, 1.4]; for the time being, however, we will concern ourselves with the physical kind only. Until further notice, therefore, the unqualified term *data independence* should be understood to mean physical data independence specifically. (We should perhaps add that the term *data independence* is not very apt—it does not capture very well the essence of what is really going on—but it is the term traditionally used, and we will stay with it in this book.)

Data independence can most easily be understood by first considering its opposite. Applications implemented on older systems—prerelational or even predatabase systems—tend to be *data-dependent*. What this means is that the way the data is physically represented in secondary storage, and the techniques used to access it, are both dictated by the requirements of the application under consideration, and moreover that *knowledge of that physical representation and those access techniques is built into the application code.* For example, suppose we have an application that uses the EMPLOYEE file of Fig. 1.5, and suppose it is decided, for performance reasons, that the file is to be indexed on its "employee name" field (see Appendix D, online). In an older system, the application in question will typically be aware of the fact that the index exists, and aware also of the sequence of records as defined by that index, and the internal structure of the application will be built around that knowledge. In particular, the precise form of the various data access and exception-checking routines within the application will depend very heavily on details of the interface presented to the application by the data management software.

We say that an application such as the one in this example is data-dependent, because it is impossible to change the physical representation (how the data is physically represented in storage) and access techniques (how it is physically accessed) without affecting the application, probably drastically. For instance, it would not be possible to replace the index in the example by a hashing scheme without making major modifications to the application code. What is more, the portions of that code requiring alteration in such a situation are precisely those portions that communicate with the data management software; the difficulties involved are quite irrelevant to the problem the application was originally written to solve—that is, they are difficulties *introduced* by the nature of the data management interface.

In a database system, however, it would be extremely undesirable to allow applications to be data-dependent in the foregoing sense, for at least the following two reasons:

1. Different applications will require different views of the same data. For example, suppose that before the enterprise introduces its integrated database there are two applications, *A* and *B*, each owning a private file that includes the field "customer balance." Suppose, however, that application *A* stores that field in decimal, whereas application *B* stores it in binary. It will still be possible to integrate the two files, and to eliminate the redundancy, provided the DBMS is ready and able to perform all necessary conversions between the stored representation chosen (which might be decimal or binary or something else again) and the form in which each application wishes to see it. For example, if it is decided to store the field in decimal, then every access by *B* will require a conversion to or from binary.

   This is a fairly trivial example of the kind of difference that might exist in a database system between the data as seen by a given application and the data as physically stored. We will consider many other possible differences later in this section.

2. The DBA—or possibly the DBMS—must have the freedom to change the physical representation and access technique in response to changing requirements, without existing applications having to be modified. For example, new kinds of data might be added to the database; new standards might be adopted; application priorities (and therefore relative performance requirements) might change; new storage devices

might become available; and so on. If applications are data-dependent, such changes will typically require corresponding changes to program code, thus tying up programmer effort that would otherwise be available for the creation of new applications. It is still not uncommon, even today, to find a significant fraction of the available programming effort devoted to this kind of maintenance (think of all the work that went into addressing the "Year 2000" problem)—hardly the best use of a scarce and valuable resource.

It follows that the provision of data independence is a major objective for database systems. Data independence can be defined as the immunity of applications to change in physical representation and access technique—which implies that the applications in question do not depend on any particular physical representation or access technique. In Chapter 2, we describe an architecture for database systems that provides a basis for achieving this objective. Before then, however, let us consider in more detail some examples of the types of changes that the DBA might wish to make, and that we might therefore wish applications to be immune to.

We start by defining three terms: *stored field*, *stored record*, and *stored file* (refer to Fig. 1.7).

- A stored field is, loosely, the smallest unit of stored data. The database will contain many occurrences (or instances) of each of several types of stored field. For example, a database containing information about different kinds of parts might include a stored field type called "part number," and then there would be one occurrence of that stored field for each kind of part (screw, hinge, lid, etc.).

  Note: The foregoing paragraph notwithstanding, you should be aware that it is common in practice to drop the qualifiers *type* and *occurrence* and to rely on context to indicate which is meant. Despite a small risk of confusion, the practice is convenient, and we will adopt it ourselves from time to time in what follows. (These remarks apply to stored records as well—see the paragraph immediately following.)

- A stored record is a collection of related stored fields. Again we distinguish between type and occurrence. A stored record occurrence (or instance) consists of a group of related stored field occurrences. For example, a stored record occurrence in the "parts" database might consist of an occurrence of each of the following stored fields: part number, part name, part color, and part weight. We say that the database contains many occurrences of the "part" stored record type—again, one occurrence for each kind of part.

- Finally, a stored file is the collection of all currently existing occurrences of one type of stored record. (We assume for simplicity that any given stored file contains just one type of stored record. This simplification does not materially affect any of our subsequent discussions.)

Now, in nondatabase systems it is normally the case that any given *logical* record as seen by some application is identical to some corresponding *stored* record. As we have already seen, however, this is not necessarily the case in a database system, because the DBA might need to be able to make changes to the stored representation of data—that is,

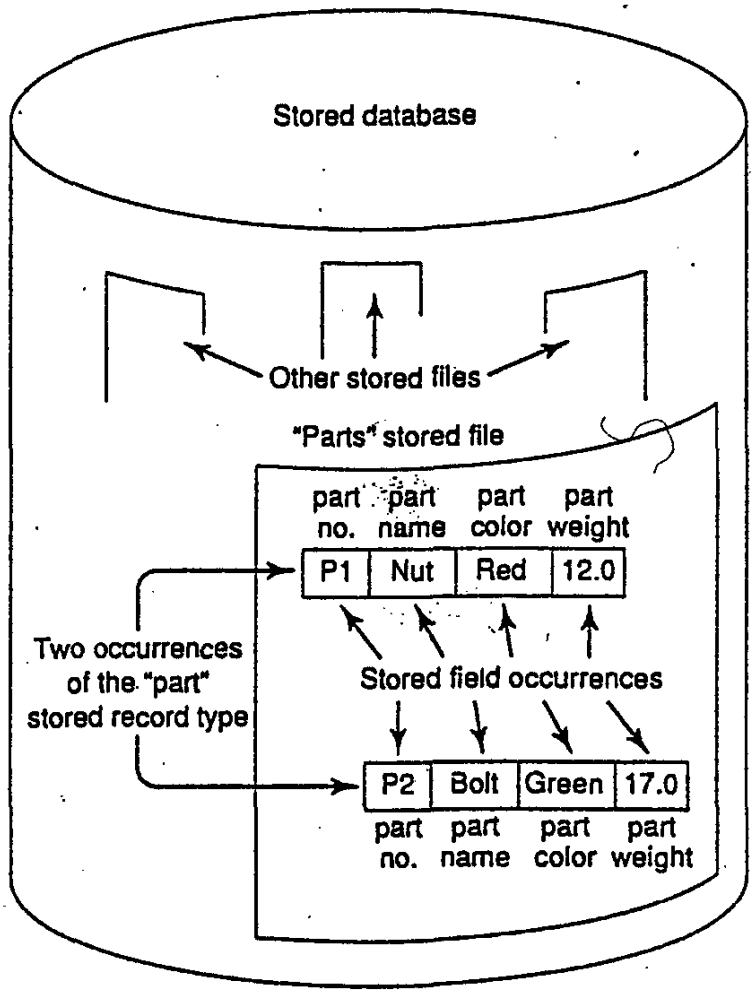


Fig. 1.7 Stored fields, records, and files

to the stored fields, records, and files—while the data as seen by applications does *not* change. For example, the SALARY field in the EMPLOYEE file might be stored in binary to economize on storage space, whereas a given COBOL application might see it as a character string. And later the DBA might decide for some reason to change the stored representation of that field from binary to decimal, say, and yet still allow the COBOL application to see it in character form.

As stated earlier, a difference such as this one—involving data type conversion on some field on each access—is comparatively minor. In general, however, the difference between what the application sees and what is physically stored might be quite considerable. To amplify this remark, we briefly consider some aspects of the stored representation that might be subject to change. You should consider in each case what the DBMS would have to do to make applications immune to such change (and indeed whether such immunity can always be achieved).

- *Representation of numeric data*

  A numeric field might be stored in internal arithmetic form (e.g., packed decimal) or as a character string. Either way, the DBA must choose whether to use *fixed* or *floating point;* what *base* or *radix* (e.g., binary or decimal) to use; what the *precision* (number of digits) should be; and, if fixed point, what the *scale factor* (number of digits after the radix point) should be. Any of these aspects might need to be changed to improve performance or to conform to a new standard or for many other reasons.

- *Representation of character data*

  A character string field might be stored using any of several distinct coded character sets—for example, ASCII, EBCDIC, or Unicode.

- *Units for numeric data*

  The units in a numeric field might change—from inches to centimeters, for example, during a process of metrication.

- *Data coding*

  In some situations it might be desirable to represent data in storage by coded values. For example, the "part color" field, which an application sees as a character string ("Red" or "Blue" or "Green" . . .), might be stored as a single decimal digit, interpreted according to the coding scheme 1 = "Red," 2 = "Blue," and so on.

- *Data materialization*

  In practice the logical field seen by an application usually does correspond directly to some specific stored field (although there might be differences in data type, coding, and so on, as we have seen). If it does, then the process of *materialization*—that is, constructing an occurrence of the logical field from the corresponding stored field occurrence and presenting it to the application—is said to be *direct.* Sometimes, however, a logical field will have no single stored counterpart; instead, its values will be materialized by means of some computation, perhaps on several stored field occurrences. For example, values of the logical field "total quantity" might be materialized by summing a number of individual stored quantities. In such a case, the materialization process is said to be *indirect.*

- *Structure of stored records*

  Two existing stored records might be combined into one. For example, the stored records

part no.	part color

  and

part no.	part weight

  might be combined to form

part no.	part color	part weight

  Such a change might occur as new applications are integrated into the database system. The implication is that a given application's logical record might consist of a

proper subset of the corresponding stored record—that is, certain fields in that stored record would be invisible to the application in question.

Alternatively, a single stored record type might be split into two. Reversing the previous example, the stored record

part no.	part color	part weight

might be split into

part no.	part color

and

part no.	part weight

Such a split would allow less frequently used portions of the original record to be stored on a slower device, for example. The implication is that a given application's logical record might contain fields from several distinct stored records—that is, it might be a proper superset of any given one of those stored records.

- *Structure of stored files*

  A given stored file can be physically implemented in storage in a wide variety of ways (see Appendix D, online). For example, it might be entirely contained within a single storage volume (e.g., a single disk), or it might be spread across several volumes (possibly on several different device types); it might or might not be physically sequenced according to the values of some stored field; it might or might not be sequenced in one or more additional ways by some other means (e.g., by one or more indexes or one or more embedded pointer chains or both); it might or might not be accessible via some hashing scheme; the stored records might or might not be physically grouped into blocks; and so on. But none of these considerations should affect applications in any way (other than performance, of course).

This concludes our partial list of aspects of the stored data representation that are subject to possible change. Note that the list implies in particular that the database should be able to grow without impairing existing applications; indeed, enabling the database to grow without logically impairing existing applications is one of the most important reasons for requiring data independence in the first place. For example, it should be possible to extend an existing stored record by the addition of new stored fields, representing, typically, further information concerning some existing type of entity (e.g., a "unit cost" field might be added to the "part" stored record). Such new fields should simply be invisible to existing applications. Likewise, it should be possible to add entirely new stored record types and new stored files, again without requiring any change to existing applications; such new records and files would typically represent new entity types (e.g., a "supplier" record type might be added to the "parts" database). Again, such additions should be invisible to existing applications.

As you might have realized by now, data independence is one of the reasons why separating the data model from its implementation, as discussed near the end of Section 1.3, is so important: To the extent we do *not* make that separation, we will not achieve data independence. The widespread failure to make the separation properly, in today's SQL systems in particular, is thus particularly distressing. *Note:* We do not mean to suggest by

these remarks that today's SQL systems provide no data independence at all—only that they provide much less than relational systems are theoretically capable of.[4] In other words, data independence is not an absolute; different systems provide it to different degrees, and few if any provide none at all. Today's SQL systems typically provide more data independence than older systems did, but they are still far from perfect, as we will see in the chapters to come.

1.6 RELATIONAL SYSTEMS AND OTHERS

We have said that SQL systems have come to dominate the DBMS marketplace, and that one important reason for this state of affairs is that such systems are based on *the relational model of data*. Informally, indeed, SQL systems are often referred to as *relational systems* specifically.[5] In addition, the vast majority of database research over the last 30 years or so has also been based (albeit a little indirectly, in some cases) on the relational model. Indeed, it is fair to say that the introduction of the relational model in 1969–70 was *the single most important event in the entire history of the database field*. For these reasons, plus the fact that the relational model is solidly based on logic and mathematics and therefore provides an ideal vehicle for teaching database foundations and principles, the emphasis in this book is heavily on relational systems.

What then exactly is a relational system? It is obviously not possible to answer this question properly at this early point in the book—but it is possible, and desirable, to give a rough and ready answer, which we can make more precise later. Briefly, then (albeit very loosely), a relational system is a system in which:

1. The data is perceived by the user as tables (and nothing but tables).
2. The operators available to the user for (e.g.) retrieval are operators that derive "new" tables from "old" ones. For example, there is one operator, *restrict*, which extracts a subset of the rows of a given table, and another, *project*, which extracts a subset of the columns—and a row subset and a column subset of a table can both be regarded in turn as tables in their own right, as we will see in just a moment.

So why are such systems called "relational"? The reason is that *relation* is basically just a mathematical term for a table. (Indeed, the terms *relation* and *table*—sometimes *relational* table for emphasis—can be taken as synonymous, at least for informal purposes. See Chapters 3 and 6 for further discussion.) Please note that the reason is definitely *not* that *relation* is "basically just a mathematical term for" a *relationship* in the sense of entity/relationship diagrams as described in Section 1.3; in fact, as noted in that section, there is very little direct connection between relational systems and such diagrams.

As promised, we will make the foregoing definitions much more precise later, but they will serve for the time being. Fig. 1.8 provides an illustration. The data—see part *a* of the figure—consists of a single table, named CELLAR (in fact, it is a scaled-down version

[4] A striking example of what relational systems *are* capable of in this respect is described in Appendix A.

[5] Despite the fact that in many respects SQL is quite notorious for its *departures* from the relational model, as we will see.

a. *Given table:* CELLAR

WINE	YEAR	BOTTLES
Zinfandel	1999	2
Fumé Blanc	2000	2
Pinot Noir	1997	3
Zinfandel	1998	9

b. *Operators (examples):*

1. *Restrict:* *Result:*

```
SELECT WINE, YEAR, BOTTLES
FROM CELLAR
WHERE YEAR > 1998 ;
```

WINE	YEAR	BOTTLES
Zinfandel	1999	2
Fumé Blanc	2000	2

2. *Project:* *Result:*

```
SELECT WINE, BOTTLES
FROM CELLAR ;
```

WINE	BOTTLES
Zinfandel	2
Fumé Blanc	2
Pinot Noir	3
Zinfandel	9

Fig. 1.8 Data structure and operators in a relational system (examples)

of the CELLAR table from Fig. 1.1, reduced in size to make it more manageable). Two sample retrievals, one involving a *restriction* or row-subsetting operation and the other a *projection* or column-subsetting operation, are shown in part *b* of the figure. The examples are expressed in SQL once again.

We can now distinguish between relational and nonrelational systems. In a relational system, the user sees the data as tables, and nothing but tables (as already explained). In a nonrelational system, by contrast, the user sees *other data structures* (either instead of or as well as the tables of a relational system). Those other structures, in turn, require other operators to access them. For example, in a hierarchic system like IBM's IMS, the data is represented to the user in the form of trees (hierarchies), and the operators provided for accessing such trees include operators for *following pointers* (namely, the pointers that implement the hierarchic paths up and down the trees). By contrast, as the examples in this chapter have shown, it is precisely an important distinguishing characteristic of relational systems that they involve no pointers (at least, no pointers visible to the user—i.e., no pointers at the model level—though there might well be pointers at the level of the physical implementation).

As the foregoing discussion suggests, database systems can be conveniently categorized according to the data structures and operators they present to the user. According to this scheme, the oldest (prerelational) systems fall into three broad categories: inverted list, hierarchic, and network systems.[6] (*Note:* The term *network* here has nothing to do

[6] By analogy with the relational model, earlier editions of this book referred to inverted list, hierarchic, and network *models* (and much of the literature still does). To talk in such terms is a little misleading, however, because—unlike the relational model—the inverted list, hierarchic, and network "models" were all invented *after the fact;* that is, commercial inverted list, hierarchic, and network products were implemented *first*, and the corresponding "models" were defined *subsequently* by a process of induction (in this context, a polite term for guesswork) from those existing implementations. See the annotation to reference [1.1] for further discussion.

with networks in the data communications sense, as described in the next chapter.) We do not discuss these categories in detail in this book because—from a technological point of view, at least—they must be regarded as obsolete. You can find tutorial descriptions of all three in reference [1.5] if you are interested.

As an aside, we remark that network systems are sometimes called either CODASYL systems or DBTG systems, after the committee that proposed them: namely, the Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL). Probably the best-known example of such a system is IDMS, from Computer Associates International Inc. Like hierarchic systems (but unlike relational ones), such systems all expose pointers to the user.

The first relational products began to appear in the late 1970s and early 1980s. At the time of writing, the vast majority of database systems are relational (at least, they support SQL), and they run on just about every kind of hardware and software platform available. Leading examples include, in alphabetical order, DB2 (various versions) from IBM Corp., Ingres II from Computer Associates International Inc., Informix Dynamic Server from Informix Software Inc.,[7] Microsoft SQL Server from Microsoft Corp., Oracle 9i from Oracle Corp., and Sybase Adaptive Server from Sybase Inc. *Note:* When we have cause to refer to any of these products later in this book, we will refer to them (as most of the industry does, informally) by the abbreviated names DB2, Ingres (pronounced "ingress"), Informix, SQL Server, Oracle, and Sybase, respectively.

Subsequently, **object** and **object/relational** products began to become available—object systems in the late 1980s and early 1990s, object/relational systems in the late 1990s. The object/relational systems are extended versions of certain of the original SQL products (e.g., DB2, Informix); the object—sometimes *object-oriented*—systems represent attempts to do something entirely different, as in the case of GemStone from GemStone Systems Inc. and Versant ODBMS from Versant Object Technology. Such systems are discussed in Part VI of this book. (We should note that the term *object* as used in this paragraph has a rather specific meaning, which we will explain when we get to Part VI. Prior to that point, we will use the term in its normal generic sense, barring explicit statements to the contrary.)

In addition to the approaches already mentioned, research has proceeded over the years on a variety of alternative schemes, including the **multi-dimensional** approach and the **logic-based** (also called *deductive* or *expert*) approach. We discuss multi-dimensional systems in Chapter 22 and logic-based systems in Chapter 24. Also, the recent explosive growth of the World Wide Web and the use of XML has generated much interest in what has become known (not very aptly) as the **semistructured** approach. We discuss "semistructured" systems in Chapter 27.

1.7 SUMMARY

We close this introductory chapter by summarizing the main points discussed. First, a database system can be thought of as a computerized record-keeping system. Such a sys-

[7] The DBMS division of Informix Software Inc. was acquired by IBM Corp. in 2001.

tem involves the **data** itself (stored in the **database**), **hardware**, **software** (in particular the **database management system** or DBMS), and—most important!—**users**. Users in turn can be divided into **application programmers**, **end users**, and the **database administrator** or DBA. The DBA is responsible for administering the database and database system in accordance with policies established by the **data administrator** or DA.

Databases are **integrated** and (usually) **shared**; they are used to store **persistent** data. Such data can usefully, albeit informally, be considered as representing **entities**, together with **relationships** among those entities—although in fact a relationship is really just a special kind of entity. We very briefly examined the idea of entity/relntionship diagrams.

Database systems provide a number of benefits, of which one of the most important is (physical) **data independence**. Data independence can be defined as the immunity of application programs to changes in the way the data is physically stored and accessed. Among other things, data independence requires that a sharp distinction be made between the data model and its **implementation**. (We remind you in passing that the term *data model*, perhaps unfortunately, has two rather different meanings.)

Database systems also usually support **transactions** or logical units of work. One advantage of transactions is that they are guaranteed to be atomic (all or nothing), even if the system fails in the middle of the transaction in question.

Finally, database systems can be based on a number of different approaches. Relational systems in particular are based on a formal theory called the **relational model**, according to which data is represented as rows in tables (interpreted as **true propositions**), and operators are provided that directly support the process of **inferring** additional true propositions from the given ones. From both an economic and a theoretical perspective, relational systems are easily the most important (and this state of affairs is not likely to change in the foreseeable future). We have seen a few simple examples of SQL, the standard language for relational systems (in particular, examples of the SQL SELECT, INSERT, DELETE, and UPDATE statements). This book is heavily based on relational systems, although—for reasons explained in the preface—not so much on SQL *per se*.

EXERCISES

1.1 *Explain the following in your own words:*

binary relationship	menu-driven interface
command-driven interface	multi-user system
concurrent access	online application
data administration	persistent data
database	property
database system	query language
data independence	redundancy
DBA	relationship
DBMS	security

entity	sharing
entity/relationship diagram	stored field
forms-driven interface	stored file
integration	stored record
integrity	transaction

1.2 What are the advantages of using a database system? What are the disadvantages?

1.3 What do you understand by the term *relational system?* Distinguish between relational and nonrelational systems.

1.4 What do you understand by the term *data model?* Explain the difference between a data model and its implementation. Why is the difference important?

1.5 Show the effects of the following SQL retrieval operations on the wine cellar database of Fig. 1.1:

a.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  BIN# = 72 ;
```

b.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  YEAR > 2000 ;
```

c.
```
SELECT BIN#, WINE, YEAR
FROM   CELLAR
WHERE  READY < 2003 ;
```

d.
```
SELECT WINE, BIN#, YEAR
FROM   CELLAR
WHERE  PRODUCER = 'Robt. Mondavi'
AND    BOTTLES > 6 ;
```

1.6 Give in your own words an interpretation as a true proposition of a typical row from each of your answers to Exercise 1.5.

1.7 Show the effects of the following SQL update operations on the wine cellar database of Fig. 1.1:

a.
```
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 80, 'Syrah', 'Meridian', 1998, 12, 2003 ) ;
```

b.
```
DELETE
FROM   CELLAR
WHERE  READY > 2004 ;
```

c.
```
UPDATE CELLAR
SET    BOTTLES = 5
WHERE  BIN# = 50 ;
```

d.
```
UPDATE CELLAR
SET    BOTTLES = BOTTLES + 2
WHERE  BIN# = 50 ;
```

1.8 Write SQL statements to perform the following operations on the wine cellar database:

a. Get bin number, name of wine, and number of bottles for all Geyser Peak wines.

b. Get bin number and name of wine for all wines for which there are more than five bottles in stock.

c. Get bin number for all red wines.

d. Add three bottles to bin number 30.

e. Remove all Chardonnay from stock.

f. Add an entry for a new case (12 bottles) of Gary Farrell Merlot: bin number 55, year 2000, ready in 2005.

1.9 Suppose you have a music collection consisting of CDs and/or minidiscs and/or LPs and/or audiotapes, and you want to build a database that will let you find which recordings you have for a specific composer (e.g., Sibelius) or conductor (e.g., Simon Rattle) or soloist (e.g., Arthur Grumiaux) or work (e.g., Beethoven's Fifth) or orchestra (e.g., the New York Philharmonic) or kind of work (e.g., violin concerto) or chamber group (e.g., the Kronos Quartet). Draw an entity/relationship diagram like that of Fig. 1.6 for this database.

REFERENCES AND BIBLIOGRAPHY

1.1 E. F. Codd: "Data Models in Database Management," Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo. (June 1980), *ACM SIGMOD Record 11*, No. 2 (February 1981) and elsewhere.

Codd was the inventor of the relational model, which he first described in reference [6.1]. Reference [6.1], however, did not in fact define the term *data model* as such—but the present much later paper does. It also addresses the question: What purposes are data models in general, and the relational model in particular, intended to serve? And it goes on to offer evidence to support the claim that, contrary to popular belief, the relational model was in fact the first data model to be defined. In other words, Codd has some claim to being the inventor of the data model concept in general, as well as of the relational data model in particular.

1.2 Hugh Darwen: "What a Database *Really* Is: Predicates and Propositions," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994–1997*. Reading, Mass.: Addison-Wesley (1998).

This paper gives a very approachable (informal but accurate) explanation of the idea, discussed briefly near the end of Section 1.3, that a database is best thought of as a collection of true propositions.

1.3 C. J. Date and P. Hopewell: "Storage Structures and Physical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

1.4 C. J. Date and P. Hopewell: "File Definition and Logical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

References [1.3] and [1.4] were the first papers to define and distinguish between physical and logical data independence.

1.5 C. J. Date: *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

CHAPTER 2

Database System Architecture

2.1 INTRODUCTION

We are now in a position to present an architecture for a database system. Our aim in presenting this architecture is to provide a framework on which subsequent chapters can build. Such a framework is useful for describing general database concepts and for explaining the structure of specific database systems—but we do not claim that every system can neatly be matched to this particular framework, nor do we mean to suggest that

this particular architecture provides the only possible framework. In particular, "small" systems (see Chapter 1) will probably not support all aspects of the architecture. However, the architecture does seem to fit most systems reasonably well; moreover, it is basically identical to the architecture proposed by the ANSI/SPARC Study Group on Data Base Management Systems (the so-called ANSI/SPARC architecture—see references [2.1] and [2.2]). We choose not to follow the ANSI/SPARC terminology in every detail, however.

Caveat: This chapter resembles Chapter 1 inasmuch as, while an understanding of the material it contains is essential to a full appreciation of the structure and capabilities of a modern database system, it is again somewhat abstract and dry. As with Chapter 1, therefore, you might prefer just to give the material a "once over lightly" reading for now and come back to it later as it becomes more directly relevant to the topics at hand.

2.2 THE THREE LEVELS OF THE ARCHITECTURE

The ANSI/SPARC architecture is divided into three levels, usually referred to as the internal level, the external level, and the conceptual level (see Fig. 2.1), though other names are also used. Broadly speaking:

- The internal level (also known as the *storage* level) is the one closest to physical storage—that is, it is the one concerned with the way the data is stored inside the system.
- The external level (also known as the *user logical* level) is the one closest to the users—that is, it is the one concerned with the way the data is seen by individual users.
- The conceptual level (also known as the *community logical* level, or sometimes just the *logical* level, unqualified) is a level of indirection between the other two.

Observe that the external level is concerned with *individual* user perceptions, while the conceptual level is concerned with a *community* user perception. As we saw in Chapter 1, most users will not be interested in the total database, but only in some restricted

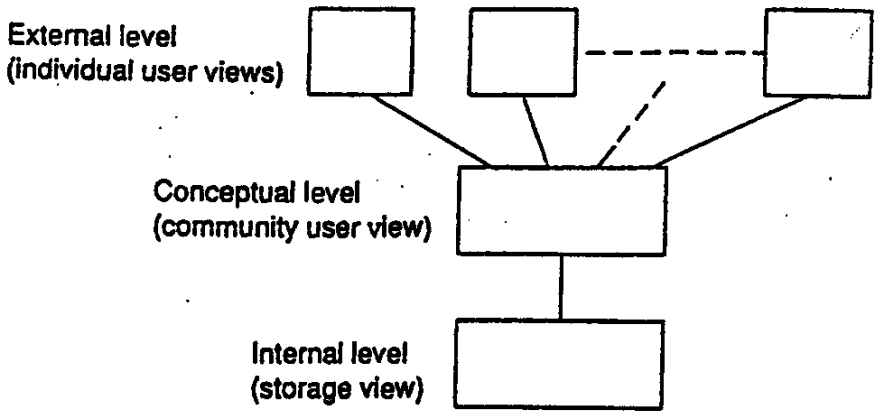


Fig. 2.1 The three levels of the architecture

portion of it; thus, there will be many distinct "external views," each consisting of a more or less abstract representation of some portion of the total database, and there will be precisely one "conceptual view," consisting of a similarly abstract representation of the database in its entirety. And then there will be precisely one "internal view," representing the database as stored internally. Note that (to use the terminology of Chapter 1) the external and conceptual levels are both *model* levels, while the internal level is an *implementation* level; in other words, the external and conceptual levels are defined in terms of user-oriented constructs such as records and fields, while the internal level is defined in terms of machine-oriented constructs such as bits and bytes.

An example will help to make these ideas clearer. Fig. 2.2 shows the conceptual view, the corresponding internal view, and two corresponding external views (one for a PL/I user and one for a COBOL user[1]), all for a simple personnel database. Of course, the example is completely hypothetical—it is not intended to resemble any real system—and many irrelevant details have deliberately been omitted. *Explanation:*

1. At the conceptual level, the database contains information concerning an entity type called EMPLOYEE. Each individual employee has an EMPLOYEE_NUMBER (six characters), a DEPARTMENT_NUMBER (four characters), and a SALARY (five decimal digits).
2. At the internal level, employees are represented by a stored record type called STORED_EMP, 20 bytes long. STORED_EMP contains four stored fields: a 6-byte prefix (presumably containing control information such as codes, flags, or pointers), and three data fields corresponding to the three properties of employees. In addition, STORED_EMP records are indexed on the EMP# field by an index called EMPX, whose definition is not shown.

External (PL/I)

```
DCL 1 EMPP,
    2 EMP# CHAR(6),
    2 SAL FIXED BIN(31);
```

External (COBOL)

```
   01 EMPC.
      02 EMPNO PIC X(6).
      02 DEPTNO PIC X(4).
```

Conceptual

```
EMPLOYEE
    EMPLOYEE_NUMBER     CHARACTER(6)
    DEPARTMENT_NUMBER   CHARACTER(4)
    SALARY              DECIMAL(5)
```

Internal

```
STORED_EMP    BYTES=20
   PREFIX     BYTES=6,OFFSET=0
   EMP#       BYTES=6,OFFSET=6,INDEX=EMPX
   DEPT#      BYTES=4,OFFSET=12
   PAY        BYTES=4,ALIGN=FULLWORD,OFFSET=16
```

Fig. 2.2 An example of the three levels

[1] We apologize for using such ancient languages as the basis for this example, but the fact is that PL/I and COBOL are both still widely used in commercial installations.

3. The PL/I user has an external view of the database in which each employee is represented by a PL/I record containing two fields (department numbers are of no interest to this user and have therefore been omitted). The record type is defined by an ordinary PL/I structure declaration in accordance with normal PL/I rules.
4. Similarly, the COBOL user has an external view in which each employee is represented by a COBOL record containing, again, two fields (this time, salaries have been omitted). The record type is defined by an ordinary COBOL record description in accordance with normal COBOL rules.

Notice that corresponding data items can have different names at different points in the foregoing scheme. For example, the employee number is called EMP# in the PL/I external view, EMPNO in the COBOL external view, EMPLOYEE_NUMBER in the conceptual view, and EMP# again in the internal view. Of course, the system must be aware of the correspondences; for example, it must be told that the COBOL field EMPNO is derived from the conceptual field EMPLOYEE_NUMBER, which in turn is derived from the stored field EMP# at the internal level. Such correspondences, or mappings, are not explicitly shown in Fig. 2.2; see Section 2.6 for further discussion.

Now, it makes little difference for the purposes of the present chapter whether the system under consideration is relational or otherwise. However, it might be helpful to indicate briefly how the three levels of the architecture are typically realized in a relational system specifically:

- First, the conceptual level in such a system will definitely be relational, in the sense that the objects visible at that level will be relational tables and the operators will be relational operators (including in particular the *restrict* and *project* operators discussed briefly in Chapter 1).
- Second, a given external view will typically be relational too, or something very close to it; for example, the PL/I and COBOL record declarations of Fig. 2.2 might loosely be regarded as PL/I and COBOL analogs of the declaration of a relational table in a relational system. *Note:* We should mention in passing that the term *external view* (usually abbreviated to just *view*) unfortunately has a rather specific meaning in relational contexts that is *not* identical to the meaning assigned to it in this chapter. See Chapters 3 and (especially) 10 for an explanation and discussion of the relational meaning.
- Third, the internal level will *not* be relational, because the objects at that level will not be just (stored) relational tables—instead, they will be the same kinds of objects found at the internal level of any other kind of system (stored records, pointers, indexes, hashes, etc.). In fact, the relational model as such has **nothing whatsoever** to say about the internal level; it is, to repeat from Chapter 1, concerned with how the database looks to the *user.*

We now proceed to discuss the three levels of the architecture in considerably more detail, starting with the external level. Throughout our discussions we will be making repeated references to Fig. 2.3, which shows the major components of the architecture and their interrelationships.

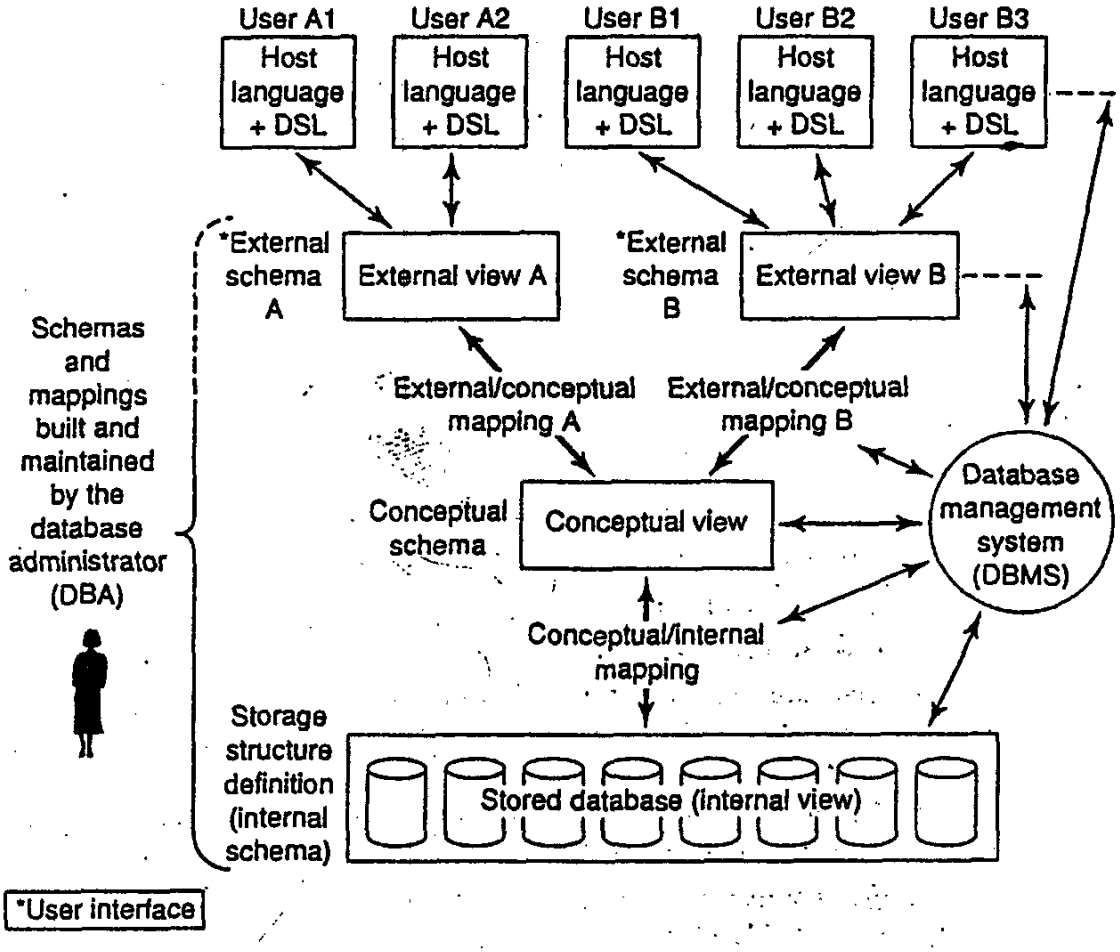


Fig. 2.3 Detailed system architecture

2.3 THE EXTERNAL LEVEL

The external level is the individual user level. As explained in Chapter 1, a given user can be an application programmer or an end user of any degree of sophistication. (The DBA is an important special case; unlike other users, however, the DBA will need to be interested in the conceptual and internal levels also. See the next two sections.)

Each user has a language at his or her disposal:

- For the application programmer, that language will be either a conventional programming language (e.g., Java, C++, or PL/I) or perhaps a proprietary language that is specific to the system in question. Such proprietary languages are sometimes called *fourth-generation* languages or 4GLs, on the (fuzzy!) grounds that (a) machine code, assembler language, and languages such as Java or C++ or PL/I can be regarded as three earlier language "generations," and (b) the proprietary languages represent the same kind of improvement over third-generation languages (3GLs) as those languages did over assembler language and assembler language did over machine code.

- For the end user, the language will be either a query language (probably SQL) or some special-purpose language, perhaps forms- or menu-driven, tailored to that user's requirements and supported by some online application as explained in Chapter 1.

For our purposes, the important thing about all such languages is that they will include a **data sublanguage**—that is, a subset of the total language that is concerned specifically with database objects and operations. The data sublanguage (abbreviated DSL in Fig. 2.3) is said to be **embedded** within the corresponding **host language**. The host language is responsible for providing various nondatabase facilities, such as local variables, computational operations, branching logic, and so on. A given system might support any number of host languages and any number of data sublanguages; however, one particular data sublanguage that is supported by almost all current systems is the language SQL, discussed briefly in Chapter 1. Most such systems allow SQL to be used both *interactively* as a stand-alone query language and also *embedded* in other languages such as Java or C++ or PL/I (see Chapter 4 for further discussion).

Now, although it is convenient for architectural purposes to distinguish between the data sublanguage and its containing host language, the two might in fact *not* be distinct as far as the user is concerned; indeed, it is probably preferable from the user's perspective if they are not. If they are not distinct, or if they can be distinguished only with difficulty, we say they are **tightly coupled** (and the combination is called a *database programming language*[2]). If they are clearly and easily separable, we say they are **loosely coupled**. Some commercial systems—including in particular certain SQL products, such as Oracle—support tight coupling, but not all do (tight coupling provides a more uniform set of facilities for the user but obviously involves more effort on the part of the system implementers, a fact that presumably accounts for the *status quo*).

In principle, any given data sublanguage is really a combination of at least two subordinate languages—a **data definition language** (DDL), which supports the *definition* or "declaration" of database objects, and a **data manipulation language** (DML), which supports the *processing* or "manipulation" of such objects.[3] For example, consider the PL/I user of Fig. 2.2 in Section 2.2. The data sublanguage for that user consists of those PL/I features that are used to communicate with the DBMS:

- The DDL portion consists of those declarative constructs of PL/I that are needed to declare database objects—the DECLARE (DCL) statement itself, certain PL/I data types, and possibly special extensions to PL/I to deal with new kinds of objects not supported by existing PL/I.
- The DML portion consists of those executable statements of PL/I that transfer information into and out of the database—again, possibly including special new statements.

[2] The language Tutorial D that we will be using in later chapters as a basis for examples—see the remarks on this topic in the preface to this book—is a database programming language in this sense.

[3] This rather inappropriate use of the term *manipulation* has become sanctioned by usage.

(In the interest of accuracy, we should make it clear that PL/I does not in fact include any specific database features at the time of writing. The "DML" statements in particular are typically just PL/I CALL statements that invoke the DBMS—though those calls might be syntactically disguised in some way to make them a little more user-friendly; see the discussion of *embedded SQL* in Chapter 4.)

To return to the architecture: We have already indicated that an individual user will generally be interested only in some portion of the total database; moreover, that user's view of that portion will generally be somewhat abstract when compared with the way the data is physically stored. The ANSI/SPARC term for an individual user's view is an external view. An external view is thus the content of the database as seen by some particular user: to that user, in other words, the external view *is* the database. For example, a user from the Personnel Department might regard the database as a collection of department and employee record occurrences, and might be quite unaware of the supplier and part record occurrences seen by users in the Purchasing Department.

In general, then, an external view consists of many occurrences of many types of external record (*not* necessarily the same thing as a stored record).[4] The user's data sublanguage is thus defined in terms of external records; for example, a DML *retrieve* operation will retrieve external record occurrences, not stored record occurrences. (Incidentally, we can now see that the term *logical record* used a couple of times in Chapter 1 actually referred to an external record. From this point forward, in fact, we will generally avoid the term *logical record*.)

Each external view is defined by means of an external schema, which consists basically of definitions of each of the various external record types in that external view (again, refer back to Fig. 2.2 for a couple of simple examples). The external schema is written using the DDL portion of the user's data sublanguage. (That DDL is therefore sometimes referred to as an external DDL.) For example, the employee external record type might be defined as a six-character employee number field plus a five-digit (decimal) salary field, and so on. In addition, there must be a definition of the *mapping* between the external schema and the underlying *conceptual* schema (see the next section). We will consider that mapping later, in Section 2.6.

2.4 THE CONCEPTUAL LEVEL

The conceptual view is a representation of the entire information content of the database, again (as with an external view) in a form that is somewhat abstract in comparison with the way in which the data is physically stored. It will also be quite different, in general, from the way in which the data is viewed by any particular user. Broadly speaking, the

[4] We are assuming here that all information is represented at the external level in the form of records specifically. However, some systems allow information to be represented in other ways instead of or as well as records. For a system using such alternative methods, the definitions and explanations given in this section will require suitable modification. Analogous remarks apply to the conceptual and internal levels also. Detailed consideration of such matters is beyond the scope of this early part of the book; see Chapters 14 (especially the "References and Bibliography" section) and 25 for further discussion. See also—in connection with the internal level in particular—Appendix A.

conceptual view is intended to be a view of the data "as it really is," rather than as users are forced to see it by the limitations of (for example) the particular language or hardware they might be using.

The conceptual view consists of many occurrences of many types of conceptual record. For example, it might consist of a collection of department record occurrences, plus a collection of employee record occurrences, plus a collection of supplier record occurrences, plus a collection of part record occurrences, and so on. A conceptual record is not necessarily the same as either an external record, on the one hand, or a stored record, on the other.

The conceptual view is defined by means of the conceptual schema, which includes definitions of each of the various conceptual record types (again, refer to Fig. 2.2 for a simple example). The conceptual schema is written using another data definition language, the conceptual DDL. If physical data independence is to be achieved, then those conceptual DDL definitions must not involve any considerations of physical representation or access technique at all—they must be definitions of information content *only*. Thus, there must be no reference in the conceptual schema to stored field representation, stored record sequence, indexes, hashing schemes, pointers, or any other storage and access details. If the conceptual schema is made truly data-independent in this way, then the external schemas, which are defined in terms of the conceptual schema (see Section 2.6), will *a fortiori* be data-independent too.

The conceptual view, then, is a view of the total database content, and the conceptual schema is a definition of that view. However, it would be misleading to suggest that the conceptual schema is nothing more than a set of definitions much like the simple record definitions found in (e.g.) a COBOL program today. The definitions in the conceptual schema are intended to include a great many additional features, such as the security and integrity constraints mentioned in Chapter 1. Some authorities would go as far as to suggest that the ultimate objective of the conceptual schema is to describe the complete enterprise—not just its data *per se*, but also how that data is used: how it flows from point to point within the enterprise, what it is used for at each point, what audit or other controls are to be applied at each point, and so on [2.3]. It must be emphasized, however, that no system today actually supports a conceptual schema of anything approaching this degree of comprehensiveness;[5] in most existing systems, the "conceptual schema" is little more than a simple union of all of the individual external schemas, plus certain security and integrity constraints. But it is certainly possible that systems of the future will be much more sophisticated in their support of the conceptual level.

2.5 THE INTERNAL LEVEL

The third level of the architecture is the internal level. The internal view is a low-level representation of the entire database; it consists of many occurrences of many types of internal record. *Internal record* is the ANSI/SPARC term for the construct that we have

[5] Some might argue that the so-called *business rule* systems come close (see Chapters 9 and 14).

been calling a *stored* record (and we will continue to use this latter term). The internal view is thus still at one remove from the physical level, since it does not deal in terms of *physical* records—also called **blocks** or **pages**—nor with any device-specific considerations such as cylinder or track sizes. In other words, the internal view effectively assumes an unbounded linear address space; details of how that address space is mapped to physical storage are highly system-specific and are deliberately omitted from the general architecture. *Note:* In case you are not familiar with the term, we should explain that the block, or page, is the **unit of I/O**—that is, it is the amount of data transferred between secondary storage and main memory in a single I/O operation. Typical page sizes can be anywhere from 1KB or less to 64KB or so, where (as we will see later) 1KB = one kilobyte = 1024 bytes.

The internal view is described by means of the **internal schema**, which not only defines the various stored record types but also specifies what indexes exist, how stored fields are represented, what physical sequence the stored records are in, and so on (once again, see Fig. 2.2 for a simple example; see also Appendix D, online). The internal schema is written using yet another data definition language—the **internal DDL**.

Note: In what follows, we will tend to use the more intuitive terms *stored database* and *stored database definition* in place of *internal view* and *internal schema*, respectively. Also, we observe that, in certain exceptional situations, application programs—in particular, those of a utility nature (see Section 2.11)—might be permitted to operate directly at the internal level rather than at the external level. Needless to say, the practice is not recommended; it represents a security risk (since the security constraints are bypassed) and an integrity risk (since the integrity constraints are bypassed likewise), and the program will be data-dependent to boot; but sometimes it might be the only way to obtain the required functionality or performance—just as an application programmer might occasionally have to descend to assembler language in order to satisfy certain functionality or performance objectives in a programming language system.

2.6 MAPPINGS

In addition to the three levels *per se*, the architecture of Fig. 2.3 involves certain mappings—one conceptual/internal mapping and several external/conceptual mappings, in general:

- The *conceptual/internal* mapping defines the correspondence between the conceptual view and the stored database; it specifies how conceptual records and fields are represented at the internal level. If the structure of the stored database is changed—that is, if a change is made to the stored database definition—then the conceptual/internal mapping must be changed accordingly, so that the conceptual schema can remain invariant. (It is the responsibility of the DBA, or possibly the DBMS, to manage such changes.) In other words, the effects of such changes must be isolated below the conceptual level, in order to preserve physical data independence.
- An *external/conceptual* mapping defines the correspondence between a particular external view and the conceptual view. In general, the differences that can exist

dump/restore purposes. In this connection, note that *multi-terabyte systems*[6]—that is, commercial database installations that store several trillions of bytes of data, loosely speaking—already exist, and systems of the future are predicted to be much larger. It goes without saying that such *VLDB* ("very large database") systems require very careful and sophisticated administration, especially if there is a requirement for continuous availability (which there usually is). Nevertheless, we will continue to talk (for the sake of simplicity) as if there were in fact just a single database.

- *Monitoring performance and responding to changing requirements*

  As indicated in Chapter 1, the DBA is responsible for organizing the system in such a way as to get the performance that is "best for the enterprise," and for making the appropriate adjustments—that is, **tuning**—as requirements change. For example, it might be necessary to **reorganize** the stored database from time to time to ensure that performance levels remain acceptable. As already mentioned, any change to the internal level of the system must be accompanied by a corresponding change to the definition of the conceptual/internal mapping, so that the conceptual schema can remain constant.

Of course, the foregoing is not an exhaustive list—it is merely intended to give some idea of the extent and nature of the DBA's responsibilities.

2.8 THE DATABASE MANAGEMENT SYSTEM

The **database management system** (DBMS) is the software that handles all access to the database. Conceptually, what happens is the following (refer to Fig. 2.3 once again):

1. A user issues an access request, using some particular data sublanguage (typically SQL).
2. The DBMS accepts that request and analyzes it.
3. The DBMS inspects, in turn, (the object versions of) the external schema for that user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping, and the stored database definition.
4. The DBMS executes the necessary operations on the stored database.

By way of an example, consider what is involved in the retrieval of a particular external record occurrence. In general, fields will be required from several conceptual record occurrences, and each conceptual record occurrence in turn will require fields from several stored record occurrences. Conceptually, then, the DBMS must first retrieve all required stored record occurrences, then construct the required conceptual record occur-

[6] 1024 bytes = 1 kilobyte (KB); 1024KB = 1 megabyte (MB); 1024MB = 1 gigabyte (GB); 1024GB = 1 terabyte (TB); 1024TB = 1 petabyte (PB); 1024PB = 1 exabyte (EB or XB); 1024XB = 1 zettabyte (ZB); 1024ZB = 1 yottabyte (YB). Note in particular that a gigabyte is a billion bytes, loosely speaking (the abbreviation BB is sometimes used instead of GB). Incidentally (and contrary to popular belief), *gigabyte* is pronounced with a soft initial *g* and the *i* is long (as in *gigantic*).

rences, and then construct the required external record occurrence. At each stage, data type or other conversions might be necessary.

Of course, the foregoing description is very much simplified: in particular, it suggests that the entire process is interpretive, inasmuch as it describes the processes of analyzing the request, inspecting the various schemas, and so on, as if they were all done at run time. Interpretation, in turn, often implies poor performance, because of the run-time overhead. In practice, however, it might be possible for access requests to be *compiled* prior to run time (in particular, several of today's SQL products do this—see, for example, the annotation to references [4.13] and [4.27] in Chapter 4).

Let us now examine the functions of the DBMS in a little more detail. Those functions will include support for at least all of the following (refer to Fig. 2.4, overleaf):

- *Data definition*

  The DBMS must be able to accept data definitions (external schemas, the conceptual schema, the internal schema, and all associated mappings) in source form and convert them to the appropriate object form. In other words, the DBMS must include DDL processor or DDL compiler components for each of the various data definition languages (DDLs). The DBMS must also "understand" the DDL definitions, in the sense that, for example, it "understands" that EMPLOYEE external records include a SALARY field, and it must be able to use this knowledge in analyzing and responding to data manipulation requests (e.g., "Get employees with salary < $50,000").

- *Data manipulation*

  The DBMS must be able to handle requests to retrieve, update, or delete existing data in the database or to add new data to the database. In other words, the DBMS must include a DML processor or DML compiler component to deal with the data manipulation language (DML).

  In general, DML requests can be planned or unplanned:

  a. A planned request is one for which the need was foreseen in advance of the time at which the request is made. The DBA will probably have tuned the physical database design in such a way as to guarantee good performance for planned requests.

  b. An unplanned request, by contrast, is an *ad hoc* query or (less likely) update—that is, a request for which the need was not seen in advance, but instead arose in a spur-of-the-moment fashion. The physical database design might or might not be well suited for the specific request under consideration.

  To use the terminology introduced in Chapter 1 (Section 1.3), planned requests are characteristic of *operational* or *production* applications, while unplanned requests are characteristic of *decision support* applications. Furthermore, planned requests will typically be issued from prewritten application programs, whereas unplanned requests, by definition, will be issued interactively, typically via some *query language processor.* (In fact, as we saw in Chapter 1, the query language processor is really a built-in online application, not part of the DBMS *per se;* we include it in Fig. 2.4 only for completeness.)

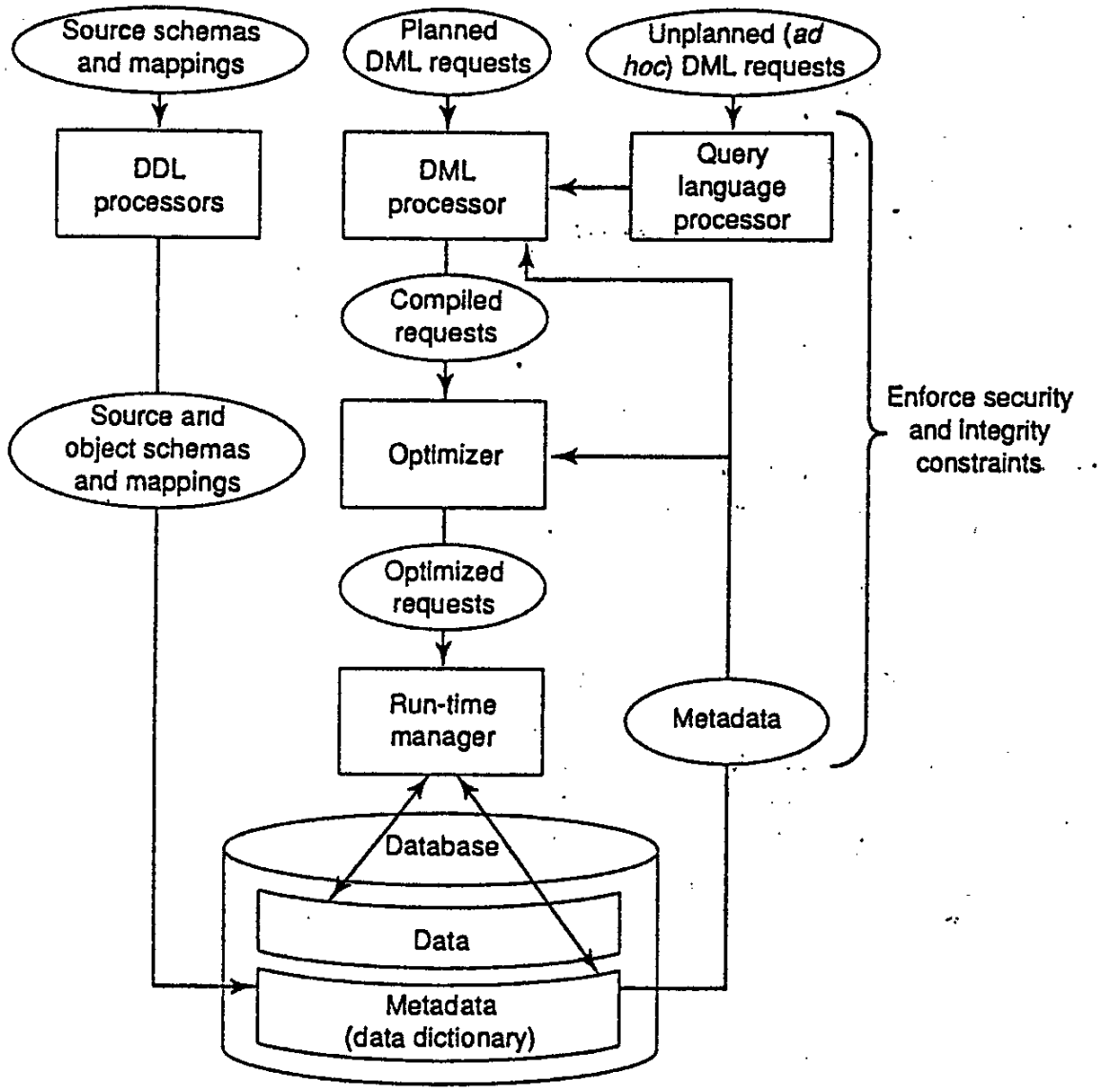


Fig. 2.4 Major DBMS functions and components

- *Optimization and execution*

  DML requests, planned or unplanned, must be processed by the optimizer component, whose purpose is to determine an efficient way of implementing the request.[7] Optimization is discussed in detail in Chapter 18. The optimized requests are then executed under the control of the run-time manager. (In practice, the run-time manager will probably invoke some kind of *file* or *storage manager* to access the stored data. File managers are discussed briefly at the end of the present section.)

[7] Throughout this book we take the term *optimization* to refer to the optimization of DML requests specifically, barring explicit statements to the contrary.

- *Data security and integrity*

  The DBMS, or some subsystem invoked by the DBMS, must monitor user requests and reject any attempt to violate the security and integrity constraints defined by the DBA (see the previous section). These tasks can be carried out at compile time or run time or some mixture of the two.

- *Data recovery and concurrency*

  The DBMS—or, more likely, another related software component called the **transaction manager** or **TP monitor**—must enforce certain recovery and concurrency controls. Details of these aspects of the system are beyond the scope of this chapter; see Part IV of this book for an in-depth discussion. The transaction manager is not shown in Fig. 2.4 because it is usually not part of the DBMS *per se*.

- *Data dictionary*

  The DBMS must provide a **data dictionary** function. The data dictionary can be regarded as a database in its own right (but a system database rather than a user database); it contains "data about the data" (sometimes called **metadata** or **descriptors**)—that is, *definitions* of other objects in the system, instead of just "raw data." In particular, all of the various schemas and mappings (external, conceptual, etc.) and all of the various security and integrity constraints will be kept, in both source and object form, in the dictionary. A comprehensive dictionary will also include much additional information, showing, for instance, which programs use which parts of the database, which users require which reports, and so on. The dictionary might even—in fact, probably should—be integrated into the database it defines and thus include its own definition. Certainly it should be possible to query the dictionary just like any other database, so that, for example, it is possible to tell which programs and/or users are likely to be affected by some proposed change to the system. See Chapter 3 for further discussion.

  Note: We are touching here on an area in which there is much terminological confusion. Some people would refer to what we are calling the dictionary as a *directory* or a *catalog*—with the tacit implication that directories and catalogs are somehow inferior to a genuine dictionary—and would reserve the term *dictionary* to refer to a specific (important) kind of application development tool. Other terms that are also sometimes used to refer to this latter kind of object are *data repository* (see Chapter 14) and *data encyclopedia*.

- *Performance*

  It goes without saying that the DBMS should perform all of its tasks as efficiently as possible.

We can summarize all of the foregoing by saying that the overall purpose of the DBMS is to provide the **user interface** to the database system. The user interface can be defined as a boundary in the system below which everything is invisible to the user. By definition, therefore, the user interface is at the *external* level. In today's SQL products, however, there are some situations—mostly having to do with update operations—in which the external level is unlikely to differ very significantly from the relevant portion of the underlying conceptual level. We will elaborate on this issue in Chapter 10.

We conclude this section by briefly contrasting database management systems (DBMSs) with *file* management systems (*file managers* or *file servers* for short). Basically, the file manager is that component of the underlying operating system that manages stored files; loosely speaking, therefore, it is "closer to the disk" than the DBMS is. (In fact, Appendix D, online, explains how the DBMS is often built *on top of* some kind of file manager.) Thus, the user of a file management system will be able to create and destroy stored files and perform simple retrieval and update operations on stored records in such files. In contrast to the DBMS, however:

- File managers are not aware of the internal structure of stored records and hence cannot handle requests that rely on a knowledge of that structure.
- File managers typically provide little or no support for security and integrity constraints.
- File managers typically provide little or no support for recovery and concurrency controls.
- There is no genuine data dictionary concept at the file manager level.
- File managers provide much less data independence than the DBMS does.
- Files are typically not "integrated" or "shared" in the same sense that the database is, but instead are usually private to some particular user or application.

2.9 DATA COMMUNICATIONS

In this section, we briefly consider the topic of **data communications**. Database requests from an end user are actually transmitted from the user's computer or workstation—which might be physically remote from the database system itself—to some online application, built-in or otherwise, and thence to the DBMS, in the form of *communication messages*. Likewise, responses back from the DBMS and online application to the user's workstation are also transmitted in the form of such messages. All such message transmissions take place under the control of another software component, the **data communications manager** (DC manager).

The DC manager is not part of the DBMS but is an autonomous system in its own right. However, since it is clearly required to work in harmony with the DBMS, the two are sometimes regarded as equal partners in a higher-level cooperative venture called a **database/data-communications system** (DB/DC system), in which the DBMS looks after the database and the DC manager handles all messages to and from the DBMS, or more accurately to and from applications that use the DBMS. In this book, however, we will have comparatively little to say about message handling as such (it is a large subject in its own right). Section 2.12 does briefly discuss the question of communication *between distinct systems* (e.g., between distinct machines in a communications network such as the Internet), but that is really a separate topic.

2.10 CLIENT/SERVER ARCHITECTURE

So far in this chapter we have been discussing database systems from the point of view of the so-called ANSI/SPARC architecture. In particular, we gave a simplified picture of that architecture in Fig. 2.3. In this section we offer a slightly different perspective on the subject.

The overall purpose of a database system is to support the development and execution of database applications. From a high-level point of view, therefore, such a system can be regarded as having a very simple two-part structure, consisting of a *server*, also called the *back end*, and a set of *clients*, also called the *front ends* (refer to Fig. 2.5). *Explanation:*

1. The server is just the DBMS itself. It supports all of the basic DBMS functions discussed in Section 2.8—data definition, data manipulation, data security and integrity, and so on. In other words, "server" in this context is just another name for the DBMS.
2. The clients are the various applications that run on top of the DBMS—both user-written applications and built-in applications (i.e., applications provided by the DBMS vendor or some third party). As far as the server is concerned, of course, there is no difference between user-written and built-in applications: they all use the same interface to the server—namely, the external-level interface discussed in Section 2.3. (We note as an aside that, as mentioned in Section 2.5, certain special "utility" applications might constitute an exception to the foregoing, inasmuch as they might sometimes need to operate directly at the *internal* level of the system. Such utilities are

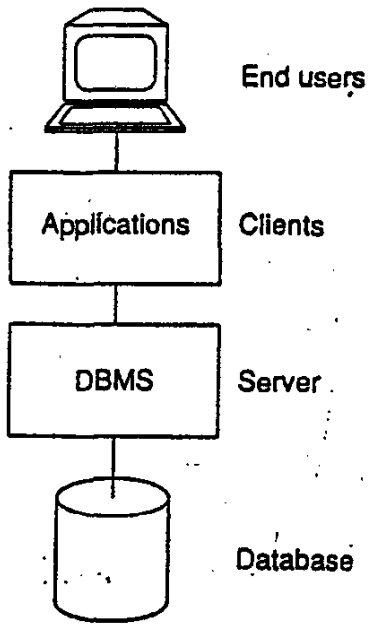


Fig. 2.5 Client/server architecture

best regarded as integral components of the DBMS, rather than as applications in the usual sense. They are discussed in more detail in the next section.)

We elaborate briefly on the question of user-written *vs.* vendor-provided applications:

- User-written applications are basically regular application programs, written (typically) either in a conventional 3GL such as C++ or COBOL or in some proprietary 4GL—though in both cases the language needs to be coupled somehow with an appropriate data sublanguage, as explained in Section 2.3.
- Vendor-provided applications (often called tools) are applications whose basic purpose is to assist in the creation and execution of other applications! The applications that are created are applications that are tailored to some specific task (they might not look much like applications in the conventional sense; indeed, the whole point of the tools is to allow users, especially end users, to create applications *without* having to write programs in a conventional programming language). For example, one of the vendor-provided tools will be a *report writer*, whose purpose is to allow end users to obtain formatted reports from the system on request. Any given report request can be regarded as a small application program, written in a very high-level (and special-purpose) *report writer language*.

  Vendor-provided tools can be divided into several more or less distinct classes:

  a. Query language processors
  b. Report writers
  c. Business graphics subsystems
  d. Spreadsheets
  e. Natural language processors
  f. Statistical packages
  g. Copy management or "data extract" tools
  h. Application generators (including 4GL processors)
  i. Other application development tools, including computer-aided software engineering (CASE) products
  j. Data mining and visualization tools

  and many others. Details of most such tools are beyond the scope of this book: however, we remark that since (as stated near the opening of this section) the whole point of a database system is to support the creation and execution of applications, the quality of the available tools is, or should be, a major factor in "the database decision" (i.e., the process of choosing the right database product). In other words, the DBMS *per se* is not the only factor that needs to be taken into account, nor even necessarily the most significant factor.

We close this section with a forward reference. Since the overall system can be so neatly divided into two parts, server and clients, the possibility arises of running the two on different machines. In other words, the potential exists for **distributed processing**.

Distributed processing means that distinct machines can be connected into some kind of communications network in such a way that a single data processing task can be spread across several machines in the network. In fact, so attractive is this possibility—for a variety of reasons, mainly economic—that the term *client/server* has come to apply almost exclusively to the case where client and server are indeed on different machines. We will discuss distributed processing in more detail in Section 2.12.

2.11 UTILITIES

Utilities are programs designed to help the DBA with various administration tasks. As mentioned in the previous section, some utility programs operate at the external level of the system, and thus are effectively nothing more than special-purpose applications; some might not even be provided by the DBMS vendor, but rather by some third party. Other utilities, however, operate directly at the internal level (in other words, they are really part of the server), and hence must be provided by the DBMS vendor.

Here are some examples of the kind of utilities that are typically needed in practice:

- Load routines, to create the initial version of the database from regular data files
- Unload/reload (or dump/restore) routines, to unload the database or portions thereof to backup storage and to reload data from such backup copies (of course, the "reload utility" is basically identical to the load utility just mentioned)
- Reorganization routines, to rearrange the data in the stored database for various reasons (usually having to do with performance)—for example, to cluster data in some particular way on the disk, or to reclaim space occupied by logically deleted data
- Statistical routines, to compute various performance statistics such as file sizes, value distributions, I/O counts, and so on
- Analysis routines, to analyze the statistics just mentioned

Of course, this list represents just a small sample of the range of functions that utilities typically provide; numerous other possibilities exist.

2.12 DISTRIBUTED PROCESSING

To repeat from Section 2.10, the term *distributed processing* means that distinct machines can be connected into a communications network—the Internet provides the obvious example—such that a single data processing task can span several machines in the network. (The term *parallel processing* is also used with essentially the same meaning, except that the distinct machines tend to be physically close together in a "parallel" system and need not be so in a "distributed" system; that is, they might be geographically dispersed in the latter case.) Communication among the various machines is handled by some kind of network management software, possibly an extension of the DC manager (discussed in Section 2.9), more likely a separate software component.

Many levels or varieties of distributed processing are possible. To repeat from Section 2.10, one simple case involves running the DBMS back end (the server) on one machine and the application front ends (the clients) on another. Refer to Fig. 2.6.

As mentioned at the end of Section 2.10, *client/server*, though strictly speaking a purely architectural term, has come to be almost synonymous with the arrangement illustrated in Fig. 2.6, in which client and server run on different machines. Indeed, there are many arguments in favor of such a scheme:

- The first is basically just the usual parallel processing argument: namely, two or more machines are now being applied to the overall task. and server (database) and client (application) processing are being done in parallel. Response time and throughput should thus be improved.
- Furthermore, the server machine might be a custom-built machine that is tailored to the DBMS function (a "database machine") and might thus provide better DBMS performance.
- Likewise, the client machine might be a personal workstation, tailored to the needs of the end user and thus able to provide better interfaces, high availability, faster responses, and overall improved ease of use to the user.
- Several different client machines might be able—in fact, typically will be able—to access the same server machine. Thus, a single database might be shared across several distinct clients (see Fig. 2.7).

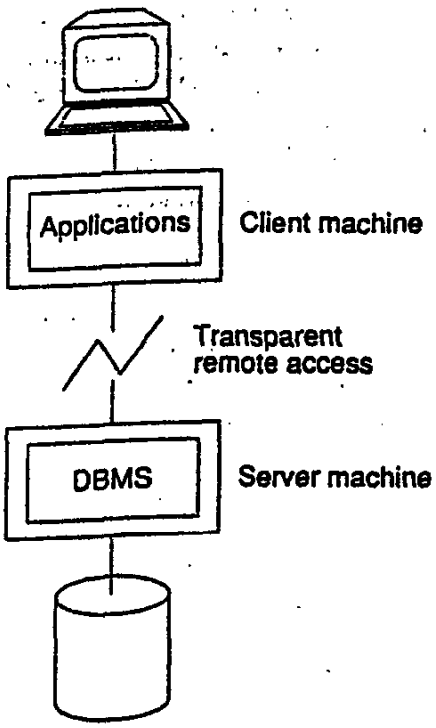


Fig. 2.6 Client(s) and server running on different machines

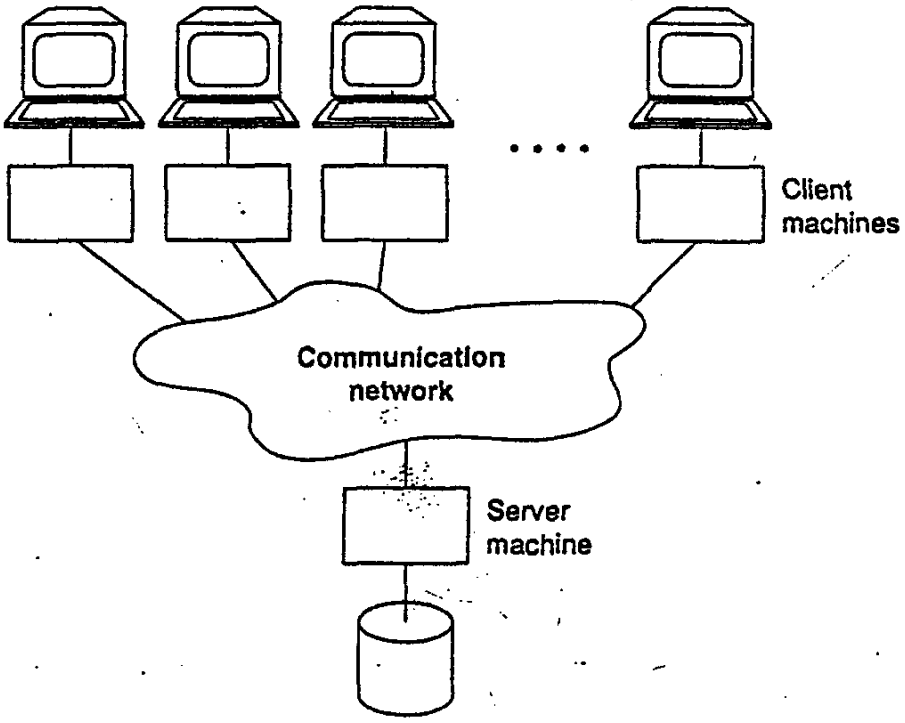


Fig. 2.7 One server machine, many client machines

In addition to the foregoing arguments, there is also the point that running the client(s) and the server on separate machines matches the way enterprises actually operate. It is quite common for a single enterprise—a bank, for example—to operate many computers, such that the data for one portion of the enterprise is stored on one computer and the data for another portion is stored on another. It is also quite common for users on one computer to need at least occasional access to data stored on another. To pursue the banking example for a moment, it is very likely that users at one branch office will occasionally need access to data stored at another. Note, therefore, that the client machines might have stored data of their own, and the server machine might have applications of its own. In general, therefore, each machine will act as a server for some users and a client for others (see Fig. 2.8); in other words, each machine will support *an entire database system*, in the sense of earlier sections of this chapter.

The final point is that a single client machine might be able to access several different server machines (the converse of the case illustrated in Fig. 2.7). This capability is desirable because, as already mentioned, enterprises do typically operate in such a manner that the totality of their data is not stored on one single machine but rather is spread across many distinct machines, and applications will sometimes need the ability to access data from more than one machine. Such access can basically be provided in two different ways:

- A given client might be able to access any number of servers, but only one at a time (i.e., each individual database request must be directed to just one server). In such a system it is not possible, within a single request, to combine data from two or more

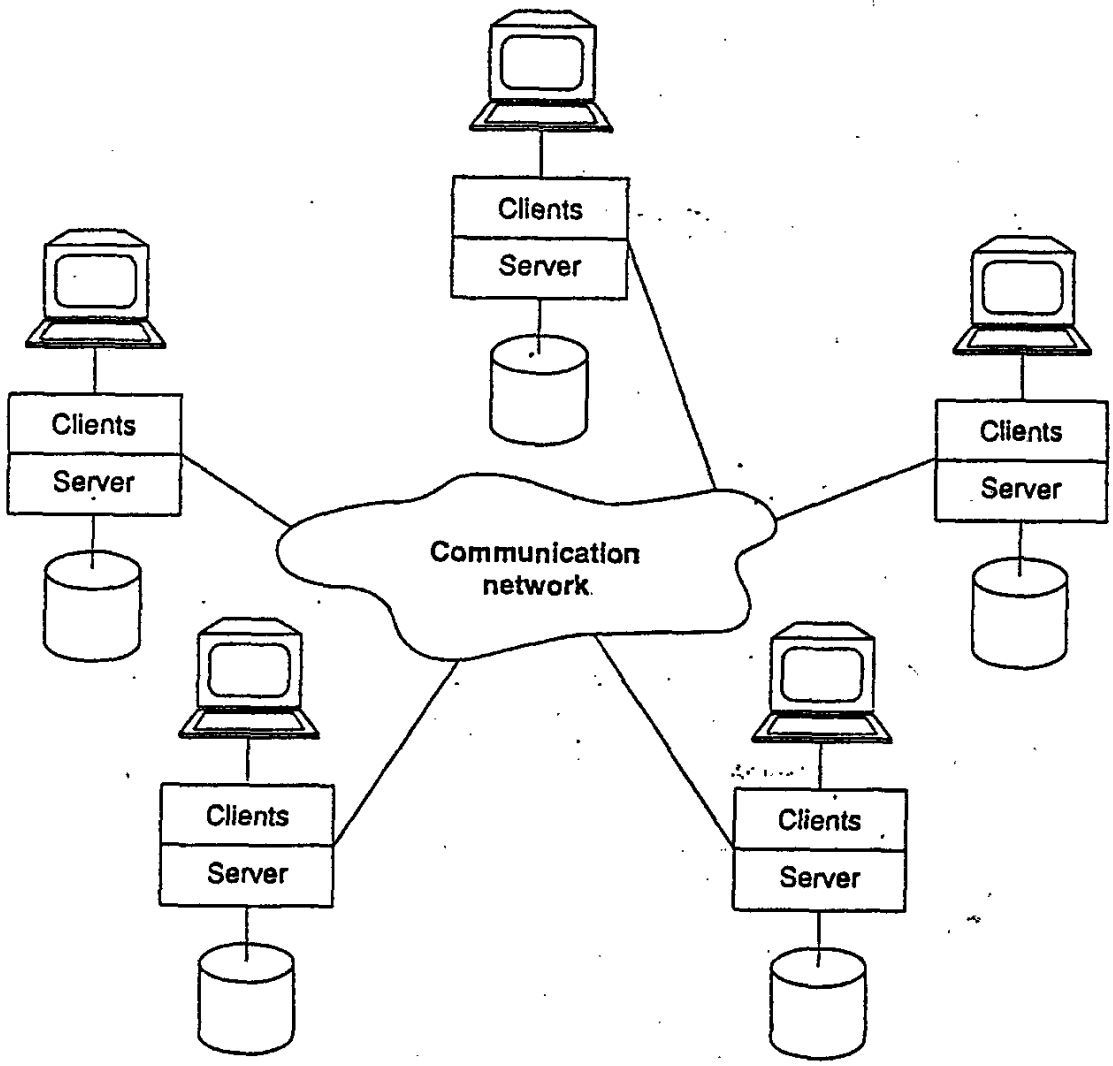


Fig. 2.8 Each machine runs both client(s) and server

different servers. Furthermore, the user in such a system has to know which particular machine holds which pieces of data.

- The client might be able to access many servers simultaneously (i.e., a single database request might be able to combine data from several servers). In this case, the servers look to the client from a logical point of view as if they were really a single server, and the user does not have to know which machines hold which pieces of data.

This latter case constitutes what is usually called a **distributed database system**. Distributed database is a big topic in its own right; carried to its logical conclusion, full distributed database support implies that a single application should be able to operate "transparently" on data that is spread across a variety of different databases, managed by a

variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected by a variety of different communication networks—where "transparently" means the application operates from a logical point of view as if the data were all managed by a single DBMS running on a single machine. Such a capability might sound like a pretty tall order, but it is highly desirable from a practical perspective, and much effort has been devoted to making such systems a reality. We will discuss such systems in detail in Chapter 21.

2.13 SUMMARY

In this chapter we have looked at database systems from an architectural point of view. First, we described the **ANSI/SPARC architecture**, which divides a database system into three levels, as follows: The **internal** level is the one closest to physical storage (i.e., it is the one concerned with the way the data is stored); the **external** level is the one closest to the users (i.e., it is the one concerned with the way the data is viewed by individual users); and the **conceptual** level is a level of indirection between the other two (it provides a *community view* of the data). The data as perceived at each level is described by a schema (or several schemas, in the case of the external level). Mappings define the correspondence between (a) a given external schema and the conceptual schema, and (b) the conceptual schema and the internal schema. Those mappings are the key to the provision of logical and physical **data independence**, respectively.

Users—that is, end users and application programmers, both of whom operate at the external level—interact with the data by means of a **data sublanguage**, which consists of at least two components, a **data definition language** (DDL) and a **data manipulation language** (DML). The data sublanguage is embedded in a **host language**. Please note, however, that the boundaries (a) between the host language and the data sublanguage and (b) between the DDL and the DML are primarily conceptual in nature; ideally they should be "transparent to the user."

We also took a closer look at the functions of the DBA and the DBMS. Among other things, the DBA is responsible for creating the internal schema (**physical database design**); by contrast, creating the conceptual schema (**logical** or **conceptual** database design) is the responsibility of the *data* administrator. And the DBMS is responsible, among other things, for implementing DDL and DML requests from the user. The DBMS is also responsible for providing some kind of **data dictionary** function.

Database systems can also be conveniently thought of as consisting of a **server** (the DBMS itself) and a set of **clients** (the applications). Client and server can and often will run on distinct machines, thus providing one simple kind of **distributed processing**. In general, each server can serve many clients, and each client can access many servers. If the system provides total "transparency"—meaning that each client can behave as if it were dealing with a single server on a single machine, regardless of the overall physical state of affairs—then we have a genuine **distributed database system**.

EXERCISES

2.1 Draw a diagram of the database system architecture presented in this chapter (the ANSI/SPARC architecture).

2.2 Explain the following in your own words:

back end	front end
client	host language
conceptual DDL, schema, view	load
conceptual/internal mapping	logical database design
data definition language	internal DDL, schema, view
data dictionary	physical database design
data manipulation language	planned request
data sublanguage	reorganization
DB/DC system	server
DC manager	stored database definition
distributed database	unload/reload
distributed processing	unplanned request
external DDL, schema, view	user interface
external/conceptual mapping	utility

2.3 Describe the sequence of steps involved in retrieving a particular external record occurrence.

2.4 List the major functions performed by the DBMS.

2.5 Distinguish between logical and physical data independence.

2.6 What do you understand by the term *metadata*?

2.7 List the major functions performed by the DBA.

2.8 Distinguish between the DBMS and a file management system.

2.9 Give some examples of vendor-provided tools.

2.10 Give some examples of database utilities.

2.11 Examine any database system that might be available to you. Try to map that system to the ANSI/SPARC architecture as described in this chapter. Does it cleanly support the three levels of the architecture? How are the mappings between levels defined? What do the various DDLs (external, conceptual, internal) look like? What data sublanguage(s) does the system support? What host languages? Who performs the DBA function? Are there any security or integrity facilities? Is there a dictionary? Is it self-describing? What vendor-provided applications does the system support? What utilities? Is there a separate DC manager? Are there any distributed processing capabilities?

REFERENCES AND BIBLIOGRAPHY

Most of the following references are fairly old by now, but they are still relevant to the concepts introduced in the present chapter. See also the references in Chapter 14.

2.1 ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report. *FDT* (bulletin of ACM SIGMOD) 7, No. 2 (1975).

2.2 Dionysios C. Tsichritzis and Anthony Klug (eds.): "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems," *Information Systems 3* (1978).

References [2.1] and [2.2] are the Interim and Final Report, respectively, of the so-called ANSI/SPARC Study Group. The ANSI/X3/SPARC Study Group on Data Base Management Systems (to give it its full title) was established in late 1972 by the Standards Planning and Requirements Committee (SPARC) of the American National Standards Institute (ANSI) Committee on Computers and Information Processing (X3). (*Note:* Some 25 years later, the name X3 was changed to NCITS—National Committee on Information Technology Standards. A few years later again, it was changed to INCITS—IN for International instead of National.) The objectives of the Study Group were to determine which areas, if any, of database technology were appropriate for standardization, and to produce a set of recommendations for action in each such area. In working to meet these objectives, the Study Group took the reasonable position that *interfaces* were the only aspect of a database system that could possibly be suitable for standardization, and accordingly defined a generalized database system architecture, or framework, that emphasized the role of such interfaces. The Final Report provides a detailed description of that architecture and of some of the 42 (!) identified interfaces. The Interim Report is an earlier working document that is still of some interest; in some areas it provides additional detail.

2.3 J. J. van Griethuysen (ed.): "Concepts and Terminology for the Conceptual Schema and the Information Base," International Organization for Standardization (ISO) Technical Report ISO/TR 9007:1987(E) (March 1982; revised July 1987).

This document is the report of an ISO Working Group whose objectives included "the definition of concepts for conceptual schema languages." It includes an introduction to three competing candidates (more accurately, three *sets* of candidates) for an appropriate set of formalisms, and applies each of the three to a common example involving the activities of a hypothetical car registration authority. The three sets of contenders are (1) "entity-attribute-relationship" schemes, (2) "binary relationship" schemes, and (3) "interpreted predicate logic" schemes. The report also includes a discussion of the fundamental concepts underlying the notion of the conceptual schema, and suggests some principles as a basis for implementation of a system that properly supports that notion. Heavy going in places, but an important document for anyone seriously interested in the conceptual level of the system.

2.4 William Kent: *Data and Reality.* Amsterdam, Netherlands: North-Holland/New York, N.Y.: Elsevier Science (1978).

A stimulating and thought-provoking discussion of the nature of information, and in particular of the conceptual schema. "This book projects a philosophy that life and reality are at bottom amorphous, disordered, contradictory, inconsistent, nonrational, and nonobjective" (excerpt from the final chapter). The book can be regarded in large part as a compendium of real-world problems that (it is suggested) existing database formalisms—in particular, formalisms that are based on conventional record-like structures, which includes the relational model—have difficulty dealing with. Recommended.

2.5 Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis: "The GMAP: A Versatile Tool for Physical Data Independence," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

GMAP stands for *Generalized Multi-level Access Path*. The authors of the paper note correctly that today's database products "force users to frame their queries in terms of a logical schema that is directly tied to physical structures," and hence are rather weak on physical data independence. In their paper, therefore, they propose a conceptual/internal mapping language (to use the terminology of the present chapter) that can be used to specify far more kinds of mappings than are typically supported in products today. Given a particular "logical schema," the language (which is based on relational algebra—see Chapter 7—and is therefore declarative, not procedural, in nature) allows the specification of numerous different "physical" or internal schemas, all of them formally derived from that logical schema. Among other things, those physical schemas can include vertical and horizontal partitioning (or "fragmentation"—see Chapter 21), any number of physical access paths, clustering, and controlled redundancy.

The paper also gives an algorithm for transforming user operations against the logical schema into equivalent operations against the physical schema. A prototype implementation shows that the DBA can tune the physical schema to "achieve significantly better performance than is possible with conventional techniques."

CHAPTER 3

An Introduction to Relational Databases

3.1 INTRODUCTION

As explained in Chapter 1, the emphasis in this book is heavily on relational systems. In particular, Part II covers the theoretical foundations of such systems—that is, the relational model—in considerable depth. The purpose of the present chapter is to give a preliminary, intuitive, and very informal introduction to the material to be addressed in Part II (and to some extent in subsequent parts too), in order to pave the way for a better understanding of those later parts of the book. Most of the topics mentioned will be discussed much more formally, and in much more detail, in those later chapters.

3.2 AN INFORMAL LOOK AT THE RELATIONAL MODEL

We claimed in Chapter 1 that relational systems are based on a formal foundation, or theory, called *the relational model of data*. The relational model is often described as having the following three aspects:

- *Structural aspect:* The data in the database is perceived by the user as tables, and nothing but tables.
- *Integrity aspect:* Those tables satisfy certain integrity constraints, to be discussed toward the end of this section.
- *Manipulative aspect:* The operators available to the user for manipulating those tables—for example, for purposes of data retrieval—are operators that derive tables from tables. Of those operators, three particularly important ones are *restrict*, *project*, and *join*.

A simple relational database, the departments-and-employees database, is shown in Fig. 3.1. As you can see, that database is indeed "perceived as tables" (and the meanings of those tables are intended to be self-evident). Fig. 3.2 shows some sample restrict, project, and join operations against the database of Fig. 3.1. Here are (very loose!) definitions of those operations:

- The *restrict* operation extracts specified rows from a table. *Note:* Restrict is sometimes called *select*; we prefer *restrict* because the operator is not the same as the SELECT of SQL.
- The *project* operation extracts specified columns from a table.
- The *join* operation combines two tables into one on the basis of common values in a common column.

Of the examples in Fig. 3.2, the only one that seems to need any further explanation is the join example. Join requires the two tables to have a common column, which tables DEPT and EMP do (they both have a column called DEPT#), and so they can be joined on

DEPT

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.1 The departments-and-employees database (sample values)

Restrict: *Result:*

DEPTs where BUDGET > 8M

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M

Project: *Result:*

DEPTs over DEPT#, BUDGET

DEPT#	BUDGET
D1	10M
D2	12M
D3	5M

Join:

DEPTs and EMPs over DEPT#

Result:

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K
D1	Marketing	10M	E2	Cheng	42K
D2	Development	12M	E3	Finzi	30K
D2	Development	12M	E4	Saito	35K

Fig. 3.2 Restrict, project, and join (examples)

the basis of common values in that column. To be specific, a given row from table DEPT will join to a given row in table EMP (to yield a row of the result table) if and only if the two rows in question have a common DEPT# value. For example, the DEPT and EMP rows

DEPT#	DNAME	BUDGET
D1	Marketing	10M

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K

(column names shown for explicitness) join together to produce the result row

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

because they have the same value, D1, in the common column. Note that the common value appears once, not twice, in the result row. The overall result of the join contains all possible rows that can be obtained in this manner, and no other rows. Observe in particular that since no EMP row has a DEPT# value of D3 (i.e., no employee is currently assigned to that department), no row for D3 appears in the result, even though there *is* a row for D3 in table DEPT.

Now, one point that Fig. 3.2 clearly shows is that *the result of each of the three operations is another table* (in other words, the operators are indeed "operators that derive tables from tables," as required). This is the closure property of relational systems, and it is very important. Basically, because the output of any operation is the same kind of object as the input—they are all tables—*the output from one operation can become input*

to another. Thus it is possible, for example, to take a projection of a join, a join of two restrictions, a restriction of a projection, and so on. In other words, it is possible to write *nested relational expressions*—that is, relational expressions in which the operands themselves are represented by relational expressions, not necessarily just by simple table names. This fact in turn has numerous important consequences, as we will see later, both in this chapter and in many subsequent ones.

By the way, when we say that the output from each operation is another table, it is important to understand that we are talking *from a conceptual point of view.* We do not mean to imply that the system actually has to materialize the result of every individual operation in its entirety.[1] For example, suppose we are trying to compute a restriction of a join. Then, as soon as a given row of the join is formed, the system can immediately test that row against the specified restriction condition to see whether it belongs in the final result, and immediately discard it if not. In other words, the intermediate result that is the output from the join might never exist as a fully materialized table in its own right at all. As a general rule, in fact, the system tries very hard *not* to materialize intermediate results in their entirety, for obvious performance reasons. *Note:* If intermediate results are fully materialized, the overall expression evaluation strategy is called (unsurprisingly) materialized evaluation; if intermediate results are passed piecemeal to subsequent operations, it is called pipelined evaluation.

Another point that Fig. 3.2 also clearly illustrates is that the operations are all set-at-a-time, not row-at-a-time; that is, the operands and results are whole tables, not just single rows, and tables contain *sets* of rows. (A table containing a set of just one row is legal, of course, as is an *empty* table, i.e., one containing no rows at all.) For example, the join in Fig. 3.2 operates on two tables of three and four rows respectively, and returns a result table of four rows. By contrast, the operations in nonrelational systems are typically at the row- or record-at-a-time level; thus, this *set processing capability* is a major distinguishing characteristic of relational systems (see further discussion in Section 3.5).

Let us return to Fig. 3.1 for a moment. There are a couple of additional points to be made in connection with the sample database of that figure:

- First, note that relational systems require only that the database be *perceived by the user* as tables. Tables are the logical structure in a relational system, not the physical structure. At the physical level, in fact, the system is free to store the data any way it likes—using sequential files, indexing, hashing, pointer chains, compression, and so on—provided only that it can map that stored representation to tables at the logical level. Another way of saying the same thing is that tables represent an *abstraction* of the way the data is physically stored—an abstraction in which numerous storage-level details (such as stored record placement, stored record sequence, stored data value representations, stored record prefixes, stored access structures such as indexes, and so forth) are all *hidden from the user.*

  Incidentally, the term *logical structure* in the foregoing paragraph is intended to encompass both the conceptual and external levels, in ANSI/SPARC terms. The point is that—as explained in Chapter 2—the conceptual and external levels in a relational

[1] In other words, to repeat from Chapter 1, the relational model is indeed a *model*—it has nothing to say about implementation.

system will both be relational, but the internal level will not be. In fact, relational theory as such has nothing to say about the internal level at all; it is, to repeat, concerned with how the database looks to the *user*.[2] The only requirement is that, to repeat, whatever physical structure is chosen at the internal level must fully support the required logical structure.

- Second, relational databases abide by a very nice principle, called *The Information Principle: The entire information content of the database is represented in one and only one way—namely, as explicit values in column positions in rows in tables.* This method of representation is the *only* method available (at the logical level, that is) in a relational system. In particular, there are no *pointers* connecting one table to another. In Fig. 3.1, for example, there is a connection between the D1 row of table DEPT and the E1 row of table EMP, because employee E1 works in department D1; but that connection is represented, not by a pointer, but by the appearance of the *value* D1 in the DEPT# position of the EMP row for E1. In nonrelational systems such as IMS or IDMS, by contrast, such information is typically represented—as mentioned in Chapter 1—by some kind of *pointer* that is explicitly visible to the user.

  Note: We will explain in Chapter 26 just why allowing such user-visible pointers would constitute a violation of *The Information Principle*. Also, when we say there are no pointers in a relational database, we do not mean there cannot be pointers at the *physical level*—on the contrary, there certainly can, and indeed there almost certainly will. But, to repeat, all such physical storage details are concealed from the user in a relational system.

So much for the structural and manipulative aspects of the relational model; now we turn to the integrity aspect. Consider the departments-and-employees database of Fig. 3.1 once again. In practice, that database might be required to satisfy any number of integrity constraints—for example, employee salaries might have to be in the range 25K to 95K (say), department budgets might have to be in the range 1M to 15M (say), and so on. Certain of those constraints are of such major pragmatic importance, however, that they enjoy some special nomenclature. To be specific:

1. Each row in table DEPT must include a unique DEPT# value; likewise, each row in table EMP must include a unique EMP# value. We say, loosely, that columns DEPT# in table DEPT and EMP# in table EMP are the **primary keys** for their respective tables. (Recall from Chapter 1 that we indicate primary keys in our figures by double underlining.)
2. Each DEPT# value in table EMP must exist as a DEPT# value in table DEPT, to reflect the fact that every employee must be assigned to an existing department. We say, loosely, that column DEPT# in table EMP is a **foreign key**, referencing the primary key of table DEPT.

[2] It is an unfortunate fact that most of today's SQL products do not support this aspect of the theory properly. To be more specific, they typically support only rather restrictive conceptual/internal mappings; typically, in fact, they map one logical table directly to one stored file. This is one reason why (as noted in Chapter 1) those products do not provide as much data independence as relational technology is theoretically capable of. See Appendix A for further discussion.

A More Formal Definition

We close this section with a somewhat more formal definition of the relational model, for purposes of subsequent reference (despite the fact that the definition is quite abstract and will not make much sense at this stage). Briefly, the relational model consists of the following five components:

1. An open-ended collection of **scalar types** (including in particular the type *boolean* or *truth value*)
2. A **relation type generator** and an intended interpretation for relations of types generated thereby
3. Facilities for defining **relation variables** of such generated relation types
4. A relational **assignment** operation for assigning relation values to such relation variables
5. An open-ended collection of generic **relational operators** ("the relational algebra") for deriving relation values from other relation values

As you can see, the relational model is very much more than just "tables plus restrict, project, and join," though it is often characterized in such a manner informally.

By the way, you might be surprised to see no explicit mention of integrity constraints in the foregoing definition. The fact is, however, such constraints represent just one application of the relational operators (albeit a very important one); that is, such constraints are formulated in terms of those operators, as we will see in Chapter 9.

3.3 RELATIONS AND RELVARS

If it is true that a relational database is basically just a database in which the data is perceived as tables—and of course it *is* true—then a good question to ask is: Why exactly do we call such a database relational? The answer is simple (in fact, we mentioned it in Chapter 1): *Relation* is just a mathematical term for a table—to be precise, a table of a specific kind (details to be pinned down in Chapter 6). Thus, for example, we can say that the departments-and-employees database of Fig. 3.1 contains two *relations*.

Now, in informal contexts it is usual to treat the terms *relation* and *table* as if they were synonymous; in fact, the term *table* is used much more often than the term *relation* in practice. But it is worth taking a moment to understand why the term *relation* was introduced in the first place. Briefly:

- As we have seen, relational systems are based on the relational model. The relational model in turn is an abstract theory of data that is based on certain aspects of mathematics (mainly set theory and predicate logic).
- The principles of the relational model were originally laid down in 1969–70 by E. F. Codd, at that time a researcher at IBM. It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into a field (database management) that prior to

that time was all too deficient in any such qualities. Codd's ideas were first widely disseminated in a now classic paper, "A Relational Model of Data for Large Shared Data Banks" (reference [6.1] in Chapter 6).

- Since that time, those ideas—by now almost universally accepted—have had a wide-ranging influence on just about every aspect of database technology, and indeed on other fields as well, such as the fields of artificial intelligence, natural language processing, and hardware design.

Now, the relational model as originally formulated by Codd very deliberately made use of certain terms, such as the term *relation* itself, that were not familiar in IT circles at that time (even though the concepts in some cases were). The trouble was, many of the more familiar terms were very fuzzy—they lacked the precision necessary to a formal theory of the kind that Codd was proposing. For example, consider the term *record*. At different times and in different contexts, that single term can mean either a record *occurrence* or a record *type;* a *logical* record or a *physical* record; a *stored* record or a *virtual* record; and perhaps other things besides. The relational model therefore does not use the term *record* at all—instead, it uses the term *tuple* (rhymes with *couple*), to which it gives a very precise definition. We will discuss that definition in detail in Chapter 6; for present purposes, it is sufficient to say that the term *tuple* corresponds approximately to the notion of a row (just as the term *relation* corresponds approximately to the notion of a table).

In the same kind of way, the relational model does not use the term *field;* instead, it uses the term *attribute*, which for present purposes we can say corresponds approximately to the notion of a column in a table.

When we move on to study the more formal aspects of relational systems in Part II, we will make use of the formal terminology, but in this chapter we are not trying to be so formal (for the most part, at any rate), and we will mostly stick to terms such as *row* and *column* that are reasonably familiar. However, one formal term we will start using a lot from this point forward is the term *relation* itself.

We return to the departments-and-employees database of Fig. 3.1 to make another important point. The fact is, DEPT and EMP in that database are really relation variables: variables, that is, whose values are relation values (different relation values at different times). For example, suppose EMP currently has the value—the *relation* value, that is—shown in Fig. 3.1, and suppose we delete the row for Saito (employee number E4):

```
DELETE EMP WHERE EMP# = EMP# ('E4') ;
```

The result is shown in Fig. 3.3.

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K

Fig. 3.3 Relation variable EMP after deleting E4 row

Conceptually, what has happened here is that *the old relation value of EMP has been replaced* en bloc *by an entirely new relation value.* Of course, the old value (with four rows) and the new one (with three) are very similar, but conceptually they *are* different values. Indeed, the delete operation in question is basically just shorthand for a certain relational assignment operation that might look like this:

```
EMP  :=  EMP WHERE NOT ( EMP# = EMP# ('E4') ) ;
```

As in all assignments, what is happening here, conceptually, is that (a) the *expression* on the right side is evaluated, and then (b) the result of that evaluation is assigned to the *variable* on the left side (naturally that left side must identify a variable specifically). As already stated, the net effect is thus to replace the "old" EMP value by a "new" one. (As an aside, we remark that we have now seen our first examples of the use of the Tutorial D language—both the original DELETE and the equivalent assignment are expressed in that language.)

In analogous fashion, relational INSERT and UPDATE operations are also basically shorthand for certain relational assignments. See Chapter 6 for further details.

Now, it is an unfortunate fact that much of the literature uses the term *relation* when what it really means is a relation *variable* (as well as when it means a relation *per se*—that is, a relation *value*). Historically, however, this practice has certainly led to some confusion. Throughout this book, therefore, we will distinguish very carefully between relation variables and relations *per se;* following reference [3.3], in fact, we will use the term relvar as a convenient shorthand for *relation variable,* and we will take care to phrase our remarks in terms of relvars, not relations, when it really is relvars that we mean.[3] Please note, therefore, that from this point forward we take the unqualified term *relation* to mean a relation value specifically (just as we take, e.g., the unqualified term *integer* to mean an integer value specifically), though we will also use the qualified term *relation value* on occasion, for emphasis.

Before going any further, we should warn you that the term *relvar* is not in common usage—but it should be! We really do feel it is important to be clear about the distinction between relations *per se* and relation variables. (We freely admit that earlier editions of this book failed in this respect, but then so did the rest of the literature. What is more, most of it still does.) Note in particular that, by definition, update operations and integrity constraints—see Chapters 6 and 9, respectively—both apply specifically to relvars, not relations.

3.4 WHAT RELATIONS MEAN

In Chapter 1, we mentioned the fact that columns in relations have associated data types (*types* for short, also known as *domains*). And at the end of Section 3.2, we said that the relational model includes "an open-ended set of . . . types." Note carefully that the fact that the set is open-ended implies among other things that users will be able to define their

[3] The distinction between relation values and relation variables is actually a special case of the distinction between values and variables in general. We will examine this latter distinction in depth in Chapter 5.

own types (as well as being able to make use of system-defined or *built-in* types, of course). For example, we might have user-defined types as follows (Tutorial D syntax again; the ellipses ". . ." denote portions of the definitions that are not germane to the present discussion):

```
TYPE EMP# ... ;
TYPE NAME ... ;
TYPE DEPT# ... ;
TYPE MONEY ... ;
```

Type EMP#, for example, can be regarded (among other things) as *the set of all possible employee numbers;* type NAME as *the set of all possible names;* and so on.

Now consider Fig. 3.4, which is basically the EMP portion of Fig. 3.1 expanded to show the column data types. As the figure indicates, every relation—to be more precise, every relation *value*—has two parts, a set of column-name:type-name pairs (the heading), together with a set of rows that conform to that heading (the body). *Note:* In practice we often ignore the type-name components of the heading, as indeed we have done in all of our examples prior to this point, but you should understand that, conceptually, they are always there.

Now, there is a very important (though perhaps unusual) way of thinking about relations, and that is as follows:

1. Given a relation *r*, the heading of *r* denotes a certain predicate (where a predicate is just a *truth-valued function* that, like all functions, takes a set of *parameters*).
2. As mentioned briefly in Chapter 1, each row in the body of *r* denotes a certain true proposition, obtained from the predicate by substituting certain *argument* values of the appropriate type for the parameters of the predicate ("instantiating the predicate").

   In the case of Fig. 3.4, for example, the predicate looks something like this:

   Employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY

(the parameters are EMP#, ENAME, DEPT#, and SALARY, corresponding of course to the four EMP columns). And the corresponding true propositions are:

Employee E1 is named Lopez, works in department D1, and earns salary 40K

(obtained by substituting the EMP# value E1, the NAME value Lopez, the DEPT# value D1, and the MONEY value 40K for the appropriate parameters);

EMP# : EMP#	ENAME : NAME	DEPT# : DEPT#	SALARY : MONEY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.4 Sample EMP relation value, showing column types

Employee E2 is named Cheng, works in department D1, and earns salary 42K

(obtained by substituting the EMP# value E2, the NAME value Cheng, the DEPT# value D1, and the MONEY value 42K for the appropriate parameters); and so on. In a nutshell, therefore:

- *Types* are (sets of) things we can talk about.
- *Relations* are (sets of) things we say about the things we can talk about.

(There is a nice analogy here that might help you appreciate and remember these important points: *Types are to relations as nouns are to sentences.*) Thus, in the example, the things we can talk about are employee numbers, names, department numbers, and money values, and the things we say are true utterances of the form "The employee with the specified employee number has the specified name, works in the specified department, and earns the specified salary."

It follows from all of the foregoing that:

1. Types and relations are both *necessary* (without types, we have nothing to talk about; without relations, we cannot say anything).
2. Types and relations are *sufficient*, as well as necessary—that is, we do not need anything else, logically speaking.
3. *Types and relations are not the same thing.* It is an unfortunate fact that certain commercial products—not relational ones, by definition!—are confused over this very point. We will discuss this confusion in Chapter 26 (Section 26.2).

By the way, it is important to understand that *every* relation has an associated predicate, including relations that are derived from others by means of operators such as join. For example, the DEPT relation of Fig. 3.1 and the three result relations of Fig. 3.2 have predicates as follows:

- DEPT: *Department DEPT# is named DNAME and has budget BUDGET*
- Restriction of DEPT where BUDGET > 8M: *Department DEPT# is named DNAME and has budget BUDGET, which is greater than eight million dollars*
- Projection of DEPT over DEPT# and BUDGET: *Department DEPT# has some name and has budget BUDGET*
- Join of DEPT and EMP over DEPT#: *Department DEPT# is named DNAME and has budget BUDGET and employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY* (note that this predicate has six parameters, not seven—the two references to DEPT# denote the same parameter)

Finally, we observe that relvars have predicates too: namely, the predicate that is common to all of the relations that are possible values of the relvar in question. For example, the predicate for relvar EMP is:

Employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY

3.5 OPTIMIZATION

As explained in Section 3.2, the relational operators (restrict, project, join, and so on) are all *set-level*. As a consequence, relational languages are often said to be nonprocedural, on the grounds that users specify *what*, not *how*—that is, they say what they want, without specifying a procedure for getting it. The process of "navigating" around the stored data in order to satisfy user requests is performed automatically by the system, not manually by the user. For this reason, relational systems are sometimes said to perform automatic navigation. In nonrelational systems, by contrast, navigation is generally the responsibility of the user. A striking illustration of the benefits of automatic navigation is shown in Fig. 3.5, which contrasts a certain SQL INSERT statement with the "manual navigation" code the user might have to write to achieve an equivalent effect in a nonrelational system (actually a CODASYL network system: the example is taken from the chapter on network databases in reference [1.5]). *Note:* The database is the well-known suppliers-and-parts database. See Section 3.9 for further explanation.

```
INSERT INTO SP ( S#, P#, QTY )
       VALUES ( 'S4', 'P3', 1000 ) ;
```

```
MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
   ACCEPT S-SP-ADDR FROM S-SP CURRENCY
   FIND OWNER WITHIN P-SP
   GET P
   IF P# IN P < 'P3'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN S-SP .
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
   ACCEPT P-SP-ADDR FROM P-SP CURRENCY
   FIND OWNER WITHIN S-SP
   GET S
   IF S# IN S < 'S4'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP
```

Fig. 3.5 Automatic *vs.* manual navigation

Despite the remarks of the previous paragraph, it has to be said that *nonprocedural* is not a very satisfactory term, common though it is, because procedurality and nonprocedurality are not absolutes. The best that can be said is that some language *A* is either more or less procedural than some other language *B*. Perhaps a better way of putting matters would be to say that relational languages are at *a higher level of abstraction* than nonrelational languages (as Fig. 3.5 suggests). Fundamentally, it is this raising of the level of abstraction that is responsible for the increased productivity that relational systems can provide.

Deciding just how to perform the automatic navigation referred to above is the responsibility of a very important DBMS component called the optimizer (we mentioned this component briefly in Chapter 2). In other words, for each access request from the user, it is the job of the optimizer to choose an efficient way to implement that request. By way of an example, let us suppose the user issues the following query (Tutorial D once again):

```
( EMP WHERE EMP# = EMP# ('E4') ) { SALARY }
```

Explanation: The expression inside the outer parentheses ("EMP WHERE . . .") denotes a restriction of the current value of relvar EMP to just the row for employee E4. The column name in braces ("SALARY") then causes the result of that restriction to be projected over the SALARY column. The result of that projection is a single-column, single-row relation that contains employee E4's salary. (Incidentally, note that we are implicitly making use of the relational *closure* property in this example—we have written a nested relational expression, in which the input to the projection is the output from the restriction.)

Now, even in this very simple example, there are probably at least two ways of performing the necessary data access:

1. By doing a physical sequential scan of (the stored version of) relvar EMP until the required data is found
2. If there is an index on (the stored version of) the EMP# column—which in practice there probably will be, because EMP# values are supposed to be unique, and many systems in fact *require* an index in order to enforce uniqueness—then by using that index to go directly to the required data

The optimizer will choose which of these two strategies to adopt. More generally, given any particular request, the optimizer will make its choice of strategy for implementing that request on the basis of considerations such as the following:

- Which relvars are referenced in the request
- How big those relvars currently are
- What indexes exist
- How selective those indexes are
- How the data is physically clustered on the disk
- What relational operations are involved

and so on. To repeat, therefore: Users specify only what data they want, not how to get to that data; the access strategy for getting to that data is chosen by the optimizer ("automatic navigation"). Users and user programs are thus independent of such access strategies, which is of course essential if data independence is to be achieved.

We will have a lot more to say about the optimizer in Chapter 18.

3.6 THE CATALOG

As explained in Chapter 2, the DBMS must provide a catalog or dictionary function. The catalog is the place where—among other things—all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal, external/external) are kept. In other words, the catalog contains detailed information, sometimes called *descriptor information* or *metadata*, regarding the various objects that are of interest to the system itself. Examples of such objects are relvars, indexes, users, integrity constraints, security constraints, and so on. Descriptor information is essential if the system is to do its job properly. For example, the optimizer uses catalog information about indexes and other auxiliary structures, as well as much other information, to help it decide how to implement user requests (see Chapter 18). Likewise, the authorization subsystem uses catalog information about users and security constraints to grant or deny such requests in the first place (see Chapter 17).

Now, one of the nice features of relational systems is that, in such a system, *the catalog itself consists of relvars* (more precisely, *system* relvars, so called to distinguish them from ordinary user ones). As a result, users can interrogate the catalog in exactly the same way they interrogate their own data. For example, the catalog in an SQL system might include two system relvars called TABLE and COLUMN, the purpose of which is to describe the tables (or relvars) in the database and the columns in those tables. For the departments-and-employees database of Fig. 3.1, the TABLE and COLUMN relvars might look in outline as shown in Fig. 3.6.[4]

Note: As mentioned in Chapter 2, the catalog should normally be self-describing—that is, it should include entries describing the catalog relvars themselves (see Exercise 3.3).

Now suppose some user of the departments-and-employees database wants to know exactly what columns relvar DEPT contains (obviously we are assuming that for some reason the user does not already have this information). Then the expression

```
( COLUMN WHERE TABNAME = 'DEPT' ) { COLNAME }
```

does the job.

Here is another example: "Which relvars include a column called EMP#?"

```
( COLUMN WHERE COLNAME = 'EMP#' ) { TABNAME }
```

[4] Note that the presence of column ROWCOUNT in Fig. 3.6 suggests that INSERT and DELETE operations on the database will cause an update to the catalog es a side effect. In practice, ROWCOUNT might be updated only on request (e.g., when some utility is run), meaning that values of that column might not always be current.

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
DEPT	3	3	.....
EMP	4	4	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
DEPT	DEPT#	.....
DEPT	DNAME	.....
DEPT	BUDGET	.....
EMP	EMP#	.....
EMP	ENAME	.....
EMP	DEPT#	.....
EMP	SALARY	.....
.....	.....	.....

Fig. 3.6 . Catalog for the departments-and-employees database (in outline)

Exercise: What does the following do?

```
( ( TABLE JOIN COLUMN )
        WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

3.7 BASE RELVARS AND VIEWS

We have seen that, starting with a set of relvars such as DEPT and EMP, together with a set of relation values for those relvars, relational expressions allow us to obtain further relation values from those given ones. It is time to introduce a little more terminology. The original (given) relvars are called **base relvars**, and their values are called **base relations**; a relation that is not a base relation but can be obtained from the base relations by means of some relational expression is called a **derived**, or **derivable**, relation. *Note:* Base relvars are called real relvars in reference [3.3].

Now, relational systems obviously have to provide a means for creating the base relvars in the first place. In SQL, for example, this task is performed by the CREATE TABLE statement (TABLE here meaning, very specifically, a base relvar, or what SQL calls a base table). And base relvars obviously have to be *named*—for example:

```
CREATE TABLE EMP ... ;
```

However, relational systems usually support another kind of named relvar also, called a view, whose value at any given time is a *derived* relation (and so a view can be thought of, loosely, as a **derived relvar**). The value of a given view at a given time is whatever results from evaluating a certain relational expression at that time; the relational expression in question is specified when the view in question is created. For example, the statement

```
CREATE VIEW TOPEMP AS
     ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ;
```

TOPEMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.7 TOPEMP as a view of EMP (unshaded portions)

might be used to define a view called TOPEMP. (For reasons that are unimportant at this juncture, this example is expressed in a mixture of SQL and Tutorial D.)

When this statement is executed, the relational expression following the AS—the view-defining expression—is not evaluated but is merely remembered by the system in some way (actually by saving it in the catalog, under the specified name TOPEMP). To the user, however, it is now as if there really were a relvar in the database called TOPEMP, with current value as indicated in the unshaded portions (only) of Fig. 3.7. And the user should be able to operate on that view exactly as if it were a base relvar. *Note:* If (as suggested previously) DEPT and EMP are thought of as real relvars, then TOPEMP might be thought of as a *virtual* relvar—that is, a relvar that appears to exist in its own right, but in fact does not (its value at any given time depends on the value(s) of certain other relvar(s)). In fact, views are called virtual relvars in reference [3.3].

Note carefully, however, that although we say that the value of TOPEMP is the relation that would result if the view-defining expression were evaluated, we do *not* mean we now have *a separate copy* of the data; that is, we do not mean the view-defining expression actually *is* evaluated and the result materialized. On the contrary, the view is effectively just a kind of "window" into the underlying base relvar EMP. As a consequence, any changes to that underlying relvar will be automatically and instantaneously visible through that window (assuming they lie within the unshaded portion). Likewise, changes to TOPEMP will automatically and instantaneously be applied to relvar EMP, and hence be visible through the window (see later for an example).

Here is a sample retrieval operation against view TOPEMP:

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

Given the sample data of Fig. 3.7, the result will look like this:

EMP#	SALARY
E1	40K
E4	35K

Conceptually, operations against a view like the retrieval operation just shown are handled by replacing references to the view name by the view-defining expression (i.e., the expression that was saved in the catalog). In the example, therefore, the original expression

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

is modified by the system to become

```
( ( ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ) .
          WHERE SALARY < 42K ) .{ EMP#, SALARY }
```

(we have italicized the view name in the original expression and the replacement text in the modified version). The modified expression can then be simplified to just

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) { EMP#, SALARY }
```

(see Chapter 18), and this latter expression when evaluated yields the result shown earlier. In other words, the original operation against the view is effectively converted into an equivalent operation against the underlying base relvar, and that equivalent operation is then executed in the normal way (more accurately, *optimized* and executed in the normal way).

By way of another example, consider the following DELETE operation:

```
DELETE TOPEMP WHERE SALARY < 42K ;
```

The DELETE that is actually executed looks something like this:

```
DELETE EMP WHERE SALARY > 33K AND SALARY < 42K ;
```

Now, the view TOPEMP is very simple, consisting as it does just of a row-and-column subset of a single underlying base relvar (loosely speaking). In principle, however, a view definition, since it is essentially just a named relational expression, can be *of arbitrary complexity* (it can even refer to other views). For example, here is a view whose definition involves a join of two underlying base relvars:

```
CREATE VIEW JOINEX AS
   ( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) { EMP#, DEPT# } ;
```

We will return to the whole question of view definition and view processing in Chapter 10.

Incidentally, we can now explain the remark in Chapter 2, near the end of Section 2.2, to the effect that the term *view* has a rather specific meaning in relational contexts that is not identical to the meaning assigned to it in the ANSI/SPARC architecture. At the external level of that architecture, the database is perceived as an "external view," defined by an external schema (and different users can have different external views). In relational systems, by contrast, a view is, specifically, a *named, derived, virtual relvar,* as previously explained. Thus, the relational analog of an ANSI/SPARC "external view" is (typically) a collection of several relvars, each of which is a view in the relational sense, and the "external schema" consists of definitions of those views. (It follows that views in the relational sense are the relational model's way of providing **logical data independence**, though once again it has to be said that today's SQL products are sadly deficient in this regard. See Chapter 10.)

Now, the ANSI/SPARC architecture is quite general and allows for arbitrary variability between the external and conceptual levels. In principle, even the *types* of data structures supported at the two levels could be different; for example, the conceptual level

could be relational, while a given user could have an external view that was hierarchic.[5] In practice, however, most systems use the same type of structure as the basis for both levels, and relational products are no exception to this general rule—views are still relvars, just like the base relvars are. And since the same type of object is supported at both levels, the same data sublanguage (usually SQL) applies at both levels. Indeed, the fact that a view is a relvar is precisely one of the strengths of relational systems; it is important in just the same way as the fact that a subset is a set is important in mathematics. *Note:* SQL products and the SQL standard (see Chapter 4) often seem to miss this point, however, inasmuch as they refer repeatedly to "tables and views," with the tacit implication that a view is not a table. You are strongly advised *not* to fall into this common trap of taking "tables" (or "relvars") to mean, specifically, *base* tables (or relvars) only.

There is one final point that needs to be made on the subject of base relvars *vs.* views, as follows. The base relvar *vs.* view distinction is frequently characterized thus:

- Base relvars "really exist," in the sense that they represent data that is physically stored in the database.
- Views, by contrast, do not "really exist" but merely provide different ways of looking at "the real data."

However, this characterization, though perhaps useful in informal contexts, does not accurately reflect the true state of affairs. It is true that users can *think* of base relvars as if they were physically stored; in a way, in fact, the whole point of relational systems is to allow users to think of base relvars as physically existing, while not having to concern themselves with how those relvars are actually represented in storage. But—and it is a big but—this way of thinking should *not* be construed as meaning that a base relvar is physically stored in any kind of direct way (e.g., as a single stored file). As explained in Section 3.2, base relvars are best thought of as an *abstraction* of some collection of stored data—an abstraction in which all storage-level details are concealed. In principle, there can be an arbitrary degree of differentiation between a base relvar and its stored counterpart.[6]

A simple example might help to clarify this point. Consider the departments-and-employees database once again. Most of today's relational systems would probably implement that database with two stored files, one for each of the two base relvars. But there is absolutely no logical reason why there should not be just one stored file of *hierarchic* stored records, each consisting of (a) the department number, name, and budget for some given department, together with (b) the employee number, name, and salary for each employee who happens to be in that department. In other words, the data can be physically stored in whatever way seems appropriate (see Appendix A for a discussion of further possibilities), but it always looks the same at the logical level.

[5] We will see an example of this possibility in Chapter 27.

[6] The following quote from a recent book displays several of the confusions discussed in this paragraph, as well as others discussed in Section 3.3 earlier: "[It] is important to make a distinction between stored relations, which are *tables*, and virtual relations, which are *views* . . . [We] shall use *relation* only where a table or a view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term *base relation* or *base table*." The quote is, regrettably, not at all atypical.

3.8 TRANSACTIONS

Note: The topic of this section is not peculiar to relational systems. We cover it here nevertheless, because an understanding of the basic idea is needed in order to appreciate certain aspects of the material to come in Part II. However, our coverage at this point is deliberately not very deep.

In Chapter 1 we said that a transaction is a "logical unit of work," typically involving several database operations. Clearly, the user needs to be able to inform the system when distinct operations are part of the same transaction, and the BEGIN TRANSACTION, COMMIT, and ROLLBACK operations are provided for this purpose. Basically, a transaction begins when a BEGIN TRANSACTION operation is executed, and terminates when a corresponding COMMIT or ROLLBACK operation is executed. For example (pseudocode):

```
BEGIN TRANSACTION ; /* move $$$ from account A to account B */
UPDATE account A ;                          /* withdrawal   */
UPDATE account B ;                          /* deposit      */
IF everything worked fine
   THEN COMMIT ;                            /* normal end */
   ELSE ROLLBACK ;                          /* abnormal end */
END IF ;
```

Points arising:

1. Transactions are guaranteed to be atomic—that is, they are guaranteed (from a logical point of view) either to execute in their entirety or not to execute at all,[7] even if (say) the system fails halfway through the process.
2. Transactions are also guaranteed to be durable, in the sense that once a transaction successfully executes COMMIT, its updates are guaranteed to appear in the database, even if the system subsequently fails at any point. (It is this durability property of transactions that makes the data in the database *persistent*, in the sense of Chapter 1.)
3. Transactions are also guaranteed to be isolated from one another, in the sense that database updates made by a given transaction *T1* are not made visible to any distinct transaction *T2* until and unless *T1* successfully executes COMMIT. COMMIT causes database updates made by the transaction to become visible to other transactions; such updates are said to be *committed*, and are guaranteed never to be canceled. If the transaction executes ROLLBACK instead, all database updates made by the transaction are canceled (*rolled back*). In this latter case, the effect is as if the transaction never ran in the first place.
4. The interleaved execution of a set of concurrent transactions is usually guaranteed to be serializable, in the sense that it produces the same result as executing those same transactions one at a time in some unspecified serial order.

Chapters 15 and 16 contain an extended discussion of all of the foregoing points, and much else besides.

[7] Since a transaction is the execution of some piece of code, a phrase such as "the execution of a transaction" is really a solecism (if it means anything at all, it has to mean the execution of an execution). However, such phraseology is common and useful, and for want of anything better we will use it ourselves in this book.

3.9 THE SUPPLIERS-AND-PARTS DATABASE

Most of our examples in this book are based on the well-known suppliers-and-parts database. The purpose of this section is to explain that database, in order to serve as a point of reference for later chapters. Fig. 3.8 shows a set of sample data values; subsequent examples will actually assume these specific values, where it makes any difference.[8] Fig. 3.9 shows the database definition, expressed in Tutorial D once again (the Tutorial D keyword VAR means "variable"). Note the primary and foreign key specifications in particular. Note too that (a) several columns have data types of the same name as the column in question; (b) the STATUS column and the two CITY columns are defined in terms of system-defined types—INTEGER (integers) and CHAR (character strings of arbitrary length)—instead of user-defined ones. Note finally that there is an important point that needs to be made regarding the column values as shown in Fig. 3.8, but we are not yet in a position to make it; we will come back to it in Chapter 5, Section 5.3, near the end of the subsection "Possible Representations."

The database is meant to be understood as follows:

- Relvar S represents *suppliers* (more accurately, suppliers *under contract*). Each supplier has a supplier number (S#), unique to that supplier; a supplier name (SNAME), not necessarily unique (though the SNAME values do happen to be unique in Fig. 3.8); a rating or status value (STATUS); and a location (CITY). We assume that each supplier is located in exactly one city.
- Relvar P represents *parts* (more accurately, *kinds* of parts). Each kind of part has a part number (P#), which is unique; a part name (PNAME); a color (COLOR); a

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig. 3.8 The suppliers-and-parts database (sample values)

[8] For ease of reference, Fig. 3.8 is repeated on the inside back cover. For the benefit of readers who might be familiar with the sample data values from earlier editions, we note that part P3 has moved from Rome to Oslo. The same change has also been made in Fig. 4.5 in the next chapter.

```
TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ... ;
TYPE QTY ... ;

VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR
    PRIMARY KEY { S# } ;

VAR P BASE RELATION
    { P#     P#,
      PNAME  NAME,
      COLOR  COLOR,
      WEIGHT WEIGHT,
      CITY   CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#    S#,
       P#    P#,
       QTY   QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

Fig. 3.9 The suppliers-and-parts database (data definition)

weight (WEIGHT); and a location where parts of that kind are stored (CITY). We assume where it makes any difference that part weights are given in pounds (but see the discussion of *units of measure* in Chapter 5, Section 5.4). We also assume that each kind of part comes in exactly one color and is stored in a warehouse in exactly one city.

- Relvar SP represents *shipments*. It serves in a sense to link the other two relvars together, logically speaking. For example, the first row in SP as shown in Fig. 3.8 links a specific supplier from relvar S (namely, supplier S1) to a specific part from relvar P (namely, part P1)—in other words, it represents a shipment of parts of kind P1 by the supplier called S1 (and the shipment quantity is 300). Thus, each shipment has a supplier number (S#), a part number (P#), and a quantity (QTY). We assume there is at most one shipment at any given time for a given supplier and a given part; for a given shipment, therefore, the combination of S# value and P# value is unique with respect to the set of shipments currentiy appearing in SP. Note that the database of Fig. 3.8 includes one supplier, supplier S5, with no shipments at all.

We remark that (as already pointed out in Chapter 1, Section 1.3) suppliers and parts can be regarded as entities, and a shipment can be regarded as a relationship between a particular supplier and a particular part. As also pointed out in that chapter, however, a relationship is best regarded as just a special case of an entity. One advantage of relational databases over all other known kinds is precisely that all entities, regardless of whether

they are in fact relationships, are represented in the same uniform way: namely, by means of rows in relations, as the example indicates.

A couple of final remarks:

- First, the suppliers-and-parts database is clearly very simple, much simpler than any real database is likely to be; most real databases will involve many more entities and relationships (and, more important, many more *kinds* of entities and relationships) than this one does. Nevertheless, it is at least adequate to illustrate most of the points that we need to make in the rest of the book, and (as already stated) we will use it as the basis for most—not all—of our examples as we proceed.
- Second, there is nothing wrong with using more descriptive names such as SUPPLIERS, PARTS, and SHIPMENTS in place of the rather terse names S, P, and SP in Figs. 3.8 and 3.9; indeed, descriptive names are generally to be recommended in practice. But in the case of the suppliers-and-parts database specifically, the relvars are referenced so frequently in what follows that very short names seemed desirable. Long names tend to become irksome with much repetition.

3.10 SUMMARY

This brings us to the end of our brief overview of relational technology. Obviously we have barely scratched the surface of what by now has become a very extensive subject, but the whole point of the chapter has been to serve as a gentle introduction to the much more comprehensive discussions to come. Even so, we have managed to cover quite a lot of ground. Here is a summary of the major topics we have discussed.

A **relational database** is a database that is perceived by its users as a collection of **relation variables**—that is, *relvars*—or, more informally, tables. A **relational system** is a system that supports relational databases and operations on such databases, including in particular the operations **restrict**, **project**, and **join**. These operations, and others like them, are collectively known as **the relational algebra**,[9] and they are all set-level. The closure property of relational systems means the output from every operation is the same kind of object as the input (they are all relations), which means we can write **nested relational expressions**. Relvars can be updated by means of the **relational assignment** operation; the familiar update operations **INSERT**, **DELETE**, and **UPDATE** can be regarded as shorthands for certain common relational assignments.

The formal theory underlying relational systems is called **the relational model of data**. The relational model is concerned with logical matters only, not physical matters. It addresses three principal aspects of data: **data structure**, **data integrity**, and **data manipulation**. The *structural* aspect has to do with relations *per se;* the *integrity* aspect has to do with (among other things) **primary and foreign keys**; and the *manipulative* aspect has to do with the operators (restrict, project, join, etc.). ***The Information Principle***—which we

[9] We mentioned this term in the formal definition of the relational model in Section 3.2. However, we will not start using it in earnest until we reach Chapter 6.

now observe might better be called *The Principle of Uniform Representation*—states that the entire information content of a relational database is represented in one and only one way, as explicit values in column positions in rows in relations. Equivalently: *The only variables allowed in a relational database are, specifically, relvars.*

Every relation has a **heading** and a **body**; the heading is a set of column-name:type-name pairs, the body is a set of rows that conform to the heading. The heading of a given relation can be regarded as a **predicate**, and each row in the body denotes a certain true **proposition**, obtained by substituting certain **arguments** of the appropriate type for the **parameters** of the predicate. Note that these remarks are true of derived relations as well as base ones; they are also true of relvars, *mutatis mutandis*. In other words, *types* are (sets of) things we can talk about, and *relations* are (sets of) things we say about the things we can talk about. Together, types and relations are necessary and sufficient to represent any data we like (at the logical level, that is).

The **optimizer** is the system component that determines how to implement user requests (which are concerned with *what*, not *how*). Since relational systems therefore assume responsibility for navigating around the stored database to locate the desired data, they are sometimes described as **automatic navigation** systems. Optimization and automatic navigation are prerequisites for **physical data independence**.

The **catalog** is a set of system relvars that contain descriptors for the various items that are of interest to the system (base relvars, views, indexes, users, etc.). Users can interrogate the catalog in exactly the same way they interrogate their own data.

The original (given) relvars in a given database are called **base relvars**, and their values are called **base relations**; a relation that is not a base relation but is obtained from the base relations by means of some relational expression is called a **derived relation** (collectively, base and derived relations are sometimes referred to as **expressible relations**). A view is a relvar whose value at any given time is such a derived relation (loosely, it can be thought of as a **derived relvar**); the value of such a relvar at any given time is whatever results from evaluating the associated **view-defining expression** at that time. Note, therefore, that base relvars have *independent existence*, but views do not—they depend on the applicable base relvars. (Another way of saying the same thing is that base relvars are *autonomous*, but views are not.) Users can operate on views in exactly the same way as they operate on base relvars, at least in theory. The system implements operations on views by replacing references to the name of the view by the view-defining expression, thereby converting the operation into an equivalent operation on the underlying base relvar(s).

A **transaction** is a *logical unit of work*, typically involving several database operations. A transaction begins when **BEGIN TRANSACTION** is executed and terminates when **COMMIT** (normal termination) or **ROLLBACK** (abnormal termination) is executed. Transactions are **atomic**, **durable**, and **isolated** from one another. The interleaved execution of a set of concurrent transactions is usually guaranteed to be **serializable**.

Finally, the base example for most of the book is the **suppliers-and-parts database**. It is worth taking the time to familiarize yourself with that example now, if you have not done so already; that is, you should at least know which relvars have which columns and what the primary and foreign keys are (it is not as important to know exactly what the sample data values are!).

EXERCISES

3.1 Explain the following in your own words:

automatic navigation	primary key
base relvar	projection
catalog	proposition
closure	relational database
commit	relational DBMS
derived relvar	relational model
foreign key	restriction
join	rollback
optimization	set-level operation
predicate	view

3.2 Sketch the contents of the catalog relvars TABLE and COLUMN for the suppliers-and-parts database.

3.3 As explained in Section 3.6, the catalog is self-describing—that is, it includes entries for the catalog relvars themselves. Extend Fig. 3.6 to include the necessary entries for the TABLE and COLUMN relvars themselves.

3.4 Here is a query on the suppliers-and-parts database. What does it do? What is the predicate for the result?

```
( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY }
```

3.5 Suppose the expression in Exercise 3.4 is used in a view definition:

```
CREATE VIEW V AS
   ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY } ;
```

Now coasider this query:

```
( V WHERE CITY = 'London' ) { S# }
```

What does this query do? What is the predicate for the result? Show what is involved on the part of the DBMS in processing this query.

3.6 What do you understand by the terms *atomicity*, *durability*, *isolation*, and *serializability* as applied to transactions?

3.7 State *The Information Principle*.

3.8 If you are familiar with the hierarchic data model, identify as many differences as you can between it and the relational model as briefly described in this chapter.

REFERENCES AND BIBLIOGRAPHY

3.1 E. F. Codd: "Relational Database: A Practical Foundation For Productivity," *CACM 25*, No. 2 (February 1982). Republished in Robert L. Ashenhurst (ed.), *ACM Turing Award Lectures: The First Twenty Years 1966–1985*. Reading, Mass.: Addison-Wesley (*ACM Press Anthology Series*, 1987).

This is the paper Codd presented on the occasion of his receiving the 1981 ACM Turing Award for his work on the relational model. It discusses the well-known *application backlog* problem. To quote: "The demand for computer applications is growing fast—so fast that information

systems departments (whose responsibility it is to provide those applications) are lagging further and further behind in their ability to meet that demand." There are two complementary ways of attacking this problem:

1. Provide IT professionals with new tools to increase their productivity.
2. Allow end users to interact directly with the database, thus bypassing the IT professional entirely.

Both approaches are needed, and in this paper Codd gives evidence to suggest that the necessary foundation for both is provided by relational technology.

3.2 C. J. Date: "Why Relational?" in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

An attempt to provide a succinct yet reasonably comprehensive summary of the major advantages of relational systems. The following observation from the paper is worth repeating here: Among all the numerous advantages of "going relational," there is one in particular that cannot be overemphasized, and that is *the existence of a sound theoretical base*. To quote: "Relational really is different. It is different because it is not *ad hoc*. Older systems, by contrast, were *ad hoc;* they may have provided solutions to certain important problems of their day, but they did not rest on any solid theoretical base. Relational systems, by contrast, do rest on such a base . . . which means [they] are *rock solid* . . . Thanks to this solid foundation, relational systems behave in well-defined ways; and (possibly without realizing the fact) users have a simple model of that behavior in their mind, one that enables them to predict with confidence what the system will do in any given situation. There are (or should be) no surprises. This predictability means that user interfaces are easy to understand, document, teach, learn, use, and remember."

3.3 C. J. Date and Hugh Darwen: *Foundation for Future Database Systems: The Third Manifesto* (2d edition). Reading, Mass.: Addison-Wesley (2000). See also *http://www.thethirdmanifesto.com*, which contains certain formal extracts from the book, an errata list, and much other relevant material. Reference [20.1] is also relevant.

The Third Manifesto is a detailed, formal, and rigorous proposal for the future direction of databases and DBMSs. It can be seen as an abstract blueprint for the design of a DBMS and the language interface to such a DBMS. It is based on the classical core concepts type, value, variable, and operator. For example, we might have a *type* INTEGER; the integer "3" might be a *value* of that type; N might be a *variable* of that type, whose value at any given time is some integer value (i.e., some value of that type); and "+" might be an *operator* that applies to integer values (i.e., to values of that type). *Note:* The emphasis on types in particular is brought out by the book's subtitle: *A Detailed Study of the Impact of Type Theory on the Relational Model of Data, Including a Comprehensive Model of Type Inheritance*. Part of the point here is that type theory and the relational model are more or less independent of each other. To be more specific, the relational model does not prescribe support for any particular types (other than type *boolean*); it merely says that attributes of relations must be of some type, thus implying that *some* (unspecified) types must be supported.

The term *relvar* is taken from this book. In this connection, the book also says this: "The first version of this *Manifesto* drew a distinction between database values and database variables, analogons to the distinction between relation values and relation variables. It also introduced the term *dbvar* as shorthand for *database variable*. While we still believe this distinction to be a valid one, we found it had little direct relevance to other aspects of these proposals. We therefore decided, in the interest of familiarity, to revert to more traditional terminology." This

decision subsequently turned out to be a bad one . . . To quote reference [23.4]: "With hindsight, it would have been much better to bite the bullet and adopt the more logically correct terms *database value* and *database variable* (or dbvar), despite their lack of familiarity." In the present book we do stay with the familiar term *database*, but we decided to do so only against our own better judgment (somewhat).

One more point. As the book itself says: "We [confess] that we do feel a little uncomfortable with the idea of calling what is, after all, primarily a technical document a *manifesto*. According to *Chambers Twentieth Century Dictionary*, a manifesto is *a written declaration of the intentions, opinions, or motives* of some person or group (e.g., a political party). By contrast, *The Third Manifesto* is . . . a matter of science and logic, not mere intentions, opinions, or motives." However, *The Third Manifesto* was specifically written to be compared and contrasted with two previous ones, *The Object-Oriented Database System Manifesto* [20.2, 25.1] and *The Third-Generation Database System Manifesto* [26.44], and our title was thus effectively chosen for us.

3.4 C. J. Date: "Great News, The Relational Model Is Very Much Alive!," *http://www.dbdebunk.com* (Augnst 2000).

Ever since it first appeared in 1969, the relational model has been subjected to an extraordinary variety of attacks by a number of different writers. One recent example was entitled, not at all atypically, "Great News, The Relational Model Is Dead!" This article was written as a rebuttal to this position.

3.5 C. J. Date: "There's Only One Relational Model!" *http://www.dbdebunk.com* (February 2001).

Ever since it first appeared in 1969, the relational model has been subjected to an extraordinary variety of misrepresentation and obfuscation by a number of different writers. One recent example was a book chapter titled "Different Relational Models," the first sentence of which read: "There is no such thing as *the* relational model for databases anymore [*sic*] than there is just one geometry." This article was written as a rebuttal to this position.

CHAPTER 4

An Introduction to SQL

4.1 INTRODUCTION

As noted in Chapter 1, SQL is the standard language for relational systems, and it is supported by just about every database product on the market today. SQL was originally developed by IBM Research in the early 1970s [4.9, 4.10]; it was first implemented on a large scale in an IBM prototype called System R [4.1–4.3, 4.12–4.14], and subsequently reimplemented in numerous commercial products from both IBM [4.8, 4.14, 4.21] and many other vendors. In this chapter we present an overview of the major features of the SQL language (more detailed aspects, having to do with such matters as integrity, security, etc., are deferred to the chapters devoted to those topics). Throughout our discussions, we take the unqualified name SQL to refer to the current version of the standard (*viz.*, SQL:1999), barring explicit statements to the contrary.[1] Reference [4.23] is the formal SQL:1999 specification; reference [4.24] is an extensive set of corrections to that specification.

[1] A new version of the standard ("SQL:2003") is anticipated in late 2003, and we will occasionally make explicit reference to that version as well.

Note: The previous version of the standard was SQL:1992, and SQL:1999 is meant to be a compatible extension to that previous version. However, it is only fair to point out that no product today supports even SQL:1992 in its entirety; instead, products typically support what might be called "a superset of a subset" of the standard (either SQL:1999 or, more likely, SQL:1992). To be more specific, most products fail to support the standard in some respects and yet go beyond it in others.[2] For example, IBM's DB2 product does not support all of the standard integrity features, but it does support an operator to rename a base table, which the standard does not.

A few additional preliminary remarks:

- SQL was originally intended to be a "data sublanguage" specifically (see Chapter 2). However, with the addition in 1996 of the SQL Persistent Stored Modules feature (SQL/PSM, *PSM* for short), the standard became computationally complete—it now includes statements such as CALL, RETURN, SET, CASE, IF, LOOP, LEAVE, WHILE, and REPEAT, as well as several related features such as the ability to declare variables and exception handlers. Further details of PSM are beyond the scope of this book, but a tutorial description can be found in reference [4.20].
- SQL uses the term *table* in place of both of the terms *relation* and *relvar*, and the terms *row* and *column* in place of *tuple* and *attribute*, respectively. For consistency with the SQL standard and SQL products, therefore, we will do likewise in this chapter (and throughout this book whenever we are concerned with SQL specifically).
- SQL is an enormous language. The specification [4.23] is well over 2,000 pages long—not counting the more than 300 hundred pages of corrigenda in reference [4.24]. As a consequence, it is not possible, or even desirable, to treat the subject exhaustively in a book of this nature, and we have not attempted to do so; rather, we have omitted many features and simplified many others.
- Finally, it has to be said that (as already indicated at various points in Chapters 1–3) SQL is very far from being the perfect relational language—it suffers from sins of both omission and commission. Nevertheless, it is the standard, it is supported by just about every product on the market, and every database professional needs to know something about it. Hence the coverage in this book.

4.2 OVERVIEW

SQL includes both data definition and data manipulation operations. We consider definitional operations first. Fig. 4.1 gives an SQL definition for the suppliers-and-parts database (compare and contrast Fig. 3.9 in Chapter 3). As you can see, the definition includes one CREATE TYPE statement for each of the six user-defined types (UDTs) and one CREATE TABLE statement for each of the three base tables (as noted in Chapter 3, the keyword TABLE in CREATE TABLE means a base table specifically). Each such

[2] In fact, no product possibly could support the standard in its entirety, because there are simply too many gaps, mistakes, and inconsistencies in references [4.23] and [4.24]. Reference [4.20] includes a detailed discussion of this problem at the SQL:1992 level.

```
CREATE TYPE S# ... ;
CREATE TYPE NAME ... ;
CREATE TYPE P# ... ;
CREATE TYPE COLOR ... ;
CREATE TYPE WEIGHT ... ;
CREATE TYPE QTY ... ;

CREATE TABLE S
     ( S#     S#,
       SNAME  NAME,
       STATUS INTEGER,
       CITY   CHAR(15),
     PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
     ( P#     P#,
       PNAME  NAME,
       COLOR  COLOR,
       WEIGHT WEIGHT,
       CITY   CHAR(15),
     PRIMARY KEY ( P# ) ) ;

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

Fig. 4.1 The suppliers-and-parts database (SQL definition)

CREATE TABLE statement specifies the name of the base table to be created, the names and types of the columns of that table, and the primary key and any foreign keys in that table (possibly some additional information also, not illustrated in Fig. 4.1). Also, please note the following:

- We often make use of the "#" character in (e.g.) type names and column names, but in fact that character is not legal in the standard.
- We use the semicolon ";" as a statement terminator. Whether SQL actually uses such terminators depends on the context. The specifics are beyond the scope of this book.
- The built-in type CHAR in SQL requires an associated *length*—15 in the figure—to be specified.

Having defined the database, we can now start operating on it by means of the SQL manipulative operations SELECT, INSERT, DELETE, and UPDATE. In particular, we can perform relational restrict, project, and join operations on the data, in each case by using the SQL data manipulation statement SELECT. Some examples are given in Fig. 4.2. *Note:* The join example in that figure illustrates the point that dot-qualified names (e.g., S.S#, SP.S#) are sometimes necessary in SQL to "disambiguate" column references. The general rule—though there are exceptions—is that qualified names are always acceptable, but unqualified names are acceptable too as long as they cause no ambiguity.

Restrict:

```
SELECT S#, P#, QTY
FROM   SP
WHERE  QTY < QTY ( 150 ) ;
```

Result:

S#	P#	QTY
S1	P5	100
S1	P6	100

Project:

```
SELECT S#, CITY
FROM   S ;
```

Result:

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

Join:

```
SELECT S.S#, SNAME, STATUS, CITY, P#, QTY
FROM   S, SP
WHERE  S.S# = SP.S# ;
```

Result:

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
..	.....	..	.....	..	...
S4	Clark	20	London	P5	400

Fig. 4.2 Restrict, project, and join examples in SQL

We remark that SQL also supports a shorthand form of the SELECT clause as illustrated by the following example:

```
SELECT *                        /* or "SELECT S.*" (i.e., the */
FROM   S ;                      /* "*" can be dot-qualified)  */
```

The result is a copy of the entire S table; the star or asterisk is shorthand for a "comma-list"—see Section 4.6 for a formal explanation of this term—of (a) names of all columns in the first table referenced in the FROM clause, in the left-to-right order in which those columns are defined within that table, followed by (b) names of all columns in the second table referenced in the FROM clause, in the left-to-right order in which those columns are defined within that table (and so on for all of the other tables referenced in the FROM clause). *Note:* The expression SELECT * FROM *T*, where *T* is a table name, can be further abbreviated to just TABLE *T*.

The SELECT statement is discussed at much greater length in Chapter 8, Section 8.6.

Turning now to update operations: Examples of the SQL INSERT, DELETE, and UPDATE statements have already been given in Chapter 1, but those examples deliberately involved single-row operations only. Like SELECT, however, INSERT, DELETE, and UPDATE are all *set-level* operations, in general (and some of the exercises in Chapter 1 did in fact illustrate this point). Here are some set-level update examples for the suppliers-and-parts database:

```
INSERT
INTO    TEMP ( P#, WEIGHT )
        SELECT P#, WEIGHT
        FROM   P
        WHERE  COLOR = COLOR ('Red') ;
```

This example assumes that we have already created another table TEMP with two columns, P# and WEIGHT. The INSERT statement inserts into that table part numbers and corresponding weights for all red parts.

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

This DELETE statement deletes all shipments for part P2.

```
UPDATE S
SET     STATUS = 2 * STATUS ,
        CITY = 'Rome'
WHERE   CITY = 'Paris' ;
```

This UPDATE statement doubles the status for all suppliers in Paris and moves those suppliers to Rome.

Note: SQL does not include a direct analog of the relational assignment operation. However, we can simulate that operation by first deleting all rows from the target table and then performing an INSERT . . . SELECT. . . . (as in the first example above) into that table.

4.3 THE CATALOG

The SQL standard does include specifications for a standard catalog called the Information Schema. In fact, the conventional terms *catalog* and *schema* are both used in SQL, but with highly SQL-specific meanings; loosely speaking, an SQL catalog consists of the descriptors for an individual database,[3] and an SQL schema consists of the descriptors for that portion of that database that belongs to some individual user. In other words, there can be any number of catalogs (one per database), each consisting of any number of schemas. However, each catalog is required to include exactly one schema called INFORMATION_SCHEMA, and from the user's perspective it is that schema that, as already indicated, performs the normal catalog function.

The Information Schema thus consists of a set of SQL tables whose contents effectively echo, in a precisely defined way, all of the definitions from all of the other schemas in the catalog in question. More precisely, the Information Schema is defined to contain a set of *views* of a hypothetical "Definition Schema." The implementation is not required to support the Definition Schema as such, but it is required (a) to support *some* kind of "Definition Schema" and (b) to support views of that "Definition Schema" that do look like those of the Information Schema. Points arising:

[3] In the interest of accuracy, we should also say that there is no such thing as a "database" in the SQL standard! Exactly what the collection of data is called that is described by a catalog is implementation-defined. However, it is not unreasonable to think of it as a database.

1. The rationale for stating the requirement in terms of two separate constructs as just described is as follows. First, existing products certainly do support something akin to the "Definition Schema." However, those "Definition Schemas" vary widely from one product to another (even when the products in question come from the same vendor). Hence the idea of requiring only that the implementation support certain predefined views of its "Definition Schema" does make sense.
2. We should really say "an" (not "the") Information Schema, since as we have seen there is one such in every catalog. In general, therefore, the totality of data available to a given user will *not* be described by a single Information Schema. For simplicity, however, we will continue to talk as if there were just one.

It is not worth going into great detail on the content of the Information Schema here; instead, we simply list some of the more important Information Schema views, in the hope that their names alone will be sufficient to give some idea of what those views contain. One point that is worth calling out explicitly, however, is that the TABLES view contains information for *all* named tables, views as well as base tables, while the VIEWS view contains information for views only.

```
ASSERTIONS                  TABLES
CHECK_CONSTRAINTS           TABLE_CONSTRAINTS
COLUMNS                     TABLE_PRIVILEGES
COLUMN_PRIVILEGES           USAGE_PRIVILEGES
COLUMN_UDT_USAGE            USER_DEFINED_TYPES
CONSTRAINT_COLUMN_USAGE     UDT_PRIVILEGES
CONSTRAINT_TABLE_USAGE      VIEWS
KEY_COLUMN_USAGE            VIEW_COLUMN_USAGE
REFERENTIAL_CONSTRAINTS     VIEW_TABLE_USAGE
SCHEMATA
```

Reference [4.20] gives more details; in particular, it shows how to formulate queries against the Information Schema (which is not quite as simple as you might expect).

4.4 VIEWS

Here is an example of an SQL view definition:

```
CREATE VIEW GOOD_SUPPLIER
    AS SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ;
```

And here is an example of an SQL qnery against this view:

```
SELECT S#, STATUS
FRON   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

Substituting the view definition for the reference to the view name, we obtain an expression that looks something like this (note the snbquery in the FROM clause):

```
SELECT GOOD_SUPPLIER.S#, GOOD_SUPPLIER.STATUS
FROM ( SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ) AS GOOD_SUPPLIER
WHERE  GOOD_SUPPLIER.CITY = 'London' ;
```

And this expression can then be simplified to something like this:

```
SELECT S#, STATUS
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

This latter query is what is actually executed.

By way of another example, consider the following DELETE operation:

```
DELETE
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

The DELETE actually executed looks something like this:

```
DELETE
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

4.5 TRANSACTIONS

SQL includes direct analogs of the BEGIN TRANSACTION, COMMIT, and ROLLBACK statements from Chapter 3, called START TRANSACTION, COMMIT WORK, and ROLLBACK WORK, respectively (the keyword WORK is optional).

4.6 EMBEDDED SQL

Most SQL products allow SQL statements to be executed both directly (i.e., interactively from an online terminal) and as part of an application program (i.e., the SQL statements can be embedded, meaning they can be intermixed with the programming language statements of such a program). In the embedded case, moreover, the application program can typically be written in a variety of host languages; the SQL standard includes support for Ada, C, COBOL, Fortran, Java, M (formerly known as MUMPS), Pascal, and PL/I. In this section we consider the embedded case specifically.

A fundamental principle underlying embedded SQL, which we call the **dual-mode principle**, is that *any SQL statement that can be used interactively can also be embedded in an application program.* Of course, there are various differences of detail between a given interactive SQL statement and its embedded counterpart, and retrieval operations in particular require significantly extended treatment in the embedded case—see later in this section—but the principle is nevertheless broadly true. (Its converse is not, by the way; several embedded SQL statements cannot be used interactively, as we will see.)

Before we can discuss the actual statements of embedded SQL, it is necessary to cover a number of preliminary details. Most of those details are illustrated by the program

```
EXEC SQL BEGIN DECLARE SECTION ;

   DCL SQLSTATE CHAR(5) ;
   DCL P#       CHAR(6) ;
   DCL WEIGHT   FIXED DECIMAL(5,1) ;

EXEC SQL END DECLARE SECTION ;

P# = 'P2' ;                        /* for example              */
EXEC SQL SELECT P.WEIGHT
         INTO   :WEIGHT
         FROM   P
         WHERE  P.P# = P# ( :P# ) ;
IF SQLSTATE = '00000'
THEN ... ;                          /* WEIGHT = retrieved value */
ELSE ... ;                          /* some exception occurred  */
```

Fig. 4.3 Fragment of a *PL/I* program with embedded SQL

fragment shown in Fig. 4.3. (To fix our ideas we assume the host language is PL/I. Most of the ideas translate into other host languages with only minor changes.) Points arising:

1. Embedded SQL statements are prefixed by **EXEC SQL**, to distinguish them from statements of the host language, and are terminated by a special terminator symbol (a semicolon for PL/I).
2. An *executable* SQL statement (for the rest of this section we will mostly drop the "embedded" qualifier) can appear wherever an executable host statement can appear. Note that "executable," by the way: Unlike interactive SQL, embedded SQL includes some statements that are purely declarative, not executable. For example, DECLARE CURSOR is not an executable statement (see the subsection "Operations Involving Cursors" later), nor are BEGIN and END DECLARE SECTION (see point 5), and nor is WHENEVER (see point 9).
3. SQL statements can include references to host variables; such references must include a **colon prefix** to distinguish them from SQL column names. Host variables can appear in embedded SQL wherever a literal can appear in interactive SQL. They can also appear in an INTO clause on SELECT (see point 4) or FETCH (again, see the subsection "Operations Involving Cursors" later) to designate targets for retrieval operations.
4. Notice the **INTO** clause on the SELECT statement in Fig. 4.3. The purpose of that clause is (as just indicated) to specify the target variables into which values are to be retrieved; the *i*th target variable mentioned in the INTO clause corresponds to the *i*th value to be retrieved, as specified by the SELECT clause.
5. All host variables referenced in SQL statements must be declared (DCL in PL/I) within an **embedded SQL declare section**, which is delimited by the **BEGIN** and **END DECLARE SECTION** statements.
6. Every program containing embedded SQL statements must include a host variable called **SQLSTATE**. After any SQL statement has been executed, a status code is returned to the program in that variable; a value of 00000 means the statement exe-

cuted successfully, and a value of 02000 means the statement did execute but no data was found to satisfy the request (see reference [4.23] for details of other values). In principle, therefore, every SQL statement in the program should be followed by a test on SQLSTATE, and appropriate action taken if the value is not what was expected. In practice, however, such testing can often be implicit (see point 9).

7. Every host variable must have a data type appropriate to the uses to which it is put. For example, a host variable that is to be used as a target (e.g., on SELECT) must have a data type that is compatible with that of the expression that provides the value to be assigned to that target: likewise, a host variable that is to be used as a source (e.g., on INSERT) must have a data type that is compatible with that of the SQL column to which values of that source are to be assigned. The details are a little complicated, however, and we therefore ignore the issue for the remainder of this chapter (for the most part, at any rate), deferring further discussion to Chapter 5, Section 5.7.
8. Host variables and SQL columns can have the same name.
9. As already mentioned, every SQL statement should in principle be followed by a test of the returned SQLSTATE value. The WHENEVER statement is provided to simplify this process. The WHENEVER statement takes the form:

   ```
   EXEC SQL WHENEVER <condition> <action> ;
   ```

   Possible *<condition>*s include NOT FOUND, SQLWARNING, and SQLEXCEPTION (others include specific SQLSTATE values and violation of specified integrity constraints); *<action>* is either CONTINUE or a GO TO statement. WHENEVER is not an executable statement—rather, it is a directive to the SQL compiler: "WHENEVER *<condition>* GO TO *<label>*" causes the compiler to insert a statement of the form "IF *<condition>* THEN GO TO *<label>* . . ." after each executable SQL statement it encounters; "WHENEVER *<condition>* CONTINUE" causes it not to insert any such statements, the implication being that the programmer will insert appropriate statements by hand. The *<condition>*s NOT FOUND, SQLWARNING, and SQLEXCEPTION are defined as follows:

`NOT FOUND`	*means*	no data was found —SQLSTATE = 02xxx
`SQLWARNING`	*means*	a minor error occurred —SQLSTATE = 01xxx
`SQLEXCEPTION`	*means*	a major error occurred —see reference [4.23] for SQLSTATE

   Each WHENEVER statement the compiler encounters on its sequential scan through the program text for a particular condition overrides the previous one it found for that condition.
10. Note finally that, to use the terminology of Chapter 2, embedded SQL constitutes a *loose coupling* between SQL and the host language.

So much for the preliminaries. In the rest of this section we concentrate on data manipulation operations specifically. As already indicated, most of those operations can be handled in a fairly straightforward fashion (i.e., with only minor changes to their

syntax). Retrieval operations require special treatment, however. The problem is that such operations retrieve many rows (in general), not just one, and host languages are generally not equipped to handle the retrieval of more than one row at a time. It is therefore necessary to provide some kind of bridge between the set-level retrieval capabilities of SQL and the row-level retrieval capabilities of the host. Such is the purpose of cursors. A cursor consists essentially of a kind of (logical) *pointer*—a pointer in the application, that is, not one in the database—that can be used to run through a collection of rows, pointing to each of the rows in turn and thereby providing addressability to those rows one at a time. However, we defer detailed discussion of cursors to the subsection "Operations Involving Cursors," and consider first those statements that have no need of them.

Operations Not Involving Cursors

The data manipulation statements that do not need cursors are as follows:

- Singleton SELECT
- INSERT
- DELETE (except the CURRENT form—see the next subsection)
- UPDATE (except the CURRENT form—see the next subsection)

We give examples of each in turn.

Singleton SELECT: Get status and city for the supplier whose supplier number is given by the host variable GIVENS#.

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK, :TOWN
         FROM   S
         WHERE  S# = S# ( :GIVENS# ) ;
```

We use the term *singleton SELECT* to mean a SELECT expression that evaluates to a table containing at most one row. In the example, if there is exactly one row in table S satisfying the condition in the WHERE clause, then the STATUS and CITY values from that row will be assigned to the host variables RANK and TOWN as requested, and SQLSTATE will be set to 00000; if no S row satisfies the WHERE condition, SQLSTATE will be set to 02000; and if more than one does, the program is in error, and SQLSTATE will be set to an error code.

INSERT: Insert a new part (part number, name, and weight given by host variables P#, PNAME, PWT, respectively; color and city unknown) into table P.

```
EXEC SQL INSERT
         INTO   P ( P#, PNAME, WEIGHT )
         VALUES ( :P#, :PNAME, :PWT ) ;
```

The COLOR and CITY values for the new part will be set to the applicable *default* values (see Chapter 6, Section 6.6). *Note:* For reasons beyond the scope of this book, the default for a column that is of some user-defined type will necessarily be *null*. (We defer detailed discussion of nulls to Chapter 19. Occasional references prior to that point are unavoidable, however.)

DELETE: Delete all shipments for suppliers whose city is given by the host variable CITY.

```
EXEC SQL DELETE
         FROM   SP
         WHERE  :CITY =
              ( SELECT CITY
                FROM   S
                WHERE  S.S# = SP.S# ) ;
```

If no supplier rows satisfy the WHERE condition, SQLSTATE will be set to 02000. Again, note the subquery (in the WHERE clause this time).

UPDATE: Increase the status of all London suppliers by the amount given by the host variable RAISE.

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CITY = 'London' ;
```

Again SQLSTATE will be set to 02000 if no SP rows satisfy the WHERE condition.

Operations Involving Cursors

Now we turn to the question of set-level retrieval—that is, retrieval of a set containing an arbitrary number of rows, instead of at most one row as in the singleton SELECT case discussed in the previous subsection. As explained earlier, what is needed here is a mechanism for accessing the rows in the set one by one, and cursors provide such a mechanism. The process is illustrated in outline by the example of Fig. 4.4, which is intended to retrieve S#, SNAME, and STATUS information for all suppliers in the city given by the host variable Y.

Explanation: The statement "DECLARE X CURSOR . . ." defines a cursor called X, with an associated *table expression* (i.e., an expression that evaluates to a table), specified by the SELECT that forms part of that DECLARE. That table expression is not evaluated at this point; DECLARE CURSOR is a purely declarative statement. The expression *is*

```
EXEC SQL DECLARE X CURSOR FOR      /* define the cursor   */
         SELECT S.S#, S.SNAME, S.STATUS
         FROM   S
         WHERE  S.CITY = :Y
         ORDER  BY S# ASC ;

EXEC SQL OPEN X ;                  /* execute the query   */
         DO for all S rows accessible via X ;
            EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                   /* fetch next supplier */
            .........
         END ;
EXEC SQL CLOSE X ;                 /* deactivate cursor X */
```

Fig. 4.4 Multi-row retrieval example

evaluated when the cursor is opened ("OPEN X"). The statement "FETCH X INTO . . ." is then used to retrieve rows one at a time from the resulting set, assigning retrieved values to host variables in accordance with the specifications of the INTO clause in that statement. (For simplicity we have given the host variables the same names as the corresponding database columns. Notice that the SELECT in the cursor declaration does not have an INTO clause of its own.) Since there will be many rows in the result set, the FETCH will normally appear inside a loop; the loop will be repeated as long as there are more rows still to come in that result set. On exit from the loop, cursor X is closed ("CLOSE X").

Now we consider cursors and cursor operations in more detail. First of all, a cursor is declared by means of a DECLARE CURSOR statement, which takes the general form

```
EXEC SQL DECLARE <cursor name> CURSOR
         FOR <table exp> [ <ordering> ] ;
```

(we are ignoring a few optional specifications in the interest of brevity). The optional *<ordering>* takes the form

```
ORDER BY <order item commalist>
```

where (a) the commalist contains at least one *<order item>* and (b) each *<order item>* consists of a column name—unqualified, please note[4]—optionally followed by ASC (ascending) or DESC (descending), where ASC is the default. If no ORDER BY clause is specified, the ordering is system-determined. (As a matter of fact the same is true if an ORDER BY clause *is* specified, at least as far as rows with the same value for the specified *<order item commalist>* are concerned.)

Note: The useful term *commalist* is defined as follows. Let *<xyz>* denote an arbitrary syntactic category (i.e., anything that appears on the left side of some BNF production rule). Then the expression *<xyz commalist>* denotes a sequence of zero or more *<xyz>*s in which each pair of adjacent *<xyz>*s is separated by a comma (and optionally one or more blanks). We will be making extensive use of the commalist shorthand in future syntax rules (all syntax rules, that is, not just SQL ones).

As previously stated, the DECLARE CURSOR statement is declarative, not executable; it declares a cursor with the specified name and having the specified table expression and ordering permanently associated with it. The table expression can include host variable references. A program can include any number of DECLARE CURSOR statements, each of which must (of course) be for a different cursor.

Three executable statements are provided to operate on cursors: OPEN, FETCH, and CLOSE.

- The statement

  ```
  EXEC SQL OPEN <cursor name> ;
  ```

[4] Actually, the column name *can* be qualified if the specified *<table exp>* abides by a rather complex set of rules. The rules in question were introduced with SQL:1999, which also introduced rules according to which an *<order item>* can sometimes specify either (a) a computational expression, as in (e.g.) ORDER BY A+B, or (b) the name of a column that is not part of the result table, as in (e.g.) SELECT CITY FROM S ORDER BY STATUS. Details of these rules are beyond the scope of this book.

opens the specified cursor (which must not currently be open). In effect, the table expression associated with the cursor is evaluated (using the current values for any host variables referenced within that expression); a set of rows is thus identified and becomes the current active set for the cursor. The cursor also identifies a position within that active set: namely, the position just before the first row. *Note:* Active sets are always considered to have an ordering—see the earlier discussion of ORDER BY—and so the concept of position has meaning.[5]

- The statement

```
EXEC SQL FETCH <cursor name>
         INTO <host variable reference commalist> ;
```

advances the specified cursor (which must be open) to the next row in the active set and then assigns the *i*th value from that row to the *i*th host variable referenced in the INTO clause. If there is no next row when FETCH is executed, SQLSTATE is set to 02000 and no data is retrieved.

- The statement

```
EXEC SQL CLOSE <cursor name> ;
```

closes the specified cursor (which must currently be open). The cursor now has no current active set. However, it can subsequently be opened again, in which case it will acquire another active set—probably not exactly the same one as before, especially if the values of any host variables referenced in the cursor declaration have changed in the meantime. Note that changing the values of those host variables while the cursor is open has no effect on the current active set.

Two further statements can include references to cursors, the CURRENT forms of DELETE and UPDATE. If a cursor, X say, is currently positioned on a particular row, then it is possible to DELETE or UPDATE "the current of X"—that is, the row on which X is positioned. For example:

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CURRENT OF X ;
```

The CURRENT forms of DELETE and UPDATE are not permitted if the *<table exp>* in the cursor declaration would define a nonupdatable view if it were part of a CREATE VIEW statement (see Chapter 10, Section 10.6).

4.7 DYNAMIC SQL AND SQL/CLI

The previous section tacitly assumed we could compile any given program in its entirety—SQL statements and all—"ahead of time," as it were (i.e., prior to run time). For certain applications, however, that assumption is unwarranted. By way of example, consider an online application (recall from Chapter 1 that an online application is one that

[5] Sets *per se* do not have an ordering (see Chapter 6), so an "active set" is not really a set, as such, at all. It would be better to think of it as an *ordered list* or *array* (of rows).

supports access to the database from an online terminal or something analogous). Typically, the steps such an application must go through are as follows (in outline):

1. Accept a command from the terminal.
2. Analyze that command.
3. Execute appropriate SQL statements on the database.
4. Return a message and/or results to the terminal.

Now, if the set of commands the program can accept in Step 1 is fairly small, as in the case of (perhaps) a program handling airline reservations, then the set of possible SQL statements to be executed will probably also be small and can be "hardwired" into the program. In this case, Steps 2 and 3 will consist simply of logic to examine the input command and then branch to the part of the program that issues the predefined SQL statement(s). On the other hand, if there can be great variability in the input, then it might not be practicable to predefine and "hardwire" SQL statements for every possible command. Instead, what we need to do is *construct* the necessary SQL statements dynamically, and then compile and execute those constructed statements dynamically. The SQL facilities described in this section are provided to assist in this process.

Dynamic SQL

Dynamic SQL is part of embedded SQL. It consists of a set of "dynamic statements"—which themselves *are* compiled ahead of time—whose purpose is precisely to support the compilation and execution of regular SQL statements that are constructed at run time. Thus, the two principal dynamic statements are PREPARE (in effect, *compile*) and EXECUTE. Their use is illustrated in the following unrealistically simple, but accurate, PL/I example.

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE QTY < QTY ( 300 )' ;
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

Explanation:

1. The name SQLSOURCE identifies a PL/I variable (of type "varying length character string") in which, at run time, the program will somehow construct the source form of some SQL statement—a DELETE statement, in our particular example—as a character string.
2. The name SQLPREPPED, by contrast, identifies an *SQL* variable, *not* a PL/I variable, that will be used to hold the compiled form of the SQL statement whose source form is given in SQLSOURCE. (The names SQLSOURCE and SQLPREPPED are arbitrary, of course.)
3. The PL/I assignment statement "SQLSOURCE = . . . ;" assigns to SQLSOURCE the source form of an SQL DELETE statement. In practice, the process of constructing such a source statement is likely to be much more complex—perhaps involving the

input and analysis of some request from the end.user, expressed in natural language or some other form more user-friendly than SQL.

4. The PREPARE statement then takes that source statement and "prepares" (compiles) it to produce an executable version, which it stores in SQLPREPPED.
5. Finally, the EXECUTE statement executes that SQLPREPPED version and thus causes the actual DELETE to occur. SQLSTATE information from the DELETE is returned exactly as if the DELETE had been executed directly in the normal way.

Note that because it denotes an SQL variable, not a PL/I variable, the name SQLPREPPED does not have a colon prefix when it is referenced in the PREPARE and EXECUTE statements. Note too that such SQL variables do not have to be explicitly declared.

By the way, the process just described is essentially what happens when SQL statements themselves are entered interactively. Most systems provide some kind of interactive SQL query processor. That query processor is in effect just a particular online application; it is ready to accept an extremely wide variety of input—*viz.*, any valid (or invalid!) SQL statement. It then uses the facilities of dynamic SQL to construct suitable SQL statements corresponding to its input, to compile and execute those constructed statements, and to return messages and results back to the terminal.

Of course, there is much more to dynamic SQL than the PREPARE and EXECUTE statements as just described; for example, there are mechanisms for parameterizing the statements to be prepared and providing arguments to be substituted for those parameters when those statements are executed, and there are counterparts to the cursor facilities as described in the previous section. In particular, there is an EXECUTE IMMEDIATE statement, which effectively combines the functions of PREPARE and EXECUTE into a single operation.

Call-Level Interfaces

The SQL Call-Level Interface feature (SQL/CLI, *CLI* for short) was added to the standard in 1995. SQL/CLI is heavily based on Microsoft's *Open Database Connectivity* interface, ODBC. Both permit an application written in one of the usual host languages to issue database requests, not via embedded SQL, but rather by invoking certain vendor-provided routines. Those routines, which must have been linked to the application in question, then use dynamic SQL to perform the requested database operations on the application's behalf. (From the DBMS's point of view, in other words, those routines can be thought of as just another application.)

As you can see, SQL/CLI and ODBC both address the same general problem as dynamic SQL does: They both allow applications to be written for which the exact SQL statements to be executed are not known until run time. However, they actually represent a better approach to the problem than dynamic SQL does. There are two principal reasons for this state of affairs:

- Dynamic SQL is a *source code* standard. Any application using dynamic SQL thus requires the services of some kind of SQL compiler in order to process the operations—PREPARE, EXECUTE, and so on—prescribed by that standard. SQL/CLI, by

contrast, merely standardizes the details of certain *routine invocations* (i.e., subroutine calls, basically); no special compiler services are needed, only the regular services of the standard host language compiler. As a result, applications can be distributed (perhaps by third-party software vendors) in "shrink-wrapped" *object code* form.

- What is more, those applications can be *DBMS-independent;* that is, SQL/CLI includes features that permit the creation (again, perhaps by third-party software vendors) of generic applications that can be used with several different DBMSs, instead of having to be specific to some particular DBMS.

Here by way of illustration is an SQL/CLI analog of the dynamic SQL example from the previous subsection:

```
char sqlsource [65000] ;

strcpy ( sqlsource,
         "DELETE FROM SP WHERE QTY < QTY ( 300 )" ) ;
rc = SQLExecDirect ( hstmt, (SQLCHAR *)sqlsource, SQL_NTS ) ;
```

Explanation:

1. Since real-world SQL/CLI applications tend to use C as host language, we use C instead of PL/I as the basis for this example. We also follow the SQL/CLI specification in using lowercase (or mixed uppercase and lowercase) for variable names, routine names, and the like, instead of all uppercase as elsewhere in this book (and we show such names in **boldface** in these explanatory notes in order to set them off from surrounding material). Note too that, precisely because it is a standard for invoking routines from a host language, SQL/CLI syntax—though not of course the corresponding semantics—will vary from one host language to another, in general.
2. The C function **strcpy** is invoked to copy the source form of a certain SQL DELETE statement into the C variable **sqlsource**.
3. The C assignment statement ("=") invokes the SQL/CLI routine SQLExecDirect—the analog of dynamic SQL's EXECUTE IMMEDIATE—to execute the SQL statement contained in **sqlsource**, and assigns the return code resulting from that invocation to the C variable **rc**.

As you would probably expect, SQL/CLI includes analogs of more or less everything in dynamic SQL, plus a few extra things as well. Further details are beyond the scope of this book. However, you should be aware that interfaces such as SQL/CLI, ODBC, and JDBC (which is a Java variant of ODBC, in effect) are becoming increasingly important in practice, for reasons to be discussed in Chapter 21, Section 21.6.

4.8 SQL IS NOT PERFECT

As stated in Section 4.1, SQL is very far from being the "perfect" relational language—it suffers from numerous sins of both omission and commission. Specific criticisms will be offered at appropriate points in subsequent chapters, but the overriding issue is simply that SQL fails in all too many ways to support the relational model properly. As a consequence, it is not at all clear that today's SQL products really deserve to be called

"relational" at all! Indeed, as far as this writer is aware, *there is no product on the market today that supports the relational model in its entirety.*[6] This is not to say that some parts of the model are unimportant; on the contrary, every detail of the model is important, and important, moreover, for genuinely practical reasons. Indeed, the point cannot be stressed too strongly that the purpose of relational theory is not just "theory for its own sake"; rather, the purpose is to provide a base on which to build systems that are *100 percent practical.* But the sad fact is that the vendors have not yet really stepped up to the challenge of implementing the theory in its entirety. As a consequence, the "relational" products of today regrettably all fail, in one way or another, to deliver on the full promise of relational technology.

4.9 SUMMARY

This concludes our introduction to some of the major features of the SQL standard. We have stressed the fact that SQL is important from a commercial perspective, though it is sadly deficient from a relational one.

SQL includes both a data definition language (DDL) component and a data manipulation language (DML) component. The SQL DML can operate at both the external level (on views) and the conceptual level (on base tables). Likewise, the SQL DDL can be used to define objects at the external level (views), the conceptual level (base tables), and even—in most commercial systems, though not in the standard *per se*—the internal level as well (indexes or other auxiliary structures). Moreover, SQL also provides certain *data control* facilities—that is, facilities that cannot really be classified as belonging to either the DDL or the DML. An example of such a facility is the GRANT statement, which allows users to grant *access privileges* to each other (see Chapter 17).

We showed how SQL can be used to create base tables, using the CREATE TABLE statement (we also touched on the CREATE TYPE statement in passing). We then gave some examples of the SELECT, INSERT, DELETE, and UPDATE statements, showing in particular how SELECT can be used to express the relational restrict, project, and join operations. We also briefly described the Information Schema, which consists of a set of prescribed views of a hypothetical "Definition Schema," and we took a quick look at the SQL facilities for dealing with views and transactions.

A large part of the chapter was concerned with embedded SQL. The basic idea behind embedded SQL is the dual-mode principle—that is, the principle that (insofar as possible) *any SQL statement that can be used interactively can also be used in an application program.* The major exception to this principle arises in connection with multi-row retrieval operations, which require the use of a cursor to bridge the gap between the set-level retrieval capabilities of SQL and the row-level retrieval capabilities of a host language such as PL/I.

Following a number of necessary, though mostly syntactic, preliminaries (including in particular a brief explanation of SQLSTATE), we considered those operations—singleton SELECT, INSERT, DELETE, and UPDATE—that have no need for cursors.

[6] But see reference [20.1].

Then we turned to the operations that *do* need cursors, and discussed DECLARE CURSOR, OPEN, FETCH, CLOSE, and the CURRENT forms of DELETE and UPDATE. (The standard refers to the CURRENT forms of these operators as *positioned* DELETE and UPDATE, and uses the term *searched* DELETE and UPDATE for the non-CURRENT or "out of the blue" forms.) Finally, we introduced the concept of dynamic SQL, describing the PREPARE and EXECUTE statements in particular, and we also briefly explained the SQL Call-Level Interface, SQL/CLI (we also mentioned ODBC and JDBC).

EXERCISES

4.1 Fig. 4.5 shows some sample data values for an extended form of the suppliers-and-parts database called the suppliers-parts-projects database.[7] Suppliers (S), parts (P), and projects (J) are uniquely identified by supplier number (S#), part number (P#), and project number (J#), respectively. The predicate for SPJ (shipments) is: *Supplier S# supplies part P# to project J# in quantity QTY* (the combination {S#,P#,J#} is the primary key, as the figure indicates). Write an appropriate set of SQL definitions for this database. *Note:* This database will be used as the basis for numerous exercises in subsequent chapters.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Fig. 4.5 The suppliers-parts-projects database (sample values)

[7] For ease of reference, Fig. 4.5 is repeated (along with Fig. 3.8) on the inside back cover of the book.

4.2 In Section 4.2 we described the CREATE TABLE statement as defined by the SQL standard *per se*. Many commercial SQL products support additional options on that statement, however, typically having to do with indexes, disk space allocation, and other implementation matters (thereby undermining the objectives of physical data independence and intersystem compatibility). Investigate any SQL product that might be available to you. Do the foregoing criticisms apply to that product? Specifically, what additional CREATE TABLE options does that product support?

4.3 Once again, investigate any SQL product that might be available to you. Does that product support the Information Schema? If not, what *does* its catalog support look like?

4.4 Give SQL formulations for the following updates to the suppliers-parts-projects database:

a. Insert a new supplier S10 into table S (the name and city are Smith and New York, respectively; the status is not yet known).

b. Delete all projects for which there are no shipments.

c. Change the color of all red parts to orange.

4.5 Again using the suppliers-parts-projects database, write a program with embedded SQL statements to list all suppliers in supplier number order. Each supplier should be immediately followed in the listing by all projects supplied by that supplier, in project number order.

4.6 Let tables PART and PART_STRUCTURE be defined as follows:

```
CREATE TABLE PART
     ( P# P#, DESCRIPTION CHAR(100),
       PRIMARY KEY ( P# ) ) ;

CREATE TABLE PART_STRUCTURE
     ( MAJOR_P# P#, MINOR_P# P#, QTY QTY,
       PRIMARY KEY ( MAJOR_P#, MINOR_P# ),
       FOREIGN KEY ( MAJOR_P# ) REFERENCES PART,
       FOREIGN KEY ( MINOR_P# ) REFERENCES PART ) ;
```

Table PART_STRUCTURE shows which parts (MAJOR_P#) contain which other parts (MINOR_P#) as first-level components. Write an SQL program to list all component parts of a given part, to all levels (the so-called part explosion problem). *Note:* The sample data shown in Fig. 4.6 might help you visualize this problem. We remark that table PART_STRUCTURE shows how *bill-of-materials* data—see Section 1.3, subsection "Entities and Relationships"—is typically represented in a relational system.

PART_STRUCTURE

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

Fig. 4.6 Table PART_STRUCTURE (sample value)

REFERENCES AND BIBLIOGRAPHY

Appendix H of reference [3.3] gives a detailed comparison between SQL:1999 and the proposals of *The Third Manifesto*. See also Appendix B of the present book.

4.1 M. M. Astrahan and R. A. Lorie: "SEQUEL-XRM: A Relational System." Proc. ACM Pacific Regional Conf., San Francisco, Calif. (April 1975).

Describes the first prototype implementation of SEQUEL [4.9], the earliest version of SQL. See also references [4.2] and [4.3], which perform an analogous function for System R.

4.2 M. M. Astrahan *et al.*: "System R: Relational Approach to Database Management." *ACM TODS 1*, No. 2 (June 1976).

System R was the major prototype implementation of the SEQUEL/2 (later SQL) language [4.10]. This paper describes the architecture of System R as originally planned. See also reference [4.3].

4.3 M. W. Blasgen *et al.*: "System R: An Architectural Overview." *IBM Sys. J. 20*, No. 1 (February 1981).

This paper describes the architecture of System R as it became by the time the system was fully implemented (compare and contrast reference [4.2]).

4.4 Joe Celko: *SQL for Smarties: Advanced SQL Programming*. San Francisco, Calif.: Morgan Kaufmann (1995).

"This is the first advanced SQL book available that provides a comprehensive presentation of the techniques necessary to support your progress from casual user of SQL to expert programmer" (from the book's own cover).

4.5 Surajit Chaudhuri and Gerhard Weikum: "Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System," Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt (September 2000).

This paper includes some severe criticisms of SQL. To quote: "*SQL is painful.* A big headache that comes with a database system is the SQL language. It is the union of all conceivable features (many of which are rarely used or should be discouraged [from] use anyway) and is way too complex for the typical application developer. Its core, say selection-projection-join queries and aggregation, is extremely useful, but we doubt that there is wide and wise use of all the bells and whistles. Understanding semantics of [SQL:1992, let alone SQL:1999], covering all combinations of nested (and correlated) subqueries, [nulls], triggers, ADT functions, etc. is a nightmare. Teaching SQL typically focuses on the core, and leaves the featurism as a 'learning-on-the-job' life experience. Some trade magazines occasionally pose SQL quizzes where the challenge is to express a complicated information request in a single SQL request. Those statements run over several pages, and are hardly comprehensible."

4.6 Andrew Eisenberg and Jim Melton: "SQL:1999, Formerly Known as SQL3." *ACM SIGMOD Record 28, No. 1* (March 1999).

A brief introduction to the new features that were added to the SQL standard with the publication of SQL:1999.

4.7 Andrew Eisenberg and Jim Melton: "SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings)." *ACM SIGMOD Record 27*, No. 4 (December 1998); "SQLJ—Part 1: SQL Routines Using the Java™ Programming Language," *ACM SIGMOD Record 28*, No. 4 (December 1999). See also Gray Clossman *et al.*: "Java and Relational Databases: SQLJ," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

The name *SQLJ* originally referred to a project to consider possible degrees of integration between SQL and Java (a joint effort involving some of the best-known SQL vendors). Part 0 of that project dealt with embedded SQL in Java programs; Part 1 was concerned with the idea of invoking Java from SQL (e.g., calling a stored procedure—see Chapter 21—that is written in Java); and Part 2 addressed the possibility of using Java classes as SQL data types (e.g., as a basis for defining columns in SQL tables). Part 0 was included in SQL:1999, and Parts 1 and 2 will almost certainly be included in SQL:2003 (see the annotation to reference [4.23]).

4.8 Donald D. Chamberlin: *Using the New DB2*. San Francisco, Calif.: Morgan Kaufmann (1996).

A readable and comprehensive description of a state-of-the-art commercial SQL product, by one of the principal designers of the original SQL language [4.9–4.11]. *Note:* The book also discusses "some controversial decisions" that were made in the design of SQL—primarily the decisions to support (a) nulls and (b) duplicate rows. "My [i.e., Chamberlin's] purpose . . . is historical rather than persuasive—I recognize that nulls and duplicates are religious issues . . . For the most part, the designers of [SQL] were practical people rather than theoreticians, and this orientation was reflected in many [design] decisions." This position is very different from ours! Nulls and duplicates are *scientific* issues, not religious ones; they are discussed, scientifically, in this book in Chapters 19 and 6, respectively. As for "practical . . . rather than [theoretical]," we categorically reject the suggestion that theory is not practical; we have already stated in Section 4.8 our position that relational theory, at least, is very practical indeed.

4.9 Donald D. Chamberlin and Raymond F. Boyce: "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich. (May 1974).

The paper that first introduced the SQL language (or SEQUEL, as it was originally called; the name was subsequently changed for legal reasons).

4.10 Donald D. Chamberlin *et al.*: "SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. R&D. 20*, No. 6 (November 1976). See also the errata in *IBM J. R&D. 21*, No. 1 (January 1977).

Experience from the early prototype implementation of SEQUEL discussed in reference [4.1] and results from certain usability tests led to the design of a revised version of the language called SEQUEL/2. The language supported by System R [4.2, 4.3] was basically SEQUEL/2 (with the conspicuous absence of the so-called "assertion" and "trigger" facilities—see Chapter 9), plus certain extensions suggested by early user experience [4.11].

4.11 Donald D. Chamberlin: "A Summary of User Experience with the SQL Data Sublanguage," Proc. Int. Conf. on Databases, Aberdeen, Scotland (July 1980). Also available as IBM Research Report RJ2767 (April 1980).

Discusses early user experience with System R and proposes some extensions to the SQL language in light of that experience. Certain of those extensions—EXISTS, LIKE, PREPARE, and EXECUTE—were in fact implemented in the final version of System R. They are described in Section 8.6 (EXISTS), Appendix B (LIKE), and Section 4.7 (PREPARE and EXECUTE).

4.12 Donald D. Chamberlin *et al.*: "Support for Repetitive Transactions and *Ad Hoc* Queries in System R," *ACM TODS 6*, No. 1 (March 1981).

Gives some measurements of System R performance in both the *ad hoc* query and "canned transaction" environments. (A "canned transaction" is a simple application that accesses only a small part of the database and is compiled prior to execution time. It corresponds to what we called a *planned request* in Chapter 2, Section 2.8.) The paper shows, among other things, that

in a system like System R (a) compilation is almost always superior to interpretation, even for *ad hoc* queries, and (b) as long as appropriate indexes exist in the database, many transactions can be executed per second. The paper is notable because it was one of the first to give the lie to the claim, frequently heard at the time, that "relational systems will never perform." Commercial SQL products subsequently achieved transaction rates in the hundreds and even thousands of transactions per second.

4.13 Donald D. Chamberlin *et al.*: "A History and Evaluation of System R," *CACM 24*, No. 10 (October 1981).

Describes the three principal phases of the System R project (preliminary prototype, multi-user prototype, evaluation), with emphasis on the technologies of compilation and optimization that were pioneered in System R. There is some overlap between this paper and reference [4.14].

4.14 Donald D. Chamberlin, Arthur M. Gilbert, and Robert A. Yost: "A History of System R and SQL/Data System," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

Discusses the lessons learned from the System R prototype and describes the evolution of that prototype into the first of IBM's DB2 product family, SQL/DS (subsequently renamed "DB2 for VM and VSE").

4.15 C. J. Date: "A Critique of the SQL Database Language," *ACM SIGMOD Record 14*, No. 3 (November 1984). Republished in *Relational Database: Selected Writings*, Reading, Mass.: Addison-Wesley (1986).

As noted in the body of the chapter, SQL is not perfect. This paper presents a critical analysis of a number of the language's principal shortcomings, mainly from the standpoint of formal computer languages in general rather than database languages specifically. *Note:* Certain of this paper's criticisms do not apply to SQL:1999.

4.16 C. J. Date: "What's Wrong with SQL?" in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

Discusses some additional shortcomings of SQL, over and above those identified in reference [4.15], under the headings "What's Wrong with SQL *per se*," "What's Wrong with the SQL Standard," and "Application Portability." *Note:* Again, certain of this paper's criticisms do not apply to SQL:1999.

4.17 C. J. Date: "SQL Dos and Don'ts," in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

This paper offers some practical advice on how to use SQL in such a way as (a) to avoid some of the potential pitfalls arising from the problems discussed in references [4.15], [4.16], and [4.19] and (b) to realize the maximum possible benefits in terms of productivity, portability, connectivity, and so forth.

4.18 C. J. Date: "How We Missed the Relational Boat," in *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

A succinct summary of SQL's shortcomings with respect to its support (or lack thereof) for the structural, integrity, and manipulative aspects of the relational model.

4.19 C. J. Date: "Grievous Bodily Harm" (in two parts), *DBP&D 11*, No. 5 (May 1998) and No. 6 (June 1998); "Fifty Ways to Quote Your Query," *http://www.dbpd.com* (July 1998).

SQL is an extremely redundant language, in the sense that all but the most trivial of queries can be expressed in many different ways. These papers illustrate this point and discuss some of its

implications. In particular, they show that the GROUP BY clause, the HAVING clause, and range variables could all be dropped from the language with effectively no loss of functionality (and the same is true of the "IN *<subquery>*" construct). *Note:* All of these SQL constructs are explained in Chapter 8, Section 8.6.

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

A comprehensive tutorial on the SQL standard (1992 version), including SQL/CLI (1995), SQL/PSM (1996), and a preliminary look at SQL:1999. In particular, the book contains an appendix, Appendix D, that documents "many aspects of the standard that appear to be inadequately defined, or even incorrectly defined, at this time." Most of the problems identified in that appendix still exist in SQL:1999.

4.21 C. J. Date and Colin J. White: *A Guide to DB2* (4th edition). Reading, Mass.: Addison-Wesley (1993).

Provides an extensive and thorough overview of IBM's original DB2 product (as of 1993) and some of its companion products. DB2 was based on System R, though not as closely as SQL/DS was [4.14].

4.22 Neal Fishman: "SQL *du Jour*," *DBP&D 10*, No. 10 (October 1997).

A depressing survey of some of the incompatibilities to be found among SQL products that all claim to "support the SQL standard."

4.23 International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL*, Document ISO/IEC 9075:1999. *Note:* See also reference [22.21].

The original ISO SQL:1999 definition (known to the cognoscenti as *ISO/IEC 9075*, or sometimes just *ISO 9075*). Note, however, that the original monolithic document has since been replaced by an open-ended series of separate "parts" (ISO/IEC 9075-1, -2, etc.). At the time of writing, the following parts have been defined:

Part 1: Framework (SQL/Framework)
Part 2: Foundation (SQL/Foundation)
Part 3: Call-Level Interface (SQL/CLI)
Part 4: Persistent Stored Modules (SQL/PSM)
Part 5: Host Language Bindings (SQL/Bindings)
Part 6: *There is no Part 6*
Part 7: *There is no Part 7*
Part 8: *There is no Part 8*
Part 9: Management of External Data (SQL/MED)
Part 10: Object Language Bindings (SQL/OLB)

As noted earlier in the chapter, the next edition of the standard is expected in 2003, at which time the following changes to the foregoing list are likely:

- The material from Part 5 will be folded into Part 2 and Part 5 will be dropped.
- The material from Part 2 defining the standard database catalog (the "Information Schema") will be moved to a new Part 11, "SQL Schemata."
- A new Part 13, "Java Routines and Types (SQL/JRT)," will standardize further integration of Java with SQL (see the annotation to reference [4.7]).
- A new Part 14, "XML-Related Specifications (SQL/XML)," will standardize features having to do with the relationship between SQL and XML (see Chapter 27).

By the way, it is worth mentioning that, although SQL is widely recognized as the international "relational" database standard, the standard document does not describe itself as such; in fact, it never actually uses the term *relation* at all! (As mentioned in a footnote earlier in this chapter, it does not use the term *database* either, come to that.)

4.24 International Organization for Standardization (ISO): *(ISO Working Draft)—Database Language SQL—Technical Corrigendum 5*, Document ISO/IEC JTC1/SC32/WG3 (December 2, 2001).

Contains a large number of revisions and corrections to the specifications of reference [4.23].

4.25 Raymond A. Lorie and Jean-Jacques Daudenarde: *SQL and Its Applications*. Englewood Cliffs, N.J.: Prentice-Hall (1991).

An SQL "how to" book (almost half the book consists of a detailed series of case studies involving realistic applications).

4.26 Raymond A. Lorie and J. F. Nilsson: "An Access Specification Language for a Relational Data Base System," *IBM J. R&D. 23*, No. 3 (May 1979).

Gives more details on one particular aspect of the System R compilation mechanism [4.12, 4.27]. For any given SQL statement, the System R optimizer generates a program in an internal language called ASL (Access Specification Language). ASL serves as the interface between the optimizer and the *code generator*. (The code generator, as its name implies, converts an ASL program into machine code.) ASL consists of operators such as "scan" and "insert" on objects such as indexes and stored files. The purpose of ASL is to make the overall translation process more manageable, by breaking it down into a set of well-defined subprocesses.

4.27 Raymond A. Lorie and Bradford W. Wade: "The Compilation of a High-Level Data Language," IBM Research Report RJ2598 (August 1979).

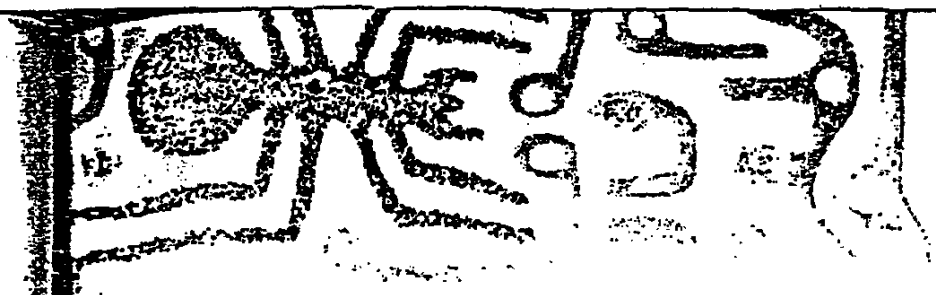
System R pioneered a scheme for compiling queries ahead of run time and then automatically recompiling them if the physical database structure had changed significantly in the interim. This paper describes the System R compilation and recompilation mechanism in some detail, without however getting into questions of optimization. See reference [18.33] for information on this latter topic.

4.28 Jim Melton and Alan R. Simon: *SQL:1999—Understanding Relational Components*. San Francisco, Calif.: Morgan Kaufmann (2002).

A tutorial on SQL:1999 (basics only—advanced topics are deferred to reference [26.32]). Melton is the editor of the SQL standard at the time of writing.

4.29 David Rozenshtein, Anatoly Abramovich, and Eugene Birger: *Optimizing Transact-SQL: Advanced Programming Techniques*. Fremont, Calif.: SQL Forum Press (1995).

Transact-SQL is the dialect of SQL supported by the Sybase and SQL Server products. This book presents a series of programming techniques for Transact-SQL based on the use of *characteristic functions* (defined by the authors as "devices that allow programmers to encode conditional logic as . . . expressions within SELECT, WHERE, GROUP BY, and SET clauses"). Although expressed in terms of Transact-SQL specifically, the ideas are actually of wider applicability. *Note:* We should perhaps add that the "optimizing" mentioned in the book's title refers not to the DBMS optimizer component but rather to "optimizations" that can be done by users themselves by hand.



PART II

THE RELATIONAL MODEL

The foundation of modern database technology is without question the relational model: it is that foundation that makes the field a science. Thus, any book on the fundamentals of database technology that does not include thorough coverage of the relational model is by definition shallow. Likewise, any claim to expertise in the database field can hardly be justified if the claimant does not understand the relational model in depth. Not that the material is at all "difficult," we hasten to add—it is not—but, to repeat, it *is* the foundation, and will remain so for as far out as anyone can see.

As explained in Chapter 3, the relational model is concerned with three principal aspects of data: data *structure*, data *manipulation*, and data *integrity*. In this part of the book, we consider each of these aspects in turn:

- Chapters 5 and 6 discuss structure (Chapter 5 deals with *types* and Chapter 6 with *relations*).
- Chapters 7 and 8 discuss manipulation (Chapter 7 deals with the *relational algebra* and Chapter 8 with the *relational calculus*).
- Chapter 9 discusses integrity.

Finally, Chapter 10 discusses the important topic of *views*. *Note:* Perhaps we should add that the division of the relational model into three parts, useful though it can be at a high conceptual level, tends to break down once we start looking more closely. As you will soon see, in fact, individual components of the model are highly interconnected and rely on one another in a variety of ways; thus, it is not possible in general (even in principle) to remove any given component without wrecking the entire model. One consequence of this fact is that Chapters 5–10 include numerous cross-references to one another.

It is also important to understand that the model is not a static thing—it has evolved and expanded over the years, and it continues to do so.[1] The text that follows reflects the current thinking of the author and other workers in this field (in particular, as mentioned in the preface, it is informed throughout by the ideas of *The Third Manifesto* [3.3]). The treatment is meant to be fairly complete, even definitive (as of the time of writing),

[1] It resembles mathematics in this respect (mathematics is not static, either, but grows over time): in fact, the relational model can be regarded itself as a small branch of mathematics.

though of course it is pedagogic in style, bnt you should not take what follows as the last word on the subject.

To say it again, the relational model is not hard to understand—but it is a theory, and most theories come equipped with their own special terminology, and (for reasons already explained in Section 3.3) the relational model is no exception in this regard. And we will be using that special terminology in this part of the book, naturally. However, it cannot be denied that the terminology can be a little bewildering at first, and indeed can serve as a barrier to understanding. (This latter fact is particularly unfortunate, given that the underlying ideas are not really difficult at all.) So, if you are having trouble in understanding some of the material that follows, please be patient: you will probably find that the concepts do become very straightforward, once you have become familiar with the terminology.

Now, it has to be said that the chapters that follow are very long (they almost form a book in their own right). But the length reflects the importance of the subject matter! It would be quite possible to provide an overview of the topic in just one or two pages; indeed, it is a major strength of the relational model that its basic ideas can be explained and understood very easily. However, a one- or two-page treatment cannot do justice to the subject, nor illustrate its wide range of applicability. The considerable length of this part of the book should thus be seen, not as a comment on the model's complexity, but as a tribute to its importance and its success as a foundation for numerous far-reaching developments. Effort invested in fully understanding the material will repay the reader many times over in his or her subsequent database activities.

Finally, a word regarding SQL. We have already said in Part I of this book that SQL is the standard "relational" database language, and just about every database product on the market supports it (or, more accurately, some dialect of it—see reference [4.22]). As a consequence, no modern database book would be complete without extensive coverage of SQL. The chapters that follow on various aspects of the relational model therefore do also discuss the relevant SQL facilities, where applicable (they build on Chapter 4, which covers basic SQL concepts).

CHAPTER 5

Types

5.1 INTRODUCTION

Note: You might want to give this chapter a "once over lightly" reading on a first pass. The chapter does logically belong here, but large parts of the material are not really needed very much prior to Chapter 20 in Part V and Chapters 25–27 in Part VI.

The data type concept (*type* for short) is fundamental; every value, every variable, every parameter, every read-only operator, and in particular every relational attribute is of some type. So what is a type? Among other things, it is a set of values. Examples include type INTEGER (the set of all integers), type CHAR (the set of all character strings), type S# (the set of all supplier numbers), and so on. Thus, when we say that, for example, the suppliers relvar S has an attribute STATUS of type INTEGER, what we mean is that values of that attribute are integers, and nothing but integers.

Note: Two points arise immediately:

- First, types are also called *domains*, especially in relational contexts; in fact, we used this latter term ourselves in earlier editions of this book, but we now prefer *types*.

- Second, a caveat: We are trying to be reasonably precise in this part of the book. Therefore, instead of saying that, for example, type INTEGER is the set of *all* integers, we ought really to say that it is the set of *all integers that are capable of representation in the computer system under consideration* (there will obviously be some integers that are beyond the representational capability of any given computer system). An analogous qualification applies to many subsequent statements and examples in this chapter, as you might expect; we will not bother to spell the point out explicitly every time, but will let this one caveat do duty for all.

Any given type is either system-defined (i.e., built in) or user-defined. We assume for the purposes of this chapter that, of the three types mentioned earlier, INTEGER and CHAR are system-defined and S# is user-defined. Any type whatsoever, regardless of whether it is system- or user-defined, can be used as the basis for declaring relational attributes (and variables, parameters, and read-only operators—see Section 5.2).

Any given type has an associated set of operators that can validly be applied to values of the type in question; that is, values of a given type can be operated upon *solely* by means of the operators defined for that type (where by "defined for that type" we mean, precisely, that the operator in question has a parameter that is declared to be of that type). For example, in the case of the system-defined type INTEGER:

- The system provides operators "=", "<", and so on, for comparing integers.
- It also provides operators "+", "*", and so on, for performing arithmetic on integers.
- It does *not* provide operators "||" (concatenate), SUBSTR (substring), and so on, for performing string operations on integers (in other words, string operations on integers are not supported).

By contrast, in the case of the user-defined type S#, we would probably define operators "=", "<", and so on, for comparing supplier numbers. However, we would probably not define operators "+", "*", and so on, which would mean that arithmetic on supplier numbers would not be supported (why would we ever want to add or multiply two supplier numbers?).

We now proceed to explore the foregoing ideas in depth, using the type theory of reference [3.3] as a basis.

5.2 VALUES *VS.* VARIABLES

The first thing we need to do is pin down the crucial, and fundamental, *logical difference*[1] between values and variables (there is a surprising amount of confusion on this issue in the literature). Following reference [5.1], we adopt the following definitions:

- A value is an "individual constant"—for example, the individual constant that is the integer 3. A value has *no location in time or space.* However, values can be represented in memory by means of some encoding, and such representations, or (our preferred term) *appearances,* do have locations in time and space. Indeed, distinct

[1] See reference [3.3] for an explanation of this useful and important concept.

appearances of the same value can exist at any number of distinct locations in time and space, meaning, loosely, that any number of different variables can have the same value, at the same time or different times. Note in particular that, by definition, a value cannot be updated; for if it could, then after such an update it would not be that value any longer.

- A variable is a holder for an appearance of a value. A variable does have a location in time and space. Also, of course, variables, unlike values, can be updated; that is, the current value of the variable in question can be replaced by another value, probably different from the previous one. (Of course, the variable in question is still the same variable after the update.)

Please note very carefully that it is not just simple things like the integer 3 that are legitimate values. On the contrary, values can be arbitrarily complex; for example, a value might be a geometric point, or a polygon, or an X ray, or an XML document, or a fingerprint, or an array, or a stack, or a list, or a relation (and on and on). Analogous remarks apply to variables too, of course.

Next, observe that it is important to distinguish between a value *per se*, on the one hand, and an appearance of that value in some particular context (in particular, as the current value of some variable), on the other. As already explained, the very same value can appear in many different contexts simultaneously. Each of those appearances consists, internally, of some *encoding* or *physical representation* of the value in question: furthermore, those encodings are not necessarily all the same. For example, the integer value 3 occurs exactly once in the set of integers (there is exactly one integer 3 "in the universe," as it were), but any number of variables might simultaneously contain an appearance of that integer as their current value. Furthermore, some of those appearances might be physically represented by means of (say) a decimal encoding, and others by means of a binary encoding, of that particular integer. Thus, there is also a logical difference between an appearance of a value, on the one hand, and the internal encoding or physical representation of that appearance, on the other.

The foregoing remarks notwithstanding, we usually find it convenient, for fairly obvious reasons, to abbreviate "encoding of an appearance of a value" to just "appearance of a value," or (more often) to just "value," as long as there is no risk of ambiguity in doing so. Note that "appearance of a value" is a *model* concept, whereas "encoding of an appearance" is an *implementation* concept. For example, users certainly might need to know whether two distinct variables contain appearances of the same value (i.e., whether they "compare equal"); however, they do not need to know whether those two appearances make use of the same physical encoding.

Values and Variables Are Typed

Every value has (equivalently, is of) some type. In other words, if v is a value, then v can be thought of as carrying around with it a kind of flag that announces "I am an integer" or "I am a supplier number" or "I am a geometric point" (etc.). Note that, by definition, any

given value always has exactly one type,[2] which never changes. It follows that distinct types are *disjoint*, meaning they have no values in common. Moreover:

- Every variable is explicitly declared to be of some type, meaning that every possible value of the variable in question is a value of the type in question.
- Every attribute of every relvar—see Chapter 6—is explicitly declared to be of some type, meaning that every possible value of the attribute in question is a value of the type in question.
- Every operator—see Section 5.5—that returns a result is explicitly declared to be of some type, meaning that every possible result that can be returned by an invocation of the operator in question is a value of the type in question.
- Every parameter of every operator—again, see Section 5.5—is explicitly declared to be of some type, meaning that every possible argument that can be substituted for the parameter in question is a value of the type in question. (Actually, this statement is not quite precise enough. Operators in general fall into two disjoint classes, read-only *vs.* update operators; read-only operators return a result, while update operators update one or more of their arguments instead. For an update operator, any argument that is subject to update is required to be a *variable*, not a *value*, of the same type as the corresponding parameter.)
- More generally, every expression is at least implicitly declared to be of some type: namely, the type declared for the outermost operator involved, where by "outermost operator" we mean the operator executed last in the evaluation of the expression in question. For example, the declared type of the expression $a * (b + c)$ is the declared type of the operator "*" (multiply).

As an aside, we observe that the foregoing remarks concerning operators and operator parameters need some slight refinement if the operators in question are polymorphic. An operator is said to be polymorphic if it is defined in terms of some parameter *P* and the arguments corresponding to *P* can be of different types on different invocations. The equality operator "=" is an obvious example: We can test *any* two values *v1* and *v2* for equality (just as long as *v1* and *v2* are of the same type), and so "=" is polymorphic—it applies to integers, and to character strings, and to supplier numbers, and in fact to values of every possible type. Analogous remarks apply to the assignment operator ":=" (which is also defined for every type): We can assign any value *v* to any variable *V*, just as long as *v* and *V* are of the same type. (Of course, the assignment will fail if it violates some integrity constraint—see Chapter 9—but it cannot fail on a type error as such.[3]) We will meet further examples of polymorphic operators in Chapter 20 and elsewhere.

[2] Except possibly if type inheritance is supported, a possibility we ignore until Chapter 20.

[3] More precisely, it cannot fail on a type error *at run time*. We are assuming here, reasonably enough, that the system does do "static" or compile-time type checking; clearly, a run-time error cannot occur if the compile-time check succeeds.

5.3 TYPES *VS.* REPRESENTATIONS

We have already touched on the fact that there is a logical difference between a type *per se*, on the one hand, and the physical representation of values of that type inside the system, on the other. In fact, types are a *model* issue, while physical representations are an *implementation* issue. For example, supplier numbers might be physically represented as character strings, but it does not follow that we can perform character string operations on supplier numbers; rather, we can perform such operations only if appropriate operators have been defined for the type. And the operators we define for a given type will naturally depend on the intended meaning of the type in question, not on the way values of that type happen to be physically represented—indeed, those physical representations should be hidden from the user. In other words, the distinction we draw between type and physical representation is one important aspect of *data independence* (see Chapter 1).

We note in passing that data types (especially user-defined ones) are sometimes called abstract data types or ADTs in the literature, to stress the foregoing point: the point, that is, that types must be distinguished from their physical representation. We do not use this term ourselves, however, because it suggests there might be some types that are not "abstract" in this sense, and we believe a distinction should *always* be drawn between a type and its physical representation.

Scalar *vs.* Nonscalar Types

Any given type is either *scalar* or *nonscalar:*

- A nonscalar type is a type whose values are explicitly defined to have a set of user-visible, directly accessible components. In particular, relation types (see Chapter 6) are nonscalar in this sense, since relations have both tuples and attributes as user-visible components. (Moreover, tuple types are nonscalar in turn, since tuples have attribute values as user-visible components.)
- A scalar type is a type that is not nonscalar (!). *Note:* The terms encapsulated and atomic are also sometimes used instead of *scalar; atomic* in particular tends to be used in relational contexts (including earlier editions of this book). Regarding *encapsulated,* see Chapter 25.

Values of type T are *scalar* or *nonscalar* according as T is scalar or nonscalar: thus, a nonscalar value has a set of user-visible components, while a scalar value does not. Analogous remarks apply to variables, attributes, operators, parameters, and expressions in general, *mutatis mutandis.*

Possible Representations, Selectors, and THE_ Operators

Let T be a scalar type. We have seen that the physical representation of values of type T is hidden from the user. In fact, such representations can be arbitrarily complex—in particular, they can certainly have components—but, to repeat, any such components will be hidden from the user. However, we do require that values of type T have at least one possible

representation[4] (declared as part of the definition of type *T*), and such possible representations are *not* hidden from the user; in particular, they have user-visible components. Understand, however, that the components in question are *not* components of the type, they are components of the possible representation—the type as such is still scalar in the sense already explained. By way of illustration, consider the user-defined type QTY ("quantity"), whose definition in Tutorial D might look like this:

```
TYPE QTY POSSREP { INTEGER } ;
```

This type definition says, in effect, that quantities can "possibly be represented" by integers. Thus, the declared possible representation ("possrep") certainly does have user-visible components—in fact, it has exactly one such, of type INTEGER—but quantities *per se* do not.

Here is another example to illustrate the same point:

```
TYPE POINT      /* geometric points in two-dimensional space */
    POSSREP CARTESIAN { X RATIONAL, Y RATIONAL }
    POSSREP POLAR { R RATIONAL, θ RATIONAL } ;
```

The definition of type POINT here includes declarations of two distinct possible representations, CARTESIAN and POLAR, reflecting the fact that points in two-dimensional space can indeed "possibly be represented" by either Cartesian or polar coordinates. Each of those possible representations in turn has two components, both of which are of type RATIONAL.[5] Note carefully, however, that (to spell it out once again) type POINT *per se* is still scalar—it has no user-visible components.

Syntax: We adopt the convention that if a given type *T* has a possible representation with no explicit name, then that possible representation is named *T* by default. We also adopt the convention that if a given possible representation *PR* has a component with no explicit name, then that component is named *PR* by default. In addition, each POSSREP declaration causes automatic definition of the following more or less self-explanatory operators:

- A selector operator, which allows the user to specify or *select* a value of the type in question by supplying a value for each component of the possible representation
- A set of THE_ operators (one such for each component of the possible representation), which allow the user to access the corresponding possible-representation components of values of the type in question

Note: When we say a POSSREP declaration causes "automatic definition" of these operators, we mean that whatever agency—possibly the system, possibly some human user—is responsible for implementing the type in question is also responsible for implementing the operators.

[4] Unless type *T* is a "dummy type" (see Chapter 20).

[5] Tutorial D uses the more accurate RATIONAL over the more familiar REAL. We remark in passing that RATIONAL might well be an example of a *built-in* type with more than one declared possible representation. For example, the expressions 530.00 and 5.3E2 might well denote the same RATIONAL value—that is, they might constitute distinct, but equivalent, invocations of two distinct RATIONAL selectors (see subsequent discussion).

Here by way of example are some sample selector and THE_ operator invocations for type POINT:

```
CARTESIAN ( 5.0, 2.5 )
/* selects the point with x = 5.0, y = 2.5 */

CARTESIAN ( X1, Y1 )
/* selects the point with x = X1, y = Y1, where */
/* X1 and Y1 are variables of type RATIONAL     */

POLAR ( 2.7, 1.0 )
/* selects the point with r = 2.7, θ = 1.0 */

THE_X ( P )
/* denotes the x coordinate of the point in */
/* P, where P is a variable of type POINT   */

THE_R ( P )
/* denotes the r coordinate of the point in P */

THE_Y ( exp )
/* denotes the y coordinate of the point denoted  */
/* by the expression exp (which is of type POINT) */
```

Note that (a) selectors have the same name as the corresponding possible representation; (b) THE_ operators have names of the form THE_*C*, where *C* is the name of the corresponding component of the corresponding possible representation. Note too that selectors—or, more precisely, selector *invocations*—are a generalization of the more familiar concept of a literal (all literals are selector invocations, but not all selector invocations are literals; in fact, a selector invocation is a literal if and only if all of its arguments are literals in turn).

To see how the foregoing might work in practice, suppose the physical representation of points is in fact Cartesian coordinates (though there is no need, in general, for a physical representation to be identical to any of the declared possible ones). Then the system will provide certain highly protected operators, denoted in what follows by *italic pseudocode*, that effectively expose that physical representation, and the *type implementer* will use those operators to implement the necessary CARTESIAN and POLAR selectors. (Obviously the type implementer is—in fact, must be—an exception to the general rule that users are not aware of physical representations.) For example:

```
OPERATOR CARTESIAN ( X RATIONAL, Y RATIONAL ) RETURNS POINT ;
   BEGIN ;
      VAR P POINT ;     /* P is a variable of type POINT */
      X component of physical representation of P  :=  X ;
      Y component of physical representation of P  :=  Y ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;

OPERATOR POLAR ( R RATIONAL, θ RATIONAL ) RETURNS POINT ;
   RETURN ( CARTESIAN ( R * COS ( θ ), R * SIN ( θ ) ) ) ;
END OPERATOR ;
```

Observe that the POLAR definition makes use of the CARTESIAN selector, as well as the (presumably built-in) operators SIN and COS. Alternatively, the POLAR definition could be expressed directly in terms of the protected operators, as follows:

```
OPERATOR POLAR ( R RATIONAL, θ RATIONAL ) RETURNS POINT ;
   BEGIN ;
      VAR P POINT ;
      X component of physical representation of P
                                          := R * COS ( θ ) ;
      Y component of physical representation of P
                                          := R * SIN ( θ ) ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;
```

The type implementer will also use those protected operators to implement the necessary THE_ operators, thus:

```
OPERATOR THE_X ( P POINT ) RETURNS RATIONAL ;
   RETURN ( X component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_Y ( P POINT ) RETURNS RATIONAL ;
   RETURN ( Y component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_R ( P POINT ) RETURNS RATIONAL ;
   RETURN ( SQRT ( THE_X ( P ) ** 2 + THE_Y ( P ) ** 2 ) ) ;
END OPERATOR ;

OPERATOR THE_θ ( P POINT ) RETURNS RATIONAL ;
   RETURN ( ARCTAN ( THE_Y ( P ) / THE_X ( P ) ) ) ;
END OPERATOR ;
```

Observe that the definitions of THE_R and THE_θ make use of THE_X and THE_Y, as well as the (presumably built-in) operators SQRT and ARCTAN. Alternatively, THE_R and THE_θ could be defined directly in terms of the protected operators (details left as an exercise).

So much for the POINT example. However, it is important to understand that all of the concepts discussed apply to simpler types as well[6]—for example, type QTY. Here are some sample selector invocations for that type:

```
QTY ( 100 )

QTY ( N )

QTY ( N1 - N2 )
```

And here are some sample THE_ operator invocations:

```
THE_QTY ( Q )

THE_QTY ( Q1 - Q2 )
```

Note: We are assuming in these examples that (a) N, N1, and N2 are variables of type INTEGER, (b) Q, Q1, and Q2 are variables of type QTY, and (c) "–" is a polymorphic operator—it applies to both integers and quantities.

Now, since values are always typed, it is strictly incorrect to say that (e.g.) the quantity for a certain shipment is 100. A quantity is a value of type QTY, not a value of type

[6] Including built-in types in particular, although (partly for historical reasons) the corresponding selectors and THE_ operators might deviate somewhat from the syntactic and other rules we have prescribed in this section. See reference [3.3] for further discussion.

INTEGER! For the shipment in question, therefore, we should more properly say the quantity is QTY(100), not simply 100 as such. In informal contexts, however, we usually do not bother to be quite as precise, thus using (e.g.) 100 as a convenient shorthand for QTY(100). Note in particular that we have used such shorthands in the suppliers-and-parts and suppliers-parts-projects databases (see Figs. 3.8 and 4.5, both repeated on the inside back cover).

We give one further example of a type definition:

```
TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;
```

Type LINESEG denotes line segments. The example illustrates the point that a given possible representation can be defined in terms of *user-defined* types, not just system-defined types as in all of our previous examples (in other words, a user-defined type is indeed a type).

Finally, note that all of our examples in this subsection on possible representations and related matters have involved scalar types specifically. However, nonscalar types have possible representations, too. We will return to this issue in Section 5.6.

5.4 TYPE DEFINITION

New types can be introduced in Tutorial D either by means of the TYPE statement already illustrated in several examples in previous sections or by means of some *type generator*. We defer discussion of type generators, and the related question of how to define nonscalar types, to Section 5.6; in this section, we discuss the TYPE statement specifically. Here by way of example is a definition for the scalar type WEIGHT:

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

Explanation: Weights can possibly be represented by decimal numbers of five digits precision with one digit after the decimal point, where the decimal number in question is greater than zero and less than 5,000. *Note:* The foregoing sentence in its entirety constitutes a type constraint for type WEIGHT. In general, a type constraint for type *T* is, precisely, a definition of the set of values that make up type *T*. If a given POSSREP declaration contains no explicit CONSTRAINT specification, then CONSTRAINT TRUE is assumed by default (in the example, omitting the CONSTRAINT specification would thus mean that valid WEIGHT values are precisely those that can be represented by decimal numbers of five digits precision with one digit after the decimal point).

The WEIGHT example raises another point, however. In Chapter 3, Section 3.9, we said part weights were given in pounds. But it might not be a good idea to bundle the type notion *per se* with the somewhat separate *units* notion (where by the term *units* we mean units of measure). Indeed, following reference [3.3], we can allow users to think of weights as being measured *either* in pounds *or* in (say) grams, by providing a distinct possible representation for each, thus:

```
TYPE WEIGHT
     POSSREP LBS ( L DECIMAL (5,1)
                   CONSTRAINT L > 0.0 AND L < 5000.0 )
     POSSREP GMS ( G DECIMAL (7,1)
                   CONSTRAINT G > 0.0 AND G < 2270000.0
                              AND MOD ( G, 45.4 ) = 0.0 ) ;
```

Note that both POSSREP declarations include a CONSTRAINT specification, and those two specifications are logically equivalent (MOD is an operator that takes two numeric operands and returns the remainder that results after dividing the first by the second; we are assuming for simplicity that one pound = 454 grams). Given this definition:

- If W is an expression of type WEIGHT, then THE_L(W) will return a DECIMAL (5,1) value denoting the corresponding weight in pounds, while THE_G(W) will return a DECIMAL(7,1) value denoting the same weight in grams.
- If N is an expression of type DECIMAL(5,1), then the expressions LBS(N) and GMS(454*N) will both return the same WEIGHT value.

Here then is the Tutorial D syntax for defining a scalar type:

```
<type def>
    ::=   TYPE <type name> <possrep def list> ;

<possrep def>
    ::=   POSSREP [ <possrep name> ]
                  { <possrep component def commalist>
                          [ <possrep constraint def> ] }

<possrep component def>
    ::=   [ <possrep component name> ] <type name>

<possrep constraint def>
    ::=   CONSTRAINT <bool exp>
```

Points arising from this syntax (most of which are illustrated by the two WEIGHT examples shown earlier):

1. The syntax makes use of both lists and commalists. The term *commalist* was defined in Chapter 4 (Section 4.6); the term *list* is defined analogously, as follows. Let <xyz> denote an arbitrary syntactic category (i.e., anything that appears on the left side of some BNF production rule). Then the expression <*xyz list*> denotes a sequence of zero or more <xyz>s in which each pair of adjacent <xyz>s is separated by one or more blanks.
2. The <*possrep def list*> must contain at least one <*possrep def*>. The <*possrep component def commalist*> must contain at least one <*possrep component def*>.
3. Brackets "[" and "]" indicate that the material they enclose is optional (as is normal with BNF notation). By contrast, braces "{" and "}" stand for themselves; that is, they are symbols in the language being defined, not (as they usually are) symbols of the metalanguage. To be specific, we use braces to enclose commalists of items when the commalist in question is intended to denote a *set* of some kind (implying among

other things that the order in which the items appear within the commalist is immaterial, and implying also that no item can appear more than once).

4. In general, a *<bool exp>* ("boolean expression") is an expression that denotes a truth value (TRUE or FALSE). In the context at hand, the *<bool exp>* must not mention any variables, but *<possrep component name>*s from the containing *<possrep def>* can be used to denote the corresponding components of the applicable possible representation of an arbitrary value of the scalar type in question. *Note:* Boolean expressions are also called *conditional, truth-valued,* or *logical* expressions.
5. Observe that *<type def>*s have absolutely nothing to say about physical representations. Rather, such representations must be specified as part of the conceptual/internal mapping (see Chapter 2, Section 2.6).
6. Defining a new type causes the system to make an entry in the catalog to describe that new type (refer to Chapter 3, Section 3.6, if you need to refresh your memory regarding the catalog). Analogous remarks apply to operator definitions also (see Section 5.5).

Here for future reference are definitions for the scalar types used in the suppliers-and-parts database (except for type WEIGHT, which has already been discussed). CONSTRAINT specifications are omitted for simplicity.

```
TYPE S#     POSSREP { CHAR } ;
TYPE NAME   POSSREP { CHAR } ;
TYPE P#     POSSREP { CHAR } ;
TYPE COLOR  POSSREP { CHAR } ;
TYPE QTY    POSSREP { INTEGER } ;
```

(Recall from Chapter 3 that the supplier STATUS attribute and the supplier and part CITY attributes are defined in terms of built-in types instead of user-defined ones, so no type definitions are shown corresponding to these attributes.)

Of course, it must also be possible to get rid of a type if we have no further use for it:

```
DROP TYPE <type name> ;
```

The *<type name>* must identify a user-defined type, not a built-in one. The operation causes the catalog entry describing the type to be deleted, meaning the type in question is no longer known to the system. For simplicity, we assume that DROP TYPE will fail if the type in question is still being used somewhere—in particular, if some attribute of some relvar somewhere is defined on it.

We close this section by pointing out that the operation of defining a type does not actually create the corresponding set of values; conceptually, those values already exist, and always will exist (think of type INTEGER, for example). Thus, all the "define type" operation—for example, the TYPE statement, in Tutorial D—really does is introduce a *name* by which that set of values can be referenced. Likewise, the DROP TYPE statement does not actually drop the corresponding values, it merely drops the name that was introduced by the corresponding TYPE statement.

5.5 OPERATORS

All of the operator definitions we have seen in this chapter so far have been either for selectors or for THE_ operators; now we take a look at operator definitions in general. Our first example shows a user-defined operator, ABS, for the built-in type RATIONAL:

```
OPERATOR ABS ( Z RATIONAL ) RETURNS RATIONAL ;
   RETURN ( CASE
               WHEN Z ≥ 0.0 THEN +Z
               WHEN Z < 0.0 THEN -Z
            END CASE ) ;
END OPERATOR ;
```

Operator ABS ("absolute value") is defined in terms of just one parameter, Z, of type RATIONAL, and returns a result of that same type. Thus, an invocation of ABS—for example, ABS (AMT1 + AMT2)—is, by definition, an expression of type RATIONAL.

The next example, DIST ("distance between"), takes two parameters of one user-defined type (POINT) and returns a result of another (LENGTH):

```
OPERATOR DIST ( P1 POINT, P2 POINT ) RETURNS LENGTH ;
   RETURN ( WITH THE_X ( P1 ) AS X1 ,
                 THE_X ( P2 ) AS X2 ,
                 THE_Y ( P1 ) AS Y1 ,
                 THE_Y ( P2 ) AS Y2 :
            LENGTH ( SQRT ( ( X1 - X2 ) ** 2
                          + ( Y1 - Y2 ) ** 2 ) ) ) ;
END OPERATOR ;
```

We are assuming that the LENGTH selector takes an argument of type RATIONAL. Also, note the use of a WITH clause to introduce names for the results of certain subexpressions. We will be making heavy use of this construct in the chapters to come.

Our next example is the required "=" (equality[7]) comparison operator for type POINT:

```
OPERATOR EQ ( P1 POINT, P2 POINT ) RETURNS BOOLEAN ;
   RETURN ( THE_X ( P1 ) = THE_X ( P2 ) AND
            THE_Y ( P1 ) = THE_Y ( P2 ) ) ;
END OPERATOR ;
```

Observe that the expression in the RETURN statement here makes use of the *built-in* "=" operator for type RATIONAL. For simplicity, we will assume from this point forward that the usual infix notation "=" can be used for the equality operator (for all types, that is, not just type POINT); we omit consideration of how such an infix notation might be specified in practice, since it is basically just a matter of syntax.

Here is the ">" operator for type QTY:

```
OPERATOR GT ( Q1 QTY, Q2 QTY ) RETURNS BOOLEAN ;
   RETURN ( THE_QTY ( Q1 ) > THE_QTY ( Q2 ) ) ;
END OPERATOR ;
```

The expression in the RETURN statement here makes use of the built-in ">" operator for type INTEGER. Again, we will assume from this point forward that the usual infix notation can be used for this operator—for all "ordinal types," that is, not just type QTY. (An

[7] Our "equality" operator might better be called *identity*, since *v1* = *v2* is true if and only if *v1* and *v2* are in fact the very same value.

ordinal type is, by definition, a type to which ">" applies. A simple example of a "nonordinal" type is POINT.)

Here finally is an example of an *update* operator definition (all previous examples have been of *read-only* operators, which are not allowed to update anything except possibly local variables).[8] As you can see, the definition involves an UPDATES specification instead of a RETURNS specification; update operators do not return a value and must be invoked by explicit CALLs [3.3].

```
OPERATOR REFLECT ( P POINT ) UPDATES P ;
   BEGIN ;
      THE_X ( P )  :=  - THE_X ( P ) ;
      THE_Y ( P )  :=  - THE_Y ( P ) ;
      RETURN ;
   END ;
END OPERATOR ;
```

The REFLECT operator effectively moves the point with Cartesian coordinates (x,y) to the inverse position $(-x,-y)$; it does this by updating its point argument appropriately. Note the use of THE_ pseudovariables in this example. A THE_ pseudovariable is an invocation of a THE_ operator in a target position (in particular, on the left side of an assignment). Such an invocation actually *designates*—rather than just returning the value of—the specified component of (the applicable possible representation of) its argument. Within the REFLECT definition, for instance, the assignment

```
THE_X ( P )  :=  ... ;
```

actually assigns a value to the X component of (the Cartesian possible representation of) the argument variable corresponding to the parameter P. Of course, any argument to be updated by an update operator—by assignment to a THE_ pseudovariable in particular—must be specified as a variable specifically, not as some more general expression.

Pseudovariables can be nested, as here:

```
VAR LS LINESEG ;
THE_X ( THE_BEGIN ( LS ) )  :=  6.5 ;
```

We now observe that THE_ pseudovariables are in fact logically unnecessary. Consider the following assignment once again:

```
THE_X ( P )  :=  - THE_X ( P ) ;
```

This assignment, which uses a pseudovariable, is logically equivalent to the following one, which does not:

```
P  :=  CARTESIAN ( - THE_X ( P ), THE_Y ( P ) ) ;
```

Similarly, the assignment

```
THE_X ( THE_BEGIN ( LS ) )  :=  6.5 ;
```

is logically equivalent to this one:

[8] Read-only and update operators are also known as *observers* and *mutators*, respectively, especially in object systems (see Chapter 25). *Function* is another synonym for *read-only operator* (and is occasionally used as such in this book).

```
LS  :=  LINESEG ( CARTESIAN ( 6.5,
                              THE_Y ( THE_BEGIN ( LS ) ) ) ,
                THE_END ( LS ) ) ;
```

In other words, pseudovariables *per se* are not strictly necessary in order to support the kind of component-level updating we are discussing here. However, the pseudovariable approach does seem intuitively more attractive than the alternative (for which it can be regarded as a shorthand); moreover, it also provides a higher degree of imperviousness to changes in the syntax of the corresponding selector. (It might also be easier to implement efficiently.)

While we are on the subject of shorthands, we should point out that the only update operator that is logically necessary is in fact assignment (":="); all other update operators can be defined in terms of assignment alone (as in fact we already know from Chapter 3, in the case of relational update operators in particular). However, we do require support for a multiple form of assignment, which allows any number of individual assignments to be performed "simultaneously" [3.3]. For example, we could replace the two assignments in the definition of the operator REFLECT by the following multiple assignment:

```
THE_X ( P )  :=  - THE_X ( P ) ,
THE_Y ( P )  :=  - THE_Y ( P ) ;
```

(note the comma separator). The semantics are as follows: First, all of the source expressions on the right sides are evaluated; second, all of the individual assignments are then executed in sequence as written.[9] *Note:* Since multiple assignment is considered to be a single operation, no integrity checking is performed "in the middle of" such an assignment; indeed, this fact is the major reason why we require multiple assignment support in the first place. See Chapters 9 and 16 for further discussion.

Finally, it must be possible to get rid of an operator if we have no further use for it. For example:

```
DROP OPERATOR REFLECT ;
```

The specified operator must be user-defined, not built in.

Type Conversions

Consider the following type definition once again:

```
TYPE S# POSSREP { CHAR } ;
```

By default, the possible representation here has the inherited name S#, and hence the corresponding selector operator does, too. The following is thus a valid selector invocation:

```
S# ('S1')
```

(it returns a certain supplier number). Note, therefore, that the S# selector might be regarded, loosely, as a **type conversion** operator that converts character strings to supplier

[9] This definition requires some refinement in the case where two or more of the individual assignments refer to the same target variable. The details are beyond the scope of this book; suffice it to say they are carefully specified to give the desired result when—as in the example, in fact—distinct individual assignments update distinct parts of the same target variable (an important special case).

numbers. Analogously, the P# selector might be regarded as a conversion operator that converts character strings to part numbers; the QTY selector might be regarded as a conversion operator that converts integers to quantities; and so on.

By the same token, THE_ operators might be regarded as operators that perform type conversion in the opposite direction. For example, recall the definition of type WEIGHT from the beginning of Section 5.4:

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

If W is of type WEIGHT, then the expression

```
THE_D ( W )
```

effectively converts the weight denoted by W to a DECIMAL(5,1) number.

Now, we said in Section 5.2 that (a) the source and target in an assignment must be of the same type, and (b) the comparands in an equality comparison must be of the same type. In some systems, however, these rules are not directly enforced; thus, it might be possible in such a system to request, for example, a comparison between a part number and a character string—in a WHERE clause, perhaps, as here:

```
... WHERE P# = 'P2'
```

Here the left comparand is of type P# and the right comparand is of type CHAR; on the face of it, therefore, the comparison should fail on a type error (a *compile-time* type error, in fact). Conceptually, however, what happens is that the system realizes that it can use the P# "conversion operator" (in other words, the P# selector) to convert the CHAR comparand to type P#, and so it effectively rewrites the comparison as follows:

```
... WHERE P# = P# ('P2')
```

The comparison is now valid.

Invoking a conversion operator implicitly in this way is known as coercion. However, it is well known that coercion can lead to program bugs. For that reason, we adopt the conservative position in this book that *coercions are not permitted*—operands must always be of the appropriate types, not merely coercible to those types. Of course, we do allow type conversion operators (or "CAST" operators, as they are usually called) to be defined and invoked explicitly when necessary—for example:

```
CAST_AS_CHAR ( 530.00 )
```

As we have already pointed out, selectors (at least, those that take just one argument) can also be thought of as explicit conversion operators, of a kind.

Now, you might have realized that what we are talking about here is what is known in programming language circles as strong typing. Different writers have slightly different definitions for this term; as we use it, however, it means, among other things, that (a) every value *has* a type, and (b) whenever we try to perform an operation, the system checks that the operands are of the right types for the operation in question. For example, consider the following expressions:

```
WEIGHT + QTY   /* e.g., part weight plus shipment quantity  */
WEIGHT * QTY   /* e.g., part weight times shipment quantity */
```

The first of these expressions makes no sense, and the system should reject it. The second, on the other hand, does make sense—it denotes the total weight for all parts involved in the shipment. So the operators we would define for weights and quantities in combination would presumably include "*" but not "+".

Here are a couple more examples, involving comparison operations this time:

```
WEIGHT > QTY
EVEN > ODD
```

(In the second example, we are assuming that EVEN is of type EVEN_INTEGER and ODD is of type ODD_INTEGER, with the obvious semantics.) Again, then, the first expression makes no sense, but the second does make sense. So the operators we would define for weights and quantities in combination presumably would not include ">", but those for even and odd integers presumably would.[10] (With respect to this question of deciding which operators are valid for which types, incidentally, we note that historically most of the database literature—the first few editions of this book included—considered comparison operators only and ignored other operators, such as "+" and "*".)

Concluding Remarks

Complete support for operators along the lines sketched in the present section has a number of significant implications, which we briefly summarize here:

- First, and most important, it means the system will know (a) exactly which expressions are valid, and (b) the type of the result for each such valid expression.
- It also means that the total collection of types for a given database will be a closed set—that is, the type of the result of every valid expression will be a type that is known to the system. Observe in particular that this closed set of types must include the type *boolean* or *truth value*, if comparisons are to be valid expressions!
- In particular, the fact that the system knows the type of the result of every valid expression means it knows which assignments are valid, and also which comparisons.

We close this section with a forward reference. We have claimed that what the relational community has historically called *domains* are really data types, system- or user-defined, of arbitrary internal complexity, whose values can be operated on solely by means of the operators defined for the type in question (and whose physical representation is therefore hidden from the user). Now, if we turn our attention for a moment to object systems, we find that the most fundamental object concept, the *object class*, is really a data type, system- or user-defined, of arbitrary internal complexity, whose values can be operated on solely by means of the operators defined for the type in question (and whose physical representation is therefore hidden from the user) . . . In other words, domains and

[10] In practice EVEN_INTEGER and ODD_INTEGER might both be subtypes of type INTEGER, in which case the ">" operator would probably be *inherited* from this latter type (see Chapter 20).

object classes are the *same thing!*—and so we have here the key to marrying the two technologies (relations and objects) together. We will elaborate on this important issue in Chapter 26.

5.6 TYPE GENERATORS

We turn now to types that are not defined by means of the TYPE statement but are obtained by invoking some type generator. Abstractly, a type generator is just a special kind of operator: it is special because it returns a type instead of, for example, a simple scalar value. In a conventional programming language, for example, we might write

```
VAR SALES ARRAY INTEGER [12] ;
```

to define a variable called SALES whose valid values are one-dimensional arrays of 12 integers. In this example, the expression ARRAY INTEGER [12] can be regarded as an invocation of the ARRAY type generator, and it returns a specific array type. That specific array type is a generated type. Points arising:

1. Type generators are known by many different names in the literature, including *type constructors*, *parameterized types*, *polymorphic types*, *type templates*, and *generic types*. We will stay with the term *type generator*.
2. Generated types are indeed types, and can be used wherever ordinary "nongenerated" types can be used. For example, we might define some relvar to have some attribute of type ARRAY INTEGER [12]. By contrast, type generators as such are *not* types.
3. Most generated types, though not all, will be *nonscalar* types specifically (array types are a case in point). As promised in Section 5.4, therefore, we have now shown how nonscalar types might be defined. *Note:* While it might be possible to define nonscalar types without directly invoking some type generator, we do not consider such a possibility any further in this book.
4. For definiteness, we regard generated types as *system-defined* types specifically, since they are obtained by invoking a system-defined type generator. *Note:* Actually we are oversimplifying slightly here. In particular, we do not rule out the possibility of users being able to define their own type generators. However, we do not consider such a possibility any further in this book.

Now, generated types clearly have possible representations ("possreps" for short) that are derived in the obvious way from (a) a *generic* possrep that applies to the type generator in question and (b) the specific possrep(s) of the user-visible component(s) of the specific generated type in question. In the case of ARRAY INTEGER [12], for example:

- There will be some generic possrep defined for one-dimensional arrays in general, probably as a contiguous sequence of *array elements* that can be identified by subscripts in the range from *lower* to *upper* (where *lower* and *upper* are the applicable bounds—1 and 12, in our example).

- The user-visible components are precisely the 12 array elements just mentioned, and they have whatever possrep(s) are defined for type INTEGER.

In like manner, there will be operators that provide the required selector and THE_ operator functionality. For example, the expression

```
ARRAY INTEGER ( 2, 5, 9, 9, 15, 27, 33, 32, 25, 19, 5, 1 )
```

—an array literal, in fact—might be used to specify a particular value of type ARRAY INTEGER [12] ("selector functionality"). Likewise, the expression

```
SALES [3]
```

might be used to access the third component (i.e., the third array element) of the array value that happens to be the current value of the array variable SALES ("THE_ operator functionality"). It might also be used as a pseudovariable.

Assignment and equality comparison operators also apply. For example, here is a valid assignment:

```
SALES  :=  ARRAY INTEGER ( 2,  5,  9,  9, 15, 27,
                           33, 32, 25, 19,  5,  1 ) ;
```

And here is a valid equality comparison:

```
SALES  =  ARRAY INTEGER ( 2,  5,  9,  9, 15, 27,
                          33, 32, 25, 19,  5,  1 )
```

Note: Any given type generator will also have a set of generic constraints and operators associated with it (generic, in the sense that the constraints and operators in question will apply to every specific type obtained via invocation of the type generator in question). For example, in the case of the ARRAY type generator:

- There might be a generic constraint to the effect that the lower bound *lower* must not be greater than the upper bound *upper*.
- There might be a generic "reverse" operator that takes an arbitrary one-dimensional array as input and returns as output another such array containing the elements of the given one in reverse order.

(In fact, the "selectors," "THE_ operators," and assignment and equality comparison operators discussed previously are also effectively derived from certain generic operators.)

Note finally that two type generators that are of particular importance in the relational world are TUPLE and RELATION. They are discussed in detail in the next chapter.

5.7 SQL FACILITIES

Built-In Types

SQL provides the following more or less self-explanatory built-in types:

```
BOOLEAN                          NUMERIC (p,q)      DATE
BIT [ VARYING ] (n)              DECIMAL (p,q)      TIME
BINARY LARGE OBJECT (n)          INTEGER            TIMESTAMP
CHARACTER [ VARYING ] (n)        SMALLINT           INTERVAL
CHARACTER LARGE OBJECT (n)       FLOAT (p)
```

A number of defaults, abbreviations, and alternative spellings—for example, CHAR for CHARACTER, CLOB for CHARACTER LARGE OBJECT, BLOB for BINARY LARGE OBJECT—are also supported; we omit the details. Points arising:

1. BIT and BIT VARYING were added in SQL:1992 and will be dropped again in SQL:2003 (!).
2. Their names notwithstanding, (a) CLOB and BLOB are really *string* types (they have nothing to do with objects in the sense of Chapter 25); (b) BLOB in particular is really a *byte* or "octet" string type (it has nothing to do with binary numbers). Also, since values of such types can be very large—they are sometimes referred to, informally, as *long strings*—SQL provides a construct called a *locator* that (among other things) allows them to be accessed piecemeal.
3. Assignment and equality comparison operators are available for all of these types. Equality comparison is essentially straightforward (but see point 5). The assignment statement looks like this:

   ```
   SET <target> = <source> ;
   ```

   Of course, assignments are also performed implicitly when database retrievals and updates are executed. However, relational assignment as such is not supported. Multiple assignment also is not supported, except as follows:[11] If row *r* is updated by means of a statement of the form

   ```
   UPDATE T SET C1 = exp1, ..., Cn = expn WHERE p ;
   ```

   (*r* here being a row in the result of *T* WHERE *p*), all of the expressions *exp1*, ..., *expn* are evaluated before any of the individual assignments to *C1*, ... *Cn* are executed.
4. Strong typing is supported, but only to a limited extent. To be specific, a certain taxonomy can be imposed on the built-in types that divides them into 10 disjoint categories, thus:

   - boolean
   - bit string
   - binary
   - character string
   - numeric
   - date
   - time
   - timestamp
   - year/month interval
   - day/time interval

   Type checking is performed on the basis of these 10 categories (on assignment and equality comparison operations in particular). Thus, for example, an attempt to compare a number and a character string is illegal; however, an attempt to compare two

[11] Two further exceptions are explained briefly in Chapter 9, Section 9.12, subsection "Base Table Constraints," and Chapter 10, Section 10.6, subsection "View Updates." These exceptions apart, we know of no product on the market today that supports multiple assignment. We do believe such support is desirable, however, and ultimately required; indeed, it is planned for inclusion in SQL:2003, though not for relations.

numbers is legal, even if those numbers are of different nnmeric types, say INTEGER and FLOAT (in this example, the INTEGER value will be coerced to type FLOAT before the comparison is done).

5. For character string types in particular—CHAR(*n*), CHAR VARYING(*n*), and CLOB(*n*)—the type checking rules are quite complex. Full details are beyond the scope of this book, but we do need to elaborate briefly on the case of fixed-length character strings (i.e., type CHAR(*n*)) in particular:
   - *Comparison*: If values of type CHAR(*n1*) and CHAR(*n2*) are compared, the shorter is conceptually padded at the right with blanks to make it the same length as the longer before the comparison is done.[12] Thus, for example, the strings 'P2' (of length two) and 'P2 ' (of length 3) are considered to "compare equal."
   - *Assignment:* If a value of type CHAR(*n1*) is assigned to a variable of type CHAR(*n2*), then, before the assignment is done, the CHAR(*n1*) value is padded at the right with blanks if *n1* < *n2*, or truncated at the right if *n1* > *n2*, to make it of length *n2*. It is an error if any nonblank characters are lost in any such truncation.

   For further explanation and discussion, see reference [4.20].

DISTINCT Types

SQL supports two kinds of user-defined types, *DISTINCT* types and *structured* types, both of which are defined by means of the CREATE TYPE statement.[13] We consider DISTINCT types in this subsection and structured types in the next (we set "DISTINCT" in uppercase to stress the point that the word is not being used in this context in its usual natural language sense). The following is an SQL definition for the DISTINCT type WEIGHT (compare and contrast the various Tutorial D definitions for this type in Section 5.4):

```
CREATE TYPE WEIGHT AS DECIMAL (5,1) FINAL ;
```

In its simplest form (i.e., ignoring a variety of optional specifications), the syntax is:

```
CREATE TYPE <type name> AS <representation> FINAL ;
```

Points arising:

1. The required FINAL specification is explained in Chapter 20.
2. The *<representation>* is the name of another type (and the type in question must not be either user-defined or generated). Note in particular that, given these rules regarding the *<representation>*, we cannot define our POINT type from Section 5.3 as an SQL DISTINCT type.
3. Note further that the *<representation>* specifies, not a possible representation as discussed earlier in this chapter, but rather the actual *physical* representation of the

[12] We are assuming here that PAD SPACE applies to such comparisons [4.20].

[13] It also supports something it calls a *domain*, but SQL's domains have nothing to do with domains in the relational sense. Reference [4.20] discusses SQL's domains in detail.

DISTINCT type in question. In fact, SQL does not support the "possrep" notion at all. One consequence of this omission is that it is not possible to define a DISTINCT type—or a structured type, come to that—with two or more distinct possreps.

4. There is nothing analogous to the Tutorial D CONSTRAINT specification. In the case of type WEIGHT, for example, there is no way to specify that for each WEIGHT value, the corresponding DECIMAL(5,1) value must be greater than zero (!) or less than 5,000.

5. The comparison operators that apply to the DISTINCT type being defined are precisely those that apply to the underlying physical representation. *Note:* Apart from assignment (see point 8), other operators that apply to the physical representation do *not* apply to the DISTINCT type. For example, none of the following expressions is valid, even if WT is of type WEIGHT:

```
WT + 14.7     WT * 2     WT + WT
```

6. "Selector" and "THE_" operators *are* supported. For example, if NW is a host variable of type DECIMAL(5,1), then the expression WEIGHT(:NW) returns the corresponding weight value; and if WT is a column of type WEIGHT, then the expression DECIMAL(WT) returns the corresponding DECIMAL(5,1) value.[14] Hence, the following are valid SQL statements:

```
DELETE
FROM   P
WHERE  WEIGHT = WEIGHT ( 14.7 ) ;

EXEC SQL DELETE
         FROM   P
         WHERE  WEIGHT = WEIGHT ( :NW ) ;

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM   P
         WHERE  WEIGHT > WEIGHT ( :NW ) ;
```

7. With one important exception (see point 8), strong typing does apply to DISTINCT types. Note in particular that comparisons between values of a DISTINCT type and values of the underlying representation type are not legal. Hence, the following are *not* valid SQL statements, even if (as before) NW is of type DECIMAL(5,1):

```
DELETE
FROM   P
WHERE  WEIGHT = 14.7 ;              /* warning -- invalid !!! */

EXEC SQL DELETE
         FROM   P
         WHERE  WEIGHT = :NW ;      /* warning -- invalid !!! */

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM   P
         WHERE  WEIGHT > :NW ;      /* warning -- invalid !!! */
```

8. The exception mentioned under point 7 has to do with assignment operations. For example, if we want to retrieve some WEIGHT value into some DECIMAL(5,1)

[14] Actually DECIMAL(WT) is not syntactically valid in SQL:1999 but is expected to become so in SQL:2003. Note, however, that (unlike Tutorial D's THE_ operators) it cannot be used as a pseudovariable.

variable, some type conversion has to occur. Now, we can certainly perform that conversion explicitly, as here:

```
SELECT DECIMAL ( WEIGHT ) AS DWT
INTO   :NW
FROM   P
WHERE  P# = P# ('P1') ;
```

However, the following is also legal (and an appropriate coercion will occur):

```
SELECT WEIGHT
INTO   :NW
FROM   P
WHERE  P# = P# ('P1') ;
```

Analogous remarks apply to INSERT and UPDATE operations.

9. Explicit CAST operators can also be defined for converting to, from, or between DISTINCT types. We omit the details here.

10. Additional operators can be defined (and subsequently dropped) as required. *Note:* The SQL term for operators is *routines*, and there are three kinds: *functions*, *procedures*, and *methods*. (Functions and procedures correspond very roughly to our read-only and update operators, respectively; methods behave like functions, but are invoked using a different syntactic style.[15]) So we could define a function—a polymorphic function, in fact—called ADDWT ("add weight") that would allow two values to be added regardless of whether they were WEIGHT values or DECIMAL(5,1) values or a mixture of the two. All of the following expressions would then be legal:

```
ADDWT ( WT, 14.7 )
ADDWT ( 14.7, WT )
ADDWT ( WT, WT )
ADDWT ( 14.7, 3.0 )
```

More information regarding SQL routines can be found in references [4.20] and [4.28]. Further details are beyond the scope of this book.

11. The following statement is used to drop a user-defined type:

```
DROP TYPE <type name> <behavior> ;
```

Here *<behavior>* is either RESTRICT or CASCADE; loosely, RESTRICT means the DROP will fail if the type is currently in use anywhere, while CASCADE means the DROP will always succeed and will cause an implicit DROP . . . CASCADE for everything currently using the type (!).

Structured Types

Now we turn to structured types. Here are a couple of examples:

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;

CREATE TYPE LINESEG AS ( BEGIN POINT, END POINT ) NOT FINAL ;
```

[15] They also, unlike functions and procedures, involve some *run-time binding* (see Chapter 20). *Note:* The term *method*, and the slightly strange meaning that must be ascribed to it in contexts like the one at hand, derive from the world of object orientation (see Chapter 25).

(Actually the second example fails because BEGIN and END are reserved words in SQL, but we choose to overlook this point.) In its simplest form, then—that is, ignoring a variety of optional specifications—the syntax for creating a structured type is:

```
CREATE TYPE <type name> AS <representation> NOT FINAL ;
```

Points arising:

1. The required NOT FINAL specification is explained in Chapter 20. *Note:* SQL:2003 is expected to allow the alternative FINAL to be specified instead.
2. The *<representation>* is an *<attribute definition commalist>* enclosed in parentheses, where an *<attribute>* consists of an *<attribute name>* followed by a *<type name>*. Note carefully, however, that those "attributes" are not attributes in the relational sense, in part because structured types are not relation types (see Chapter 6). Moreover, that *<representation>* is the actual physical representation, not just some possible representation, of the structured type in question. *Note:* The type designer can effectively conceal that fact, however—the fact, that is, that the representation is physical—by a judicious choice and design of operators. For example, given the foregoing definition of type POINT, the system will automatically provide operators to expose the Cartesian representation (see points 3 and 6), but the type designer could provide operators "manually" to expose a polar representation as well.
3. Each attribute definition causes automatic definition of two associated operators (actually "methods"), one *observer* and one *mutator*, that provide functionality analogous to that of Tutorial D's THE_ operators.[16] For example, if LS, P, and Z are of types LINESEG, POINT, and FLOAT, respectively, the following assignments are valid:

```
SET Z = P.X ;                 /* "observes" X attribute of P */
SET P.X = Z ;                 /* "mutates" X attribute of P  */
SET X = LS.BEGIN.X ;          /* "observes" X attribute of   */
                              /* BEGIN attribute of LS       */
SET LS.BEGIN.X = Z ;          /* "mutates" X attribute of    */
                              /* BEGIN attribute of LS       */
```

4. There is nothing analogous to the Tutorial D CONSTRAINT specification.
5. The comparison operators that apply to the structured type being defined are specified by means of a separate CREATE ORDERING statement. Here are two examples:

```
CREATE ORDERING FOR POINT EQUALS ONLY BY STATE ;

CREATE ORDERING FOR LINESEG EQUALS ONLY BY STATE ;
```

EQUALS ONLY means that "=" and "≠" (or "<>", rather, this latter being the SQL syntax for "not equals") are the only valid comparison operators for values of the type in question. BY STATE means that two values of the type in question are equal if and only if, for all *i*, their *i*th attributes are equal. Other possible CREATE ORDERING specifications are beyond the scope of this book; suffice it to say that, for example, the

[16] In the interest of accuracy, we should say that SQL's mutators are not really mutators in the conventional sense of the term (i.e., they are not update operators), but they can be used in such a way as to achieve conventional mutator functionality. For example, "SET P.X = Z" (which in fact does not explicitly contain a mutator invocation!) is defined to be shorthand for "SET P = P.X(Z)" (which does).

semantics of ">" can also be defined for a structured type if desired. Note, however, that if a given structured type has no associated "ordering," then no comparisons at all, not even equality comparisons, can be performed on values of that type—a state of affairs with far-reaching consequences, as you might imagine.

6. No selectors are provided automatically, but their effect can be achieved as follows. First, SQL does automatically provide what it calls *constructor functions*, but such functions return the same value on every invocation—namely, that value of the type in question whose attributes all have the applicable *default* value.[17] For example, the constructor function invocation

   ```
   POINT ()
   ```

   returns the point with default X and Y values. Now, however, we can immediately invoke the X and Y mutators (see point 3) to obtain whatever point we want from the result of that constructor function invocation. Moreover, we can bundle the initial "construction" and the subsequent "mutations" into a single expression, as illustrated by the following example:

   ```
   POINT () . X ( 5.0 ) . Y ( 2.5 )
   ```

   Here is a more complex example:

   ```
   LINESEG () . BEGIN ( POINT () . X ( 5.0 ) . Y ( 2.5 ) )
              . END   ( POINT () . X ( 7.3 ) . Y ( 0.8 ) )
   ```

   Note: Constructor function invocations can optionally be preceded by the noiseword NEW without changing the semantics. For example:

   ```
   NEW LINESEG () . BEGIN ( NEW POINT () . X ( 5.0 ) . Y ( 2.5 ) )
                  . END   ( NEW POINT () . X ( 7.3 ) . Y ( 0.8 ) )
   ```

7. Strong typing does apply to structured types, except possibly as described in Chapter 6, Section 6.6 (subsection "Structured Types").
8. Operators in addition to those already mentioned can be defined (and subsequently dropped) as required.
9. Structured types and orderings can be dropped. Such types can be "altered," too, via an ALTER TYPE statement—for example, new attributes can be added or existing ones dropped (in other words, the representation can be changed).

We will have more to say regarding SQL's structured types in the next chapter (Section 6.6) and in Chapters 20 and 26.

[17] The default value for a given attribute can be specified as part of the corresponding attribute definition. If no such value is specified explicitly, the default value—the "default default"—will be null. *Note:* For reasons beyond the scope of this book, the default *must* be null if the type of the attribute is either a row type or a user-defined type (like POINT), and it must be either null or empty—specified as ARRAY[]—if it is an array type. Thus, for example, the constructor function invocation LINESEG() will necessarily return the line segment whose BEGIN and END components are both null.

Type Generators

SQL supports three type generators (the SQL term is *type constructors*): REF, ROW, and ARRAY.[18] In this chapter we discuss ROW and ARRAY only, deferring REF to Chapter 26. Here is an example illustrating the use of ROW:

```
CREATE TABLE CUST
     ( CUST# CHAR(3),
       ADDR  ROW ( STREET CHAR(50),
                   CITY   CHAR(25),
                   STATE  CHAR(2),
                   ZIP    CHAR(5) )
PRIMARY KEY ( CUST# ) ) ;
```

STREET, CITY, STATE, and ZIP here are the *fields* of the generated row type. In general, such fields can be of any type, including other row types. Field-level references make use of dot qualification, as in the following example (the syntax is *<exp>.<field name>*, where the *<exp>* must be row-valued):

```
SELECT CX.CUST#
FROM   CUST AS CX
WHERE  CX.ADDR.STATE = 'CA' ;
```

Note: CX here is a *correlation name*. Correlation names are discussed in detail in Chapter 8 (Section 8.6); here we simply note that SQL requires explicit correlation names to be used in field references, in order to avoid a certain syntactic ambiguity that might otherwise occur.

Here now is an INSERT example:

```
INSERT INTO CUST ( CUST#, ADDR )
VALUES ( '666', ROW ( '1600 Pennsylvania Ave.',
                      'Washington', 'DC', '20500' ) ) ;
```

Note the row literal in this example (actually, that should be "row literal," in quotes—formally, there is no such thing as a row literal in SQL, and the expression in the example is a *row value constructor*).

One more example:

```
UPDATE CUST AS CX
SET    CX.ADDR.STATE = 'TX'
WHERE  CUST# = '999' ;
```

Note: In fact the standard does not currently permit field-level updating as in this example, but the omission looks like an oversight.

The ARRAY type generator is somewhat similar. Here is an example:

```
CREATE TABLE ITEM_SALES
     ( ITEM# CHAR(5),
       SALES INTEGER ARRAY [12],
       PRIMARY KEY ( ITEM# ) ) ;
```

[18] SQL:2003 is likely to add MULTISET.

Types generated by means of ARRAY are always one-dimensional; the specified element type (INTEGER in the example) can be anything except another array type.[19] Let *a* be a value of some array type. Then *a* can contain any number *n* of elements ($n \geq 0$), up to but not greater than the specified upper bound (12 in the example). If *a* contains exactly *n* elements ($n > 0$), then those elements are precisely—and can be referenced as—*a*[1], *a*[2], ..., *a*[*n*]. The expression CARDINALITY(*a*) returns the value *n*.

Here now are some examples that use the ITEM_SALES table. Note the array literal (or "array literal," rather—officially, it is an *array value constructor*) in the second example.

```
SELECT ITEM#
FROM   ITEM_SALES
WHERE  SALES [3] > 10 ;

INSERT INTO ITEM_SALES ( ITEM#, SALES )
VALUES ( 'X4320',
         ARRAY [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ;

UPDATE ITEM_SALES
SET    SALES [3] = 10
WHERE  ITEM# = 'Z0564' ;
```

We close this section by noting that assignment and equality comparison operators do apply, for both ROW and ARRAY types—unless the ROW or ARRAY type in question involves an element type for which equality comparision is not defined, in which case it is not defined for the ROW or ARRAY type in question either.

5.8 SUMMARY

In this chapter we have taken a comprehensive look at the crucial notion of data types (also known as domains or simply types). A type is a set of values: namely, the set of all values that satisfy a certain type constraint (specified in Tutorial D by a POSSREP clause, including an optional CONSTRAINT specification). Every type has an associated set of operators (both *read-only* and *update* operators) for operating on values and variables of the type in question. Types can be as simple or as complex as we like; thus, we can have types whose values are numbers, or strings, or dates, or times, or audio recordings, or maps, or video recordings, or geometric points (etc.). Types constrain operations, in that the operands to any given operation are required to be of the types defined for that operation (strong typing). Strong typing is a good idea because it allows certain logical errors to be caught, and caught moreover at compile time instead of run time. Note that strong typing has important implications for the relational operations in particular (join, union, etc.), as we will see in Chapter 7.

[19] This restriction is likely to be removed in SQL:2003. In any case, the element type can be a row type, and that row type can include a field of some array type. Thus (e.g.) the following is a legal variable definition:

```
VX ROW (FX INTEGER ARRAY [12]) ARRAY [12]
```

And then (e.g.) VX[3].FX[5] refers to the fifth element of the array that is the sole field value within the row that is the third element of the array that is the value of the variable VX.

We also discussed the important *logical difference* between values and variables, and pointed out that the essential property of a value is that *it cannot be updated*. Values and variables are always typed; so also are (relational) *attributes*, (read-only) *operators*, *parameters*, and more generally *expressions* of arbitrary complexity.

Types can be system- or user-defined; they can also be scalar or nonscalar. A scalar type has no user-visible components. (The most important *non*scalar types in the relational model are relation types, which are discussed in the next chapter.) We distinguish carefully between a type and its physical representation (types are a *model* issue, physical representations are an *implementation* issue). However, we do require that every type have at least one declared possible representation (possibly more than one). Each such possible representation causes automatic definition of one selector operator and, for each component of that possible representation, one THE_ operator (including a THE_ pseudovariable). We support explicit type conversions but no implicit type coercions. We also support the definition of any number of additional operators for scalar types, and we require that equality comparison and (multiple) assignment be defined for every type.

We also discussed type generators, which are operators that return types (ARRAY is an example). The constraints and operators that apply to generated types are derived from the *generic* constraints and operators that are associated with the applicable type generator.

Finally, we sketched SQL's type facilities. SQL provides a variety of built-in types—BOOLEAN, INTEGER, DATE, TIME, and so on (each with its associated set of operators, of course)—but supports only a limited form of strong typing in connection with those types. It also allows users to define their own types, which it divides into DISTINCT types and structured types, and it supports certain type generators (ARRAY and ROW, also REF). We offered an analysis of all of this SQL functionality in terms of the ideas presented earlier in the chapter.

EXERCISES

5.1 State the type rules for the assignment (":=") and equality comparison ("=") operators.

5.2 Distinguish:

value	*vs.*	variable
type	*vs.*	representation
physical representation	*vs.*	possible representation
scalar	*vs.*	nonscalar
read-only operator	*vs.*	update operator

5.3 Explain the following in your own words:

coercion	pseudovariable
generated type	selector
literal	strong typing
ordinal type	THE_ operator
polymorphic operator	type generator

5.4 Why are pseudovariables logically unnecessary?

5.5 Define an operator that, given a rational number, returns the cube of that number.

5.6 Define a read-only operator that, given a point with Cartesian coordinates x and y, returns the point with Cartesian coordinates $f(x)$ and $g(y)$, where f and g are predefined operators.

5.7 Repeat Exercise 5.6 but make the operator an update operator.

5.8 Give a type definition for a scalar type called CIRCLE. What selectors and THE_ operators apply to this type? Also:

a. Define a set of read-only operators to compute the diameter, circumference, and area of a given circle.

b. Define an update operator to double the radius of a given circle (more precisely, to update a given CIRCLE variable in such a way that its circle value is unchanged except that the radius is twice what it was before).

5.9 Give some examples of types for which it might be useful to define two or more distinct possible representations. Can you think of an example where distinct possible representations for the same type have different numbers of components?

5.10 Given the catalog for the departments-and-employees database shown in outline in Fig. 3.6 in Chapter 3, how could that catalog be extended to take account of user-defined types and operators?

5.11 What types are the catalog relvars themselves defined on?

5.12 Give an appropriate set of scalar type definitions for the suppliers-parts-projects database (see Fig. 4.5 on the inside back cover). Do not attempt to write the relvar definitions.

5.13 We pointed out in Section 5.3 that it is strictly incorrect to say that (e.g.) the quantity for a certain shipment is 100 ("a quantity is a value of type QTY, not a value of type INTEGER"). As a consequence, Fig. 4.5 is rather sloppy, inasmuch as it pretends that it *is* correct to think of, for example, quantities as integers. Given your answer to Exercise 5.12, show the correct way of referring to the various scalar values in Fig. 4.5.

5.14 Given your answer to Exercise 5.12, which of the following scalar expressions are legal? For the legal ones, state the type of the result; for the others, show a legal expression that will achieve what appears to be the desired effect.

a. `J.CITY = P.CITY`

b. `JNAME || PNAME`

c. `QTY * 100`

d. `QTY + 100`

e. `STATUS + 5`

f. `J.CITY < S.CITY`

g. `COLOR = P.CITY`

h. `J.CITY = P.CITY || 'burg'`

5.15 It is sometimes suggested that types are really variables too, like relvars. For example, legal employee numbers might grow from three digits to four as a business expands, so we might need to update "the set of all possible employee numbers." Discuss.

5.16 Give SQL analogs of all type definitions from Sections 5.3 and 5.4.

5.17 Give an SQL answer to Exercise 5.12.

5.18 In SQL:

a. What is a DISTINCT type? What are values of a DISTINCT type called generically? Is there such a thing as an indistinct type?

b. What is a structured type? What are values of a structured type called generically? Is there such a thing as an unstructured type?

5.19 Explain the terms *observer*, *mutator*, and *constructor function* as used in SQL.

5.20 What are the consequences of the "=" operator not being defined for some given type?

5.21 A type is a set of values, so we might define the *empty* type to be the (necessarily unique) type where the set in question is empty. Can you think of any uses for such a type?

5.22 "SQL has no formal row or array literals." Explain and justify this observation.

5.23 Consider the SQL type POINT as defined in the subsection "Structured Types" in Section 5.7. That type has a representation involving Cartesian coordinates X and Y. What happens if we replace that type by a revised type POINT with a representation involving polar coordinates R and θ instead?

5.24 What is the difference between the SQL COUNT and CARDINALITY operators? *Note:* COUNT is discussed in Chapter 8, Section 8.6.

REFERENCES AND BIBLIOGRAPHY

5.1 J. Craig Cleaveland: *An Introduction to Data Types*. Reading, Mass.: Addison-Wesley (1986).

CHAPTER 6

Relations

6.1 INTRODUCTION

In the previous chapter, we discussed types, values, and variables in general; now we turn our attention to relation types, values, and variables in particular. And since relations are built out of tuples (speaking a trifle loosely), we need to examine tuple types, values, and variables as well. Note immediately, however, that tuples are not all that important in themselves, at least from a relational perspective; their significance lies primarily in the fact that they form a necessary stepping-stone on the way to relations.

6.2 TUPLES

We begin by defining the term *tuple* precisely. Given a collection of types TI (i = 1, 2, ..., n), not necessarily all distinct, a tuple value (tuple for short) on those types—t, say—is a set of ordered triples of the form $<Ai,TI,vi>$, where Ai is an attribute name, TI is a type name, and vi is a value of type TI, and:

- The value n is the degree or arity of t.
- The ordered triple $<Ai,TI,vi>$ is a component of t.

- The ordered pair <*Ai*,*Ti*> is an **attribute** of *t*, and it is uniquely identified by the attribute name *Ai* (attribute names *Ai* and *Aj* are the same only if $i = j$). The value *vi* is the **attribute value** for attribute *Ai* of *t*.[1] The type *Ti* is the corresponding **attribute type**.
- The complete set of attributes is the **heading** of *t*.
- The tuple type of *t* is determined by the heading of *t*, and the heading and that tuple type both have the same attributes (and hence the same attribnte names and types) and the same degree as *t* does. The tuple type name is precisely:

  ```
  TUPLE { A1 T1, A2 T2, ..., An Tn }
  ```

  Here is a sample tuple:

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P2	P4	7

The attribute names here are MAJOR_P#, MINOR_P#, and QTY; the corresponding type names are P#, P# again, and QTY; and the corresponding values are P#('P2'), P#('P4'), and QTY(7) (for simplicity, these values have been abbreviated to just P2, P4, and 7, respectively, in the picture). The degree of this tuple is three. Its heading is:

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY

And its type is:

```
TUPLE { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

Note: It is common in informal contexts to omit the type names from a tuple heading, showing just the attribute names. Informally, therefore, we might represent the foregoing tuple thus:

MAJOR_P#	MINOR_P#	QTY
P2	P4	7

In Tutorial D, the following expression could be used to denote the tuple we have been discussing:

```
TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P4'), QTY QTY(7) }
```

(an example of a tuple selector invocation—see the next subsection but one). Observe in particular in this expression that the types of the tuple attributes are determined unambiguously by the specified attribute values (e.g., attribute MINOR_P# is of type P# because the corresponding attribute value is of type P#).

[1] There is, of course, a logical difference between an attribute name and an attribute *per se*. This fact notwithstanding, we often use expressions such as "attribute *Ai*," informally, to mean the attribute whose *name* is *Ai* (indeed, we did exactly this several times in the previous chapter).

Properties of Tuples

Tuples satisfy a variety of important properties, all of them immediate consequences of the definitions given in this section so far. To be specific:

- Every tuple contains exactly one value (of the appropriate type) for each of its attributes.
- There is no left-to-right ordering to the components of a tuple. This property follows because a tuple is defined to involve a *set* of components, and sets in mathematics have no ordering to their elements.
- Every subset of a tuple is a tuple (and every subset of a heading is a heading). What is more, these remarks are true of the empty subset in particular!—see the next paragraph.

More terminology: A tuple of degree one is said to be *unary*, a tuple of degree two *binary*, a tuple of degree three *ternary* (and so on); more generally, a tuple of degree *n* is said to be *n-ary*.[2] A tuple of degree zero (i.e., a tuple with no components) is said to be *nullary*. We elaborate briefly on this last possibility. Here is a nullary tuple in Tutorial D notation:

```
TUPLE { }
```

Sometimes we refer to a tuple of degree zero more explicitly as a "0-tuple," in order to emphasize the fact that it has no components. Now, it might seem that a 0-tuple is not likely to be very useful in practice; in fact, however, it turns out that the concept is crucially important. We will have more to say about it in Section 6.4.

The TUPLE Type Generator

Tutorial D provides a TUPLE type generator that can be invoked in the definition of (e.g.) some relvar attribute or some tuple variable.[3] Here is an example of the latter case:

```
VAR ADDR TUPLE { STREET CHAR,
                 CITY   CHAR,
                 STATE  CHAR,
                 ZIP    CHAR } ;
```

A TUPLE type generator invocation takes the general form

```
TUPLE { <attribute commalist> }
```

(where each *<attribute>* consists of an *<attribute name>* followed by a *<type name>*). The tuple type produced by a specific invocation of the TUPLE type generator—for example, the one just shown in the definition of variable ADDR—is, of course, a generated type.

Every tuple type has an associated *tuple selector* operator. Here is an example of a selector invocation for the tuple type shown in the definition of variable ADDR:

[2] The term *n-tuple* is sometimes used in place of *tuple* (and so we speak of, e.g., 4-tuples, 2-tuples, and so on). However, it is usual to drop the "*n*-" prefix.

[3] Tuple variables are not part of the relational model, and they are not permitted within a relational database. But a system that supports the relational model fully will probably support tuple variables within individual applications (i.e., tuple variables that are "application-local").

```
TUPLE { STREET '1600 Pennsylvania Ave.',
        CITY 'Washington', STATE 'DC', ZIP '20500' }
```

The tuple denoted by this expression could be assigned to the tuple variable ADDR, or tested for equality with another tuple of the same type. Note in particular that, in order for two tuples to be of the same type, it is necessary and sufficient that they have the same attributes. Note too that the attributes of a given tuple type can be of any type whatsoever (they can even be of some relation type or some other tuple type).

Operators on Tuples

We have already mentioned the tuple selector, assignment, and equality comparison operators briefly. However, it is worth spelling out the semantics of tuple equality in detail, since so much in later chapters depends on it. To be specific, all of the following are defined in terms of it:

- Essentially all of the operators of the relational algebra (see Chapter 7)
- Candidate keys (see Chapter 9)
- Foreign keys (again, see Chapter 9)
- Functional and other dependencies (see Chapters 11–13)

and more besides. Here then is a precise definition:

- Tuple equality: Tuples *t1* and *t2* are equal (i.e., *t1* = *t2* is true) if and only if they have the same attributes *A1*, *A2*, ..., *An* and, for all *i* (*i* = 1, 2, ..., *n*), the value *v1* of *Ai* in *t1* is equal to the value *v2* of *Ai* in *t2*.
- Furthermore—this might seem obvious but it needs to be said—tuples *t1* and *t2* are duplicates of each other if and only if they are equal (meaning they are in fact the very same tuple).

Note that it is an immediate consequence of the foregoing definition that all 0-tuples are duplicates of one another! For this reason, we are justified in talking in terms of *the* 0-tuple instead of "a" 0-tuple, and indeed we usually do.

Note too that the comparison operators "<" and ">" do *not* apply to tuples (i.e., tuple types are not "ordinal types").

In addition to the foregoing, reference [3.3] proposes analogs of certain of the well-known relational operators (to be discussed in Chapter 7)—tuple project, tuple join, and so on. These operators are mostly self-explanatory; we content ourselves here with just one example, a tuple project (that operator being probably the most useful in practice). Let variable ADDR be as defined in the previous subsection, and let its current value be as follows:

```
TUPLE { STREET '1600 Pennsylvania Ave.',
        CITY 'Washington', STATE 'DC', ZIP '20500' }
```

Then the tuple projection

```
ADDR { CITY, ZIP }
```

denotes the tuple

```
TUPLE { CITY 'Washington', ZIP '20500' }
```

We also need to be able to extract a given attribute value from a given tuple. By way of example, if ADDR is as before, then the expression

```
ZIP FROM ADDR
```

denotes the value

```
'20500'
```

Tuple type inference: One important advantage of the tuple type naming scheme, as defined near the start of this section, is that it facilitates the task of determining the type of the result of an arbitrary tuple expression. For example, consider this tuple projection again:

```
ADDR { CITY, ZIP }
```

As we have seen, this expression evaluates to a tuple that is derived from the current value of ADDR by "projecting away" attributes STREET and STATE. And the tuple type of that derived tuple is precisely:

```
TUPLE { CITY CHAR, ZIP CHAR }
```

Analogous remarks apply to all possible tuple expressions.

WRAP and UNWRAP: Consider the following tuple types:

```
TUPLE { NAME NAME, ADDR TUPLE { STREET CHAR, CITY CHAR,
                                STATE  CHAR, ZIP  CHAR } }

TUPLE { NAME NAME, STREET CHAR, CITY CHAR,
                   STATE  CHAR, ZIP  CHAR }
```

Let us refer to these tuple types as *TT1* and *TT2*, respectively. Observe in particular that type *TT1* includes an attribute that is itself of some tuple type (the degree of *TT1* is two, not five). Now let NADDR1 and NADDR2 be tuple variables of types *TT1* and *TT2*, respectively. Then:

- The expression

  ```
  NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR
  ```

  takes the current value of NADDR2 and *wraps* the STREET, CITY, STATE, and ZIP components of that value to yield a single tuple-valued ADDR component. The result of the expression is thus of type *TT1*, and so (e.g.) the following assignment is valid:

  ```
  NADDR1  :=  NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR ;
  ```

- The expression

  ```
  NADDR1 UNWRAP ADDR
  ```

  takes the current value of NADDR1 and *unwraps* the (tuple-valued) ADDR component of that value to yield four separate components STREET, CITY, STATE, and ZIP. The result of the expression is thus of type *TT2*, and so (e.g.) the following assignment is valid:

  ```
  NADDR2  :=  NADDR1 UNWRAP ADDR ;
  ```

Tuple Types *vs.* Possible Representations

You might have noticed a certain similarity between our *TUPLE type generator* syntax as described in this section and our *declared possible representation* syntax as described in Chapter 5—both involve a commalist of items, where each item specifies the name of something and a corresponding type name—and you might be wondering whether there are two concepts here or only one. In fact there are two (and the syntactic similarity is unimportant). For example, if *X* is a tuple type, then we might very well want to take a projection of some value of that type, as described in the previous subsection. However, if *X* is a possible representation for some scalar type *T*, then there is no question of wanting to take a projection of a value of that scalar type *T*. For further discussion, see reference [3.3].

6.3 RELATION TYPES

Now we turn to relations. Our treatment parallels that for tuples in the previous section; however, we have rather more to say regarding relations than we had for tuples, and we have therefore split the material over several sections—Section 6.3 discusses relation types, Section 6.4 relation values, and Section 6.5 relation variables (relvars).

Here first is a precise definition of the term *relation*. A relation value (relation for short), *r* say, consists of a *heading* and a *body*,[4] where:

- The heading of *r* is a tuple heading as defined in Section 6.2. Relation *r* has the same attributes (and hence the same attribute names and types) and the same degree as that heading does.
- The body of *r* is a set of tuples, all having that same heading; the cardinality of that set is said to be the cardinality of *r*. (In general, the *cardinality* of a set is the number of elements in the set.)

The relation type of *r* is determined by the heading of *r*, and it has the same attributes (and hence attribute names and types) and degree as that heading does. The relation type name is precisely:

```
RELATION { A1 T1, A2 T2, ..., An Tn }
```

Here is a sample relation (it is similar but not identical to the relation shown in Fig. 4.6 in Chapter 4):

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P1	P2	5
P1	P3	3
P2	P3	2
P2	P4	7
P3	P5	4
P4	P6	8

[4] In terms of the usual tabular picture of a relation, the heading corresponds to the row of column names and the body to the set of data rows. The heading is also referred to in the literature as a (relation) schema, or sometimes scheme. It is also referred to as the intension (note the spelling), in which case the body is referred to as the extension.

The type of this relation is:

```
RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

It is common in informal contexts to omit the type names from a relation heading, showing just the attribute names. Informally, therefore, we might represent the foregoing relation thus:

MAJOR_P#	MINOR_P#	QTY
P1	P2	5
P1	P3	3
P2	P3	2
P2	P4	7
P3	P5	4
P4	P6	8

In Tutorial D, the following expression could be used to denote this relation:

```
RELATION {
  TUPLE { MAJOR_P# P#('P1'), MINOR_P# P#('P2'), QTY QTY(5) } ,
  TUPLE { MAJOR_P# P#('P1'), MINOR_P# P#('P3'), QTY QTY(3) } ,
  TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P3'), QTY QTY(2) } ,
  TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P4'), QTY QTY(7) } ,
  TUPLE { MAJOR_P# P#('P3'), MINOR_P# P#('P5'), QTY QTY(4) } ,
  TUPLE { MAJOR_P# P#('P4'), MINOR_P# P#('P6'), QTY QTY(8) } }
```

This expression is an example of a relation selector invocation. The general format is:

```
RELATION [ <heading> ] { <tuple exp commalist> }
```

(where the optional *<heading>*, which is a commalist of *<attribute>*s enclosed in braces, is required only if the *<tuple exp commalist>* is empty). Of course, all of the *<tuple exp>*s must be of the same tuple type, and that tuple type must be exactly the one determined by the *<heading>* if a *<heading>* is specified.

Note that, strictly speaking, a relation does not contain tuples—it contains a body, and that body in turn contains tuples. Informally, however, it is convenient to talk as if relations contained tuples directly, and we will follow this simplifying convention throughout this book.

As with tuples, a relation of degree one is said to be *unary*, a relation of degree two *binary*, a relation of degree three *ternary* (and so on); more generally, a relation of degree *n* is said to be *n-ary*. A relation of degree zero—that is, a relation with no attributes—is said to be *nullary* (we will discuss this last possibility in detail in the next section). Also, observe that:

- Every subset of a heading is a heading (as with tuples).
- Every subset of a body is a body.

In both cases, the subset in question might be the empty subset in particular.

The RELATION Type Generator

Tutorial D provides a RELATION type generator that can be invoked in (e.g.) the definition of some relvar. Here is an example:

```
VAR PART_STRUCTURE ...
    RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY } ... ;
```

(We have omitted irrelevant portions of this definition for simplicity.) In general, the RELATION type generator takes the same form as the TUPLE type generator, except for the appearance of RELATION in place of TUPLE. The relation type produced by a specific invocation of the RELATION type generator—for example, the one just shown in the definition of relvar PART_STRUCTURE—is, of course, a generated type.

Every relation type has an associated *relation selector* operator. We have already seen an example of a selector invocation for the relation type just illustrated. The relation denoted by that selector invocation could be assigned to the relvar PART_STRUCTURE, or tested for equality with another relation of the same type. Note in particular that, in order for two relations to be of the same type, it is necessary and sufficient that they have the same attributes. Note too that the attributes of a given relation type can be of any type whatsoever (they can even be of some tuple type or some other relation type).

6.4 RELATION VALUES

Now we can begin to take a more detailed look at relations as such (relation *values*, that is). The first point to note is that relations satisfy certain properties, all of them immediate consequences of the definition of *relation* given in the previous section, and all of them very important. We first state the properties in question, then discuss them in detail. They are as follows. Within any given relation:

1. Every tuple contains exactly one value (of the appropriate type) for each attribute.
2. There is no left-to-right ordering to the attributes.
3. There is no top-to-bottom ordering to the tuples.
4. There are no duplicate tuples.

We use the suppliers relation from Fig. 3.8 (see the inside back cover) to illustrate these properties. For convenience we show that relation again in Fig. 6.1, except that now we have expanded the heading to include the type names. *Note:* By rights we should have expanded the body, too, to include the attribute and type names. For example, the S# entry for supplier S1 should really look something like this:

```
S# S# S#('S1')
```

S# : S#	SNAME : NAME	STATUS : INTEGER	CITY : CHAR
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 6.1 The suppliers relation from Fig. 3.8

For simplicity, however, we have left the body as originally shown in Fig. 3.8.

1. *Relations Are Normalized*

As we know from Section 6.2, every tuple contains exactly one value for each of its attributes; thus, it certainly follows that every tuple in every relation contains exactly one value for each of its attributes. A relation that satisfies this property is said to be normalized, or equivalently to be in first normal form, 1NF.[5] The relation of Fig. 6.1 is normalized in this sense.

Note: This first property might seem very obvious, and indeed so it is—especially since, as you must have realized, *all* relations are normalized according to the definition! Nevertheless, the property does have some important consequences. See the subsection "Relation-Valued Attributes" later in this section, also Chapter 19 (on missing information).

2. *Attributes Are Unordered, Left to Right*

We already know that the components of a tuple have no left-to-right ordering, and the same is true for the attributes of a relation (for essentially the same reason—namely, that the heading of a relation involves a *set* of attributes, and sets in mathematics have no ordering to their elements). Now, when we represent a relation as a table on paper, we are naturally forced to show the columns of that table in some left-to-right order, but you should ignore that order if you can. In Fig. 6.1, for example, the columns could just as well have been shown in (say) the left-to-right order SNAME, CITY, STATUS, S#—the figure would still have represented the same relation, at least as far as the relational model is concerned.[6] Thus, there is no such thing as "the first attribute" or "the second attribute" (etc.), and there is no "next attribute" (i.e., there is no concept of "nextness"); attributes are always referenced by name, never by position. As a result, the scope for errors and obscure programming is reduced. For example, there is no way to subvert the system by somehow "flopping over" from one attribute into another. This situation contrasts with that found in many programming systems, where it often is possible to exploit the physical adjacency of logically discrete items, deliberately or otherwise, in a variety of subversive ways.

3. *Tuples Are Unordered, Top to Bottom*

This property follows from the fact that the body of the relation is also a set (of tuples); to say it again, sets in mathematics are not ordered. When we represent a relation as a table on paper, we are forced to show the rows of that table in some top-to-bottom order, but again you should ignore that order if you can. In Fig. 6.1, for example, the rows could just as well have been shown in reverse order—the figure would still have represented the same relation. Thus, there is no such thing as "the first tuple" or "the fifth tuple" or "the 97th tuple" of a relation, and there is no such thing as "the next tuple"; in other words,

[5] So called because certain "higher" normal forms—second, third, and so on—can also be defined (see Chapters 12 and 13).

[6] For reasons that need not concern us here, relations in mathematics, unlike their counterparts in the relational model, do have a left-to-right order to their attributes (and likewise for tuples, of course).

there is no concept of positional addressing, and there is no concept of "nextness." Note that if we did have such concepts, we would need certain additional operators as well—for example, "retrieve the *n*th tuple," "insert this new tuple *here*," "move this tuple from *here* to *there*," and so on. It is a very strong feature of the relational model (and a direct consequence of Codd's *Information Principle*) that, because there is one and only one way to represent information in that model, we need one and only one set of operators to deal with it.

To pursue this latter point a moment longer: In fact, it is axiomatic that if we have N different ways to represent information, then we need N different sets of operators. And if $N > 1$, then we have more operators to implement, document, teach, learn, remember, and use. But those extra operators add complexity, not power! There is nothing useful that can be done if $N > 1$ that cannot be done if $N = 1$. We will revisit this issue in Chapter 26 (see references [26.12–26.14] and [26.17]), and it will crop up again in Chapter 27.

Back to relations specifically. Of course, some notion of top-to-bottom tuple ordering—and of left-to-right attribute ordering too, come to that—is required in the interface between the database and a host language such as C or COBOL (see the discussion of SQL cursors and ORDER BY in Chapter 4). But it is the host language, not the relational model, that imposes that requirement; in effect, the host language requires unordered relations to be converted into ordered lists or arrays (of tuples), precisely so that operations such as "retrieve the *n*th tuple" can have a meaning. Likewise, some notion of tuple ordering is also needed when results of queries are presented to the end user. However, such notions are not part of the relational model as such; rather, they are part of the environment in which the relational implementation resides.

4. *There Are No Duplicate Tuples*

This property also follows from the fact that the body of the relation is a set; sets in mathematics do not contain duplicate elements (equivalently, the elements are all distinct). *Note:* This property serves yet again to illustrate the point that a relation and a table are not the same thing, because a table might contain duplicate rows (in the absence of any discipline to prevent such a possibility), whereas a relation, by definition, *never* contains any duplicate tuples.

As a matter of fact, it is (or should be) obvious that the concept of "duplicate tuples" makes no sense. Suppose for simplicity that the relation of Fig. 6.1 had just two attributes, S# and CITY, with the intended interpretation—see Section 6.5—"Supplier S# is located in city CITY," and suppose the relation contained a tuple showing that it is a "true fact" that supplier S1 is located in London. Then if the relation contained a duplicate of that tuple (if that were possible), it would simply be informing us of that same "true fact" a second time. But if something is true, saying it twice does not make it *more* true!

Extended discussions of the problems that duplicate tuples cause can be found in references [6.3] and [6.6].

Relations *vs.* Tables

For purposes of reference, we present in this subsection a list of some of the main differences between (a) the formal object that is a relation as such and (b) the informal object that is a table, which is an informal picture on paper of that formal object:

1. Each attribute in the heading of a relation involves a type name, but those type names are usually omitted from tabular pictures.
2. Each component of each tuple in the body of a relation involves a type name and an attribute name, but those type and attribute names are usually omitted from tabular pictures.
3. Each attribute value in each tuple in the body of a relation is a value of the applicable type, but those values are usually shown in some abbreviated form—for example, S1 instead of S#('S1')—in tabular pictures.
4. The columns of a table have a left-to-right ordering, but the attributes of a relation do not. *Note:* One implication of this point is that columns might have duplicate names, or even no names at all. For example, consider this SQL query:

   ```
   SELECT S.CITY, S.STATUS * 2, P.CITY
   FROM   S, P ;
   ```

   What are the column names in the result of this query?
5. The rows of a table have a top-to-bottom ordering, but the tuples of a relation do not.
6. A table might contain duplicate rows, but a relation does not contain duplicate tuples.

The foregoing is not an exhaustive list of the differences. Others include:

- The fact that tables are usually regarded as having at least one column, while relations are not required to have at least one attribute (see the subsection "Relations with No Attributes" later in this section)
- The fact that tables—at least in SQL—are allowed to include nulls, while relations are certainly not (see Chapter 19)
- The fact that tables are "flat" or two-dimensional, while relations are *n*-dimensional (see Chapter 22)

It follows from all of the foregoing that, in order to agree that a tabular picture can properly be regarded as representing a relation, *we have to agree on how to "read" such a picture;* in other words, we have to agree on certain **rules of interpretation** for such pictures. To be specific, we have to agree that there is an underlying type for each column; that each attribute value is a value of the relevant type; that row and column orderings are irrelevant; and that duplicate rows are not allowed. If we can agree on all of these rules of interpretation, then—*and only then*—we can agree that a table is a reasonable picture of a relation.

So we can now see that a table and a relation are indeed not quite the same thing (even though it is often convenient to pretend they are). Rather, a **relation** is what the definition says it is, a rather abstract kind of object, and a **table** is a concrete picture, typically on

paper, of such an abstract object. They are not (to repeat) quite the same. Of course, they are very similar . . . and in informal contexts, at least, it is usual, and perfectly acceptable, to say they are the same. But when we are trying to be precise—and right now we *are* trying to be precise—then we do have to recognize that the two concepts are not exactly identical.

That said, it is worth pointing out too that in fact it is a major advantage of the relational model that its basic abstract object, the relation, does have such a simple representation on paper. It is that simple representation that makes relational systems easy to use and easy to understand, and makes it easy to reason about the way relational systems behave. Nevertheless, it is unfortunately also the case that that simple representation does suggest some things that are not true (e.g., that there is a top-to-bottom tuple ordering).

Relation-Valued Attributes

As noted in Section 6.3, any type whatsoever can be used as the basis for defining relational attributes, in general. It follows that relation types in particular, since they are certainly types, can be used as the basis for defining attributes of relations; in other words, attributes can be **relation-valued**, meaning we can have relations with attributes whose values are relations in turn. In other words, we can have relations that have other relations nested inside themselves. An example of such a relation is shown in Fig. 6.2. Observe with respect to that relation that (a) attribute PQ is relation-valued; (b) the cardinality and degree are both five; and in particular (c) the empty set of parts supplied by supplier S5 is represented by a PQ value that is an empty set (more precisely, an empty relation).

The main reason we mention the possibility of relation-valued attributes here is that, historically, such a possibility has usually been regarded as illegal. Indeed, it was so regarded in earlier editions of this book. For example, here is a lightly edited excerpt from the sixth edition:

S#	SNAME	STATUS	CITY	PQ
S1	Smith	20	London	P# QTY: P1 300; P2 200; .. ...; P6 100
S2	Jones	10	Paris	P# QTY: P1 300; P2 400
..	.....	..	.....	.....
S5	Adams	30	Athens	P# QTY:

Fig. 6.2 A relation with a relation-valued attribute

> Note that *all column values are atomic* . . . That is, at every row-and-column position [*sic*] in every table [*sic*] there is always exactly one data value, never a group of several values. Thus, for example, in table EMP, we have

DEPT#	EMP#
D1	E1
D1	E2
..	..

> instead of

DEPT#	EMP#
D1	E1,E2
..	..

> Column EMP# in the second version of this table is an example of what is usually called a **repeating group**. A repeating group is a column . . . that contains several values in each row (different numbers of values in different rows, in general). *Relational databases do not allow repeating groups;* the second version of the table above would not be permitted in a relational system.

And later in the same book, we find: "Domains (i.e., types) contain atomic values only . . . [Therefore,] *relations do not contain repeating groups.* A relation satisfying this condition is said to be normalized, or equivalently to be in first normal form . . . The term *relation* is always taken to mean a normalized relation in the context of the relational model."

These remarks are not correct, however (at least not in their entirety): They arose from a misunderstanding on this writer's part of the true nature of types (domains). For reasons to be discussed in Chapter 12 (Section 12.6), it is unlikely that this mistake will have caused any very serious errors in practice; nevertheless, apologies are still due to anyone who might have been misled. At least the sixth edition was correct when it said that relations in the relational model are always normalized! Again, see Chapter 12 for further discussion.

Relations with No Attributes

Every relation has a set of attributes; and, since the empty set is certainly a set, it follows that it must be possible for a relation to have the empty set of attributes, or in other words no attributes at all. (Do not be confused: We often talk about "empty relations," meaning relations whose body is an empty set of tuples, but here, by contrast, we are talking about relations whose *heading* is an empty set of *attributes*.) Thus, relations with no attributes are at least respectable from a mathematical point of view. What is perhaps more surprising is that they turn out to be extremely important from a practical point of view as well!

In order to examine this notion more closely, we first need to consider the question of whether a relation with no attributes can contain any tuples. The answer (again perhaps surprisingly) is *yes*. To be more specific, such a relation can contain *at most one* tuple:

namely, the 0-tuple (i.e., the tuple with no components; it cannot contain more than one such tuple, because all 0-tuples are duplicates of one another). There are thus precisely two relations of degree zero—one that contains just one tuple, and one that contains no tuples at all. So important are these two relations that, following Darwen [6.5], we have pet names for them: We call the first TABLE_DEE and the other TABLE_DUM, or DEE and DUM for short (DEE is the one with one tuple, DUM is the empty one). *Note:* It is hard to draw pictures of these relations! Thinking of relations as conventional tables breaks down, somewhat, in the case of DEE and DUM.

Why are DEE and DUM so important? There are several more or less interrelated answers to this question. One is that they play a role in the relational algebra—see Chapter 7—that is akin, somewhat, to the role played by the empty set in set theory or zero in ordinary arithmetic. Another has to do with what the relations mean (see reference [6.5]); essentially, DEE means TRUE, or *yes*, and DUM means FALSE, or *no*. In other words, they have *the most fundamental meanings of all.* (A good way to remember which is which is that the "E"s in DEE match the "e" in *yes*.)

In Tutorial D, the expressions TABLE_DEE and TABLE_DUM can be used as shorthand for the relation selector invocations

```
RELATION { } { TUPLE { } }
```

and

```
RELATION { } { }
```

respectively.

It is not possible to go into more detail on this topic at this juncture; suffice it to say that you will encounter DEE and DUM many times in the pages ahead. For further discussion, see reference [6.5].

Operators on Relations

We mentioned the relational selector, assignment, and equality comparison operators briefly in Section 6.3. Of course, the comparison operators "<" and ">" do *not* apply to relations; however, relations are certainly subject to other kinds of comparisons in addition to simple equality, as we now explain.

Relational comparisons: We begin by defining a new kind of *<bool exp>*, *<relation comp>*, with syntax as follows:

```
<relation exp> <relation comp op> <relation exp>
```

The relations denoted by the two *<relation exp>*s must be of the same type, and *<relation comp op>* must be one of the following:

- $=$ (equals)
- $\neq$ (not equals)
- $\subseteq$ (subset of)
- $\subset$ (proper subset of)
- $\supseteq$ (superset of)
- $\supset$ (proper superset of)

Then we allow a *<relation comp>* to appear wherever a *<bool exp>* is expected—for example, in a WHERE clause. Here are a couple of examples:

1. `S { CITY } = P { CITY }`
   Meaning: Is the projection of snppliers over CITY equal to the projection of parts over CITY?
2. `S { S# } ⊃ SP { S# }`
   Meaning (considerably paraphrased): Are there any suppliers who supply no parts at all?

One particular relational comparison often needed in practice is a test to see whether a given relation is equal to an empty relation of the same type (i.e., one that contains no tuples). It is useful to have a shorthand for this particular case. We therefore define the expression

```
IS_EMPTY ( <relation exp> )
```

to return TRUE if the relation denoted by the *<relation exp>* is empty and FALSE otherwise.

Other operators: Another common requirement is to be able to test whether a given tuple *t* appears in a given relation *r:*

```
t ∈ r
```

This expression returns TRUE if *t* appears in *r* and FALSE otherwise ("∈" is the *set membership* operator; the expression *t* ∈ *r* can be pronounced "*t* belongs to *r*" or "*t* is a member of *r*" or, more simply, just "*t* [is] in *r*").

We also need to be able to extract the single tuple from a relation of cardinality one:

```
TUPLE FROM r
```

This expression raises an exception if *r* does not contain exactly one tuple; otherwise, it returns just that one tuple.

In addition to the operators discussed so far, there are also all of the generic operators—join, restrict, project, and so on—that go to make up the relational algebra. We defer detailed treatment of these operators to the next chapter.

Relation type inference: Just as the tuple type naming scheme described in Section 6.2 facilitates the task of determining the type of the result of an arbitrary tuple expression, so the relation type naming scheme described in Section 6.3 facilitates the task of determining the type of the result of an arbitrary relational expression. Chapter 7 goes into detail on this issue; here we content ourselves with one simple example. Given the suppliers relvar S, the expression

```
S { S#, CITY }
```

yields a result (a relation) whose type is:

```
RELATION { S# S#, CITY CHAR }
```

Analogous remarks apply to all possible relational expressions.

ORDER BY: For presentation purposes it is highly desirable to support an ORDER BY operator, as SQL does (see Chapter 3). We omit a detailed definition here, since the semantics are surely obvious. Note, however, that:

- ORDER BY works (effectively) by sorting tuples into some specified sequence, and yet "<" and ">" are not defined for tuples (!).
- ORDER BY is not a *relational* operator, since what it returns is not a relation.
- ORDER BY is also not a *function*, since there are many possible outputs for a given input, in general.

As an example of this last point, consider the effect of the operation ORDER BY CITY on the suppliers relation of Fig. 6.1. Clearly, this operation can return any of four distinct results. By contrast, the operators of the relational algebra certainly are functions—for any given input, there is always just one possible output.

6.5 RELATION VARIABLES

Now we turn to relation variables (*relvars* for short). Recall from Chapter 3 that relvars come in two varieties, *base relvars* and *views* (also called real and virtual relvars, respectively). In this section, we are primarily concerned with base relvars specifically (views are discussed in detail in Chapter 10); note, however, that anything we say here that talks in terms of *relvars*, unqualified, is true of relvars in general, views included.

Base Relvar Definition

Here is the syntax for defining a base relvar:

```
VAR <relvar name> BASE <relation type>
                  <candidate key def list>
                [ <foreign key def list> ] ;
```

The *<relation type>* takes the form

```
RELATION { <attribute commalist> }
```

(it is in fact an invocation of the RELATION type generator, as discussed in Section 6.3). The *<candidate key def list>* and optional *<foreign key def list>* are explained later. Here by way of example are the base relvar definitions for the suppliers-and-parts database (repeated from Fig. 3.9):

```
VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR }
    PRIMARY KEY { S# } ;
```

```
VAR P BASE RELATION
    { P#     P#,
      PNAME  NAME,
      COLOR  COLOR,
      WEIGHT WEIGHT,
      CITY   CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#    S#,
       P#    P#,
       QTY   QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

Explanation:

1. These three base relvars have (relation) types as follows:

   ```
   RELATION { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR }

   RELATION { P# P#, PNAME NAME, COLOR COLOR,
                                 WEIGHT WEIGHT, CITY CHAR }

   RELATION { S# S#, P# P#, QTY QTY }
   ```

2. The terms *heading, body, attribute, tuple, degree* (and so on) previously defined for relation values are all interpreted in the obvious way to apply to relvars as well.
3. All possible values of any given relvar are of the same relation type—namely, the relation type specified (indirectly, if the given relvar is a view) in the relvar definition—and therefore have the same heading.
4. When a base relvar is defined, it is given an initial value that is the empty relation of the applicable type.
5. Candidate key definitions are explained in detail in Chapter 9. Prior to that point, we will simply assume that each base relvar definition includes exactly one *<candidate key def>*, of the following particular form:

   ```
   PRIMARY KEY { <attribute name commalist> }
   ```

6. Foreign key definitions are also explained in Chapter 9.
7. Defining a new relvar causes the system to make entries in the catalog to describe that relvar.
8. As noted in Chapter 3, relvars, like relations, have a corresponding predicate: namely, the predicate that is common to all of the relations that are possible values of the relvar in question. In the case of the suppliers relvar S, for example, the predicate looks something like this:

   The supplier with supplier number S# is under contract, is named SNAME, has status STATUS, and is located in city CITY

9. We assume that a means is available for specifying default values for attributes of base relvars. The default value for a given attribute is a value that is to be placed in the applicable attribute position if the user does not provide an explicit value when inserting some tuple. Suitable Tutorial D syntax for specifying defaults might take

the form of a new clause on the base relvar definition, DEFAULT { *<default spec commalist>* } say, where each *<default spec>* takes the form *<attribute name> <default>*. For example, we might specify DEFAULT { STATUS 0, CITY '' } in the definition of the suppliers relvar *S*. *Note:* Candidate key attributes will usually, though not invariably, have no default (see Chapter 19).

Here is the syntax for dropping an existing base relvar:

```
DROP VAR <relvar name> ;
```

This operation sets the value of the specified relvar to "empty" (i.e., it deletes all tuples in the relvar, loosely speaking); it then deletes all catalog entries for that relvar. The relvar is now no longer known to the system. *Note:* For simplicity, we assume the DROP will fail if the relvar in question is still being used somewhere—for example, if it is referenced in some view definition somewhere.

Updating Relvars

The relational model includes a relational assignment operation for assigning values to—that is, *updating*—relvars (base relvars in particular). Here, slightly simplified, is the Tutorial D syntax:

```
<relation assignment>
    ::=   <relation assign commalist> ;

<relation assign>
    ::=   <relvar name> := <relation exp>
```

The semantics are as follows:[7] First, all of the *<relation exp>*s on the right sides of the *<relation assign>*s are evaluated; second, the *<relation assign>*s are executed in sequence as written. Executing an individual *<relation assign>* involves assigning the relation resulting from evaluation of the *<relation exp>* on the right side to the relvar identified by the *<relvar name>* on the left side (replacing the previous value of that relvar). Of course, the relation and relvar must be of the same type.

By way of example, suppose we are given two further base relvars S' and SP' of the same types as the suppliers relvar S and the shipments relvar SP, respectively:

```
VAR S' BASE RELATION
    { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR } ... ;

VAR SP' BASE RELATION
    { S# S#, P# P#, QTY QTY } ... ;
```

Here then are some valid examples of *<relation assignment>*:

```
1. S' :=  S , SP' :=  SP ;
2. S' :=  S WHERE CITY = 'London' ;
3. S' :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

Note that each individual *<relation assign>* can be regarded as both (a) *retrieving* the relation specified on the right side and (b) *updating* the relvar specified on the left side.

[7] Except as noted in footnote 9 in Chapter 5.

Now suppose we change the second and third examples by replacing relvar S' on the left side by relvar S in each case:

```
2. S  :=  S WHERE CITY = 'London' ;
3. S  :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

Observe that these two assignments are both effectively *updates to relvar S*—one effectively deletes all suppliers not in London, the other effectively deletes all suppliers in Paris. For convenience Tutorial D does support explicit INSERT, DELETE, and UPDATE operations, but each of these operations is defined to be shorthand for a certain *<relation assign>*. Here are some examples:

```
1. INSERT S RELATION { TUPLE { S#     S# ('S6'),
                               SNAME  NAME ('Smith'),
                               STATUS 50,
                               CITY   'Rome' } } ;
```

Assignment equivalent:

```
S  :=  S UNION RELATION { TUPLE { S#     S# ('S6'),
                                  SNAME  NAME ('Smith'),
                                  STATUS 50,
                                  CITY   'Rome' } } ;
```

Note, incidentally, that this assignment will succeed if the specified tuple for supplier S6 already exists in relvar S. In practice, we might want to refine the semantics of INSERT in such a way as to raise an exception if an attempt is made "to insert a tuple that already exists." For simplicity, however, we ignore this refinement here. Analogous remarks apply to DELETE and UPDATE also.

```
2. DELETE S WHERE CITY = 'Paris' ;
```

Assignment equivalent:

```
S  :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

```
3. UPDATE S WHERE CITY = 'Paris'
             { STATUS := 2 * STATUS,
               CITY   := 'Rome' } ;
```

Assignment equivalent:

```
S  :=  WITH ( S WHERE CITY = 'Paris' ) AS T1 ,
           ( EXTEND T1 ADD ( 2 * STATUS AS NEW_STATUS,
                             'Rome' AS NEW_CITY ) ) AS T2 ,
           T2 { ALL BUT STATUS, CITY } AS T3 ,
           ( T3 RENAME ( NEW_STATUS AS STATUS,
                         NEW_CITY AS CITY ) ) AS T4 :
       ( S MINUS T1 ) UNION T4 ;
```

As you can see, the assignment equivalent is a little complicated in this case—in fact, it relies on several features that will not be explained in detail until the next chapter. For that reason, we omit further discussion here.

For purposes of reference, here is a slightly simplified summary of the syntax of INSERT, DELETE, and UPDATE:

```
INSERT <relvar name> <relation exp> ;

DELETE <relvar name> [ WHERE <bool exp> ] ;
```

```
UPDATE <relvar name> [ WHERE <bool exp> ]
                     { <attribute update commalist> } ;
```

An *<attribute update>* in turn takes the form

```
<attribute name> := <exp>
```

Also, the *<bool exp>* in DELETE and UPDATE can include references to attributes of the target relvar, with the obvious semantics.

We close this subsection by stressing the point that relational assignment, and hence INSERT, DELETE, and UPDATE, are all set-level operations.[8] UPDATE, for example, updates a *set* of tuples in the target relvar, loosely speaking. Informally, we often talk of (e.g.) updating some individual tuple, but it must be clearly understood that:

1. We are really talking about updating a *set* of tuples, a set that just happens to have cardinality one.
2. Sometimes updating a set of tuples of cardinality one is impossible!

Suppose, for example, that the suppliers relvar is subject to the integrity constraint (see Chapter 9) that suppliers S1 and S4 must have the same status. Then any "single-tuple" UPDATE that tries to change the status of just one of those two suppliers *must fail*. Instead, both must be updated simultaneously, as here:

```
UPDATE S WHERE S# = S# ('S1') OR S# = S# ('S4')
       { STATUS := some value } ;
```

To pursue the point a moment longer, we should now add that to talk (as we have just been doing) of "updating a tuple"—or set of tuples—in a relvar is really rather sloppy. Like relations, tuples are *values* and *cannot* be updated, by definition. Thus, what we really mean when we talk of (for example) "updating tuple t to t'" is that we are replacing the tuple t (the tuple *value* t, that is) by another tuple t' (which is, again, a tuple *value*).[9] Analogous remarks apply to phrases such as "updating attribute A" within some tuple. In this book, we will continue to talk in terms of "updating tuples" and "updating attributes of tuples"—the practice is convenient—but it must be understood that such usage is only shorthand, and rather sloppy shorthand at that.

Relvars and Their Interpretation

We conclude this section with a reminder to the effect that (as explained in Chapter 3, Section 3.4) (a) the heading of any given relvar can be regarded as a predicate, and (b) the tuples appearing in that relvar at any given time can be regarded as true propositions, obtained from the predicate by substituting arguments of the appropriate type for the

[8] In passing, we note that, by definition, the CURRENT forms of DELETE and UPDATE in SQL—see Section 4.6—are *tuple*-level (or row-level, rather), and are therefore contraindicated.

[9] Of course, none of this is to say that we cannot update tuple *variables*. As explained in Section 6.2, however, the notion of a tople variable is not part of the relational model, and relational databases do not contain such variables.

parameters of that predicate ("instantiating the predicate"). We can say that the predicate corresponding to a given relvar is the intended interpretation, or meaning, for that relvar, and the propositions corresponding to tuples of that relvar are understood by convention to be true ones. In fact, the Closed World Assnmption (also known as the Closed World Interpretation) says that if an otherwise valid tuple—that is, one that conforms to the relvar heading—does *not* appear in the body of the relvar, then we can assume the corresponding proposition is false. In other words, the body of the relvar at any given time contains *all* and *only* the tuples that correspond to true propositions at that time. We will have considerably more to say on such matters in Chapter 9.

6.6 SQL FACILITIES

Rows

SQL does not support tuples, as such, at all; instead, it supports *rows*, which have a left-to-right ordering to their components. Within a given row, the component values—which are called *column values* if the row is immediately contained in a table, or *field values* otherwise—are thus identified primarily by their ordinal position (even when they also have names, which is not always the case). Row types have no explicit row type name. A row value can be "selected" (the SQL term is *constructed*) by means of an expression—actually a *<row value constructor>*—of the form:

```
[ ROW ] ( <exp commalist> )
```

The parentheses can be omitted if the commalist contains just one *<exp>*; the keyword ROW must be omitted in this case, and is optional otherwise. The commalist must not be empty (SQL does not support a "0-row"). Here is an example:

```
ROW ( P#('P2'), P#('P4'), QTY(7) )
```

This expression denotes a row of degree three.

As we saw in Chapter 5, SQL also supports a ROW *type constructor* (its counterpart to the Tutorial D TUPLE type generator) that can be invoked in the definition of, for example, some table column or some variable.[10] Here is an example of the latter case:

```
DECLARE ADDR ROW ( STREET CHAR(50),
                   CITY   CHAR(25),
                   STATE  CHAR(2),
                   ZIP    CHAR(5) ) ;
```

Row assignments and comparisons are supported, with the caveat that the only strong typing that applies is the limited form described in Chapter 5, Section 5.7. Note in particular, therefore, that the fact that *r1* = *r2* is true does not imply that rows *r1* and *r2* are the same row. Moreover, "<" and ">" are legal row comparison operators! The details of such comparisons are complicated, however, and we omit them here; see reference [4.20] for further discussion.

[10] Do not be confused: SQL's "row *value* constructor" is basically a tuple selector, while its "row *type* constructor" is basically the TUPLE type generator (speaking very loosely!).

SQL does not support row versions of any of the regular relational operators ("row project," "row join," etc.), nor does it provide direct counterparts to WRAP and UNWRAP. It also does not support any "row type inferencing"—but this latter point is perhaps unimportant, given that SQL supports almost no row operators anyway.

Table Types

SQL does not support relations, as such, at all; instead, it supports *tables*. The body of a table in SQL is not a set of tuples but a *bag* of *rows* instead (a bag, also known as a *multiset*, is a collection that like a set has no ordering, but unlike a set permits duplicate elements); thus, the columns of such a table have a left-to-right ordering, and there can be duplicate rows. (Throughout this book, however, we will apply certain disciplines to guarantee that duplicate rows never occur, even in SQL contexts.) SQL does not use the terms *heading* or *body*.

Table types have no explicit table type name. A table value can be "selected" (once again, the SQL term is *constructed*) by means of an expression—actually a *<table value constructor>*—of the form:

```
VALUES <row value constructor commalist>
```

(where the commalist must not be empty). Thus, for example, the expression

```
VALUES ( P#('P1'), P#('P2'), QTY(5) ) ,
       ( P#('P1'), P#('P3'), QTY(3) ) ,
       ( P#('P2'), P#('P3'), QTY(2) ) ,
       ( P#('P2'), P#('P4'), QTY(7) ) ,
       ( P#('P3'), P#('P5'), QTY(4) ) ,
       ( P#('P4'), P#('P6'), QTY(8) )
```

evaluates to a table looking something like the relation shown in Section 6.3, except that it has no explicit column names.

SQL does not really support an explicit counterpart to the RELATION type generator at all. It also does not support an explicit table assignment operator (though it does support explicit INSERT, DELETE, and UPDATE statements). Nor does it support any table comparison operators (not even "="). However, it does support an operator for testing whether a given row appears in a given table:

```
<row value constructor> IN <table exp>
```

It also supports a counterpart to the TUPLE FROM operator:

```
( <table exp> )
```

If such an expression appears where an individual row is required, and if the *<table exp>* denotes a table that contains exactly one row, then that row is returned; otherwise, an exception is raised. *Note:* We remark in passing that *<table name>* is not a valid *<table exp>* (!).

Table Values and Variables

SQL unfortunately uses the same term *table* to mean both a table value and a table variable. In the present section, therefore, the term *table* must be understood to mean either a table value or a table variable, as the context demands. Here first, then, is the SQL syntax for defining a base table:

```
CREATE TABLE <base table name>
           ( <base table element commalist> ) ;
```

Each *<base table element>* is either a *<column definition>* or a *<constraint>*:[11]

- The *<constraint>*s specify certain integrity constraints that apply to the base table in question. We defer detailed discussion of such constraints to Chapter 9, except to note that, since they permit duplicate rows, SQL tables do not necessarily have a primary key (or, more fundamentally, any candidate keys).
- The *<column definition>*s—there must be at least one—take the following general form:

  ```
  <column name> <type name> [ <default spec> ]
  ```

  The optional *<default spec>* specifies the *default value*, or simply *default*, that is to be placed in the applicable column if the user does not provide an explicit value on INSERT (see Chapter 4, Section 4.6, subsection "Operations Not Involving Cursors," for an example). It takes the form DEFAULT *<default>*, where *<default>* is a literal, a niladic built-in operator name,[12] or the keyword NULL (see Chapter 19). If a given column does not have an explicit default, it is implicitly assumed to have a default of null—that is, null is the "default default" (as in fact is always the case in SQL). *Note:* For reasons beyond the scope of this book, the default *must* be null if the column is of a user-defined type (as already mentioned in Chapter 4). It must also be null if the column is of some row type, and it must be either null or empty (specified as ARRAY[]) if the column is of an array type.

For some examples of CREATE TABLE, see, for example, Fig. 4.1 in Chapter 4. Note that (as we already know) SQL does not support table-valued columns, nor does it support tables with no columns at all. It does support ORDER BY, together with analogs of most of the operators of the relational algebra (see Chapters 7 and 8). However, its rules for "table type inferencing" (though they do necessarily exist) are at least partly implicit; they are also complicated, and we omit the specifics here.

An existing base table can be dropped. Here is the syntax:

```
DROP TABLE <base table name> <behavior> ;
```

where (as in the case of DROP TYPE in Chapter 5) *<behavior>* is either RESTRICT or CASCADE. Loosely, RESTRICT means the DROP will fail if the table is currently in use

[11] A *<base table element>* can also take the form LIKE *T*, which allows some or all of the column definitions for the base table being defined to be copied from some existing named table *T*.

[12] A niladic operator is one that takes no explicit operands. CURRENT_DATE is an example.

anywhere, while CASCADE means the DROP will always succeed and will cause an implicit DROP . . . CASCADE for everything currently using the table.

An existing base table can also be *altered* by means of the statement ALTER TABLE. The following kinds of "alterations" are supported:

- A new column can be added.
- A new default can be defined for an existing column (replacing the previous one, if any).
- An existing column default can be deleted.
- An existing column can be deleted.
- A new integrity constraint can be specified.
- An existing integrity constraint can be deleted.

We give an example of the first case only:

```
ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1 ;
```

This statement adds a DISCOUNT column (of type INTEGER) to the suppliers base table. All existing rows in that table are extended from four columns to five; the initial value of the new fifth column is -1 in every case.

Finally, the SQL INSERT, DELETE, and UPDATE statements have already been discussed in Chapter 4.

Structured Types

Caveat: The portions of the SQL standard that are relevant to this subsection are hard to understand. What follows is this writer's best attempt to make sense of the material.

Here first, then, repeated from Chapter 5 (Section 5.7), is an example of a structured type definition:

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
```

Type POINT can now be used in the definition of variables and columns. For example:

```
CREATE TABLE NADDR
     ( NAME ... ,
       ADDR ... ,
       LOCATION POINT ... ,
       ... ) ;
```

Now, we never said as much explicitly in Chapter 5, but we at least implied that SQL's structured types were scalar types specifically, just as the Tutorial D analog of the foregoing POINT type was a scalar type. In some respects, however, they are closer to Tutorial D's *tuple* types.[13] Certainly it is true that we can access the components ("attributes") of a given POINT value, rather as if it were a tuple. Dot qualification syntax is used for this purpose, as in the following examples (note that explicit correlation names are required, as the examples indicate):

[13] Except that structured types have a left-to-right ordering to their attributes, whereas tuple types do not.

```
SELECT NT.LOCATION.X, NT.LOCATION.Y
FROM   NADDR AS NT
WHERE  NAME = ... ;

UPDATE NADDR AS NT
SET    NT.LOCATION.X = 5.0
WHERE  NAME = ... ;
```

When used as in the foregoing example, an SQL structured type effectively behaves as if it were a simple row type (again, see Section 5.7 in Chapter 5), except that:

- The components are called attributes instead of fields.
- More importantly, structured types, unlike row types, have *names* (we will revisit this point at the very end of this section).

So far, then, SQL's structured types look as if they might not be too hard to understand. But—and it is a big but!—the foregoing is not the end of the story.[14] In addition to the foregoing, SQL also allows a base table to be defined to be "OF" some structured type, in which case a number of further considerations come into play. In order to discuss some of those considerations, let us first extend the definition of type POINT as follows:

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL
       REF IS SYSTEM GENERATED ;
```

Now we can define a base table to be "OF" this type—for example:

```
CREATE TABLE POINTS OF POINT
     ( REF IS POINT# SYSTEM GENERATED ... ) ;
```

Explanation:

1. When we define a structured type *T*, the system automatically defines an associated *reference type* ("REF type") called REF(*T*). Values of type REF(*T*) are "references" to rows within some base table[15] that has been defined to be "OF" type *T* (see point 3). In the example, then, the system automatically defines a type called REF(POINT), whose values are references to rows within base tables that are defined to be "OF" type POINT.
2. The specification REF IS SYSTEM GENERATED in a CREATE TYPE statement means the actual values of the associated REF type are provided by the system (other options—for example, REF IS USER GENERATED—are available, but we omit the details here). *Note:* In fact, REF IS SYSTEM GENERATED is the default; in our example, therefore, we could have left our original definition of type POINT unchanged if we had wanted.
3. Base table POINTS has been defined to be "OF" the structured type POINT. The keyword OF is really not very appropriate, however, because the table is *not* actually "of" the type in question, and neither are its rows![16]

[14] Nor is what follows! See Chapters 20 and 26 for further discussion.

[15] Or possibly some view. Details of the view case are beyond the scope of this book.

[16] Note in particular, therefore, that if the declared type of some parameter *P* to some operator *Op* is some structured type *ST*, a row from a base table that has been defined to be "OF" type *ST* cannot be passed as a corresponding argument to an invocation of that operator *Op*.

- First of all, if the table had just one column and that column were of the structured type in question, *ST* say, then we *might* say—though not in SQL!—something to the effect that the table is of type TABLE(*ST*) and its rows are of type ROW(*ST*).
- But the table does not have just one column, in general; rather, it has one column for each attribute of *ST*. Thus, in the example, base table POINTS has two columns X and Y; it explicitly does *not* have a column of type POINT.
- Furthermore, that table has one extra column as well: namely, a column of the applicable REF type. However, the syntax for defining that column is not the normal column definition syntax but instead looks like this:

  ```
  REF IS <column name> SYSTEM GENERATED
  ```

  This extra column is called a *self-referencing column;* it is used to contain unique IDs or "references" for the rows of the base table in question. The ID for a given row is assigned when the row is inserted and remains associated with that row until it is deleted. In the example, therefore, base table POINTS actually has *three* columns (POINT#, X, and Y, in that order), not just two. *Note:* It is not at all clear why it should be necessary to define the table to be "OF" some structured type in the first place, instead of just defining an appropriate column in the usual way, in order to obtain this "unique ID" functionality, but our explanations are in accordance with the way SQL is defined.

As an aside, we note that (perhaps surprisingly) a SYSTEM GENERATED column can be a target column in an INSERT or UPDATE operation, though special considerations apply. We omit the details here.

4. Table POINTS is an example of what the standard calls, not very aptly, both a *typed table* and a *referenceable table*. As the standard puts it: "A table ... whose row type is derived from a structured type is called a *typed table*. Only a base table or a view can be a typed table." And elsewhere: "A referenceable table is necessarily also a *typed table* . . . A typed table is called a referenceable table."

Now, it seems that the foregoing features were introduced in SQL:1999 primarily to serve as a basis for incorporating some kind of "object functionality" into SQL,[17] and we will discuss that functionality in detail in Chapter 26. But nothing in the standard says the features in question can be used only in connection with that functionality, which is why we describe them in this chapter.

One final point: Recall from Chapter 5 that there is no explicit "define tuple type" operator in Tutorial D; instead, there is a TUPLE type generator, which can be invoked in (e.g.) the definition of a tuple variable. As a consequence, the only names tuple types have in Tutorial D are names of the form:

```
TUPLE { A1 T1, A2 T2, ..., An Tn }
```

One important consequence is that it is immediately clear when two tuple types are in fact one and the same, and when two tuples are of the same type.

[17] The fact that SQL structured types *always* have an associated REF type, even though that REF type serves no purpose except when the structured type in question is used as the basis for defining a "typed table," strongly suggests this conjecture is correct.

Now, row types in SQL are similar to Tutorial D's tuple types in the foregoing respect. But structured types are different; there *is* an explicit "define structured type" operator, and structured types do have additional explicit names. By way of example, consider the following SQL definitions:

```
CREATE TYPE POINT1 AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
CREATE TYPE POINT2 AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
DECLARE V1 POINT1 ;
DECLARE V2 POINT2 ;
```

Note carefully that variables V1 and V2 are of different types. Thus, they cannot be compared with one another, nor can either one be assigned to the other.

6.7 SUMMARY

In this chapter we have taken a comprehensive look at relations and related matters. We began by defining tuples precisely, stressing the points that (a) every tuple contains exactly one value for each of its attributes, (b) there is no left-to-right ordering to the attributes, and (c) every subset of a tuple is a tuple, and every subset of a heading is a heading. And we discussed the TUPLE type generator, tuple selectors, tuple assignment and equality, and other generic tuple operators.

Then we turned to relations (meaning, more specifically, relation *values*). We gave a precise definition, and pointed out that every subset of a body is a body, and (as with tuples) every subset of a heading is a heading. We discussed the RELATION type generator and relation selectors, and we observed that the attributes of a given relation type can be of any type whatsoever, in general.

Note: It is worth elaborating on this last point briefly, since there is so much confusion surrounding it in the industry. You will often hear claims to the effect that relational attributes can only be of very simple types (numbers, strings, and so forth). The truth is, however, that there is absolutely nothing in the relational model to support such claims. As noted in Chapter 5, in fact, types can be as simple or as complex as we like, and so we can have attributes whose values are numbers, or strings, or dates, or times, or audio recordings, or maps, or video recordings, or geometric points (etc.).

The foregoing message is so important—and so widely misunderstood—that we state it again in different terms:

> **The question of what data types are supported is orthogonal to the question of support for the relational model.**

Back to our summary. Next, we went on to state certain properties that all relations satisfy:

1. They are always normalized.
2. They have no left-to-right ordering to their attributes.

3. They have no top-to-bottom ordering to their tuples.
4. They never contain any duplicate tuples.

We also identified some of the main differences between relations and tables: we discussed **relation-valued attributes**; and we briefly considered TABLE_DEE and TABLE_DUM, which are the only possible relations with no attributes at all. We described **relational comparisons** in some detail, and we took a quick look at certain other operators on relations (including **ORDER BY** in particular).

In connection with operators on relations, by the way, you might have noticed that we discussed the question of user-defined operators for scalar types in some depth in Chapter 5, but did not do the same for relation types. The reason is that most of the relational operators we need—restrict, project, join, relational comparisons, and so forth—are in fact built into the relational model itself and do not require any "user definition." (What is more, those operations are *generic*, in that they apply to relations of all types, loosely speaking.) However, there is no reason why those built-in operators should not be augmented with a set of user-defined ones, if the system provides a means for defining them.

We remind you that the heading of any given relation can be regarded as a predicate and the tuples of that relation can be regarded as true propositions, derived from that predicate by supplying argument values of the appropriate types for the parameters of the predicate.

Next, we went on to consider **base relvars**, pointing out that, like relations, relvars have predicates. The **Closed World Assumption** says we can assume that if an otherwise valid tuple does not appear in the body of the relvar, then the corresponding proposition is false.

Next, we discussed relational **assignment** (and the **INSERT**, **DELETE**, and **UPDATE** shorthands) in some detail. We emphasized the point that relational assignment was a set-level operation; we also noted that it was not really correct to speak of "updating tuples" or "updating attributes."

Finally, we sketched the SQL counterparts to the foregoing ideas, where applicable. An SQL table is not a set of tuples but a bag of rows (also, SQL uses the same term *table* for both table values and table variables). Base tables can be "altered" by means of **ALTER TABLE**. They can also be defined in terms of structured types, a possibility that we will be considering in much more detail later in this book (in Chapter 26).

EXERCISES

6.1 What do you understand by the term *cardinality*?

6.2 Define as precisely as you can the terms *tuple* and *relation*.

6.3 State as precisely as you can what it means for (a) two tuples to be equal; (b) two tuple types to be equal; (c) two relations to be equal; (d) two relation types to be equal.

6.4 Write (a) a set of predicates and (b) a set of Tutorial D relvar definitions for the suppliers-parts-projects database of Fig. 4.5 (see the inside back cover).

6.5 Write tuple selector invocations for a typical tuple from each of the relvars in the suppliers-parts-projects database.

6.6 Define a local tuple variable into which an individual tuple could be retrieved from the shipments relvar in the suppliers-parts-projects database.

6.7 What do the following Tutorial D expressions denote?

a. `RELATION { S# S#, P# P#, J# J#, QTY QTY } { }`

b.
```
RELATION { TUPLE { S# S#('S1'), P# P#('P1'),
                   J# J#('J1'), QTY QTY(200) } }
```

c. `RELATION { TUPLE { } }`

d. `RELATION { } { TUPLE { } }`

e. `RELATION { } { }`

6.8 What do you understand by the term *first normal form?*

6.9 List as many differences as you can think of between relations and tables.

6.10 Give an example of your own of a relation with (a) one relation-valued attribute and (b) two such attributes. Also, give two more relations that represent the same information as those relations but do not involve relation-valued attributes.

6.11 Write an expression that returns TRUE if the current value of the parts relvar P is empty and FALSE otherwise. Do not use the IS_EMPTY shorthand.

6.12 In what respects is ORDER BY a rather unusual operator?

6.13 State the Closed World Assumption.

6.14 It is sometimes suggested that a relvar is really just a traditional computer file, with "tuples" instead of records and "attributes" instead of fields. Discuss.

6.15 Give Tutorial D formulations for the following updates to the suppliers-parts-projects database:

a. Insert a new shipment with supplier number S1, part number P1, project number J2, quantity 500.

b. Insert a new supplier S10 into table S (the name and city are Smith and New York, respectively; the status is not yet known).

c. Delete all blue parts.

d. Delete all projects for which there are no shipments.

e. Change the color of all red parts to orange.

f. Replace all appearances of supplier number S1 by appearances of supplier number S9 instead.

6.16 We have seen that data definition operations cause updates to be made to the catalog. But the catalog is only a collection of relvars, just like the rest of the database; so could we not use the regular update operations INSERT, DELETE, and UPDATE to update the catalog appropriately? Discuss.

6.17 In the body of the chapter, we said that any type whatsoever can be used as the basis for defining relational attributes, in general. That qualifier "in general" was there for a reason, however. Can you think of any exceptions to this general rule?

6.18 What do you understand by the SQL terms *column, field,* and *attribute?*

6.19 (*Modified version of Exercise 5.23*) Consider the SQL type POINT and the SQL table POINTS as defined in the subsection "Structured Types" in Section 6.6. Type POINT has a representation involving Cartesian coordinates X and Y. What happens if we replace that type by a revised type POINT with a representation involving polar coordinates R and θ instead?

REFERENCES AND BIBLIOGRAPHY

Most of the following references are applicable to all aspects of the relational model, not just to relations as such.

6.1 E. F. Codd: "A Relational Model of Data for Large Shared Data Banks," *CACM 13*, No. 6 (June 1970). Republished in *Milestones of Research—Selected Papers 1958–1982* (*CACM 25th Anniversary Issue*), *CACM 26*, No. 1 (January 1983). See also the earlier version, "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19, 1969), which was Codd's very first publication on the relational model.

The paper that started it all. Although now over 30 years old, it stands up remarkably well to repeated rereading. Of course, many of the ideas have been refined somewhat since the paper was first published, but by and large the changes have been evolutionary, not revolutionary, in nature. Indeed, there are some ideas in the paper whose implications have still not been fully explored.

We remark on a small matter of terminology. In his paper, Codd uses the term *time-varying relations* in place of our preferred *relation variables* (relvars). But *time-varying relations* is really not a very good term. First, relations as such are *values* and simply do not "vary with time" (there is no notion in mathematics of a relation having different values at different times). Second, if we say in some programming language, for example,

```
DECLARE N INTEGER ;
```

we do not call N a "time-varying integer," we call it an *integer variable*. In this book, therefore, we use our "relation variable" terminology, not the "time-varying" terminology; however, you should at least be aware of the existence of this latter.

6.2 E. F. Codd: *The Relational Model for Database Management Version 2*. Reading, Mass.: Addison-Wesley (1990).

Codd spent much of the late 1980s revising and extending his original model (which he renamed "the Relational Model Version 1" or RM/V1), and this book is the result. It describes "the Relational Model Version 2" (RM/V2). The essential difference between RM/V1 and RM/V2 is as follows: Whereas RM/V1 was intended as an abstract blueprint for one particular aspect of the total database problem (essentially the foundational aspect), RM/V2 was intended as an abstract blueprint for *the entire system*. Thus, where RM/V1 had just three parts—structure, integrity, and manipulation—RM/V2 had 18; and those 18 parts include not only the original three (of course), but also parts having to do with the catalog, authorization, naming, distributed database, and various other aspects of database management. For purposes of reference, here is a complete list of the 18 parts:

A	Authorization	M	Manipulation
B	Basic operators	N	Naming
C	Catalog	P	Protection
D	Principles of DBMS design	Q	Qualifiers
E	Commands for the DBA	S	Structure
F	Functions	T	Data types
I	Integrity	V	Views
J	Indicators	X	Distributed database
L	Principles of language design	Z	Advanced operators

The ideas advocated in this book are by no means universally accepted, however [6.7, 6.8]. We comment on one particular issue here. As we saw in Chapter 5, domains (i.e., types) constrain comparisons. In the case of suppliers and parts, for instance, the comparison S.S# = SP.P# is not valid, because the comparands are of different types; hence, an attempt to join suppliers and shipments over matching supplier and part numbers will fail. Codd therefore proposes "domain check override" (DCO) versions of certain of the relational algebra operations, which allow the operations in question to be performed even if they involve a comparison between values of different types. A DCO version of the join just mentioned, for example, will cause the join to be done even though attributes S.S# and SP.P# are of different types (presumably it will be done on the basis of matching *representations* instead of matching *types*).

But therein lies the problem. *The whole DCO idea is based on a confusion between types and representations.* Recognizing domains for what they are (i.e., types)—with all that such recognition entails—gives us the domain checking we want *and* gives us something like the DCO capability as well. For example, the following expression constitutes a valid representation-level comparison between a supplier number and a part number:

```
THE_S# ( S# ) = THE_P# ( P# )
```

(both comparands here are of type CHAR). Thus, it is our claim that the kind of mechanism discussed in Chapter 5 gives us all the facilities we want, but does so in a manner that is clean, systematic (i.e., not *ad hoc*), and fully orthogonal. In particular, there is now no need to clutter up the relational model with new constructs such as "DCO join" (etc.).

6.3 Hugh Darwen: "The Duplicity of Duplicate Rows," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

This paper was written as further support for the arguments previously presented in reference [6.6] (first version) in support of the prohibition against duplicate rows. The paper not only offers novel versions of some of those same arguments, it also manages to come up with some additional ones. In particular, it stresses the fundamental point that, in order to discuss in any intelligent manner the question of whether two objects are duplicates, it is essential to have a clear *criterion of equality* (called a criterion of *identity* in the paper) for the class of objects under consideration. In other words, what does it mean for two objects, be they rows in a table or anything else, to be "the same"?

6.4 Hugh Darwen: "Relation-Valued Attributes," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

6.5 Hugh Darwen: "The Nullologist in Relationland," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

Nullology is, as Darwen puts it, "the study of nothing at all"—in other words, the study of the empty set. (It has nothing to do with SQL-style nulls!) Sets are ubiquitous in relational theory, and the question of what happens if such a set happens to be empty is far from a frivolous one. In fact, it turns out that very often the empty-set case turns out to be absolutely fundamental. *Note:* As far as the present chapter is concerned, the most immediately applicable portions of this paper are Sections 2 ("Tables with No Rows") and 3 ("Tables with No Columns").

6.6 C. J. Date: "Double Trouble, Double Trouble" (in two parts), *http://www.dbdebunk.com* (April 2002). An earlier version of this paper, "Why Duplicate Rows Are Prohibited," appeared in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

Presents an extensive series of arguments, with examples, in support of the prohibition against duplicate rows. In particular, the paper shows that duplicate rows constitute a major *optimization inhibitor* (see Chapter 18). See also reference [6.3].

6.7 C. J. Date: "Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1)," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

Summarizes and criticizes Codd's "RM/V1" (see the annotation to reference [6.2]) and offers an alternative definition. The assumption is that it is crucially important to get "Version 1" right before we can even think about moving on to some "Version 2." *Note:* The version of the relational model described in the present book is based on the "reconstituted" version as sketched in this paper, and further clarified and described in reference [3.3].

6.8 C. J. Date: "A Critical Review of the Relational Model Version 2 (RM/V2)," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

Summarizes and criticizes Codd's "RM/V2" [6.2].

6.9 C. J. Date: *The Database Relational Model: A Retrospective Review and Analysis*. Reading, Mass.: Addison-Wesley (2001).

This short book (160 pages) is offered as a careful, unbiased, retrospective review and assessment of Codd's relational contribution as documented in his 1970s publications. To be specific, it examines the following papers in detail (as well as touching on several others in passing):

- "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks" (the first version of reference [6.1])
- "A Relational Model of Data for Large Shared Data Banks" [6.1]
- "Relational Completeness of Data Base Sublanguages" [7.1]
- "A Data Base Sublanguage Founded on the Relational Calculus" [8.1]
- "Further Normalization of the Data Base Relational Model" [11.6]
- "Extending the Relational Database Model to Capture More Meaning" [14.7]
- "Interactive Support for Nonprogrammers: The Relational and Network Approaches" [26.12]

6.10 Mark A. Roth, Henry F. Korth, and Abraham Silberschatz: "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS 13*, No. 4 (December 1988).

Many people over the years have proposed the possibility of supporting relation-valued attributes; this paper is a case in point. Such proposals usually go by the name of "NF^2 relations," where NF^2 (pronounced "NF squared") is short for NFNF and stands for "non first normal form." However, there are at least two major differences between such proposals and support for relation-valued attributes as described in this chapter:

- First, NF^2 relation advocates assume that relation-valued attributes are prohibited in the relational model and therefore advertise their proposals as "extensions" to that model (in this connection, note the title of reference [6.10]).
- Second, the NF^2 advocates are correct—they *are* extending the relational model! For example, Roth *et al.* propose an extended form of union that, in our terms, (a) ungroups both operands recursively until they involve no relation-valued attributes at all, either directly or indirectly; (b) performs a regular union on those ungrouped operands; and (c) finally, recursively (re)groups the result again. And it is the recursion that constitutes the extension. That is, while any *specific* extended union is shorthand for some *specific* combination of existing relational operators, it is not possible to say that extended union *in general* is just shorthand for some combination of existing operators.

CHAPTER 7

Relational Algebra

7.1 INTRODUCTION

The relational algebra is a collection of operators that take relations as their operands and return a relation as their result. The first version of the algebra was defined by Codd in references [5.1] and [7.1]; reference [7.1] in particular has come to be regarded as the source of the "original" algebra. That original algebra had eight operators, two groups of four each:

1. The traditional set operators *union*, *intersection*, *difference*, and *Cartesian product* (all of them modified somewhat to take account of the fact that their operands are, specifically, relations instead of arbitrary sets)
2. The special relational operators *restrict* (also known as *select*), *project*, *join*, and *divide*

Fig. 7.1 gives an informal picture of how these operators work.

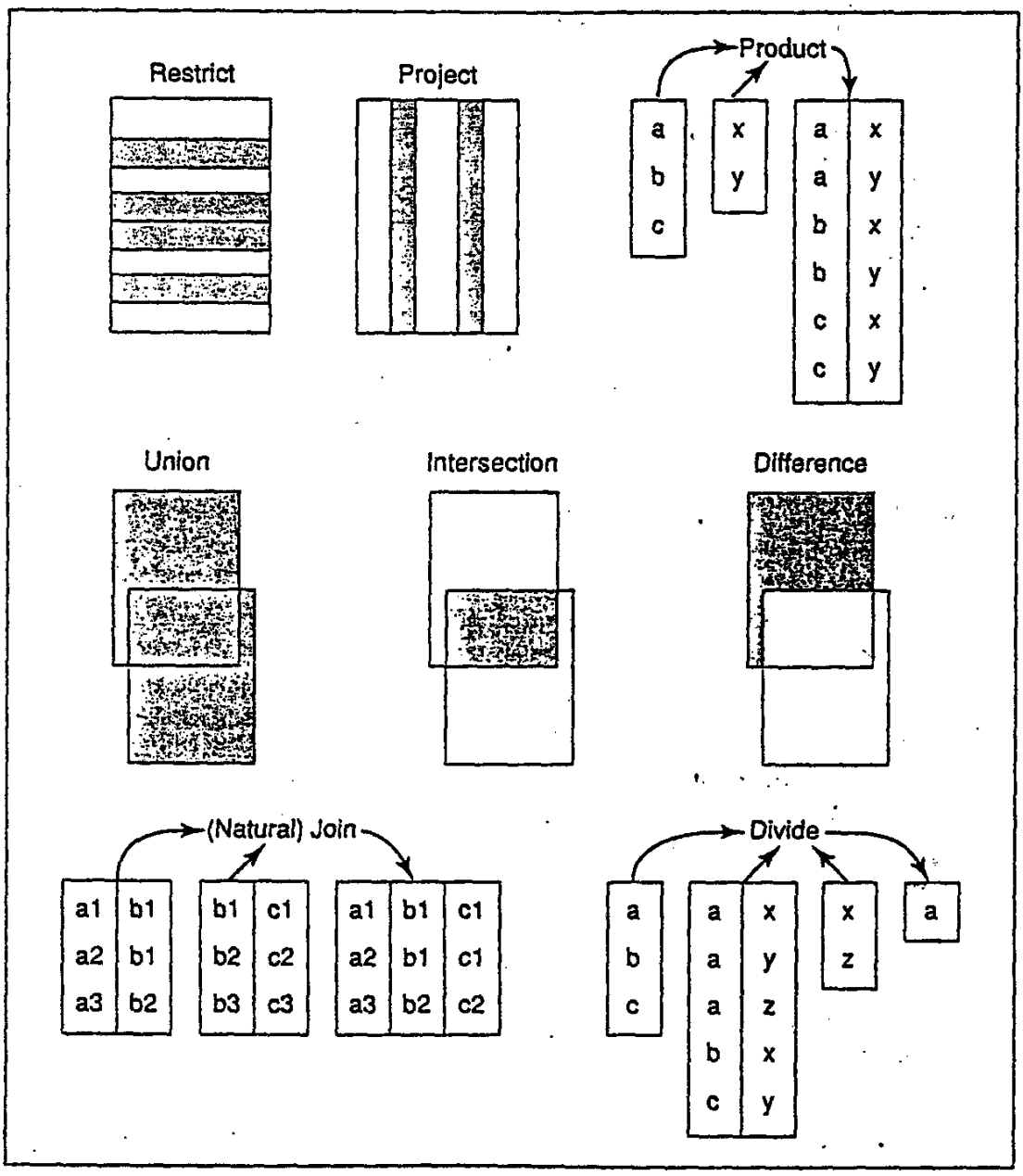


Fig. 7.1 The original eight operators (overview)

Now, Codd had a very specific purpose in mind, which we will examine in the next chapter, for defining just the eight operators he did. But those eight operators are not the end of the story; rather, *any number* of operators can be defined that satisfy the simple requirement of "relations in, relations out," and many additional operators have indeed been defined, by many different writers. In this chapter, we will discuss the original eight

operators first—not exactly as they were originally defined but as they have since become—and use them as the basis for discussing a variety of algebraic ideas; then we will go on to consider some of the many useful operators that have subsequently been added to the original set.

Before we can discuss the algebra in detail, however, there are a few more preliminary remarks we need to make:

- First of all, the operators apply to all relations, loosely speaking; in fact, they are really *generic* operators, associated with the RELATION type generator and hence applicable to any specific relation type obtained by invoking that type generator.
- Second, almost all of the operators we will be discussing are really just shorthand anyway! We will have more to say on this important point in Section 7.10.
- Third, the operators are all *read-only* (i.e., they "read" but do not update their operands). Thus, they apply specifically to *values*—relation values, of course—and hence, harmlessly, to those relation values that happen to be the current values of relvars.
- Last, it follows from the previous point that it makes sense to talk about, for example, "the projection over attribute *A* of relvar *R*," meaning the relation that results from taking the projection over that attribute *A* of the current value of that relvar *R*. Occasionally, however, it is convenient to use expressions like "the projection over attribute *A* of relvar *R*" in a slightly different sense. For example, suppose we define a view SC of the suppliers relvar S that consists of just the S# and CITY attributes of that relvar. Then we might say, loosely but very conveniently, that relvar SC is "the projection over S# and CITY of relvar S"—meaning, more precisely, that the value of SC at any given time is the projection over S# and CITY of the value of relvar S at that time. In a sense, therefore, we can talk in terms of projections of relvars *per se*, rather than just in terms of projections of current values of relvars. We hope this kind of dual usage of the terminology on our part does not cause any confusion.

The plan of the chapter is as follows. Following this introductory section, Section 7.2 revisits the question of relational closure and elaborates on it considerably. Sections 7.3 and 7.4 then discuss Codd's original eight operators in detail, and Section 7.5 gives examples of how those operators can be used to formulate queries. Next, Section 7.6 considers the more general question of what the algebra is for. Section 7.7 discusses a number of miscellaneous points. Then Section 7.8 describes some useful additions to Codd's original algebra, including in particular the important EXTEND and SUMMARIZE operators. Section 7.9 discusses operators for mapping between relations with relation-valued attributes and relations without such attributes. Finally, Section 7.10 offers a brief summary. *Note:* We defer discussion of the pertinent SQL facilities to Chapter 8, for reasons to be explained in that chapter.

7.2 CLOSURE REVISITED

As we saw in Chapter 3, the fact that the output from any given relational operation is another relation is referred to as the relational *closure* property. To recap briefly, closure

means we can write nested relational expressions—that is, relational expressions in which the operands are themselves represented by relational expressions of arbitrary complexity. (There is an obvious analogy between the ability to nest relational expressions in the relational algebra and the ability to nest arithmetic expressions in ordinary arithmetic; indeed, the fact that relations are closed under the algebra is important for exactly the same kinds of reasons that the fact that numbers are closed under ordinary arithmetic is important.)

Now, when we discussed closure in Chapter 3, there was one very significant point we deliberately glossed over. Recall that every relation has two parts, a heading and a body; loosely speaking, the heading is the attributes and the body is the tuples. The heading for a base relation—where, as you will recall from Chapter 5, a *base relation* is the value of a base relvar—is obviously known to the system, because it is specified as part of the relevant base relvar definition. But what about derived relations? For example, consider the expression

```
S JOIN P
```

(which represents the join of suppliers and parts over matching cities, CITY being the only attribute common to the two relations). We know what the body of the result looks like—but what about the heading? Closure dictates that it must *have* a heading, and the system needs to know what it is (in fact the user does too, as we will see in a moment). In other words, that result must—of course!—be of some well-defined relation type. If we are to take closure seriously, therefore, we need to define the relational operations in such a way as to guarantee that every operation produces a result with a proper relation type: in particular, with proper attribute names. (We remark in passing that this is an aspect of the algebra that has been much overlooked in the literature—and also, regrettably, in the SQL language and hence in SQL products—with the notable exception of the treatment found in references [7.2] and [7.10]. The algebra as presented in this chapter is very much influenced by these two references.)

One reason we require every result relation to have proper attribute names is to allow us to refer to those attributes in subsequent operations—in particular, in operations invoked elsewhere within the overall nested expression. For example, we could not sensibly even write an expression such as

```
( S JOIN P ) WHERE CITY = 'Athens'
```

if we did not know that the result of evaluating the expression S JOIN P had an attribute called CITY.

What we need, therefore, is a set of **relation type inference rules** built into the algebra, such that if we know the type(s) of the input relation(s) for any given relational operation, we can infer the type of the output from that operation. Given such rules, it will follow that an arbitrary relational expression, no matter how complex, will produce a result that also has a well-defined type, and in particular a well-defined set of attribute names.

As a preparatory step to achieving this goal, we introduce a new operator, RENAME, whose purpose is (loosely) to rename attributes within a specified relation. More precisely, the RENAME operator takes a given relation and returns another that is identical to the given one except that one of its attributes has a different name. (The given relation is

specified by means of a relational expression, possibly involving other relational operations.) For example, we might write:

```
S RENAME CITY AS SCITY
```

This expression—please note that it *is* an expression, not a "command" or statement, and hence that it can be nested inside other expressions—yields a relation with the same heading and body as the relation that is the current value of relvar S, except that the city attribute is named SCITY instead of CITY:

S#	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Important: Please note that this RENAME expression has *not* changed the suppliers relvar in the database! It is just an expression (exactly as, e.g., S JOIN SP is just an expression), and like any other expression it simply denotes a certain value—a value that, in this particular case, happens to look very much like the current value of the suppliers relvar.

Here is another example (a "multiple renaming" this time):

```
P RENAME ( PNAME AS PN, WEIGHT AS WT )
```

This expression is shorthand for the following:

```
( P RENAME PNAME AS PN ) RENAME WEIGHT AS WT
```

The result looks like this:

P#	PN	COLOR	WT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

It is worth noting that the availability of RENAME means that the relational algebra, unlike SQL, has no need for (and in fact does not support) dot-qualified attribute names such as S.S#.

7.3 THE ORIGINAL ALGEBRA: SYNTAX

In this section, we present a concrete syntax, based on Tutorial D, for relational algebra expressions that use the original eight operators, plus RENAME. The syntax is included here primarily for purposes of subsequent reference. A few notes on semantics are also

included. *Note:* Most database texts use a "mathematical" or "Greek" notation for the relational operators: σ for restriction ("selection"), π for projection, ∩ for intersection, ⋈ ("bow tie") for join, and so on. As you can see, we prefer to use keywords such as JOIN and WHERE. Keywords make for lengthier expressions, but we think they also make for more user-friendly ones.

```
<relation exp>
    ::=   RELATION { <tuple exp commalist> }
        | <relvar name>
        | <relation op inv>
        | <with exp>
        | <introduced name>
        | ( <relation exp> )
```

A *<relation exp>* is an expression that denotes a relation (i.e., a relation *value*). The first format is a relation selector invocation (see Chapter 6); we do not spell out the syntax of *<tuple exp>* in detail here, since examples should be sufficient to give the general idea. The *<relvar name>* and (*<relation exp>*) formats are self-explanatory; the others are explained in what follows.

```
<relation op inv>
    ::=   <project> | <nonproject>
```

A relational operator invocation, *<relation op inv>*, is either a *<project>* or a *<nonproject>*. *Note:* We distinguish these two cases in the syntax merely for operator precedence reasons (it is convenient to assign a high precedence to projection).

```
<project>
    ::=   <relation exp>
                { [ ALL BUT ] <attribute name commalist> }
```

The *<relation exp>* must not be a *<nonproject>*.

```
<nonproject>
    ::=   <rename> | <union> | <intersect> | <minus> | <times>
        | <where> | <join> | <divide>

<rename>
    ::=   <relation exp> RENAME ( <renaming commalist> )
```

The *<relation exp>* must not be a *<nonproject>*. The individual *<renaming>*s are executed in sequence as written (for the syntax of *<renaming>*, see the examples in the previous section). The parentheses can be omitted if the commalist contains just one *<renaming>*.

```
<union>
    ::=   <relation exp> UNION <relation exp>
```

The *<relation exp>*s must not be *<nonproject>*s, except that either or both can be another *<union>*.

```
<intersect>
    ::=   <relation exp> INTERSECT <relation exp>
```

The *<relation exp>*s must not be *<nonproject>*s, except that either or both can be another *<intersect>*.

```
<minus>
    ::=   <relation exp> MINUS <relation exp>
```

The *<relation exp>*s must not be *<nonproject>*s.

```
<times>
    ::=   <relation exp> TIMES <relation exp>
```

The *<relation exp>*s must not be *<nonproject>*s, except that either or both can be another *<times>*.

```
<where>
    ::=   <relation exp> WHERE <bool exp>
```

The *<relation exp>* must not be a *<nonproject>*. The *<bool exp>* can include references to attributes of the relation denoted by the *<relation exp>*, with the obvious semantics.

```
<join>
    ::=   <relation exp> JOIN <relation exp>
```

The *<relation exp>*s must not be *<nonproject>*s, except that either or both can be another *<join>*.

```
<divide>
    ::=   <relation exp> DIVIDEBY <relation exp> PER <per>
```

The *<relation exp>*s must not be *<nonproject>*s.

```
<per>
    ::=   <relation exp> | ( <relation exp>, <relation exp> )
```

The *<relation exp>*s must not be *<nonproject>*s.

```
<with exp>
    ::=   WITH <name intro commalist> : <exp>
```

The *<with exp>*s we are primarily concerned with in this book are relational expressions specifically, which is why we are discussing them in this chapter. However, scalar and tuple *<with exp>*s are supported too: in fact, a given *<with exp>* is a *<relation exp>*, a *<tuple exp>*, or a *<scalar exp>* according as the *<exp>* after the colon is a *<relation exp>*, a *<tuple exp>*, or a *<scalar exp>* in turn. In all cases, the individual *<name intro>*s are executed in sequence as written, and the semantics of the *<with exp>* are defined to be the same as those of a version of *<exp>* in which each occurrence of each introduced name is replaced by a reference to a variable whose value is the result of evaluating the corresponding expression. *Note:* WITH is not really an operator of the relational algebra as such; it is really just a device to help with the formulation of what otherwise might be rather complicated expressions (especially ones involving common subexpressions). Several examples are given in Section 7.5.

```
<name intro>
    ::=   <exp> AS <introduced name>
```

The *<introduced name>* can be used within the containing *<with exp>* wherever the *<exp>* (enclosed in parentheses if necessary) would be allowed.

7.4 THE ORIGINAL ALGEBRA: SEMANTICS

Union

In mathematics, the union of two sets is the set of all elements belonging to either or both of the given sets. Since a relation is—or, rather, contains—a set (namely, a set of tuples), it is obviously possible to construct the union of two such sets; the result will be a set consisting of all tuples appearing in either or both of the given relations. For example, the union of the set of supplier tuples currently appearing in relvar S and the set of part tuples currently appearing in relvar P is certainly a set.

However, although that result is a set, *it is not a relation;* relations cannot contain a mixture of different kinds of tuples, they must be "tuple-homogeneous." And we do want the result to be a relation, because we want to preserve the closure property. Therefore, the union in the relational algebra is not the completely general mathematical union; rather, it is a special kind of union, in which we require the two input relations to be of the same type—meaning, for example, that they both contain supplier tuples, or both part tuples, but not a mixture of the two. If the two relations are of the same type, then we can take their union, and the result will also be a relation of the same type; in other words, the closure property will be preserved. *Note:* Historically, much of the database literature (earlier editions of this book included) used the term *union compatibility* to refer to the notion that two relations must be of the same type. This term is not very apt, however, for a variety of reasons, the most obvious of which is that the notion does not apply just to union.

Here, then, is a definition of the relational union operator: Given two relations *a* and *b* of the same type, the uuion of those two relations, *a* UNION *b*, is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in *a* or *b* or both.

Example: Let relations A and B be as shown in Fig. 7.2 opposite (both are derived from the current value of the suppliers relvar S; A is the suppliers in London, and B is the suppliers who supply part P1, intuitively speaking). Then A UNION B (see part 1 of the figure) is the suppliers who either are located in London or supply part P1, or both. Notice that the result has three tuples, not four; relations never contain duplicate tuples, by definition (we say, loosely, that union "eliminates duplicates"). We remark in passing that the only other operation of the original eight for which this question of duplicate elimination arises is projection (see later in this section).

By the way, observe how the definition of union relies on the concept of tuple equality. Here is a different but equivalent definition that makes the point very clear (the revised text is highlighted): Given two relations *a* and *b* of the same type, the union of those two relations, *a* UNION *b*, is a relation of the same type, with body consisting of all tuples *t* such that ***t is equal to (i.e., is a duplicate of) some tuple*** in *a* or *b* or both. Analogous remarks apply directly to the intersect and difference operations, as you will soon see.

Intersect

Like union, and for essentially the same reason, the relational intersection operator requires its operands to be of the same type. Given two relations *a* and *b* of the same type,

A

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

B

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

1. *Union*
(A UNION B)

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris

2. *Intersection*
(A INTERSECT B)

S#	SNAME	STATUS	CITY
S1	Smith	20	London

3. *Difference*
(A MINUS B)

S#	SNAME	STATUS	CITY
S4	Clark	20	London

4. *Difference*
(B MINUS A)

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris

Fig. 7.2 Union, intersection, and difference examples

then, the intersection of those two relations, *a* INTERSECT *b*, is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in both *a* and *b*.

Example: Again, let A and B be as shown in Fig. 7.2. Then A INTERSECT B (see part 2 of the figure) is the suppliers who are located in London and supply part P1.

Difference

Like union and intersection, the relational difference operator also requires its operands to be of the same type. Given two relations *a* and *b* of the same type, then, the difference between those two relations, *a* MINUS *b* (in that order), is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in *a* and not *b*.

Example: Let A and B again be as shown in Fig. 7.2. Then A MINUS B (see part 3 of the figure) is the suppliers who are located in London and do not supply part P1, and B MINUS A (see part 4 of the figure) is the suppliers who supply part P1 and are not located in London. Observe that MINUS has a directionality to it, just as subtraction does in ordinary arithmetic (e.g., "5 – 2" and "2 – 5" are not the same thing).

Product

In mathematics, the Cartesian product (prodnct for short) of two sets is the set of all ordered pairs such that, in each pair, the first element comes from the first set and the second element comes from the second set. Thus, the Cartesian product of two relations

would be a set of ordered pairs of *tuples*, loosely speaking. But again we want to preserve the closure property; in other words, we want the result to contain tuples *per se*, not *ordered pairs* of tuples. Therefore, the relational version of Cartesian product is an *extended form* of the operation, in which each ordered pair of tuples is replaced by the single tuple that is the *union* of the two tuples in question (using "union" in its normal set theory sense, not its special relational sense). That is, given the tuples[1]

```
{ A1 a1, A2 a2, ..., Am am }
```

and

```
{ B1 b1, B2 b2, ..., Bn bn }
```

the union of the two is the single tuple

```
{ A1 a1, A2 a2, ..., Am am, B1 b1, B2 b2, ..., Bn bn }
```

Note: We are assuming for simplicity here that the two tuples have no attribute names in common. The paragraph immediately following elaborates on this point.

Another problem that occurs in connection with Cartesian product is that, of course, we require the result relation to have a well-formed heading (i.e., to be of a proper relation type). Now, clearly the heading of the result consists of all of the attributes from both of the two input headings. A problem will therefore arise if the two input headings have any attribute names in common; if we were allowed to form the product, the result heading would then have two attributes with the same name and so would not be well-formed. If we need to construct the Cartesian product of two relations that do have any such common attribute names, therefore, we must use the RENAME operator first to rename attributes appropriately.

We therefore define the (relational) Cartesian product of two relations a and b, a TIMES b, where a and b have no common attribute names, to be a relation with a heading that is the (set theory) union of the headings of a and b and with a body consisting of the set of all tuples t such that t is the (set theory) union of a tuple appearing in a and a tuple appearing in b. Note that the cardinality of the result is the product of the cardinalities, and the degree of the result is the sum of the degrees, of the input relations a and b.

Example: Let relations A and B be as shown in Fig. 7.3 opposite (A is all current supplier numbers and B is all current part numbers, intuitively speaking). Then A TIMES B—see the lower part of the figure—is all current supplier-number/part-number pairs.

Restrict

Let relation a have attributes X and Y (and possibly others), and let θ be an operator—typically "=", "≠", ">", "<", and so on—such that the boolean expression $X\ \theta\ Y$ is well-formed and, given particular values for X and Y, evaluates to a truth value (TRUE or FALSE). Then the θ-restriction, or just *restriction* for short, of relation a on attributes X and Y (in that order)—

```
a WHERE X θ Y
```

[1] Tutorial D would require the keyword TUPLE to appear in front of each of these expressions.

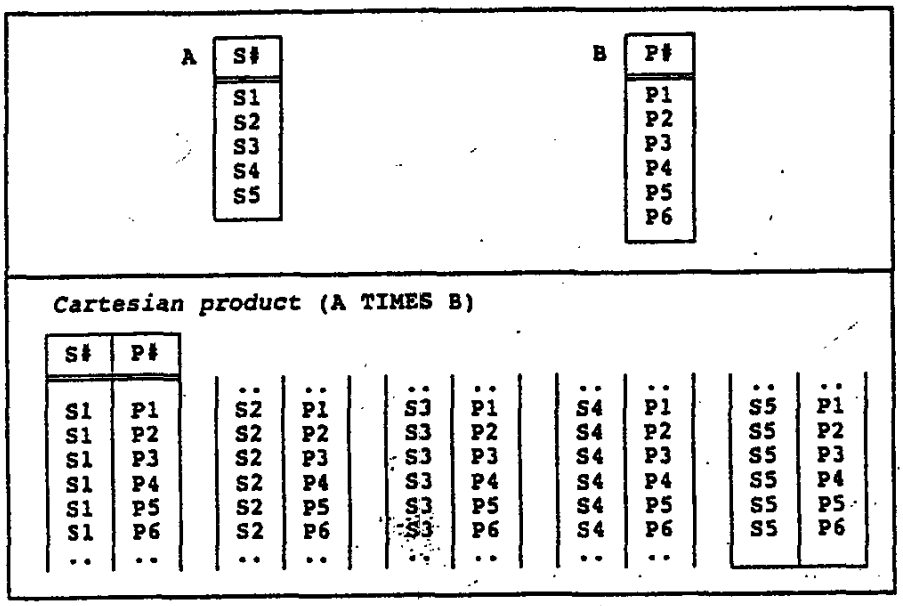


Fig. 7.3 Cartesian product example

—is a relation with the same heading as *a* and with body consisting of all tuples of *a* such that the expression *X θ Y* evaluates to TRUE for the tuple in question.

Note: The foregoing is essentially the definition of restriction given in most of the literature (including earlier editions of this book). However, it is possible to generalize it, and we will, as follows. Let relation *a* have attributes *X*, *Y*, ..., *Z* (and possibly others), and let *p* be a truth-valued function whose parameters are, precisely, some subset of *X*, *Y*, ..., *Z*. Then the restriction of *a* according to *p*—

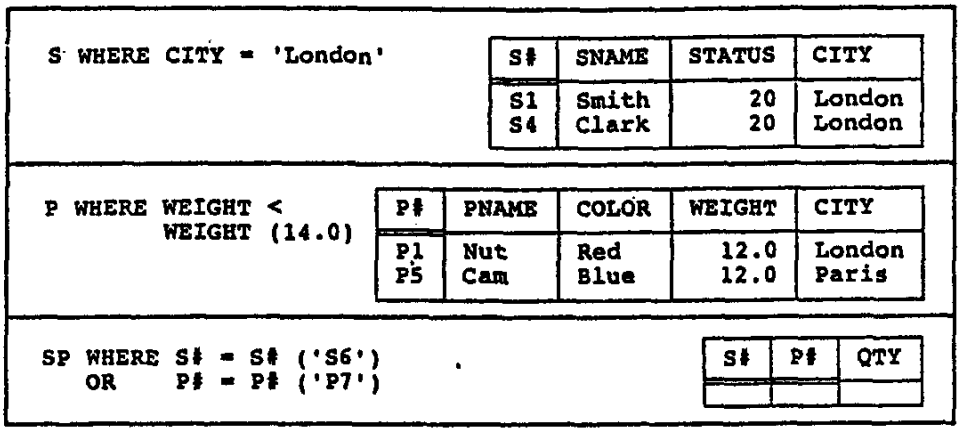
```
a WHERE p
```

—is a relation with the same heading as *a* and with body consisting of all tuples of *a* such that *p* evaluates to TRUE for the tuple in question.

The restriction operator effectively yields a "horizontal" subset of a given relation: that is, that subset of the tuples of the given relation for which some specified condition is satisfied. Some examples (all of them illustrating the generalized version of restriction as just defined) are given in Fig. 7.4, overleaf.

Points arising:

1. The expression *p* following the keyword WHERE is, of course, a boolean expression: in fact, it is a *predicate*, in a sense to be discussed in detail in Chapter 9.
2. We refer to that predicate as a **restriction condition**. If that condition is such that it can be evaluated for a given tuple *t* without examining any tuple other than *t* (and hence *a fortiori* without examining any relation other than *a*), then it is a simple restriction condition. All of the restriction conditions in Fig. 7.4 are simple in this sense. Here by contrast is an example that involves a nonsimple restriction condition:



```
S WHERE CITY = 'London'
```

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

```
P WHERE WEIGHT <
        WEIGHT (14.0)
```

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P5	Cam	Blue	12.0	Paris

```
SP WHERE S# = S# ('S6')
   OR    P# = P# ('P7')
```

S#	P#	QTY

Fig. 7.4 Restriction examples

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

We will examine this example in detail later in this section, following the discussion of divide.

3. The following equivalences are worthy of note:

```
a WHERE p1 OR p2   =  ( a WHERE p1 ) UNION ( a WHERE p2 )
a WHERE p1 AND p2  =  ( a WHERE p1 ) INTERSECT ( a WHERE p2 )
a WHERE NOT ( p )  =  a MINUS ( a WHERE p )
```

Project

Let relation *a* have attributes *X*, *Y*, ..., *Z* (and possibly others). Then the projection of relation *a* on *X*, *Y*, ..., *Z*—

```
a { X, Y, ..., Z }
```

—is a relation with:

- A heading derived from the heading of *a* by removing all attributes not mentioned in the set { *X*, *Y*, ..., *Z* }
- A body consisting of all tuples { *X x*, *Y y*, ..., *Z z* } such that a tuple appears in *a* with *X* value *x*, *Y* value *y*, ..., and *Z* value *z*

The projection operator thus effectively yields a "vertical" subset of a given relation: namely, that subset obtained by removing all attributes not mentioned in the specified commalist of attribute names and then eliminating duplicate (sub)tuples from what is left.

Points arising:

1. No attribute can be mentioned more than once in the attribute name commalist (why not?).
2. In practice, it is often convenient to be able to specify, not the attributes over which the projection is to be taken, but rather the ones that are to be "projected away" (i.e.,

removed). Instead of saying "project relation P over the P#, PNAME, COLOR, and CITY attributes," for example, we might say "project the WEIGHT attribute away from relation P," as here:

```
P { ALL BUT WEIGHT }
```

Further examples are given in Fig. 7.5. Notice in the first one (the projection of suppliers over CITY) that, although relvar S currently contains five tuples, there are only three tuples in the result ("duplicates are eliminated"). Analogous remarks apply to the other examples also, of course. Note too the reliance on tuple equality once again.

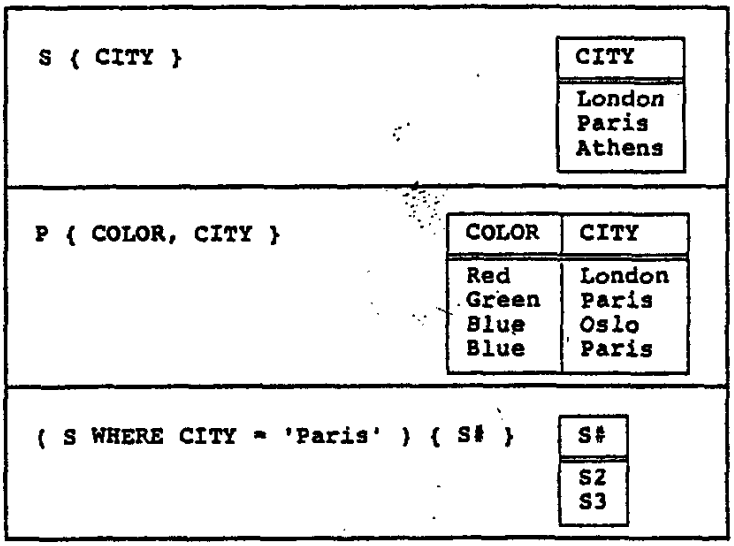


Fig. 7.5 Projection examples

Join

Join comes in several different varieties. Easily the most important, however, is the so-called *natural* join—so much so, in fact, that the unqualified term *join* is almost always taken to mean the natural join specifically, and we adopt that usage in this book. Here then is the definition (it is a little abstract, but you should already be familiar with natural join at an intuitive level from our discussions in Chapter 3). Let relations a and b have attributes

$$X1, X2, \ldots, Xm, Y1, Y2, \ldots, Yn$$

and

$$Y1, Y2, \ldots, Yn, Z1, Z2, \ldots, Zp$$

respectively; that is, the Y attributes $Y1, Y2, \ldots, Yn$ (only) are common to the two relations, the X attributes $X1, X2, \ldots, Xm$ are the other attributes of a, and the Z attributes $Z1, Z2, \ldots, Zp$ are the other attributes of b. Observe that:

- We can and do assume without loss of generality, thanks to the availability of the attribute RENAME operator, that no attribute Xi ($i = 1, 2, ..., m$) has the same name as any attribute Zj ($j = 1, 2, ..., p$).
- Every attribute Yk ($k = 1, 2, ..., n$) has the same type in both a and b (for otherwise it would not be a common attribute, by definition).

Now consider { *X1, X2, ... Xm* }, { *Y1, Y2, ..., Yn* }, and { *Z1, Z2, ..., Zp* } as three *composite* attributes *X, Y,* and *Z,* respectively. Then the (natural) join of *a* and *b*—

```
a JOIN b
```

—is a relation with heading { *X, Y, Z* } and body consisting of all tuples { *X x, Y y, Z z* } such that a tuple appears in *a* with *X* value *x* and *Y* value *y* and a tuple appears in *b* with *Y* value *y* and *Z* value *z*.

An example of a natural join (the natural join S JOIN P, over the common attribute CITY) is given in Fig. 7.6.

Note: We have illustrated the point several times—indeed, it is illustrated by Fig. 7.6—but it is still worth stating explicitly that joins are *not* always between a foreign key and a matching primary (or candidate) key, even though such joins are a very common and important special case.

Incidentally, note how the definition of natural join relies on tuple equality yet again. Note too with respect to that definition that:

- If $n = 0$ (meaning *a* and *b* have no common attributes), then *a* JOIN *b* degenerates to *a* TIMES *b*.[2]
- If $m = p = 0$ (meaning *a* and *b* are of the same type), then *a* JOIN *b* degenerates to *a* INTERSECT *b*.

Now we turn to the *θ-join* operation. This operation is intended for those occasions (comparatively rare, but by no means unknown) where we need to join two relations on the basis of some comparison operator other than equality. Let relations *a* and *b* satisfy the

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12.0
S1	Smith	20	London	P4	Screw	Red	14.0
S1	Smith	20	London	P6	Cog	Red	19.0
S2	Jones	10	Paris	P2	Bolt	Green	17.0
S2	Jones	10	Paris	P5	Cam	Blue	12.0
S3	Blake	30	Paris	P2	Bolt	Green	17.0
S3	Blake	30	Paris	P5	Cam	Blue	12.0
S4	Clark	20	London	P1	Nut	Red	12.0
S4	Clark	20	London	P4	Screw	Red	14.0
S4	Clark	20	London	P6	Cog	Red	19.0

Fig. 7.6 The natural join S JOIN P

[2] The version of Tutorial D defined in reference [3.3] includes no direct support for the TIMES operator for this very reason.

requirements for Cartesian product (i.e., they have no attribute names in common); let *a* have an attribute *X* and let *b* have an attribute *Y*; and let *X*, *Y*, and θ satisfy the requirements for θ-restriction. Then the θ-join of relation *a* on attribute *X* with relation *b* on attribute *Y* is defined to be the result of evaluating the expression:

```
( a TIMES b ) WHERE X θ Y
```

In other words, it is a relation with the same heading as the Cartesian product of *a* and *b*, and with a body consisting of the set of all tuples *t* such that *t* appears in that Cartesian product and the expression *X θ Y* evaluates to TRUE for that tuple *t*.

By way of example, suppose we wish to compute the *greater-than join* of relation S on CITY with relation P on CITY (so θ here is ">"; since the CITY attributes are defined to be of type CHAR, ">" simply means "greater in alphabetic ordering"). An appropriate relational expression is as follows:

```
( ( S RENAME CITY AS SCITY ) TIMES
  ( P RENAME CITY AS PCITY ) )
WHERE SCITY > PCITY
```

Note the attribute renaming in this example. (Of course, it would be sufficient to rename just one of the two CITY attributes; the only reason for renaming both is symmetry.) The result of the overall expression is shown in Fig. 7.7.

If θ is "=", the θ-join is called an equijoin. It follows from the definition that the result of an equijoin must include two attributes with the property that the values of those two attributes are equal in every tuple in the relation. If one of those two attributes is projected away and the other renamed appropriately (if necessary), the result is the natural join! For example, the expression representing the natural join of suppliers and parts (over cities)—

```
S JOIN P
```

—is equivalent to the following more complex expression:

```
( ( S TIMES ( P RENAME CITY AS PCITY ) )
                        WHERE CITY = PCITY )
                              { ALL BUT PCITY }
```

Note: Tutorial D does not include direct support for the θ-join operator because (a) it is not needed very often in practice and in any case (b) it is not a primitive operator (i.e., it can be defined in terms of other operators, as we have seen).

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P3	Screw	Blue	17.0	Oslo
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P3	Screw	Blue	17.0	Oslo
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London

Fig. 7.7 Greater-than join of suppliers and parts on cities

Divide

Reference [7.4] defines two distinct "divide" operators that it calls the Small Divide and the Great Divide, respectively. In Tutorial D, a *<divide>* in which the *<per>* consists of just one *<relation exp>* is a Small Divide, a *<divide>* in which it consists of a parenthesized commalist of two *<relation exp>*s is a Great Divide. The description that follows applies to the Small Divide only, and only to a particular limited form of the Small Divide at that; see reference [7.4] for a discussion of the Great Divide and for further details regarding the Small Divide as well.

We should say too that the version of the Small Divide as discussed here is not the same as Codd's original operator; in fact, it is an improved version that overcomes certain difficulties that arose with that original operator in connection with empty relations. It is also not the same as the version discussed in the first few editions of this book.

Here then is the definition. Let relations *a* and *b* have attributes

```
X1, X2, ..., Xm
```

and

```
Y1, Y2, ..., Yn
```

respectively, where no attribute *Xi* (i = 1, 2, ..., m) has the same name as any attribute *Yj* (j = 1, 2, ..., n), and let relation *c* have attributes

```
X1, X2, ..., Xm, Y1, Y2, ..., Yn
```

(i.e., *c* has a heading that is the union of the headings of *a* and *b*). Let us now regard { *X1*, *X2*, ..., *Xm* } and { *Y1*, *Y2*, ..., *Yn* } as *composite* attributes *X* and *Y*, respectively. Then the division of *a* by *b* per *c* (where *a* is the dividend, *b* is the divisor, and *c* is the "mediator")—

```
a DIVIDEBY b PER c
```

—is a relation with heading { *X* } and body consisting of all tuples { *X x* } appearing in *a* such that a tuple { *X x*, *Y y* } appears in *c* for *all* tuples { *Y y* } appearing in *b*. In other words, the result consists of those *X* values from *a* whose corresponding *Y* values in *c* include all *Y* values from *b*, loosely speaking. Note the reliance on tuple equality yet again!

Fig. 7.8 shows some examples of division. The dividend (DEND) in each case is the projection of the current value of relvar S over S#; the mediator (MED) in each case is the projection of the current value of relvar SP over S# and P#; and the three divisors (DOR) are as indicated in the figure. Notice the last example in particular, in which the divisor is a relation containing part numbers for all currently known parts: the result (obviously enough) shows supplier numbers for suppliers who supply all of those parts. As this example suggests, the DIVIDEBY operator is intended for queries of this general nature; in fact, whenever the natural language version of the query contains the word "all" in the conditional part ("Get suppliers who supply *all* parts"), there is a strong likelihood that division will be involved. (Indeed, division was specifically intended by Codd to be an algebraic counterpart to the *universal quantifier*, much as projection was intended to be an algebraic counterpart to the *existential* quantifier. See Chapter 8 for further explanation.)

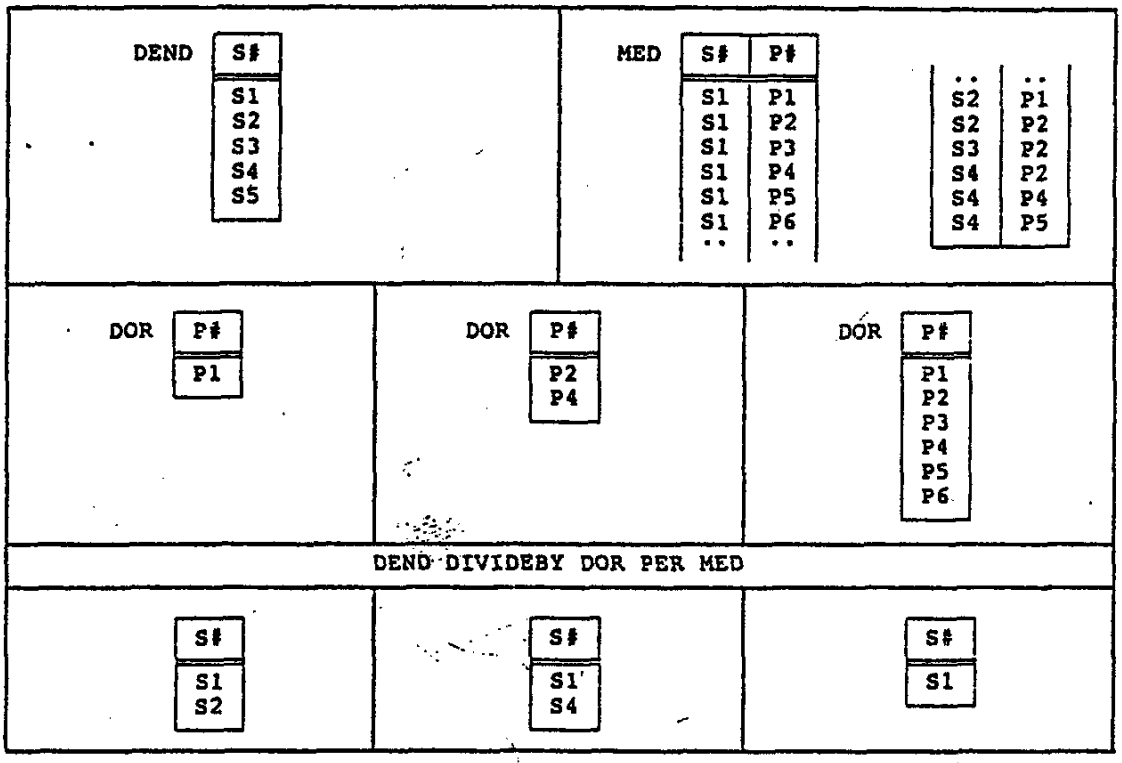


Fig. 7.8 Division examples

In connection with that last example, however, we should point out that queries of that general nature are often more readily expressed in terms of *relational comparisons*. For example:

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

This expression evaluates to a relation containing all and only the supplier tuples for suppliers who supply all currently known parts. *Explanation:*

1. For a given supplier, the expression

   ```
   ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# }
   ```

   yields the set of part numbers for parts supplied by that supplier.
2. That set of part numbers is then compared with the set of all currently known part numbers.
3. If and only if the two sets are equal, the corresponding supplier tuple appears in the result.

Here by contrast is the DIVIDEBY version, now spelled out in detail:

```
S JOIN ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
```

You might well feel that the relational comparison version is conceptually easier to deal with. In fact, there is some doubt as to whether DIVIDEBY would ever have been defined if the relational model had included relational comparisons in the first place—but it did not.

7.5 EXAMPLES

In this section we present a few examples of the use of relational algebra expressions in formulating queries. We recommend that you check these examples against the sample data of Fig. 3.8 (see the inside back cover).

7.5.1 Get supplier names for suppliers who supply part P2:

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

Explanation: First the join of relations SP and S over supplier numbers is constructed, which has the effect, conceptually, of extending each SP tuple with the corresponding supplier information (i.e., the appropriate SNAME, STATUS, and CITY values). That join is then restricted to just those tuples for part P2. Finally, that restriction is projected over SNAME. The final result has just one attribute, called SNAME.

7.5.2 Get supplier names for suppliers who supply at least one red part:

```
( ( ( P WHERE COLOR = COLOR ('Red') )
                      JOIN SP ) { S# } JOIN S ) { SNAME }
```

The sole attribute of the result is SNAME again.

Here by the way is a different formulation of the same query:

```
( ( ( P WHERE COLOR = COLOR ('Red') ) { P# }
                      JOIN SP ) JOIN S ) { SNAME }
```

This example thus illustrates the important point that there will often be several different ways of formulating any given query. See Chapter 18 for a discussion of some of the implications of this point.

7.5.3 Get supplier names for suppliers who supply all parts:

```
( ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
                                          JOIN S ) { SNAME }
```

Or:

```
( S WHERE
( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# } )
{ SNAME }
```

Once again the result has a sole attribute called SNAME.

7.5.4 Get supplier numbers for suppliers who supply at least all those parts supplied by supplier S2:

```
S { S# } DIVIDEBY ( SP WHERE S# = S# ('S2') ) { P# }
                                          PER SP { S#, P# }
```

The result has a sole attribute called S#.

7.5.5 Get all pairs of supplier numbers such that the suppliers concerned are "colocated" (i.e., located in the same city):

```
( ( ( S RENAME S# AS SA ) ( SA, CITY ) JOIN
    ( S RENAME S# AS SB ) ( SB, CITY ) )
      WHERE SA < SB ) ( SA, SB )
```

The result here has two attributes, called SA and SB (actually it would be sufficient to rename just one of the two S# attributes; we have renamed both for symmetry). We have assumed that the operator "<" has been defined for type S#. The purpose of the restriction condition SA < SB is twofold:

- It eliminates pairs of supplier numbers of the form (x,x).
- It guarantees that the pairs (x,y) and (y,x) will not both appear.

We show another formulation of this query to show how WITH can be used to simplify the business of writing what otherwise might be rather complicated expressions:[3]

```
WITH ( S RENAME S# AS SA ) ( SA, CITY ) AS T1,
     ( S RENAME S# AS SB ) ( SB, CITY ) AS T2,
     T1 JOIN T2 AS T3,
     T3 WHERE SA < SB AS T4 :
     T4 ( SA, SB )
```

WITH allows us to think about large, complicated expressions in a kind of step-at-a-time fashion, and yet it does not in any way violate the nonprocedurality of the relational algebra. We will expand on this point in the discussion following the next example.

7.5.6 Get supplier names for suppliers who do not supply part P2:

```
( ( S { S# } MINUS ( SP WHERE P# = P# ('P2') ) { S# } )
                                          JOIN S ) { SNAME }
```

The result has a sole attribute called SNAME.

As promised, we elaborate on this example in order to illustrate another point. It is not always easy to see immediately how to formulate a given query as a single nested expression. Nor should it be necessary to do so, either. Here is a step-at-a-time formulation of the example:

```
WITH S { S# } AS T1,
     SP WHERE P# = P# ('P2') AS T2,
     T2 { S# } AS T3,
     T1 MINUS T3 AS T4,
     T4 JOIN S AS T5,
     T5 { SNAME } AS T6 :
     T6
```

T6 denotes the desired result. *Explanation:* Names introduced by a WITH clause—that is, names of the form *Ti*, in the example—are assumed to be local to the statement containing that clause. Now, if the system supports "lazy evaluation" (as, for example, the PRTV system did [7.9]), then breaking the overall query down into a sequence of steps in this fashion need have *no* undesirable performance implications. Instead, the query can be processed as follows:

[3] In fact, we used the *scalar* form of WITH in the definition of operator DIST in Chapter 5, Section 5.5; we also showed the relational form in the expansion of the UPDATE shorthand in Chapter 6, Section 6.5.

- The expressions preceding the colon require no immediate evaluation by the system—all the system has to do is remember them, along with the names introduced by the corresponding AS clauses.
- The expression following the colon denotes the final result of the query (in the example, that expression is just "T6"). When it reaches this point, the system cannot delay evaluation any longer but instead must somehow compute the desired value (i.e., the value of T6).
- In order to evaluate T6, which is the projection of T5 over SNAME, the system must first evaluate T5; in order to evaluate T5, which is the join of T4 and S, the system must first evaluate T4; and so on. In other words, the system effectively has to evaluate the original nested expression, exactly as if the user had written that nested expression in the first place.

See the next section for a brief discussion of the general question of evaluating such nested expressions, and Chapter 18 for an extended treatment of the same topic.

7.6 WHAT IS THE ALGEBRA FOR?

To summarize this chapter so far: We have defined a *relational algebra*, that is, a collection of operations on relations. The operations in question are union, intersect, difference, product, restrict, project, join, and divide, plus an attribute renaming operator, RENAME (this is essentially the set that Codd originally defined in reference [7.1], except for RENAME). We have also presented a syntax for those operations, and used that syntax as a basis for a number of examples and illustrations.

As our discussions have implied, however, Codd's eight operators do not constitute a *minimal* set (nor were they ever meant to), because some of them are not primitive—they can be defined in terms of the others. For example, the operators join, intersect, and divide can be defined in terms of the other five (see Exercise 7.6), and they could therefore be dropped without any loss of functionality. Of the remaining five, however, none can be defined in terms of the other four, so we can regard those five as constituting a primitive or minimal set (not necessarily the only one, please note).[4] In practice, however, the other operators (especially join) are so useful that a good case can be made for supporting them directly.

We are now in a position to clarify an important point. Although we never said as much explicitly, the body of the chapter thus far has certainly suggested that the primary purpose of the algebra is merely *data retrieval*. Such is not the case, however. The funda-

[4] This sentence requires a certain amount of qualification. First, since we have seen that product is a special case of join, we could replace product by join in the stated set of primitives. Second, we really need to include RENAME, because our algebra (unlike that of reference [7.1]) relies on attribute naming instead of ordinal position. Third, reference [3.3] describes a kind of "reduced instruction set" version of the algebra, called A, that allows the entire functionality of Codd's original algebra (as well as RENAME and several other useful operators) to be achieved with just two primitives, called *remove* and *nor*.

mental intent of the algebra is to allow the writing of relational expressions. Those expressions in turn are intended to serve a variety of purposes, including retrieval but not limited to retrieval alone. The following list—which is not meant to be exhaustive—indicates some possible applications for such expressions:

- Defining a scope for retrieval—that is, defining the data to be fetched in some retrieval operation (as already discussed at length)
- Defining a scope for update—that is, defining the data to be inserted, changed, or deleted in some update operation (see Chapter 6)
- Defining integrity constraints—that is, defining some constraint that the database must satisfy (see Chapter 9)
- Defining derived relvars—that is, defining the data to be included in a view or snapshot (see Chapter 10)
- Defining stability requirements—that is, defining the data that is to be the scope of some concurrency control operation (see Chapter 16)
- Defining security constraints—that is, defining the data over which authorization of some kind is to be granted (see Chapter 17)

In general, in fact, the expressions serve as *a high-level, symbolic representation of the user's intent* (with regard to some particular query, for example). And precisely because they are high-level and symbolic, they can be subjected to a variety of high-level, symbolic transformation rules. For example, the expression

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

("Get supplier names for suppliers who supply part P2"—Example 7.5.1) can be transformed into the logically equivalent but probably more efficient expression

```
( ( SP WHERE P# = P# ('P2') ) JOIN S ) { SNAME }
```

(*Exercise:* In what sense is the second expression probably more efficient? Why only "probably"?)

The algebra thus serves as a convenient basis for optimization (refer back to Chapter 3, Section 3.5, if you need to refresh your memory regarding this notion). Thus, even if the user states the query in the first of the two forms just shown, the optimizer should convert it into the second form before executing it (ideally, the performance of a given query should *not* depend on the particular form in which the user happens to state it). See Chapter 18 for further discussion.

We conclude this section by noting that, precisely because of its fundamental nature, the algebra is often used as a kind of *yardstick* against which the expressive power of some given language can be measured. Basically, a language is said to be relationally complete [7.1] if it is at least as powerful as the algebra—that is, if its expressions permit the definition of every relation that can be defined by means of expressions of the algebra (the *original* algebra, that is, as described in previous sections). We will examine this notion of relational completeness in more detail in the next chapter.

7.7 FURTHER POINTS

This section covers a few miscellaneous issues related to the original eight operators.

Associativity and Commutativity

It is easy to verify that UNION is associative—that is, if *a*, *b*, and *c* are arbitrary relations of the same type, then the expressions

```
( a UNION b ) UNION c
```

and

```
a UNION ( b UNION c )
```

are logically equivalent. For convenience, therefore, we allow a sequence of UNIONs to be written without any parentheses; as a consequence, each of the foregoing expressions can be unambiguously abbreviated to just

```
a UNION b UNION c
```

Analogous remarks apply to INTERSECT, TIMES, and JOIN (but not MINUS). *Note:* For such reasons among others, some kind of prefix notation, as in, for example, UNION (*a*,*b*,*c*), might be preferable in practice to the infix style used in Tutorial D. But we stay with that infix style in this book.

UNION, INTERSECT, TIMES, and JOIN (but not MINUS) are also commutative—that is, the expressions

```
a UNION b
```

and

```
b UNION a
```

are also logically equivalent, and similarly for INTERSECT, TIMES, and JOIN.

We will revisit the whole question of associativity and commutativity in Chapter 18. Regarding TIMES, incidentally, we note that the set theory version of Cartesian product is neither associative nor commutative, but (as we have just seen) the relational version is both.

Some Equivalences

In this subsection we simply list a few important equivalences, with little by way of further comment. In what follows, *r* denotes an arbitrary relation, and *empty* denotes the empty relation of the same type as *r*.

- `r WHERE TRUE` = *r* (an *identity* restriction)
- `r WHERE FALSE` = *empty*
- `r { X, Y, ..., Z }` = *r* if *X*, *Y*, ..., *Z* are all of the attributes of *r* (an *identity* projection)

- `r { } = TABLE_DUM` if *r = empty*, `TABLE_DEE` otherwise (a *nullary* projection)
- `r JOIN r = r UNION r = r INTERSECT r = r`
- `r JOIN TABLE_DEE = TABLE_DEE JOIN r = r`
  (i.e., DEE is the *identity* with respect to join, just as zero is the identity with respect to addition, or one is the identity with respect to multiplication, in ordinary arithmetic)
- `r TIMES TABLE_DEE = TABLE_DEE TIMES r = r`
  (this equivalence is just a special case of the previous one)
- `r UNION empty = r MINUS empty = r`
- `empty INTERSECT r = empty MINUS r = empty`

Some Generalizations

JOIN, UNION, and INTERSECT were all defined originally as *dyadic* operators (i.e., each took exactly two relations as operands);[5] as we have seen, however, they can be unambiguously generalized to become n-adic operators for arbitrary $n > 1$. But what about $n = 1$? Or $n = 0$? It turns out to be desirable, at least from a conceptual point of view, to be able to perform "joins," "unions," and "intersections" of (a) just a single relation and (b) no relations at all (even though Tutorial D provides no direct syntactic support for any such operations). Here are the definitions. Let s be a set of relations (all of the same relation type RT in the case of union and intersection). Then:

- If s contains just one relation r, then the join, union, and intersection of all relations in s are all defined to be simply r.
- If s contains no relations at all, then:
  - The join of all relations in s is defined to be TABLE_DEE (the identity with respect to join).
  - The union of all relations in s is defined to be the empty relation of type RT.
  - The intersection of all relations in s is defined to be the "universal" relation of type RT—that is, that unique relation of type RT that contains all possible tuples with heading H, where H is the heading of relation type RT.[6]

7.8 ADDITIONAL OPERATORS

Numerous writers have proposed new algebraic operators since Codd defined his original eight. In this section we examine a few such operators—SEMIJOIN, SEMIMINUS, EXTEND, SUMMARIZE, and TCLOSE—in some detail. In terms of our Tutorial D syntax, these operators involve five new forms of *<nonproject>*, with specifics as follows:

[5] MINUS is dyadic, too. By contrast, restrict and project are *monadic* operators.

[6] We note in passing that the term *universal relation* is usually used in the literature with a very different meaning. See, for example, reference [13.20].

```
<semijoin>
    ::=   <relation exp> SEMIJOIN <relation exp>

<semiminus>
    ::=   <relation exp> SEMIMINUS <relation exp>

<extend>
    ::=   EXTEND <relation exp> ADD ( <extend add commalist> )
```

The parentheses can be omitted if the commalist contains just one *<extend add>*.

```
<extend add>
    ::=   <exp> AS <attribute name>

<summarize>
    ::=   SUMMARIZE <relation exp> PER <relation exp>
              ADD ( <summarize add commalist> )
```

The parentheses can be omitted if the commalist contains just one *<summarize add>*.

```
<summarize add>
    ::=   <summary type> [ ( <scalar exp> ) ]
                         AS <attribute name>

<summary type>
    ::=   COUNT | SUM | AVG | MAX | MIN | ALL | ANY
        | COUNTD | SUMD | AVGD | ...

<tclose>
    ::=   TCLOSE <relation exp>
```

The various *<relation exp>*s mentioned in the foregoing BNF production rules must not be *<nonproject>*s.

Semijoin

Let *a*, *b*, *X*, and *Y* be as defined in the subsection "Join" in Section 7.4. Then the semijoin of *a* with *b* (in that order), *a* SEMIJOIN *b*, is defined to be equivalent to:

```
( a JOIN b ) { X, Y }
```

In other words, the semijoin of *a* with *b* is the join of *a* and *b*, projected over the attributes of *a*. The body of the result is thus, loosely, the tuples of *a* that have a counterpart in *b*.

Example: Get S#, SNAME, STATUS, and CITY for suppliers who supply part P2:

```
S SEMIJOIN ( SP WHERE P# = P# ('P2') )
```

We note in passing that many real-world queries that call for the use of join are really semijoin queries in disguise—implying that direct support for SEMIJOIN might be desirable in practice. An analogous remark applies to SEMIMINUS (see the next subsection).

Semidifference

The semidifference between *a* and *b* (in that order), *a* SEMIMINUS *b*, is defined to be equivalent to:

```
a MINUS ( a SEMIJOIN b )
```

The body of the result is thus, loosely, the tuples of *a* that have no counterpart in *b*.

Example: Get S#, SNAME, STATUS, and CITY for suppliers who do not supply part P2:

```
S SEMIMINUS ( SP WHERE P# = P# ('P2') )
```

Extend

You might have noticed that the algebra as we have described it so far has no computational capabilities, as that term is conventionally understood. In practice, however, such capabilities are obviously desirable. For example, we would like to be able to retrieve the value of an arithmetic expression such as WEIGHT * 454, or to refer to such a value in a WHERE clause (we are assuming here—the discussion of units in Section 5.4 notwithstanding—that part weights are given in pounds and 1 pound = 454 grams[7]). The purpose of the extend operation is to support such capabilities. More precisely, EXTEND takes a relation and returns another that is identical to the given one except that it includes an additional attribute, values of which are obtained by evaluating some specified computational expression. For example, we might write:

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

This expression—please note that it *is* an expression, not a command or statement, and hence can be nested inside other expressions—yields a relation with the same heading as P, except that it includes an additional attribute called GMWT. Each tuple of that relation is the same as the corresponding tuple of P, except that it additionally includes a GMWT value, computed in accordance with the specified arithmetic expression WEIGHT * 454. See Fig. 7.9.

Important: Please note that this EXTEND expression has *not* changed the parts relvar in the database; it is just an expression, and like any other expression it simply denotes a certain value—a value that, in this particular case, happens to look rather like the current value of the parts relvar. (In other words, EXTEND is *not* a relational algebra analog of SQL's ALTER TABLE ... ADD COLUMN.)

P#	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Fig. 7.9 An example of EXTEND

[7] We are also assuming that "*" is a legal operation between weights and integers. What is the type of the result of such an operation?

Now we can use attribute GMWT in projections, restrictions, and so on. For example:

```
( ( EXTEND P ADD ( WEIGHT * 454 ) AS GMWT )
          WHERE GMWT > WEIGHT ( 10000.0 ) ) { ALL BUT GMWT }
```

Note: Of course, a more user-friendly language would allow the computational expression to appear directly in the WHERE clause, as here:

```
P WHERE ( WEIGHT * 454 ) > WEIGHT ( 10000.0 )
```

(see the discussion of restrict in Section 7.4). However, such a feature is really just syntactic sugar.

In general, then, the value of the extension

```
EXTEND a ADD exp AS Z
```

is a relation defined as follows:

- The heading of the result consists of the heading of *a* extended with the attribute Z.
- The body of the result consists of all tuples *t* such that *t* is a tuple of *a* extended with a value for attribute Z, computed by evaluating *exp* on that tuple of *a*.

Relation *a* must not have an attribute called Z, and *exp* must not refer to Z. Observe that the result has cardinality equal to that of *a* and degree equal to that of *a* plus one. The type of Z in that result is the type of *exp*.

Here are some more examples:

1. `EXTEND S ADD 'Supplier' AS TAG`

   This expression effectively tags each tuple of the current value of relvar S with the character string "Supplier" (a literal—or, more generally, a selector invocation—is a legal computational expression, of course).

2. `EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY ) AS SHIPWT`

   This example illustrates the application of EXTEND to the result of a relational expression that is more complicated than just a simple relvar name.

3. `( EXTEND S ADD CITY AS SCITY ) { ALL BUT CITY }`

   An attribute name such as CITY is also a legal computational expression. Observe that this particular example is equivalent to:

   ```
   S RENAME CITY AS SCITY
   ```

   In other words, RENAME is not primitive!—it can be defined in terms of EXTEND (and project). We would not want to discard our useful RENAME operator, of course, but it is at least interesting to note that it is really just shorthand.

4. `EXTEND P ADD ( WEIGHT * 454 AS GMWT, WEIGHT * 16 AS OZWT )`

   This example illustrates a "multiple EXTEND."

5. ```
   EXTEND S
   ADD COUNT ( ( SP RENAME S# AS X ) WHERE X = S# )
   AS NP
   ```

   The result of this expression is shown in Fig. 7.10. *Explanation:*
   ```

S#	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

Fig. 7.10 Another EXTEND example

a. For a given supplier, the expression

```
( ( SP RENAME S# AS X ) WHERE X = S# )
```

yields the set of shipments for that supplier.

b. The *aggregate operator* COUNT is then applied to that set of shipments and returns the corresponding cardinality (a scalar value).

Attribute NP in the result thus represents the number of parts supplied by the supplier identified by the corresponding S# value. Notice the NP value for supplier S5 in particular; the set of shipments for supplier S5 is empty, and so the COUNT invocation returns zero.

We elaborate briefly on this question of aggregate operators. The purpose of such an operator, in general, is to derive a single scalar value from the values appearing in some specified attribute of some specified relation (usually a *derived* relation). Typical examples are COUNT, SUM, AVG, MAX, MIN, ALL, and ANY. In Tutorial D, an aggregate operator invocation, *<agg op inv>*—which, since it returns a scalar value, is a special kind of *<scalar exp>*—takes the general form:

```
<agg op name> ( <relation exp> [, <attribute name> ] )
```

If the *<agg op name>* is COUNT, the *<attribute name>* is irrelevant and must be omitted; otherwise, it can be omitted if and only if the *<relation exp>* denotes a relation of degree one, in which case the sole attribute of the result of that *<relation exp>* is assumed by default. Here are a couple of examples:

```
SUM ( SP WHERE S# = S# ('S1'), QTY )
SUM ( ( SP WHERE S# = S# ('S1') ) { QTY } )
```

Note the difference between these two expressions—the first gives the total of all shipment quantities for supplier S1, the second gives the total of all *distinct* shipment quantities for supplier S1.

If the argument to an aggregate operator happens to be an empty set, COUNT (as we have seen) returns zero, and so does SUM; MAX and MIN return, respectively, the lowest and the highest value of the applicable type; ALL and ANY return TRUE and FALSE, respectively; and AVG raises an exception.

Summarize

We should begin this subsection by saying that the version of SUMMARIZE discussed here is not the same as that discussed in earlier editions of this book—in fact, it is an improved version that overcomes certain difficulties that arose with the earlier version in connection with empty relations.

As we have seen, the *extend* operator provides a way of incorporating "horizontal" or "tuple-wise" computations into the relational algebra. The summarize operator performs the analogous function for "vertical" or "attribute-wise" computations. For example, the expression

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

evaluates to a relation with attributes P# and TOTQTY, in which there is one tuple for each P# value in the projection of P over P#, containing that P# value and the corresponding total quantity (see Fig. 7.11). In other words, relation SP is conceptually partitioned into groups or sets of tuples (one such group for each part number in P), and then each such group is used to generate one tuple in the overall result.

In general, the value of the summarization

```
SUMMARIZE a PER b ADD summary AS Z
```

is a relation defined as follows:

- First, *b* must be of the same type as some projection of *a* (i.e., every attribute of *b* must be an attribute of *a*). Let the attributes of that projection (equivalently, of *b*) be *A1, A2, ..., An*.
- The heading of the result consists of the heading of *b* extended with the attribute *Z*.
- The body of the result consists of all tuples *t* such that *t* is a tuple of *b* extended with a value for attribute *Z*. That *Z* value is computed by evaluating *summary* over all tuples of *a* that have the same values for attributes { *A1, A2, ..., An* } as tuple *t* does. (Of course, if no tuples of *a* have the same value for { *A1, A2, ..., An* } as tuple *t* does, then *summary* is evaluated over an empty set.)

Relation *b* must not have an attribute called *Z* and *summary* must not refer to *Z*. Observe that the result has cardinality equal to that of *b* and degree equal to that of *b* plus one. The type of *Z* in that result is the type of *summary*.

P#	TOTQTY
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

Fig. 7.11 An example of SUMMARIZE

Here is another example:

```
SUMMARIZE ( P JOIN SP ) PER P ( CITY ) ADD COUNT AS NSP
```

The result looks like this:

CITY	NSP
London	5
Oslo	1
Paris	6

In other words, the result contains one tuple for each of the three part cities (London, Oslo, and Paris), showing in each case the number of shipments of parts stored in that city.

Points arising:

1. Observe yet again that here we have an operator whose definition relies on the notion of tuple equality.
2. Our syntax allows "multiple SUMMARIZEs"—for example:

```
SUMMARIZE SP PER P ( P# ) ADD ( SUM ( QTY ) AS TOTQTY,
                                AVG ( QTY ) AS AVGQTY )
```

3. The general form of *<summarize>* (to repeat) is as follows:

```
SUMMARIZE <relation exp> PER <relation exp>
                         ADD ( <summarize add commalist> )
```

Each *<summarize add>* in turn takes the form:

```
<summary type> [ ( <scalar exp> ) ] AS <attribute name>
```

Typical *<summary type>*s are COUNT, SUM, AVG, MAX, MIN, ALL, ANY, COUNTD, SUMD, and AVGD. The "D" ("distinct") in COUNTD, SUMD, and AVGD means "eliminate redundant duplicate values before performing the summary." The *<scalar exp>* can include references to attributes of the relation denoted by the *<relation exp>* immediately following the keyword SUMMARIZE. The *<scalar exp>* and enclosing parentheses can and must be omitted only if the *<summary type>* is COUNT.

Incidentally, please note that a *<summarize add>* is *not* the same thing as an aggregate operator invocation. An *<agg op inv>* is a scalar expression and can appear wherever a literal of the appropriate type can appear. A *<summarize add>*, by contrast, is merely a SUMMARIZE operand; it is *not* a scalar expression, it has no meaning outside the context of SUMMARIZE, and in fact it cannot appear outside that context.

4. As you might have already realized, SUMMARIZE is not a primitive operator—it can be simulated by means of EXTEND. For example, the expression

```
SUMMARIZE SP PER S ( S# ) ADD COUNT AS NP
```

is defined to be shorthand for the following:

```
( EXTEND S ( S# )
         ADD ( ( ( SP RENAME S# AS X ) WHERE X = S# ) AS Y,
               COUNT ( Y ) AS NP ) )
( S#, NP )
```

Or equivalently:

```
WITH ( S { S# } ) AS T1,
     ( SP RENAME S# AS X ) AS T2,
     ( EXTEND T1 ADD ( T2 WHERE X = S# ) AS Y ) AS T3,
     ( EXTEND T3 ADD COUNT ( Y ) AS NP ) AS T4 :
T4 { S#, NP }
```

By the way, attribute Y here is relation-valued. Refer to Section 6.4 if you need to refresh your memory regarding such a possibility.

5. Here is another example:

   ```
   SUMMARIZE S PER S { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
   ```

   Here the PER relation is not just "of the same type as" some projection of the relation to be summarized, it actually *is* such a projection. In such a case, we allow the following shorthand:

   ```
   SUMMARIZE S BY { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
   ```

   (We have replaced PER *<relation exp>* by BY *<attribute name commalist>*. The attributes named must all be attributes of the relation being summarized.)

6. Consider the following example:

   ```
   SUMMARIZE SP PER SP { } ADD SUM ( QTY ) AS GRANDTOTAL
   ```

   In accordance with the previous point, we can alternatively write this expression thus:

   ```
   SUMMARIZE SP BY { } ADD SUM ( QTY ) AS GRANDTOTAL
   ```

   Either way, the grouping and summarization here are being done on the basis of a relation that has no attributes at all. Let *sp* be the current value of relvar SP, and assume for the moment that relation *sp* does contain at least one tuple. Then all of those *sp* tuples have the same value for no attributes at all (namely, the 0-tuple); hence there is just one group, and so just one tuple in the overall result (in other words, the aggregate computation is performed precisely once for the entire relation *sp*). The expression thus evaluates to a relation with one attribute and one tuple; the attribute is called GRANDTOTAL, and the single scalar value in the single result tuple is the total of all QTY values in the original relation *sp*.

   If on the other hand the original relation *sp* has no tuples at all, then there are no groups, and hence no result tuples (i.e., the result relation is empty too). By contrast, the following expression—

   ```
   SUMMARIZE SP PER TABLE_DEE ADD SUM ( QTY ) AS GRANDTOTAL
   ```

   —will "work" (i.e., it will return the correct result, zero) even if *sp* is empty. More precisely, it will return a relation with one attribute, called GRANDTOTAL, and one tuple, containing the value zero. We therefore suggest that it should be possible to omit the PER clause entirely, as here:

   ```
   SUMMARIZE SP ADD SUM ( QTY ) AS GRANDTOTAL
   ```

   Omitting the PER clause is defined to be equivalent to specifying PER TABLE_DEE.

Tclose

"Tclose" stands for *transitive closure*. We mention it here mainly for completeness; detailed discussion is beyond the scope of this chapter. However, we do at least define the operation, as follows. Let a be a binary relation with attributes X and Y, both of the same type T. Then the **transitive closure** of a, TCLOSE a, is a relation a^+ with heading the same as that of a and body a superset of that of a, defined as follows: The tuple

```
{ X x, Y y }
```

appears in a^+ if and only if it appears in a or there exists a sequence of values $z1$, $z2$, ..., zn, all of type T, such that the tuples

```
{ X x, Y z1 }, { X z1, Y z2 }, ..., { X zn, Y y }
```

all appear in a. (In other words, if we think of relation a as representing a *graph*, then the "(x,y)" tuple appears in a^+ only if there is a path in that graph from node x to node y. Note that the body of a^+ necessarily includes the body of a as a subset.)

For further discussion of transitive closure, see Chapter 24.

7.9 GROUPING AND UNGROUPING

The fact that we can have relations with attributes whose values are relations in turn leads to the need for operators, here called *group* and *ungroup*, for mapping between relations that contain such attributes and relations that do not. For example:

```
SP GROUP { P#, QTY } AS PQ
```

Given our usual sample data, this expression yields the result shown in Fig. 7.12. *Note:* You will probably find it helpful to use that figure to check the explanations that follow, since they are (regrettably, but unavoidably) a little abstract.

We begin by observing that the original expression

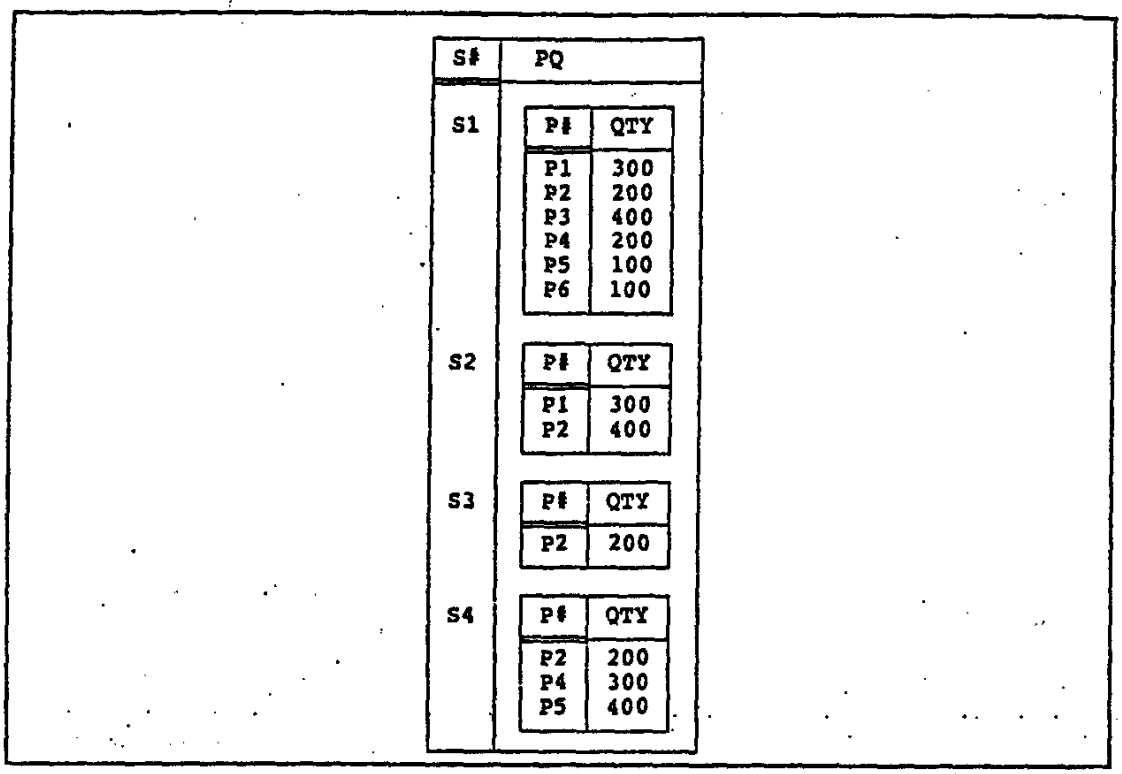
```
SP GROUP { P#, QTY } AS PQ
```

might be read as "group SP *by* S#," S# being the sole attribute of SP not mentioned in the GROUP specification. The result is a relation defined as follows. First, the heading looks like this:

```
{ S# S#, PQ RELATION { P# P#, QTY QTY } }
```

In other words, it consists of a relation-valued attribute PQ (where PQ values in turn have attributes P# and QTY), together with all of the other attributes of SP (of course, "all of the other attributes of SP" here just means attribute S#). Second, the body contains exactly one tuple for each distinct S# value in SP (and no other tuples). Each tuple in that body consists of the applicable S# value (s, say), together with a PQ value (pq, say) obtained as follows:

- Each SP tuple is replaced by a tuple (x, say) in which the P# and QTY components have been "wrapped" into a tuple-valued component (y, say).



S#	PQ

S1	P#	QTY
	P1	300
	P2	200
	P3	400
	P4	200
	P5	100
	P6	100

S2	P#	QTY
	P1	300
	P2	400

S3	P#	QTY
	P2	200

S4	P#	QTY
	P2	200
	P4	300
	P5	400

Fig. 7.12 Grouping SP by S#

- The *y* components of all such tuples *x* in which the S# value is equal to *s* are "grouped" into a relation, *pq*, and a result tuple with S# value equal to *s* and PQ value equal to *pq* is thereby obtained.

The overall result is thus indeed as shown in Fig. 7.12. Note in particular that the result includes no tuple for supplier S5 (because relvar SP does not currently do so either).

Observe that the result of *R* GROUP { *A1*, *A2*, ..., *An* } AS *B* has degree equal to *nR*–*n*+1, where *nR* is the degree of *R*.

Now we turn to *ungroup*. Let SPQ be the relation shown in Fig. 7.12. Then the expression

```
SPQ UNGROUP PQ
```

(perhaps unsurprisingly) gives us back our usual sample SP relation. To be more specific, it yields a relation defined as follows. First, the heading looks like this:

```
{ S# S#, P# P#, QTY QTY }
```

In other words, the heading consists of attributes P# and QTY (derived from attribute PQ), together with all of the other attributes of SPQ (i.e., just attribute S#, in the example). Second, the body contains exactly one tuple for each combination of a tuple in SPQ and a tuple in the PQ value within that SPQ tuple (and no other tuples). Each tuple in that body consists of the applicable S# value (*s*, say), together with P# and QTY values (*p* and *q*, say) obtained as follows:

- Each SPQ tuple is replaced by an "ungrouped" set of tuples, one such tuple (x, say) for each tuple in the PQ value in that SPQ tuple.
- Each such tuple x contains an S# component (s, say) equal to the S# component from the SPQ tuple in question and a tuple-valued component (y, say) equal to some tuple from the PQ component from the SPQ tuple in question.
- The y components of each such tuple x in which the S# value is equal to s are "unwrapped" into separate P# and QTY components (p and q, say), and a result tuple with S# value equal to s, P# value equal to p, and QTY value equal to q is thereby obtained.

The overall result is thus, as claimed, our usual sample SP relation.

Observe that the result of *R* UNGROUP *B* (where the relations that are values of the relation-valued attribute *B* have heading { *A1*, *A2*, ... *An* }) has degree equal to $nR+n-1$, where nR is the degree of *R*.

As you can see, GROUP and UNGROUP together provide what are more usually referred to as relational "nest" and "unnest" capabilities. We prefer our group/ungroup terminology, however, because the nest/unnest terminology is strongly associated with the concept of NF^2 relations [6.10], a concept we do not endorse.

For completeness, we close this section with some remarks concerning reversibility of the GROUP and UNGROUP operations (though we realize the remarks in question might not be fully comprehensible on a first reading). If we group some relation *r* in some way, there is always an inverse ungrouping that will take us back to *r* again. However, if we *un*group some relation *r* in some way, an inverse grouping to take us back to *r* again might or might not exist. Here is an example (based on one given in reference [6.4]). Suppose we start with relation TWO (see Fig. 7.13) and ungroup it to obtain THREE. If we now group THREE by A (and name the resulting relation-valued attribute RVX once again), we obtain not TWO but ONE.

If we now ungroup ONE, we return to THREE, and we have already seen that THREE can be grouped to give ONE; thus, the group and ungroup operations are indeed inverses of each other for this particular pair of relations. Note that, in ONE, RVX is

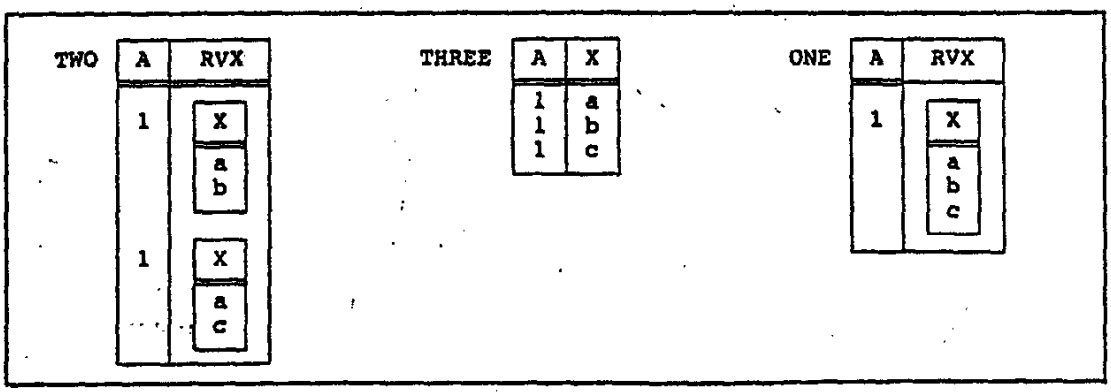


Fig. 7.13 Ungrouping and (re)grouping are not necessarily reversible

functionally dependent on A[8] (necessarily so, since ONE is of cardinality one). In general, in fact, we can say that if relation *r* has a relation-valued attribute *RVX*, then *r* is reversibly ungroupable with respect to *RVX* if and only if the following are both true:

- No tuple of *r* has an empty relation as its *RVX* value.
- *RVX* is functionally dependent on the combination of all other attributes of *r*.

7.10 SUMMARY

We have discussed the relational algebra. We began by reemphasizing the importance of closure and nested relational expressions, and explained that if we are going to take closure seriously, then we need some relation type inference rules. Such considerations led us to introduce the RENAME operator.

The original algebra consisted of eight operators—the traditional set operators union, intersect, difference, and product (all of them modified somewhat to take account of the fact that their operands are very specifically relations, not arbitrary sets), and the special relational operators restrict, project, join, and divide. (In the case of divide, however, we remarked that queries involving divide can always be formulated in terms of *relational comparisons* instead, and many people find such formulations intuitively easier to understand.) To that original set we added RENAME (as already mentioned), SEMIJOIN, SEMIMINUS, EXTEND, and SUMMARIZE, and we also mentioned TCLOSE and discussed GROUP and UNGROUP. EXTEND in particular is extremely important (in some ways it is as important as join).

Next, we pointed out that the algebraic operators are not all primitive (i.e., many of them can be defined in terms of others)—a most satisfactory state of affairs, in our opinion. As reference [7.3] puts it: "A language definition should start with a few judiciously chosen primitive operators . . . Subsequent development is, where possible, by defining new operators in terms of . . . previously defined [ones]"—in other words, by defining useful shorthands. If the shorthands in question are well chosen, then not only do they save us a great deal of writing, they also effectively raise the level of abstraction, by allowing us to talk in terms of certain "bundles" of concepts that fit together naturally. (They also pave the way for more efficient implementation.) In this connection, we remind you that, as noted in Section 7.6, reference [3.3] describes a kind of "reduced instruction set" algebra called A that is specifically intended to support the systematic definition of more powerful operators in terms of a very small number of primitives; in fact, it shows that the entire functionality of the original algebra, together with RENAME, EXTEND, SUMMARIZE, GROUP, and UNGROUP, can all be achieved with just two primitives called *remove* and *nor*.

Back to our summary. We went on to show how the algebraic operators can be combined into expressions that serve a variety of purposes: retrieval, update, and several others. We also very briefly discussed the idea of transforming such expressions for optimization purposes (but we will examine this idea in much more detail in Chapter 18).

[8] See Chapter 11, and note in particular that we are appealing here to that form of functional dependence that applies to relation values (as opposed to the more usual form, which applies to relation variables).

And we explained how the use of WITH could simplify the formulation of complex expressions; WITH effectively allows us to introduce names for subexpressions, thereby allowing us to formulate those complex expressions one step at a time, and yet does not compromise the algebra's fundamental nonprocedurality.

We also pointed out that certain of the operators were associative and commutative, and we showed certain equivalences (e.g., we showed that any relation R is equivalent to a certain restriction of R and a certain projection of R). We also considered what it means to perform joins, unions, and intersections on just one relation and on no relations at all.

EXERCISES

7.1 Which of the relational operators defined in this chapter have a definition that does not rely on tuple equality?

7.2 Given the usual suppliers-and-parts database, what is the value of the expression S JOIN SP JOIN P? What is the corresponding predicate? *Warning:* There is a trap here.

7.3 Let r be a relation of degree n. How many different projections of r are there?

7.4 Union, intersection, product, and natural join are all both commutative and associative. Verify these claims.

7.5 Consider the expression a JOIN b. If a and b have disjoint headings, this expression is equivalent to a TIMES b; if they have the same heading, it is equivalent to a INTERSECT b. Verify these claims. What is the expression equivalent to if the heading of a is a proper subset of that of b?

7.6 Of Codd's original set of eight operators, union, difference, product, restrict, and project can be considered as primitives. Give definitions of natural join, intersect, and (harder!) divide in terms of those primitives.

7.7 In ordinary arithmetic, multiplication and division are inverse operations. Are TIMES and DIVIDEBY inverse operations in the relational algebra?

7.8 In ordinary arithmetic there are two special numbers, 1 and 0, with the properties that

$$n * 1 = 1 * n = n$$

and

$$n * 0 = 0 * n = 0$$

for all numbers n. What relations (if any) play analogous roles in the relational algebra? Investigate the effect of the algebraic operations discussed in this chapter on those relations.

7.9 In Section 7.2, we said the relational closure property was important for the same kind of reason that the arithmetic closure property was important. In arithmetic, however, there is one unpleasant situation where the closure property breaks down—namely, division by zero. Is there any analogous situation in the relational algebra?

7.10 Given that intersect is a special case of join, why do not both operators give the same result when applied to no relations at all?

7.11 Which (if any) of the following expressions are equivalent?

a. `SUMMARIZE r PER r { } ADD COUNT AS CT`

b. `SUMMARIZE r ADD COUNT AS CT`

c. `SUMMARIZE r BY { } ADD COUNT AS CT`

d. `EXTEND TABLE_DEE ADD COUNT ( r ) AS CT`

7.12 Let *r* be the relation denoted by the following expression:

```
SP GROUP { } AS X
```

Show what *r* looks like, given our usual sample value for SP. Also, show the result of:

```
r UNGROUP X
```

Query Exercises

The remaining exercises are all based on the suppliers-parts-projects database. In each case you are asked to write a relational algebra expression for the indicated query. (By way of an interesting variation, you might like to try looking at some of the online answers first and stating what the given expression means in natural language.) For convenience, we repeat the structure of the database in outline here:

```
S    { S#, SNAME, STATUS, CITY }
     PRIMARY KEY { S# }
P    { P#, PNAME, COLOR, WEIGHT, CITY }
     PRIMARY KEY { P# }
J    { J#, JNAME, CITY }
     PRIMARY KEY { J# }
SPJ  { S#, P#, J#, QTY }
     PRIMARY KEY { S#, P#, J# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P
     FOREIGN KEY { J# } REFERENCES J
```

7.13 Get full details of all projects.

7.14 Get full details of all projects in London.

7.15 Get supplier numbers for suppliers who supply project J1.

7.16 Get all shipments where the quantity is in the range 300 to 750 inclusive.

7.17 Get all part-color/part-city pairs. *Note:* Here and subsequently, the term "all" means "all currently represented in the database," not "all possible."

7.18 Get all supplier-number/part-number/project-number triples such that the indicated supplier, part, and project are all colocated (i.e., all in the same city).

7.19 Get all supplier-number/part-number/project-number triples such that the indicated supplier, part, and project are not all colocated.

7.20 Get all supplier-number/part-number/project-number triples such that no two of the indicated supplier, part, and project are colocated.

7.21 Get full details for parts supplied by a supplier in London.

7.22 Get part numbers for parts supplied by a supplier in London to a project in London.

7.23 Get all pairs of city names such that a supplier in the first city supplies a project in the second city.

7.24 Get part numbers for parts supplied to any project by a supplier in the same city as that project.

7.25 Get project numbers for projects supplied by at least one supplier not in the same city.

7.26 Get all pairs of part numbers such that some supplier supplies both the indicated parts.

7.27 Get the total number of projects supplied by supplier S1.

7.28 Get the total quantity of part P1 supplied by supplier S1.

7.29 For each part being supplied to a project, get the part number, the project number, and the corresponding total quantity.

7.30 Get part numbers of parts supplied to some project in an average quantity of more than 350.

7.31 Get project names for projects supplied by supplier S1.

7.32 Get colors of parts supplied by supplier S1.

7.33 Get part numbers for parts supplied to any project in London.

7.34 Get project numbers for projects using at least one part available from supplier S1.

7.35 Get supplier numbers for suppliers supplying at least one part supplied by at least one supplier who supplies at least one red part.

7.36 Get supplier numbers for suppliers with a status lower than that of supplier S1.

7.37 Get project numbers for projects whose city is first in the alphabetic list of such cities.

7.38 Get project numbers for projects supplied with part P1 in an average quantity greater than the greatest quantity in which any part is supplied to project J1.

7.39 Get supplier numbers for suppliers supplying some project with part P1 in a quantity greater than the average shipment quantity of part P1 for that project.

7.40 Get project numbers for projects not supplied with any red part by any London supplier.

7.41 Get project numbers for projects supplied entirely by supplier S1.

7.42 Get part numbers for parts supplied to all projects in London.

7.43 Get supplier numbers for suppliers who supply the same part to all projects.

7.44 Get project numbers for projects supplied with at least all parts available from supplier S1.

7.45 Get all cities in which at least one supplier, part, or project is located.

7.46 Get part numbers for parts that are supplied either by a London supplier or to a London project.

7.47 Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

7.48 Get all pairs of supplier numbers, Sx and Sy say, such that Sx and Sy supply exactly the same set of parts each. *Note:* For simplicity, you might want to use the original suppliers-and-parts database for this exercise, instead of the expanded suppliers-parts-projects database.

7.49 Get a "grouped" version of all shipments showing, for each supplier-number/part-number pair, the corresponding project numbers and quantities in the form of a binary relation.

7.50 Get an "ungrouped" version of the relation produced in Exercise 7.49.

REFERENCES AND BIBLIOGRAPHY

7.1 E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6.* Englewood Cliffs, N.J.: Prentice-Hall (1972).

This is the paper in which Codd first *formally* defined the original algebraic operators (definitions did appear in reference [6.1] also, but they were somewhat less formal, or at least less

complete). *Note:* One slightly unfortunate aspect of the paper is that it assumes "for notational and expository convenience" that the attributes of a relation have a left-to-right ordering and hence can be identified by their ordinal position (though Codd does say that "names rather than position numbers [should] be used . . . when actually storing or retrieving information"—and he had previously said much the same thing in reference [6.1]). The paper therefore does not mention an attribute RENAME operator, and it does not consider the question of result type inference. Possibly as a consequence of these omissions, the same criticisms can still be leveled today (a) at many discussions of the algebra in the literature, (b) at today's SQL products, and (c) to a slightly lesser extent, at the SQL standard as well.

Additional commentary on this paper appears in Chapter 8, especially in Section 8.4.

7.2 Hugh Darwen (writing as Andrew Warden): "Adventures in Relationland," in C. J. Date, *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

A series of short papers that examine various aspects of the relational model and relational DBMSs in an original, entertaining, and informative style.

7.3 Hugh Darwen: "Valid Time and Transaction Time Proposals: Language Design Aspects," in Opher Etzion, Sushil Jajodia, and Suryanaryan Sripada (eds.), *Temporal Databases: Research and Practice*. New York, N.Y.: Springer-Verlag (1998).

7.4 Hugh Darwen and C. J. Date: "Into the Great Divide," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

This paper analyzes both (a) Codd's original divide as defined in reference [7.1] and (b) a generalization of that operator due to Hall, Hitchcock, and Todd [7.10] that—unlike Codd's original divide—allowed any relation to be divided by any relation (Codd's original divide was defined only for the case where the heading of the divisor was a subset of the heading of the dividend). The paper shows that both operators get into difficulties over empty relatioas, with the result that neither of them quite solves the problem it was originally intended for (i.e., neither of them is quite the counterpart of the universal quantifier it was meant to be). Revised versions of both operators (the "Small Divide" and the "Great Divide," respectively) are proposed to overcome these problems. *Note:* As the Tutorial D syntax for these two operators indicates, they really are two different operators; that is, the Great Divide is (unfortunately) not quite an upward-compatible extension of the Small Divide. The paper also suggests that the revised operators no longer merit the name "divide"! In connection with this last point, see Exercise 7.7.

For purposes of reference, we give here a definition of Codd's original divide. Let relations A and B have headings $\{X,Y\}$ and $\{Y\}$, respectively (where X and Y can be composite). Then the expression A DIVIDEBY B gives a relation with heading $\{X\}$ and body consisting of all tuples $\{X\,x\}$ such that a tuple $\{X\,x, Y\,y\}$ appears in A for *all* tuples $\{Y\,y\}$ appearing in B. In other words, loosely speaking, the result consists of those X values from A whose corresponding Y values (in A) include *all* Y values from B.

7.5 C. J. Date: "Quota Queries" (in three parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994–1997*. Reading, Mass.: Addison-Wesley (1998).

A *quota query* is a query that specifies a desired limit on the cardinality of the result—for example, the query "Get the three heaviest parts." This paper discusses one approach to formulating such queries. Using that approach, "Get the three heaviest parts" can be formulated thus:

```
P QUOTA ( 3, DESC WEIGHT )
```

This expression is defined to be shorthand for the following:

```
( ( EXTEND P
    ADD COUNT ( ( P RENAME WEIGHT AS WT ) WHERE WT > WEIGHT )
    AS #_HEAVIER )
WHERE #_HEAVIER < 3 ) ( ALL BUT #_HEAVIER )
```

(where the names WT and #_HEAVIER are arbitrary; given our usual sample data, the result consists of parts P2, P3, and P6). The paper analyzes the quota query requirement in depth and proposes several shorthands for dealing with it and related matters. *Note:* An alternative approach to formulating quota queries, involving a new relational operator called RANK, is described in reference [3.3].

7.6 R. C. Goldstein and A. J. Strnad: "The MacAIMS Data Management System." Proc. 1970 ACM SICFIDET Workshop on Data Description and Access (November 1970).

See the annotation to reference [7.7].

7.7 A. J. Strnad: "The Relational Approach to the Management of Data Bases." Proc. IFIP Congress, Ljubljana, Yugoslavia (August 1971).

We mention MacAIMS [7.6, 7.7] primarily for reasons of historical interest; it seems to have been the earliest example of a system supporting n-ary relations and an algebraic language. The interesting thing about it is that it was developed in parallel with, and at least partly independently of, Codd's work on the relational model. Unlike Codd's work, however, the MacAIMS effort seems not to have led to any significant follow-on activities.

7.8 M. G. Notley: "The Peterlee IS/1 System," IBM UK Scientific Centre Report UKSC-0018 (March 1972).

See the annotation to reference [7.9].

7.9 S. J. P. Todd: "The Peterlee Relational Test Vehicle—A System Overview," *IBM Sys. J. 15*, No. 4 (1976).

The Peterlee Relational Test Vehicle PRTV was an experimental system developed at the IBM UK Scientific Centre in Peterlee, England. It was based on an earlier prototype—possibly the very first implementation of Codd's ideas—called IS/1 [7.8]. It supported n-ary relations and a version of the algebra called ISBL (Information System Base Language), which was based on proposals documented in reference [7.10]. The ideas discussed in the present chapter regarding relation type inference can be traced back to ISBL, as well as the proposals of reference [7.10]. Significant aspects of PRTV included the following:

- It supported RENAME, EXTEND, and SUMMARIZE.
- It incorporated some sophisticated expression transformation techniques (see Chapter 18).
- It included a *lazy evaluation* feature, which was important both for optimization and for view support (see the discussion of WITH in the body of this chapter).
- It allowed users to define their own operators.

7.10 P. A. V. Hall, P. Hitchcock, and S. J. P. Todd: "An Algebra of Relations for Machine Computation," Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif. (January 1975).

7.11 Anthony Klug: "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *JACM 29*, No. 3 (July 1982).

Defines extensions to both the original relational algebra and the original relational calculus (see Chapter 8) to support aggregate operators, and demonstrates the equivalence of the two extended formalisms.

CHAPTER 8

Relational Calculus

8.1 INTRODUCTION

Relational calculus is an alternative to relational algebra. The main difference between the two is as follows: Whereas the algebra provides a set of explicit operators—join, union, project, and so on—that can be used to tell the system how to *construct* some desired relation from certain given relations, the calculus, by contrast, merely provides a notation for stating the *definition* of that desired relation in terms of those given relations. For example, consider the query "Get supplier numbers and cities for suppliers who supply part P2." An algebraic formulation of that query might specify operations as follows (we deliberately do not use the formal syntax of Chapter 7):

1. Join suppliers and shipments over S#.
2. Restrict the result of that join to tuples for part P2.
3. Project the result of that restriction over S# and CITY.

A calculus formulation, by contrast, might state simply:

Get S# and CITY for suppliers such that there exists a shipment SP with the same S# value and with P# value P2.

In this latter formulation, the user has merely stated the defining characteristics of the desired result, and has left it to the system to decide exactly what joins, restrictions, and so on, must be executed, in what sequence, in order to construct that result. Thus, we might say that—at least superficially—the calculus formulation is *descriptive*, while the algebraic one is *prescriptive:* The calculus simply describes what the problem *is*, the algebra prescribes a procedure for *solving* that problem. Or, very informally: The algebra is procedural (admittedly high-level, but still procedural); the calculus is nonprocedural.

However, we stress the point that the foregoing differences *are* only superficial. The fact is, *the algebra and the calculus are logically equivalent:* For every algebraic expression there is an equivalent calculus one, for every calculus expression there is an equivalent algebraic one. There is a one-to-one correspondence between the two. Thus, the differences are really just a matter of *style:* The calculus is arguably closer to natural language, the algebra is perhaps more like a programming language. But, to repeat, such differences are more apparent than real: in particular, neither approach is truly more nonprocedural than the other. We will examine this question of equivalence in detail in Section 8.4.

Relational calculus is based on a branch of mathematical logic called predicate calculus. The idea of using predicate calculus as the basis for a query language appears to have originated in a paper by Kuhns [8.6]. The concept of a specifically *relational* calculus—that is, an applied form of predicate calculus specifically tailored to relational databases—was first proposed by Codd in reference [6.1]; a language explicitly based on that calculus called "Data Sublanguage ALPHA" was also presented by Codd in another paper, reference [8.1]. ALPHA itself was never implemented, but a language called QUEL [8.5, 8.10–8.12]—which certainly was implemented and for some time was a serious competitor to SQL—was very similar to it; indeed, QUEL was much influenced by ALPHA.

A fundamental feature of the calculus is the **range variable**. Briefly, a range variable is a variable that "ranges over" some specified relation (i.e., it is a variable whose permitted values are tuples of that relation). Thus, if range variable *V* ranges over relation *r*, then, at any given time, the expression "*V*" denotes some tuple of *r*. For example, the query "Get supplier numbers for suppliers in London" might be expressed in QUEL as follows:

```
RANGE OF SX IS S ;
RETRIEVE ( SX.S# ) WHERE SX.CITY = "London" ;
```

The sole range variable here is SX, and it ranges over the relation that is the current value of relvar S (the RANGE statement is a *definition* of that range variable). The RETRIEVE statement can thus be paraphrased: "For every possible value of variable SX, retrieve the S# component of that value, if and only if the CITY component of that value is London."

Because of its reliance on range variables whose values are tuples (and to distinguish it from the *domain* calculus—see the next paragraph), the original relational calculus has

come to be known as the **tuple calculus**. The tuple calculus is described in detail in Section 8.2. *Note:* For simplicity, we adopt the convention throughout this book that the terms *calculus* and *relational calculus*, without a "tuple" or "domain" qualifier, refer to the tuple calculus specifically (where it makes any difference).

Subsequently, Lacroix and Pirotte [8.7] proposed an alternative version of the calculus called the **domain calculus**, in which the range variables range over domains (i.e., types) instead of relations. (The terminology is illogical: If the domain calculus is so called for the reason just stated—which it is—then the tuple calculus ought by rights to be called the *relation* calculus.) Various domain calculus languages have been proposed in the literature; probably the best known is Query-By-Example, QBE [8.14] (though QBE is really something of a hybrid—it incorporates elements of the tuple calculus as well). Several commercial QBE or "QBE-like" implementations exist. We sketch the domain calculus in Section 8.7, and discuss QBE briefly in Section 8.8.

Note: For space reasons, we omit discussion of calculus analogs of certain topics from Chapter 7 (e.g., grouping and ungrouping). We also omit consideration of calculus versions of the relational update operators. You can find a discussion of such matters in reference [3.3].

8.2 TUPLE CALCULUS

As with our discussions of the algebra in Chapter 7, we first introduce a concrete syntax—patterned after, though deliberately not quite identical to, the calculus version of Tutorial D defined in Appendix A of reference [3.3]—and then go on to discuss semantics. The subsection immediately following discusses syntax, the remaining subsections consider semantics.

Syntax

Note: Many of the syntax rules given in prose form in this subsection will not make much sense until you have studied some of the semantic material that comes later. However, we gather them all here for purposes of subsequent reference.

It is convenient to begin by repeating the syntax of *<relation exp>*s from Chapter 7:

```
<relation exp>
    ::=   RELATION { <tuple exp commalist> }
        | <relvar name>
        | <relation op inv>
        | <with exp>
        | <introduced name>
        | ( <relation exp> )
```

In other words, the syntax of *<relation exp>*s is the same as before; however, one of the most important cases, *<relation op inv>*, which is the only one we discuss in this chapter in any detail, now has a very different definition, as we will see.

```
<range var def>
    ::=   RANGEVAR <range var name>
          RANGES OVER <relation exp commalist> ;
```

A *<range var name>* can be used as a *<tuple exp>*,[1] but only in certain contexts—namely:

- Preceding the dot qualifier in a *<range attribute ref>*
- Immediately following the quantifier in a *<quantified bool exp>*
- As an operand within a *<bool exp>*
- As a *<proto tuple>* or as an *<exp>* (or an operand within an *<exp>*) within a *<proto tuple>*

```
<range attribute ref>
    ::=   <range var name> . <attribute name>
                          [ AS <attribute name> ]
```

A *<range attribute ref>* can be used as an *<exp>*, but only in certain contexts—namely:

- As an operand within a *<bool exp>*
- As a *<proto tuple>* or as an *<exp>* (or an operand within an *<exp>*) within a *<proto tuple>*

```
<bool exp>
    ::=   ... all the usual possibilities, together with:
        | <quantified bool exp>
```

References to range variables within a *<bool exp>* can be free within that *<bool exp>* only if both of the following are true:

- The *<bool exp>* appears immediately within a *<relation op inv>* (i.e., the *<bool exp>* immediately follows the keyword WHERE).
- A reference (necessarily free) to that very same range variable appears immediately within the *<proto tuple>* immediately contained within that very same *<relation op inv>* (i.e., the *<proto tuple>* immediately precedes the keyword WHERE).

Terminology: In the context of the relational calculus (either version), *<bool exp>*s are often called well-formed formulas or WFFs (pronounced "weffs"). We will use this term ourselves in much of what follows.

```
<quantified bool exp>
    ::=   <quantifier> <range var name> ( <bool exp> )

<quantifier>
    ::=   EXISTS | FORALL

<relation op inv>
    ::=   <proto tuple> [ WHERE <bool exp> ]
```

As in the algebra of Chapter 7, a *<relation op inv>* is one form of *<relation exp>*, but as noted earlier we are giving it a different definition here.

```
<proto tuple>
    ::=   ... see the body of the text
```

[1] We do not spell out the details of *<tuple exp>*s, trusting that examples will be sufficient to give the general idea; for reasons that are unimportant here, however, we do not use exactly the same syntax as we did in previous chapters.

All references to range variables appearing immediately within a *<proto tuple>* must be free within that *<proto tuple>*. *Note:* "Proto tuple" stands for "prototype tuple"; the term is apt but not standard.

Range Variables

Here are some sample range variable definitions (expressed as usual in terms of suppliers and parts):

```
RANGEVAR SX  RANGES OVER S ;
RANGEVAR SY  RANGES OVER S ;
RANGEVAR SPX RANGES OVER SP ;
RANGEVAR SPY RANGES OVER SP ;
RANGEVAR PX  RANGES OVER P ;

RANGEVAR SU RANGES OVER
       ( SX WHERE SX.CITY = 'London' ) ,
       ( SX WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                               SPX.P# = P# ('P1') ) ) ;
```

Range variable SU in this last example is defined to range over the *union* of the set of supplier tuples for suppliers who are located in London and the set of supplier tuples for suppliers who supply part P1. Observe that the definition of range variable SU makes use of the range variables SX and SPX. Note too that in such "union-style" definitions, the relations to be "unioned" must (of course) all be of the same type.

Note: Range variables are not variables in the usual programming language sense. they are variables in the sense of logic. In fact, they are analogous, somewhat, to the *parameters* to predicates as discussed in Chapter 3: the difference is that the parameters of Chapter 3 stand for values from the applicable domain (whatever that might be), whereas range variables in the tuple calculus stand specifically for tuples.

Throughout the rest of this chapter, we will assume that the range variable definitions shown in this subsection are in effect. We note that in a real language there would have to be some rules regarding the *scope* of such definitions. We ignore such matters in the present chapter (except in SQL contexts).

Free and Bound Variable References

Every reference to a range variable is either free or bound (within some context—in particular, within some WFF). We explain this notion in purely syntactic terms in this subsection, then go on to discuss its semantic significance in subsequent subsections.

Let *V* be a range variable and let *p* and *q* be WFFs. Then:

- References to *V* in NOT *p* are free or bound within that WFF according as they are free or bound in *p*. References to *V* in *p* AND *q* and *p* OR *q* are free or bound in those WFFs according as they are free or bound in *p* or *q*, as applicable.
- References to *V* that are free in *p* are bound in EXISTS *V* (*p*) and FORALL *V* (*p*). Other references to range variables in *p* are free or bound in EXISTS *V* (*p*) and FORALL *V* (*p*) according as they are free or bound in *p*.

For completeness, we need to add the following:

- The sole reference to *V* in the *<range var name> V* is free within that *<range var name>*.
- The sole reference to *V* in the *<range attribute ref> V.A* is free within that *<range attribute ref>*.
- If a reference to *V* is free in some expression *exp*, that reference is also free in any expression *exp'* that immediately contains *exp* as a subexpression, unless *exp'* introduces a quantifier that makes that reference bound instead.

Here are some examples of WFFs containing range variable references:

- *Simple comparisons:*

```
SX.S# = S# ('S1')

SX.S# = SPX.S#

SPX.P# ≠ PX.P#
```

All references to SX, PX, and SPX are free in these examples.

- *Boolean combinations of simple comparisons:*

```
PX.WEIGHT < WEIGHT ( 15.5 ) AND PX.CITY = 'Oslo'

NOT ( SX.CITY = 'London' )

SX.S# = SPX.S# AND SPX.P# ≠ PX.P#

PX.COLOR = COLOR ('Red') OR PX.CITY = 'London'
```

Again, all references to SX, PX, and SPX here are free.

- *Quantified WFFs:*

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )

FORALL PX ( PX.COLOR = COLOR ('Red') )
```

The references to SPX and PX in these two examples are bound, the reference to SX is free. See the subsection "Quantifiers" immediately following.

Quantifiers

There are two quantifiers, EXISTS and FORALL; EXISTS is the existential quantifier, FORALL is the universal quantifier.[2] Basically, if *p* is a WFF in which *V* is free, then

```
EXISTS V ( p )
```

and

```
FORALL V ( p )
```

are both valid WFFs, and *V* is bound in both of them. The first means: There exists at least one value of *V* that makes *p* true. The second means: For all values of *V*, *p* is true. For

[2] The term *quantifier* derives from the verb to *quantify*, which means, loosely, "to say how many." The symbols ∃ ("backward E") and ∀ ("upside-down A") are often used in place of EXISTS and FORALL, respectively.

example, suppose the variable *V* ranges over the set "Members of the U.S. Senate in 2003," and suppose *p* is the WFF "*V* is female" (we are not trying to use our formal syntax here!). Then EXISTS *V* (*p*) and FORALL *V* (*p*) are both valid WFFs, and they evaluate to TRUE and FALSE, respectively.

Look again at the EXISTS example from the end of the previous subsection:

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

It follows from the foregoing that we can read this WFF as follows:

> *There exists a tuple SPX, say, in the current value of relvar SP such that the S# value in that tuple SPX is equal to the value of SX.S#*—whatever that might be—*and the P# value in that tuple SPX is P2.*

Each reference to SPX here is bound. The single reference to SX is free.

We define EXISTS formally as an iterated OR. In other words, if (a) *r* is a relation with tuples *t1*, *t2*, ..., *tm*, (b) *V* is a range variable that ranges over *r*, and (c) *p*(*V*) is a WFF in which *V* occurs as a free variable, then the WFF

```
EXISTS V ( p ( V ) )
```

is defined to be equivalent to the WFF

```
FALSE OR p ( t1 ) OR ... OR p ( tm )
```

Observe in particular that this expression evaluates to FALSE if *r* is empty (equivalently, if *m* is zero).

By way of example, suppose relation *r* contains just the following tuples (we depart here from our usual syntax for simplicity):

```
( 1, 2, 3 )
( 1, 2, 4 )
( 1, 3, 4 )
```

Suppose the three attributes, in left-to-right order as shown, are called *A*, *B*, and *C*, respectively, and every attribute is of type INTEGER. Then the following WFFs have the indicated values:

```
EXISTS V ( V.C > 1 )              :  TRUE
EXISTS V ( V.B > 3 )              :  FALSE
EXISTS V ( V.A > 1 OR V.C = 4 )   :  TRUE
```

We turn now to FORALL. Here to repeat is the FORALL example from the end of the previous subsection:

```
FORALL PX ( PX.COLOR = COLOR ('Red') )
```

We can read this WFF as follows:

> *For all tuples PX, say, in the current value of relvar P, the COLOR value in that tuple PX is Red.*

The two references to PX here are both bound.

Just as we define EXISTS as an iterated OR, so we define FORALL as an iterated AND. In other words, if *r*, *V*, and *p*(*V*) are as before (in our discussion of EXISTS), then the WFF

```
FORALL V ( p ( V ) )
```

is defined to be equivalent to the WFF

```
TRUE AND p ( t1 ) AND ... AND p ( tm )
```

Observe in particular that this expression evaluates to TRUE if *r* is empty (equivalently, if *m* is zero).

By way of example, let relation *r* be as for our EXISTS examples. Then the following WFFs have the indicated values:

```
FORALL V ( V.A > 1 )              :  FALSE
FORALL V ( V.B > 1 )              :  TRUE
FORALL V ( V.A = 1 AND V.C > 2 )  :  TROE
```

Note: We support both quantifiers purely for convenience—it is not logically necessary to support both, because each can be defined in terms of the other. To be specific, the equivalence

```
FORALL V ( p )  =  NOT EXISTS V ( NOT p )
```

(loosely, "all *V*'s satisfy *p*" is equivalent to "no *V*'s do not satisfy *p*") shows that any WFF involving FORALL can always be replaced by an equivalent WFF involving EXISTS instead. and *vice versa*. For example, the (true) statement "For all integers *x*, there exists an integer *y* such that $y > x$" (i.e., every integer has a greater integer) is equivalent to the statement "There does not exist an integer *x* such that there does not exist an integer y such that $y > x$" (i.e.. there is no greatest integer). However, some problems are more naturally formulated in terms of FORALL and others in terms of EXISTS; to be more specific, if one of the quantifiers is not available, we will sometimes find ourselves having to use double negation (as the foregoing example illustrates), and double negation is always tricky. In practice, therefore, it is desirable to support both.

Free and Bound Variable References Revisited

Suppose *x* ranges over the set of all integers, and consider the WFF:

```
EXISTS x ( x > 3 )
```

Observe now that *x* here is a kind of *dummy*—it serves only to link the boolean expression inside the parentheses to the quantifier outside. The WFF simply states that there exists some integer, *x* say, that is greater than three. *Note, therefore, that the meaning of this WFF would remain totally unchanged if all references to* x *were replaced by references to some other variable* y. In other words, the WFF

```
EXISTS y ( y > 3 )
```

is semantically identical to the one shown previously.