/THEORY/IN/PRACTICE

O'REILLY®

Database In Depth

Relational Theory for Practitioners

C. J. Date

Database in Depth

Database in Depth

Relational Theory for Practitioners

C.J. Date

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Database in Depth
Relational Theory for Practitioners
by C.J. Date

Copyright © 2005 O'Reilly Media, Inc.
All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Jonathan Gennick
Production Editor:	Genevieve d'Entremont
Cover Designer:	MendeDesign, *www.mendedesign.com*
Interior Designer:	Marcia Friedman
Creative Director:	Michele Wetherbee
Printing History:	May 2005: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.
ISBN: 0-596-10012-4
[M]

Dedicated to the memory of E. F. Codd

Those who are enamored of practice without theory are like a pilot who goes into a ship without rudder or compass and never has any certainty where he is going. Practice should always be based upon a sound knowledge of theory.

Leonardo da Vinci (1452 – 1519)

The trouble with people is not that they don't know but that they know so much that ain't so.

Josh Billings (1818 – 1885)

TABLE OF CONTENTS

	FOREWORD	*xi*
	PREFACE	*xiii*
1	*INTRODUCTION*	*1*
	A Remark on Terminology	2
	Principles, Not Products	3
	A Review of the Original Model	4
	Model Versus Implementation	11
	Properties of Relations	14
	Relations Versus Relvars	17
	Values Versus Variables	19
	Summary	20
	Exercises	21
2	*RELATIONS VERSUS TYPES*	*23*
	Domain-Constrained Comparisons	24
	Data Value Atomicity	29
	So What's a Type?	32
	Scalar Versus Nonscalar Types	35
	Summary	36
	Exercises	37
3	*TUPLES AND RELATIONS*	*41*
	What's a Tuple?	41
	Some Important Consequences	44
	What's a Relation?	45
	Further Important Consequences	47
	Why Duplicate Tuples Are Prohibited	48
	Why Nulls Are Prohibited	53
	TABLE_DUM and TABLE_DEE	56
	Summary	57
	Exercises	58

4 *RELATION VARIABLES* *61*

Updating Is Set-at-a-Time 62
More on Candidate Keys 63
More on Foreign Keys 65
More on Views 67
Relvars and Predicates 72
More on Relations Versus Types 75
Summary 77
Exercises 78

5 *RELATIONAL ALGEBRA* *81*

More on Closure 83
The Original Operators 86
Evaluating SQL Expressions 93
Extend and Summarize 95
Group and Ungroup 99
Expression Transformation 100
Relational Comparisons 103
More on Relational Assignment 106
The ORDER BY Operator 108
Summary 109
Exercises 110

6 *INTEGRITY CONSTRAINTS* *115*

Type Constraints 115
Database Constraints 119
Transactions 121
Why Database Constraint Checking Must Be Immediate 122
But Doesn't Some Checking Have to Be Deferred? 125
Constraints and Predicates 127
Miscellaneous Issues 129
Summary 131
Exercises 132

7	*DATABASE DESIGN THEORY*	*135*
	The Place of Design Theory	136
	Functional Dependencies and Boyce/Codd Normal Form	138
	Join Dependencies and Fifth Normal Form	144
	Two Cheers for Normalization	150
	Orthogonality	153
	Some Remarks on Physical Design	156
	Summary	158
	Exercises	159
8	*WHAT IS THE RELATIONAL MODEL?*	*163*
	The Relational Model Defined	164
	Objectives of the Relational Model	168
	Some Database Principles	168
	The Relational Model Versus Others	169
	What Remains to Be Done?	172
	Summary	176
	Exercises	177
A	*A LITTLE BIT OF LOGIC*	*181*
B	*SUGGESTIONS FOR FURTHER READING*	*199*
	INDEX	*203*

Foreword

I remember clearly the first time I didn't buy a book by Chris Date. That's right, I said "didn't buy." It was late 1991 or early 1992. I was in a small bookstore in the Saginaw, Michigan mall, looking for a book on SQL. I found one, too, by some guy named C. J. Date, whom I'd never heard of. The book looked good, the writing was clear, half of me wanted to buy it, but...oh, the price! I stood there holding a book in my hands that was about 1/4-inch thick, had less than 200 pages, yet it cost $30—a lot for me back in those days. I struggled with myself for a bit, then, reluctantly, I put the book back on the shelf and walked away.

What a mistake! A career was in the balance—I just didn't know it at the time.

I redeemed myself, though, by convincing my boss that the company should buy the book for me. It took them a month to get around to ordering the book, but it finally arrived, and I read it, not once, but several times. One of the best and most educational parts of the book (for me) was an appendix with Chris's critique of the then-current SQL language.

I learned a lot from Chris, from that book. I became enamored of working with databases, and fascinated by the idea of a declarative language like SQL: I could just describe the results I was after, and the database engine would do the work of deriving those results

for me. I also learned more about Chris, and about his role working with Codd from almost the very beginning of the relational era. For several years I subscribed to a magazine called *Database Programming and Design* for the sole reason that Chris wrote a column in it.

The knowledge I gained about SQL from Chris—beginning with that one thin, expensive little book that I was so reluctant to buy—has had a ripple effect down through the years until today. My introduction to databases at that time was a major turning point in my career, and Chris's clarity and rigor had a profound effect on the way in which I viewed SQL and database technology in general. Ever since, SQL has been one of my favorite things to write, to learn about, and, more recently, to write about.

I shudder to think of what my career might be today had I not gotten my hands on Chris's book. Don't make the same mistake I almost made. Don't put this book back on the shelf. Read it! Learn from someone who helped to invent and refine the relational model on which your career depends. You may not agree with everything Chris has to say, and you don't need to, but do *understand* what he says. Take the time to understand his vision of the relational model today. Take the time to understand the issues he raises with respect to how vendors (sometimes wrongly) implement that model in their products. Do these things, and you'll find yourself standing head-and-shoulders above your peers who haven't taken the time to ground themselves in the fundamentals.

If I can leave you with one thought, it's on the importance of enthusiasm for learning. It's been a great honor for me to edit this book by Chris Date, and an honor I never even remotely expected to have. I learned a lot from talking with Chris and reading his drafts. How did my work on this book come about? It was the indirect result of intense discussion of subquery optimization in SQL with several very smart people on the Oracle-L list whom I'm humbled to call colleagues. Even more indirectly, I suppose, my work on this book is a result of my reading Chris's writings for the first time, so very many years ago. Curiosity and a love for learning have taken me a lot further in my career than I ever used to dream of going, and they can do the same for you.

—Jonathan Gennick
Munising, Michigan
March 2005

Preface

After many years working in the database community in various capacities, I've come to realize there's a real need for a book for practitioners (not novices) that explains the basic principles of relational theory in a way not tainted by the quirks and peculiarities of existing products, commercial practice, or the SQL standard. I wrote this book to fill that need. My intended audience is thus experienced database practitioners or other database professionals who are honest enough to admit they don't understand the theory underlying their own field as well as they might, or should. That theory is, of course, the relational model—and while it's true that the fundamental ideas of that theory are all quite simple, it's also true that they're widely misrepresented, or underappreciated, or both. Often, in fact, they don't seem to be understood at all. For example, here are a few relational questions. How many of them can you answer?

1. What exactly is first normal form?
2. What's the connection between relations and predicates?
3. What's semantic optimization?
4. What's a join dependency?
5. Why is semidifference important?
6. Why doesn't deferred integrity checking make sense?

7. What's a relation variable?
8. What's nonloss decomposition?
9. Can a relation have an attribute whose values are relations?
10. What's the difference between SQL and the relational model?
11. Why is *The Information Principle* important?
12. How does XML fit with the relational model?

This book provides answers to these and many related questions. Overall, it's meant to help database practitioners understand relational theory in depth and make good use of that understanding in their professional day-to-day activities.

What Makes This Book Different?

I must immediately explain that very little of the technical substance of this book is new. I've said most of it before, in previous books and other publications—I've just looked around and seen that it needs to be said again. But I've tried to say it differently this time: the sequence is different, the development is different, the style and treatment are different, and the intended audience is different (more on this last point in a moment). So while parts of the material have appeared before in some form or another in a variety of places, I do regard this as a totally new book. Of course, some portions of the text are, inevitably, similar to things I've written elsewhere, because the material all comes out of the same place, as it were: namely, my own brain, and my experience in teaching this material in live seminars over many years. But there's no direct plagiarism; direct plagiarism wouldn't serve my purpose. However, I have consciously reused many of my old examples, because those examples have been very carefully tailored over the years to illustrate exactly the points I want to make, no more and no less.

Let me come back to that point about the intended audience for this book being different. As already indicated, I've published several previous books in the field of database technology. So how is this one different? In particular, does it compete with any of those existing books?

In my view, the answer to the latter question is no. I have two books from Addison-Wesley that might look at first sight as if they could be competitors to this one:

- *An Introduction to Database Systems*, Eighth Edition (2004)
- *Databases, Types, and the Relational Model: The Third Manifesto*, Third Edition (coauthored with Hugh Darwen, to appear 2006)

However, the first of these, though I call it an "introduction," actually covers the whole of the database field, not just the relational model. It's meant primarily as a college text, and it doesn't assume any prior database knowledge or experience on the part of its readers; also, the style is much more formal than that of the present book, as befits a textbook.

The second is an extensive reworking of an earlier book by Hugh Darwen and myself called *Foundation for Future Database Systems: The Third Manifesto*, Second Edition (Addison-Wesley, 2000). This one is an advanced (graduate-level?) text, and it's even more formal—not to say terse—than the first book. Although there's obviously some overlap in subject matter, therefore, I don't really see any of these three books as competing with the other two.

Another significant point of difference is that the present book is mainly meant for self-study (though there are portions you might want to discuss with your friends and colleagues and coworkers). There are exercises, too, to help reinforce the material; there's no obligation to do those exercises, of course, but I think it's a good idea to have a go at some of them at least. Answers, often giving more information about the subject at hand, can be found online at *http://oreilly.com/catalog/databaseid*.

While I'm on the topic of possible competition, I should mention a couple of other books of mine (the first from Addison-Wesley again, the other from Morgan Kaufmann):

- *The Database Relational Model: A Retrospective Review and Analysis* (2001)
- *Temporal Data and the Relational Model* (coauthored with Hugh Darwen and Nikos A. Lorentzos, 2003)

In my opinion, the first of these complements the present book, in that it reviews and analyzes, in a fairly informal style, the series of papers by Ted Codd that first introduced the relational model to the world at large. And the second is concerned, as its title indicates, not with relational theory as such but with a specific application of that theory. While the first chapter of that book does contain an overview of the relational model and thus *might* be considered to compete slightly with the present book, I don't really think it does.

The net of all of the above is this: although I've written on most of these topics before in a variety of places, and sometimes in unavoidably similar terms, I don't think any other publication by myself—or anyone else, so far as I know—brings them together and covers them in a way that's even close to the way the present book does.

Further Preliminaries

I need to take care of several further preliminaries. As I've already said, I'll be using some of the same examples here as in other books and articles of mine; in particular, the running example is the famous (or infamous) suppliers-and-parts database. I apologize for dragging this old warhorse out yet one more time, but the remark I made earlier about examples having been very carefully designed to illustrate exactly the points I want to make applies to this particular example in spades.

Second, my own understanding of the relational model has evolved over the years, and continues to do so. This book represents my very latest thinking on the subject; thus, if you detect any discrepancies between this book and the ones already mentioned (and

there are a few), the treatment in this book should be taken as superseding that in those earlier ones. Though I hasten to add that such discrepancies are mostly of a fairly minor nature; what's more, I've taken care always to relate new terms and concepts to earlier ones, whenever I felt it was necessary to do so.

Third, I am, of course, going to talk about theory—but it's an article of faith with me that *theory is practical.* I mention this point explicitly because so many people seem to believe the exact opposite: namely, that if something's theoretical, it can't be practical. But the truth is that theory—at least, the theory I'm talking about here, which is relational theory, of course—is most definitely very practical indeed. The purpose of that theory is *not* just theory for its own sake; the purpose of that theory is to allow us to build systems that are 100 percent practical. Every detail of the theory is there for solid practical reasons. Indeed, much of that theory is not only practical, it's fundamental, straightforward, simple, useful, and it can be *fun* (as I hope to demonstrate in the course of this book).

(In fact, we really don't have to look any further than the relational model itself to find the most striking possible illustration of the foregoing thesis. Indeed, it really shouldn't be necessary to defend the notion that theory is practical, in a context such as ours: namely, a multibillion dollar industry totally founded on one great theoretical idea. But I suppose the cynic's position would be "Yes, but what has theory done for me lately?" In other words, those of us who do think theory is important must continually justify ourselves to our critics—which is another reason why I think a book like this one is needed.)

And another point: the standard "relational" language is SQL, of course, and I assume you're reasonably familiar with that language, as well as with basic database concepts in general. As you'll soon see, however, I'm pretty critical of SQL in what follows. The sad fact is that SQL fails in all too many ways to support the relational model properly; it suffers from numerous sins of both omission and commission (which is why I said it was "relational," in quotation marks, when I first mentioned it). But this state of affairs is precisely one of the reasons why you need to know the relational model! Because SQL's support for the model is so deficient, it gives you rope to hang yourself; so you need to know the theory in order *not* to hang yourself—that is, you need to know the theory in order to enforce for yourself the various disciplines that SQL really ought to enforce on your behalf but doesn't. A good example is duplicate rows: SQL allows duplicate rows, but the relational model doesn't. So you need to know *why* the model doesn't allow them in order to understand why it's important not to "take advantage" of this particular SQL "feature." As one reviewer of my original proposal for this book, Stephane Faroult, wrote: "When you have a bit of practice, you realize there's no way to avoid having to know the theory."

Talking of SQL, by the way, please note that I use the term *SQL* throughout the book to mean the standard version of that language exclusively, not some product-specific dialect (barring explicit statements to the contrary, of course). In particular, I follow the standard in pronouncing the name "ess cue ell," not "sequel" (though this latter pronunciation is common in the field), and therefore say things like *an* SQL table, not *a* SQL table.

Finally, I'd like to mention that a live one-day seminar is available based on the material in this book. See *http://www.dbdebunk.com* or *http://www.thethirdmanifesto.com* for further details.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Used for emphasis and indicates new terms. Italic also indicates when a variable, such as *x*, is used in place of something else during a discussion in the main text of the book.

`Constant width`

Used for code examples.

`Constant width italic`

Marks the occurrence of a variable or user-supplied element in a code example.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *"Database in Depth: Relational Theory for Practitioners*, by C. J. Date. Copyright 2005 O'Reilly Media, Inc., 0-596-10012-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari Enabled

When you see a Safari® enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-Books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it free at *http://safari.oreilly.com*.

Comments and Questions

We've done our best to make this book as error-free as we can, but you might find mistakes. If so, please notify us by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (FAX)

You can also send messages electronically. To be put on the mailing list or request a catalog, send email to:

info@oreilly.com

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for this book, where you can find examples and errata (previously reported errors and corrections are available for public view there). You can access this page at:

http://www.oreilly.com/catalog/databaseid

For more information about this book and others, see the O'Reilly web site:

http://www.oreilly.com

Acknowledgments

It's a pleasure to acknowledge my debt to the many people involved, directly or indirectly, in the production of this book. In particular, I'd like to mention my reviewers Stephane Faroult, Jonathan Gennick, Lex de Haan, Anthony Molinaro, Peter Robson, and Michael Wener for their helpful comments on earlier drafts of the manuscript. I'd also like to thank Nagraj Alur and Hugh Darwen for various technical discussions. Next, I'd like to thank my wife Lindy (as always) for her support throughout this and all of my other database projects over the years. Finally, I'm grateful to everyone at O'Reilly—especially Jonathan Gennick and Genevieve d'Entremont—for their encouragement, contributions, and support throughout the production of this book.

—C. J. Date, Healdsburg, California, 2005

CHAPTER ONE

Introduction

Professionals in any discipline need to know the foundations of their field. So if you're a database professional, you need to know the relational model, because that model is the foundation (or a huge part of the foundation, anyway) of the database field in particular. Now, every course in database management, be it academic or commercial, does at least pay lip service to the idea of teaching the relational model—but most of that teaching seems to be done very badly, if results are anything to go by. The relational model certainly isn't very well or very widely understood in the database community at large. Here are some possible reasons for this state of affairs:

- The model is taught in a vacuum. That is, for beginners at least, it's hard to see the relevance of the material, or it's hard to understand the problems it's meant to solve, or both.
- The instructors themselves don't fully understand or appreciate the significance of the material.
- (Most likely in practice.) The model as such isn't taught at all—the SQL language or some specific dialect of that language, such as the Oracle dialect, is taught instead.

So this book is aimed at database professionals, especially commercial database practitioners, who have had some exposure to the relational model but don't know as

much about it as they ought to. It's definitely *not* meant for beginners; however, it isn't a refresher course, either. To be more specific: I'm sure you know something about SQL, but—and I apologize if my tone is somewhat offensive here—if your knowledge of the relational model derives only from your knowledge of SQL, then I'm afraid you won't know the relational model as well as you should, and you'll probably know "some things that ain't so."

NOTE

SQL ≠ the relational model!

Here by way of illustration are some relational issues that SQL isn't too clear on (to put it mildly):

- What databases, relations, and tuples really are
- The difference between relations and types
- The difference between relation values and relation variables
- The relevance of predicates and propositions
- The legitimacy of relation-valued attributes
- The crucial role of integrity constraints

and so on (this isn't an exhaustive list). All of these issues, and of course many others, are addressed in this book.

I say again: if your knowledge of the relational model derives only from your knowledge of SQL, then you might know "some things that ain't so." One unfortunate consequence of this state of affairs is that you might find, in reading this book, that there are some things you have to unlearn—and unlearning something is notoriously hard to do. And a related point...I'd like to recommend, politely, that you not skip the discussion of some topic because you think you're thoroughly familiar with that topic already. For example, are you *sure* you know exactly what a key is, in relational terms? Or a join?

A Remark on Terminology

In that list of relational issues in the foregoing section, you probably noticed right away that I used the formal terms *relation, tuple,** and *attribute*. SQL doesn't use these terms, of course—it uses the more "user-friendly" terms *table, row,* and *column* instead. And I'm generally sympathetic to the idea of more user-friendly terms if they can help make the ideas more palatable. In the case at hand, however, it seems to me that, regrettably, they don't make the ideas more palatable; instead, they distort them, and in fact do the cause of genuine understanding a grave disservice. The truth is, a relation is *not* a table, a tuple

* Usually pronounced to rhyme with couple.

is *not* a row, and an attribute is *not* a column. And while it might be acceptable to pretend otherwise in informal contexts—indeed, I've done so myself, in many of my books and other writings—I would argue that it's acceptable only if we all understand that the more user-friendly terms are just an approximation to the truth and fail overall to capture the essence of what's really going on. To put it another way: if you do understand the true state of affairs, then judicious use of the user-friendly terms can be a good idea; but in order to learn and appreciate that true state of affairs in the first place, you really do need to come to grips with the more formal terms. In this book, therefore, I'll use those more formal terms most of the time—and of course I'll give precise definitions for them when we need them (mostly not in this first chapter, though, where I'm just trying to lay a certain amount of elementary groundwork).

And another point on terminology: having said that SQL tries to simplify one set of terms, I must now add that it does its best to complicate another. I refer to its use of the terms *operator, function, procedure, routine,* and *method,* all of which refer to essentially the same thing (with, perhaps, very minor differences). In this book I'll use the term *operator* throughout.

Talking of SQL, by the way, *please note that I use the term SQL to mean the standard version of that language exclusively,* not some product-specific dialect—barring explicit statements to the contrary, of course. (I did mention this point in the preface, but I know that few people actually read prefaces.) In particular, my criticisms of SQL apply to the standard version specifically. Thus, if some particular criticism happens not to apply to your own favorite product, well, good, I'm glad to hear it (and bully for you).

Principles, Not Products

It's worth taking a few moments to examine the question of why, as I claimed earlier, you as a database professional need to know the relational model. The reason is that the relational model isn't product-specific; rather, it is concerned with *principles.* What do I mean by *principles?* Well, here's a definition (from *Chambers Twentieth Century Dictionary*):

> **principle:** a source, root, origin: that which is fundamental: essential nature: theoretical basis: a fundamental truth on which others are founded or from which they spring

The point about principles is this: they *endure.* By contrast, products and technologies (and the SQL language, come to that) change all the time—but principles don't. For example, suppose you know Oracle; in fact, suppose you're an expert on Oracle. But if Oracle is *all* you know, then your knowledge is not necessarily transferable to, say, a DB2 or SQL Server environment (it might even get in the way of your making progress in that new environment). But if you know the underlying principles—in other words, if you know the relational model—then you have knowledge and skills that *will* be transferable: knowledge and skills that you'll be able to apply in *every* environment and that will never be obsolete.

In this book, therefore, we'll be concerned with principles, not products, and foundations, not fads. Of course, I realize that sometimes you do have to make compromises and trade-offs in the real world. For one example, sometimes you might have good pragmatic reasons for not designing the database in the theoretically optimal way (an issue I discuss in Chapter 7). For another, consider SQL once again. Although it's certainly possible to use SQL relationally (for the most part, at any rate), sometimes you'll find—because existing implementations are so far from perfect—that there are severe performance penalties for doing so...in which case you might more or less be forced into doing something not "truly relational" (like writing a query in some weird and unnatural way in order to get the implementation to use an index). However, I believe very firmly that you should always make such compromises and trade-offs from *a position of conceptual strength.* That is:

- You should understand what you're doing when you do have to make such a compromise.
- You should know what the theoretically correct situation is, and you should have very good reasons for departing from it.
- You should document those reasons, too, so that if they go away at some future time (for example, because a new release of the product you're using does a better job in some respect), then it might be possible to back off from the original compromise.

The following quote—which is attributed to Leonardo da Vinci (1452–1519) and is thus some 500 years old!—sums up the situation admirably:

> Those who are enamored of practice without theory are like a pilot who goes into a ship without rudder or compass and never has any certainty where he is going. *Practice should always be based on a sound knowledge of theory.*

(OK, I added the italics.)

A Review of the Original Model

You're a database professional, so you already have some familiarity with the relational model. The purpose of this section is to serve as a kickoff point for our subsequent discussions; it reviews some of the most basic aspects of that model as originally defined. Note the qualifier "as originally defined"! One widespread misconception about the relational model is that it's a totally static thing. It's not. It's like mathematics in that respect: mathematics too is not a static thing but changes over time. In fact, the relational model can itself be seen as a small branch of mathematics; as such, it evolves over time as new theorems are proved and new results discovered. What's more, those new contributions can be made by anyone who's competent to do so. Like mathematics again, the relational model, though originally invented by one man, has become a community effort and now belongs to the world.

By the way, in case you don't know, that one man was E. F. Codd, at the 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 the field of database management, which was all too deficient in such qualities prior to that time. His original definition of the relational model appeared in an IBM Research Report in 1969, and I'll have a little more to say about that paper in Appendix B.

Structural Features

The original model had three major components—structure, integrity, and manipulation—and I'll briefly describe each in turn. Please note right away, however, that all of the "definitions" I'll be giving are very loose; I'll make them more precise as and when appropriate in later chapters.

First of all, then, structure. The principal structural feature is, of course, the relation itself, and as everybody knows it's common to picture relations as tables on paper (see Figure 1-1 for a self-explanatory example). Relations are defined over *types* (also known as *domains*); a type is basically a conceptual pool of values from which actual attributes in actual relations take their actual values. With reference to the simple departments-and-employees database illustrated in Figure 1-1, for example, there might be a type called DNO ("department numbers"), which is the set of all valid department numbers. The attribute called DNO in the DEPT relation and the attribute called DNO in the EMP relation then would each contain values that are taken from that conceptual pool. (By the way, it isn't necessary for attributes to have the same name as the corresponding type, and often they won't. We'll see plenty of counterexamples later.)

DEPT

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

EMP

ENO	ENAME	DNO	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

DEPT.DNO *referenced by* EMP.DNO

FIGURE 1-1. *The departments-and-employees database—sample values*

As I've said, tables like those in Figure 1-1 depict *relations: n*-ary relations, to be precise. An *n*-ary relation can be pictured as a table with *n* columns; the columns in the picture correspond to *attributes* of the relation and the rows correspond to *tuples*. Also, the value *n* can be any nonnegative integer. A 1-ary relation is said to be *unary;* a 2-ary relation, *binary;* a 3-ary relation, *ternary;* and so on.

* E for Edgar and F for Frank—but he always signed with his initials. To his friends, among whom I was proud to count myself, he was Ted.

The relational model also supports various kinds of *keys*. To begin with, every relation has at least one *candidate key*.* A candidate key is just a unique identifier; in other words, it's a combination of attributes—often, but not always, a "combination" involving just one attribute—such that every tuple in the relation has a unique value for the combination in question. In Figure 1-1, for example, every department has a unique department number and every employee has a unique employee number, so we can say that {DNO} is a candidate key for DEPT and {ENO} is a candidate key for EMP. Note the braces, by the way: candidate keys are always *combinations*, or *sets*, of attributes—even when the set contains just one attribute—and braces are conventionally used to enclose sets of things.

Next, a *primary key* is a candidate key that's been singled out for special treatment in some way. If a given relation has just one candidate key, then it obviously makes no real difference if we say it's the primary key. But if the relation has two or more candidate keys, then we're supposed to choose one of them as primary, meaning it's somehow "more equal than the others." Suppose, for example, that every employee has a unique employee number *and* a unique employee name, so that {ENO} and {ENAME} are both candidate keys for EMP. Then we might choose {ENO}, say, to be the primary key.

Notice I said we're *supposed* to choose a primary key. If there's just one candidate key, then there's no choice and no problem. But if there are two or more, then choosing one and making it primary smacks a little bit of arbitrariness (at least to me); certainly there are situations where there don't seem to be any good reasons for making such a choice. In this book, I usually *will* follow the primary key discipline—and in pictures like Figure 1-1 I'll mark primary key attributes by double underlining—but I stress the point that it's really candidate keys, not primary keys, that are significant from a relational point of view. Partly for this reason, from this point forward I'll use the term *key*, unqualified, to mean a candidate key specifically. (In case you were wondering, the "special treatment" enjoyed by primary keys over other candidate keys is mainly syntactic in nature, anyway; it isn't fundamental, and it isn't very important.)

Finally, a *foreign key* is a set of attributes in one relation whose values are required to match the values of some candidate key in some other relation (or possibly in the same relation). With reference to Figure 1-1, for example, {DNO} is a foreign key in EMP whose values are required to match values of the candidate key {DNO} in DEPT (as I've tried to suggest by means of a suitably labeled arrow in the figure). By *required to match*, I mean that if, for example, EMP includes a tuple in which DNO has the value D2, then DEPT had better also include a tuple in which DNO has the value D2; otherwise, EMP would show some employee as being in a nonexistent department, and the database wouldn't be "a faithful model of reality."

* Strictly speaking, this sentence should read "Every *relvar* has at least one candidate key" (see the section "Relations Versus Relvars," later in this chapter). Similar remarks apply at various places elsewhere in this chapter, too (see Exercise 1-1 at the end of the chapter).

Integrity Features

An *integrity constraint* (*constraint* for short) is basically just a boolean expression that must evaluate to TRUE. In the case of departments and employees, for example, we might have a constraint to the effect that SALARY values must be greater than zero. Now, any given database will be subject to numerous constraints, but those constraints will necessarily be expressed in terms of the relations in that particular database and will be specific to that database. By contrast, the relational model (at least as originally formulated) includes two *generic* integrity rules—generic in the sense that they apply to every database, loosely speaking. One has to do with primary keys and the other with foreign keys:

Entity integrity
: Primary key attributes don't permit nulls.

Referential integrity
: There mustn't be any unmatched foreign key values.

Let me explain the second first. By the term *unmatched foreign key value,* I mean a foreign key value for which there doesn't exist an equal value of the corresponding candidate key. Thus, for example, the departments-and-employees database would be in violation of the referential integrity rule if it included an EMP tuple with, say, a DNO value of D2 but no corresponding DEPT tuple. So the referential integrity rule simply spells out the semantics of foreign keys; the name *referential integrity* derives from the fact that any given foreign key value can be regarded as a *reference* to the tuple with that same value for the corresponding candidate key. In effect, therefore, the rule just says: "If *B* references *A*, then *A* must exist."

As for the entity integrity rule, well, here I have a problem. The fact is, I reject the concept of "nulls" entirely; that is, it is my very strong opinion that *nulls have no place in the relational model.* (Codd thought otherwise, obviously, but I have strong reasons for taking the position I do.) In order to explain the entity integrity rule, therefore, I need to suspend disbelief, as it were (at least for the time being), but please understand that I'll be revisiting the whole issue of nulls in Chapter 3.

In essence, then, a null is a "marker" that means *value unknown* (crucially, it's not itself a value; it is, to repeat, a *marker,* or *flag*). For example, suppose we don't know employee E2's salary. Then, instead of entering some real SALARY value in the tuple for that employee in relation EMP—we *can't* enter a real value, by definition, precisely because we don't know what that value should be—we *mark* the SALARY position within that tuple as null:

ENO	ENAME	DNO	SALARY
E2	Cheng	D1	

As you can see, this tuple contains *nothing at all* in the SALARY position. In this book I'll use shading as just shown to highlight such empty positions; you can think of that shading as constituting the null "marker" or flag.

In terms of relation EMP, then, the entity integrity rule says, loosely, that an employee might have an unknown name, department, or salary, but *not* an unknown employee number—because if the employee number were unknown, we wouldn't even know which employee (that is, which "entity") we were talking about.

That's all I want to say about nulls for now. Forget about them until Chapter 3.

Manipulative Features

The manipulative part of the model consists of:

- A set of relational operators, such as difference (or MINUS), collectively called the *relational algebra,* together with
- A *relational assignment* operator that allows the value of some relational expression, such as *r* MINUS *s* (where *r* and *s* are relations), to be assigned to some relation.

The relational assignment operator is fundamentally how updates are done in the relational model,* and I'll have more to say about it later, in the section "Relations Versus Relvars." As for the relational algebra, it consists of a set of operators that allow "new" relations to be derived from "old" ones (speaking very loosely). More precisely, each operator takes at least one relation as input and produces another relation as output; for example, difference (or MINUS) takes two relations as input and "subtracts" one from the other to derive another relation as output. And it's very important that the output is another relation: that's the well-known *closure* property of the relational algebra. The closure property is what lets us write *nested relational expressions;* since the output from every operation is the same kind of thing as the input, the output from one operation can become the input to another—meaning, for example, that we can take the difference between *r* and *s*, feed the result as input to a union with some relation *u*, feed that result as input to an intersection with some relation *v*, and so on.

Now, any number of operators can be defined that fit the simple definition of "at least one relation in, exactly one relation out." In the following list I'll briefly describe what are usually thought of as the original eight operators (essentially the ones Codd defined in his earliest papers); in Chapter 5 I'll introduce a number of additional operators and describe them in more detail. Figure 1-2 is a pictorial representation of the original eight operators. *Note:* If you're unfamiliar with any of these operators—especially divide!—and find the following brief descriptions hard to follow, don't worry about it; I'll be going into much more detail, with lots of examples, later (mostly in Chapter 5).

* I follow convention throughout this book in using the generic term "update" to refer to the INSERT, DELETE, and UPDATE (and assignment) operators considered collectively. When I want to refer to the UPDATE operator specifically, I'll set it in all caps as just shown.

Restrict

Returns a relation containing all tuples from a specified relation that satisfy a specified condition. For example, we might restrict relation EMP to just the tuples where the DNO value is D2.

Project

Returns a relation containing all (sub)tuples that remain in a specified relation after specified attributes have been removed. For example, we might project relation EMP on just the ENO and SALARY attributes.

Product

Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations. Product is also known variously as *cartesian product, cross product, cross join,* and *cartesian join* (in fact, it is just a special case of join, as we'll see in Chapter 5).

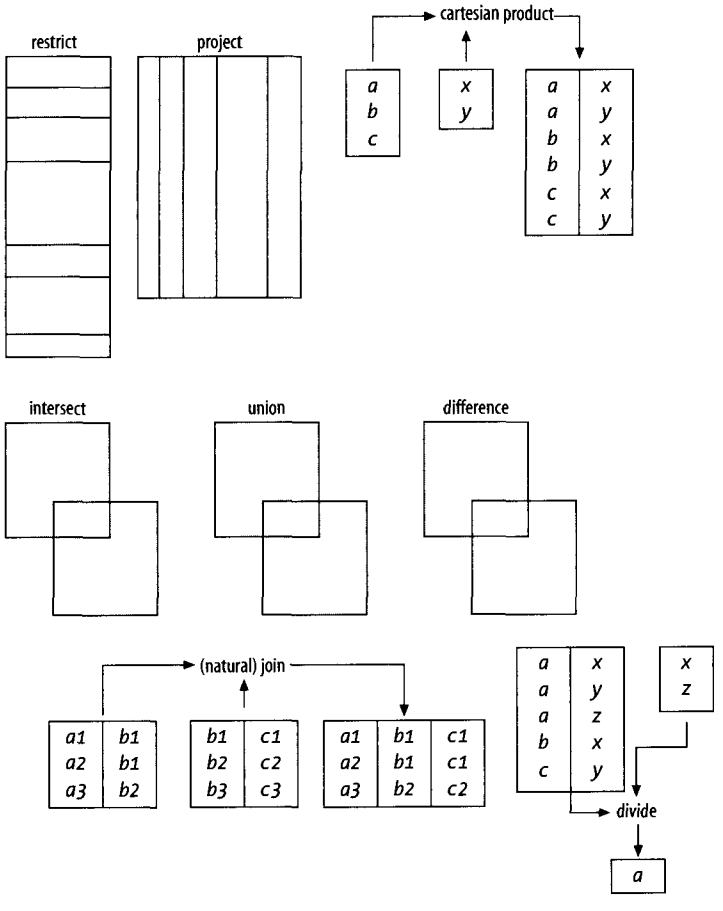


FIGURE 1-2. *The original relational algebra (results shaded)*

Intersect

Returns a relation containing all tuples that appear in both of two specified relations. (Actually, intersect also is a special case of join.)

Union

Returns a relation containing all tuples that appear in either or both of two specified relations.

Difference

Returns a relation containing all tuples that appear in the first and not the second of two specified relations.

Join

Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations, such that the two tuples contributing to any given result tuple have a common value for the common attributes of the two relations (and that common value appears just once, not twice, in that result tuple).

> **NOTE**
>
> This kind of join was originally called the *natural* join. Since natural join is far and away the most important kind, however, it's become standard practice to take the unqualified term *join* to mean the natural join specifically, and I'll follow that practice in this book.

Divide

Takes two relations, one binary and one unary, and returns a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.

One last point to close this subsection: as you probably know, there's also something called the *relational calculus*. The relational calculus can be regarded as an alternative to the relational algebra; that is, instead of saying the manipulative part of the relational model consists of the relational algebra (plus relational assignment), we can equally well say it consists of the relational calculus (plus relational assignment). The two are equivalent and interchangeable, in the sense that for every algebraic expression there's a logically equivalent expression of the calculus and vice versa. I'll have a little more to say about the calculus in Appendix A.

The Running Example

I'll finish up this brief review by introducing the example I'll use as the basis for most if not all of the discussions in the rest of the book: the well-known suppliers-and-parts database (see Figure 1-3). To elaborate:

Suppliers

Relation S denotes suppliers (more accurately, suppliers under contract). Each supplier has one supplier number (SNO), which is unique to that supplier (and so {SNO} is the

primary key); one name (SNAME), not necessarily unique (though the SNAME values in Figure 1-3 do happen to be unique); one rating or status value (STATUS); and one location (CITY).

Parts

Relation P denotes parts (more accurately, kinds of parts). Each kind of part has one part number (PNO), which is unique (so {PNO} is the primary key); one name (PNAME); one color (COLOR); one weight (WEIGHT); and one location where parts of that kind are stored (CITY).

Shipments

Relation SP denotes shipments (it shows which parts are supplied by which suppliers). Each shipment has one supplier number (SNO), one part number (PNO), and one quantity (QTY). For the sake of the example, I assume there's at most one shipment at any given time for a given supplier and a given part (and so {SNO,PNO} is the primary key; also, {SNO} and {PNO} are both foreign keys, matching the primary keys of S and P, respectively). Note that the database shown in Figure 1-3 includes one supplier, supplier S5, with no shipments at all.

S

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

SP

SNO	PNO	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

P

PNO	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

FIGURE 1-3. *The suppliers-and-parts database—sample values*

Model Versus Implementation

Before going any further, there's one very important point I need to explain, because it underpins everything else in this book. The relational model is, of course, a *data* model. Unfortunately, however, this latter term has two quite distinct meanings in the database world. The first and more fundamental meaning is this:

> *Definition:* A *data model* (first sense) is an abstract, self-contained, logical definition of the data structures, data operators, and so forth, that together make up the *abstract machine* with which users interact.

This is the meaning we have in mind when we talk about the relational model in particular. And, armed with this definition, we can usefully (and importantly) go on to

distinguish a data model in this first sense from its *implementation,* which can be defined as follows:

> *Definition:* 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.

I'll illustrate these definitions in terms of the relational model specifically. First, and obviously enough, the concept of *relation* is itself part of the model: users have to know what relations are, they have to know they're made up of tuples and attributes, they have to know how to interpret them, and so on. All that is part of the model. But they don't have to know how relations are physically stored on the disk, or how individual data values are physically encoded, or what indexes or other access paths exist; all that is part of the implementation, not part of the model.

Or consider the concept *join:* users have to know what a join is, they have to know how to invoke a join, they have to know what the result of a join looks like, and so on. Again, all that is part of the model. But they don't have to know how joins are physically implemented, or what expression transformations take place under the covers, or what indexes or other access paths are used, or what physical I/O's occur;* all that is part of the implementation, not the model.

In a nutshell, then:

- The model (first meaning) is *what the user has to know.*
- The implementation is what the user *doesn't* have to know.

(Of course, I'm not saying users aren't *allowed* to know about the implementation; I'm just saying they don't have to. In other words, everything to do with the implementation should be, at least potentially, hidden from the user.)

Here are some important consequences of the foregoing definitions. First, note that performance is fundamentally an implementation issue, not a model issue—despite extremely common misconceptions to the contrary. We're often told, for example, that "joins are slow." But such remarks make no sense! Join is part of the model, and the model as such can't be said to be either fast or slow; only implementations can be said to possess any such quality. Thus, we might reasonably say that some specific product *X* has a faster or slower implementation of some specific join than some other specific product *Y*—but that's all.

* I/O = input/output operation.

NOTE

I don't want to give the wrong impression here. It's true that performance is basically an implementation issue; but that doesn't mean a good implementation will perform well if you use the model badly! Indeed, this is precisely one of the reasons why you need to know the model (I mean, so that you *don't* use it badly). If you write an expression such as S JOIN SP, you're within your rights to expect the implementation to do a good job on it; but if you insist on (in effect) hand-coding the join yourself, perhaps like this:

```
do for all tuples in S ;
   fetch S tuple into TNO, TN, TS, TC ;
   do for all tuples in SP with SNO = TNO ;
      fetch SP tuple into TNO, TP, TQ ;
      emit tuple TNO, TN, TS, TC, TP, TQ ;
   end ;
end ;
```

then there's no way you're going to get good performance. Relational systems should not be used like simple access methods.

Second, as you probably realize, it's precisely the fact that model and implementation are logically distinct that enables us to achieve *data independence*. Data independence (not a great term, by the way, but we're probably stuck with it) means we have the freedom to change the way the data is physically stored and accessed without having to make corresponding changes in the way the data is perceived by the user. The reason we might want to change those storage and access details is, of course, performance; and the fact that we can make such changes without having to change the way the data looks to the user means that existing application programs, queries, and so on can still work. Very importantly, therefore, data independence means *protecting your investment in user training and applications.*

As you can see from the foregoing definitions, 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, most of today's database systems, even those that claim to be relational, don't make those distinctions as clearly as they should. As a direct consequence, they deliver far less data independence than they should, and far less than relational systems are theoretically capable of. I'll come back to this issue in the next section, as well as in Chapter 7.

Now I want to turn to the second meaning of the term *data model,* which I dare say you're very familiar with. It can be defined thus:

> *Definition:* A *data model* (second sense) is a model of the persistent data of some particular enterprise.

In other words, a data model in the second sense is just a (possibly somewhat abstract) *database design*. For example, we might speak of the data model for some bank, or some hospital, or some government department.

Having now explained these two different meanings, I'd like to draw your attention to an analogy that I think nicely illuminates the relationship between them:

- A data model in the first sense is like a programming language, whose constructs can be used to solve many 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—it uses the facilities provided by the model, in the first sense of that term, to solve some specific problem.

By the way, it follows from all of the above that if we're talking about data models in the second sense, we might reasonably speak of "relational models" in the plural or "a" relational model (with an indefinite article). But if we're talking about data models in the first sense, then *there's only one relational model,* and it's *the* model (with the definite article). I'll have more to say on this issue in Chapter 8.

For the rest of this book I'll use the term *data model*—or usually just *model* for short—exclusively in its first sense.

Properties of Relations

Now let's get back to an examination of basic relational concepts. In this section I want to focus on some specific properties of relations themselves. First of all, every relation has a *heading* and a *body:* the heading is a set of attributes (where an *attribute* is an attribute-name:type-name pair), and the body is a set of tuples that conform to that heading. In the case of the suppliers relation of Figure 1-3, for example, there are four attributes in the heading and five tuples in the body. Note, therefore, that a relation doesn't really contain tuples—it contains a body, and that body in turn contains the tuples—but we do usually talk as if relations contained tuples directly, for the sake of simplicity.

By the way, although it's strictly correct to say that the heading consists of attribute-name:type-name pairs, it's usual to omit the type names in pictures like Figure 1-3 and thereby pretend that the heading is a set of attribute names only. For example, the STATUS attribute does have a type (INTEGER, let's say), but I didn't show it in Figure 1-3. But you should never forget it's there!

Next, the number of attributes in the heading is the *degree* (sometimes the *arity*), and the number of tuples in the body is the *cardinality*. For example, relations S, P, and SP in Figure 1-3 have degree 4, 5, and 3, respectively, and cardinality 5, 6, and 12, respectively.

Next, relations *never* contain duplicate tuples. This property follows because a body is a *set* of tuples, and sets in mathematics do not contain duplicate elements. By the way, SQL fails here: SQL tables are allowed to contain duplicate rows, as I'm sure you know, and SQL tables are thus not relations, in general. Please understand, therefore, that in this book I *always* use the term "relation" to mean a relation—without duplicate tuples, by definition—and not an SQL table. Please understand also that relational operations *always* produce a result without duplicate tuples, again by definition. For example, projecting the

suppliers relation of Figure 1-3 on CITY produces the result shown on the left and *not* the one on the right:

CITY
London
Paris
Athens

CITY
London
Paris
Paris
London
Athens

(The result on the left can be obtained via the SQL query SELECT DISTINCT S.CITY FROM S. Omitting DISTINCT leads to the nonrelational result on the right. Note in particular that the table on the right has no double underlining; that's because it has no key, and hence *a fortiori* no primary key.)

Next, the tuples of a relation are *unordered*, top to bottom. This property follows because, again, a body is a set, and sets in mathematics have no ordering to their elements. (Thus, for example, $\{a,b,c\}$ and $\{c,a,b\}$ are the same set in mathematics, and the same kind of thing is naturally true in the relational model.) Of course, when we draw a relation as a table on paper, we do have to show the rows in some top-to-bottom order, but that ordering doesn't correspond to anything relational. In the case of the suppliers relation of Figure 1-3, for example, I could have shown the rows in any order—say, supplier S3, then S1, then S5, then S4, then S2—and the picture would still represent the same relation.

NOTE

The fact that the tuples of a relation are unordered doesn't mean queries can't include an ORDER BY specification, but it does mean that such queries produce a result that's not a relation. ORDER BY is useful for the purpose of displaying results, but it isn't a relational operator as such.

In similar fashion, the attributes of a relation are also *unordered*, left to right, because a heading too is a mathematical set. Again, when we draw a relation as a table on paper, we have to show the columns in some left-to-right order, but that ordering doesn't correspond to anything relational. In the case of the suppliers relation of Figure 1-3, for example, I could have shown the columns in any left-to-right order—say STATUS, SNAME, CITY, SNO—and the picture would still represent the same relation in the relational model. Incidentally, SQL fails here too: SQL tables do have a left-to-right ordering to their columns (another reason why SQL tables aren't relations, in general). For example, the pictures below represent the same relation, but two different SQL tables:

SNO	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

CITY	SNO
London	S1
Paris	S2
Paris	S3
London	S4
Athens	S5

(The corresponding SQL queries are, respectively, SELECT S.SNO, S.CITY FROM S and SELECT S.CITY, S.SNO FROM S. By the way, you might be thinking that the difference between these two tables is hardly very significant; in fact, however, it has some serious consequences, some of which I'll be touching on in later chapters.)

Next, relations are always *normalized* (equivalently, they're in *first normal form*, 1NF). Informally, this means that, in terms of the tabular picture of a relation, at every row-and-column intersection we always see just a single value. More formally, it means that every tuple in every relation contains just a single value, of the appropriate type, in every attribute position. I'll have quite a lot more to say on this particular issue in the next chapter.

Next, we draw a distinction between *base* and *derived* relations. As I explained earlier, the operators of the relational algebra allow us to start with some given relations—perhaps those of Figure 1-3—and obtain further relations from those given ones. The given relations are the base ones; the others are derived. Now, a relational system obviously has to provide a means for defining the base relations in the first place. In SQL, this task is performed by the CREATE TABLE statement (the SQL counterpart to a base relation being, of course, a base *table*). And base relations obviously have to be named—for example:

```
CREATE TABLE SP ... ;
```

But certain derived relations—including in particular what are called *views*—are also named. A view (also known as a *virtual relation*) is a named relation whose value at all times is the result of evaluating a certain relational expression at the time in question. Here's an SQL example:

```
CREATE VIEW SST_PARIS AS
   SELECT S.SNO, S.STATUS
   FROM   S
   WHERE  S.CITY = 'Paris' ;
```

You can operate on views as if they were base relations, but they *aren't* base relations; instead, you can think of views as being materialized—in effect, you can think of a base relation as being dynamically built—at the time they're referenced. (Though I should emphasize that thinking of views being materialized when they're referenced is *only* a way of thinking; it's not what's really supposed to happen. How views really work is explained in Chapter 4.)

There's an important point I need to make here. You'll often see people describe the difference between base relations and views like this:

- Base relations really exist—that is, they're physically stored in the database.
- Views, by contrast, don't "really exist"—they merely provide different ways of looking at the base relations.

But the relational model has nothing to say about what's physically stored! In particular, it does *not* say that base relations are physically stored. The only requirement is that there must be some mapping between whatever *is* physically stored and those base relations, so that those base relations can somehow be constructed when they're needed (conceptually speaking, at any rate). If the base relations can be constructed in this way, then so can everything else. For example, we might physically store the join of suppliers and shipments, instead of storing them separately; base relations S and SP then could be constructed, conceptually, by taking appropriate projections of that join. In other words, base relations are no more (and no less!) "physical" than views are, so far as the relational model is concerned.

The fact that the relational model says nothing about physical storage is deliberate, of course. The idea was to give implementers the freedom to implement the model in whatever way they chose—in particular, in whatever way seemed likely to yield good performance—without compromising on data independence. The sad fact is, however, that SQL vendors seem mostly not to have understood this point; instead, they map base tables fairly directly to physical storage,* and (as noted in the previous section) their products therefore provide far less data independence than relational systems are theoretically capable of. Indeed, this state of affairs is reflected in the SQL standard itself—as well as in most other SQL documentation—which typically (fairly ubiquitously, in fact) uses expressions such as "tables and views." Clearly, anyone who uses such an expression is under the impression that tables and views are different things, and probably under the impression too that "tables" are physical and views aren't. But the whole point about a view is that it *is* a table (or, as I would prefer to say, a relation); that is, we can perform the same kinds of operations on views as we can on regular relations (at least in the relational model), because views *are* "regular relations"! Throughout this book, therefore, I'll reserve the term *relation* to mean a relation (possibly a base relation, possibly a view, possibly a query result, and so on); if I mean (for example) a base relation specifically, then I'll *say* "base relation." And I suggest very strongly that you adopt the same discipline for yourself. Don't fall into the common trap of thinking that the term *relation* means a base relation specifically.

Relations Versus Relvars

Now, it's entirely possible that you already knew everything I've discussed in this chapter so far; in fact, I rather hope you did (though I also hope that doesn't mean you found the discussions boring). Anyway, now I come to something you might not know already. The fact is, historically there's been a lot of confusion between relations—I mean relations as such—on the one hand, and relation *variables* on the other.

* I say this knowing full well that today's SQL products provide a variety of options for hashing, partitioning, indexing, clustering, and otherwise organizing the data as stored on the disk. I still consider the mapping to physical storage in those products to be "fairly direct."

Forget about databases and relations for a moment; instead, consider the following simple programming language example. Suppose I say in some arbitrary programming language:

```
DECLARE N INTEGER ... ;
```

Then N here *is not an integer*. Rather, it's a *variable* whose *values* are integers as such (different integers at different times). Right? I'm sure we can agree on that. Well, in exactly the same way, if we say in SQL:

```
CREATE TABLE T ... ;
```

then T is not a table; it's a *table variable* or (as I would prefer to say, ignoring various SQL quirks such as nulls and duplicate rows) a *relation* variable, whose values are relations as such (different relations at different times).

Consider Figure 1-3 once again. That figure shows three relation values: namely, those that happen to appear in the database at some particular time. But if we were to look at the database at some different time, we would probably see three different relation values appearing in their place. In other words, S, P, and SP in that database are really variables: relation variables, to be precise. For example, suppose the relation variable S currently has the value—the relation value—shown in Figure 1-3, and suppose we delete the tuples (actually there's only one) for suppliers in Athens:

```
DELETE S WHERE CITY = 'Athens' ;
```

Here's the result:

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

Conceptually, what's happened is that the old value of S has been replaced in its entirety by a new value. Of course, the old value (with five tuples) and the new one (with four) are very similar, but they certainly are different values. In fact, the DELETE just shown is logically equivalent to, and indeed shorthand for, the following relational assignment:*

```
S := S WHERE NOT ( CITY = 'Athens' ) ;
```

* I can't show this in SQL because SQL doesn't directly support relational assignment. Throughout this book, I'll show examples in SQL wherever possible—but when it's not possible for some reason, as here, I'll use a more or less self-explanatory (and truly relational) language called **Tutorial D** instead. **Tutorial D** is the language Hugh Darwen and I use to illustrate relational ideas in our book *Databases, Types, and the Relational Model: The Third Manifesto*, Third Edition (Addison-Wesley, 2006); you can regard it as a realization in concrete syntax of the abstract constructs of the relational model (which SQL, regrettably, is not).

As with all assignments, the effect here is that (a) the *source expression* on the right side is evaluated, and (b) the result of that evaluation is then assigned to the *target variable* on the left side, with the overall result already explained.

In like fashion, of course, the familiar INSERT and UPDATE statements are also really shorthand for certain relational assignments. Thus, as I mentioned in the section "A Review of the Original Model," relational assignment is the fundamental update operator in the relational model, and indeed it's the only update operator that's really needed, logically speaking.

So we have these two different concepts, relation value and relation variable. The trouble is that the literature has historically used the same term, *relation*, to stand for both, and that practice has certainly led to confusion. In this book, therefore, I'll distinguish very carefully between the two from this point forward—I'll talk in terms of relation values when I mean relation values and relation variables when I mean relation variables. However, I'll also abbreviate *relation value* most of the time to just *relation* (exactly as we abbreviate *integer value* most of the time to just *integer*). And I'll abbreviate *relation variable* most of the time to just *relvar;* for example, I'll say the suppliers-and-parts database contains three *relvars*.

As an exercise, you might like to go back over the text of this chapter so far and see exactly where I used the term *relation* when I really ought to have used the term *relvar* instead (or as well).

Values Versus Variables

The difference between relations and relvars is actually a special case of the difference between values and variables in general, and I'd like to take a few moments to look at the more general case. (It's a bit of a digression, but I think it's worth taking the time because clear thinking here can be a tremendous help in so many areas.) Here then are some definitions:

> *Definition:* A *value* is an "individual constant": for example, the individual integer 3. A value has no location in time or space. However, values can be represented in memory by means of some encoding, and those representations do have locations in time and space. Indeed, distinct representations of the same value can appear at any number of distinct locations in time and space, meaning, loosely, that any number of different variables—see the definition below—can have the same value, at the same time or different times. Observe in particular that, by definition, *a value can't be updated;* for if it could, then after such an update it would no longer be that value.
>
> *Definition:* A *variable* is a holder for a representation of a value. A variable does have location in time and space. Also, of course, variables, unlike values, *can be updated;* that is, the current value of the variable can be replaced by another value.

Please note very carefully that it isn't 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. I'll have more to say on these matters in the next two chapters.

Now, it might be hard to imagine people getting confused over a distinction as obvious as that between values and variables, but in fact it's all too easy to fall into traps in this area. By way of illustration, consider the following extract from a tutorial on object-oriented databases (the italicized portions in brackets are comments by myself):

> We distinguish the declared type of a variable from...the type of the object that is the current value of the variable [*so an object is a value*]...We distinguish objects from values [*so an object isn't a value after all*]...A *mutator* [is an operator such that it's] possible to observe its effect on some object [*so in fact an object is a variable*].

Summary

For the most part, the aim of this preliminary chapter has been to tell you what I rather hope you knew already (and you might therefore have felt it was rather light on technical substance). Anyway, just to review briefly:

- I explained why we'd be concerned in this book with principles, not products, and why I'd be using formal terminology such as *relations*, *tuples*, and *attributes* in place of their more "user-friendly" SQL counterparts.
- I claimed that SQL and the relational model aren't the same thing. We've seen a few differences already—for example, the fact that SQL permits duplicate rows—and we'll see many more in later chapters.
- I gave an overview of the original model, touching in particular on the following concepts: *type*, *n-ary relation*, *tuple*, *attribute*, *candidate key*, *primary key*, *foreign key*, *entity integrity*, *referential integrity*, *relational assignment*, and the *relational algebra*. With regard to the algebra, I mentioned *closure* and very briefly described the operators *restrict*, *project*, *product*, *intersection*, *union*, *difference*, *join*, and *divide*.
- I discussed various properties of relations, introducing the terms *heading*, *body*, *cardinality*, and *degree*. Relations have no duplicate tuples, no tuple ordering top to bottom, and no attribute ordering left to right. I also discussed *views*.
- I discussed the differences between *model* and *implementation*, *relations* and *relvars*, and *values* and *variables* in general. The model versus implementation discussion in particular led to a discussion of *data independence*.

One last point (I didn't mention this explicitly before, but I hope it's obvious from everything I did say): overall, the relational model is *declarative*, not *procedural*, in nature; that is, we favor declarative solutions over procedural ones, wherever such solutions are feasible. The reason is obvious: *declarative* means the system does the work, *procedural*

means the user does the work (so we're talking about productivity, among other things). That's why the relational model supports declarative queries, declarative updates, declarative view definitions, declarative integrity constraints, and so on.*

Exercises

As noted in the preface, you certainly don't have to do any of the exercises, but I think it's a good idea to try at least some of them. Answers, often giving more information about the subject at hand, can be found online at *http://oreilly.com/catalog/databaseid.*

Exercise 1-1. (Repeated from the body of the chapter, but slightly reworded here.) If you haven't done so already, go through the chapter again and identify all of the places where I used the term *relation* when I should by rights have used the term *relvar* instead (or as well).

Exercise 1-2. Who was E. F. Codd?

Exercise 1-3. What's a domain?

Exercise 1-4. What do you understand by the term *referential integrity?*

Exercise 1-5. The terms *heading, body, attribute, tuple, cardinality,* and *degree,* defined in the body of the chapter for relation values, can all be interpreted in the obvious way to apply to relvars as well. Make sure you understand this remark.

Exercise 1-6. Distinguish between the two meanings of the term *data model.*

Exercise 1-7. Explain the difference between model and implementation in your own words.

Exercise 1-8. In the body of the chapter, I said that tables like those in Figures 1-1 and 1-3 weren't relations as such but, rather, *pictures* of relations. What are some of the specific points of difference between such pictures and the corresponding relations?

Exercise 1-9. Explain *data independence* in your own words.

Exercise 1-10. (Try this exercise without looking back at the body of the chapter.) What relvars are contained in the suppliers-and-parts database? What attributes do they involve? What candidate and foreign keys exist? (The point of this exercise is that it's worth making yourself as familiar as possible with the structure, at least, of the running example. Of course, it's not so important to remember the actual data values in detail—though it wouldn't hurt if you did.)

Exercise 1-11. "There's only one relational model." Explain this remark.

* As this book was going to press, I was informed that at least one well-known SQL product apparently uses the term "declarative" to mean the system *doesn't* do the work! That is, it allows the user to state certain things declaratively (for example, the fact that a certain view has a certain key), but it doesn't enforce the constraint implied by that declaration—it simply assumes the user is going to enforce it instead. Such terminological abuses do little to help the cause of genuine understanding. *Caveat lector.*

Exercise 1-12. The following is an excerpt from a recent database textbook: "[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*." This text betrays several confusions or misconceptions regarding the relational model. Identify as many as you can.

Exercise 1-13. The following is an excerpt from another recent database textbook: "[The relational] model...defines simple tables for each relation and many-to-many relationships. Cross-reference keys link the tables together, representing the relationships between entities. Primary and secondary indexes provide rapid access to data based upon qualifications." This text is intended as a *definition* of the relational model... what's wrong with it?

Exercise 1-14. Write SQL CREATE TABLE statements to define an SQL version of the suppliers-and-parts database.

Exercise 1-15. The following is a typical SQL INSERT statement against the suppliers-and-parts database:

```
INSERT INTO SP ( SNO, PNO, QTY )
       VALUES ( SNO('S5'), PNO('P6'), QTY(250) ) ;
```

Show an equivalent relational assignment operation. *Note:* I'm assuming here that attributes SNO, PNO, and QTY are of types SNO, PNO, and QTY, respectively, and the expressions SNO('S5'), PNO('P6'), and QTY(250) are literals of those types, with the obvious interpretation. I'll have more to say on such matters in the next two chapters. Also, I realize I haven't yet explained the syntax of relational assignment in detail, so don't worry too much about giving a syntactically correct answer—just do the best you can.

Exercise 1-16. (Harder.) The following is a typical SQL UPDATE statement against the suppliers-and-parts database:

```
UPDATE S
SET    STATUS = 25
WHERE  S.CITY = 'Paris' ;
```

Show an equivalent relational assignment operation. (The purpose of this exercise is to get you thinking about what's involved. I haven't told you enough in this chapter to allow you to answer it fully. See Chapter 5 for further discussion.)

Exercise 1-17. In the body of the chapter, I said that SQL doesn't directly support relational assignment. Does it support it indirectly? If so, how?

Exercise 1-18. From a *practical* point of view, why do you think duplicate tuples, top-to-bottom tuple ordering, and left-to-right attribute ordering are all very bad ideas? (These questions deliberately weren't answered in the body of the chapter, and this exercise might best serve as a basis for group discussion. We'll take a closer look at such matters later in the book.)

CHAPTER TWO

Relations Versus Types

The title of this chapter is *Relations Versus Types*, but most of it has to do with types. The point is, the relational model certainly requires a supporting type system, but it has very little to say about the nature of that system. Why does it require it? Because relations (and relvars) are defined in terms of types; that is, *every attribute of every relation (and of every relvar) is defined to be of some type.* For example, attribute STATUS of the suppliers relvar S might be of type INTEGER. If it is, then every relation *s* that's a possible value for relvar S must also have a STATUS attribute of type INTEGER—which means in turn that every tuple in such a relation *s* must have a STATUS value that's an integer.

I'll be discussing such matters in more detail later in this chapter. For now, let me just say that—with certain important exceptions, which I'll also be discussing later—relational attributes can be defined on *any types whatsoever* (implying among other things that those types can be as complex as we like, as we'll see later as well). In particular, such attributes can be defined on either system-defined (that is, built-in) or user-defined types. For our running example, I'll assume the attributes have types as follows (note

that some of the attributes have the same name as the types they're defined on and others don't):

```
Suppliers              Parts                  Shipments

SNO    : SNO           PNO    : PNO           SNO : SNO
SNAME  : NAME          PNAME  : NAME          PNO : PNO
STATUS : INTEGER       COLOR  : COLOR         QTY : QTY
CITY   : CHAR          WEIGHT : WEIGHT
                       CITY   : CHAR
```

I'll also assume, where it makes any difference, that types INTEGER (integers) and CHAR (character strings of arbitrary length) are system-defined and the others are user-defined.

By the way, SQL in particular does have a built-in type called INTEGER, as I'm sure you know. It also has a built-in type called CHAR, but (a) that type denotes fixed-length strings, not arbitrary-length ones, and (b) the length in question, *n* say, usually has to be specified along with the CHAR specification, like this: CHAR(*n*).* (CHAR without such a length specification is shorthand for CHAR(1)—not a very useful default, it might be thought.) SQL also allows users to define their own types.

In the interests of historical accuracy, I should now say that when Codd first defined the relational model, he said relations were defined over *domains*, not types. In fact, however, domains and types are *exactly the same thing*. Now, you can take this claim as a position statement on my part, if you like, but I want to present arguments in the next two sections to support that position. I'll start with the relational model as Codd originally defined it; thus, I'll use the term domain, not type, until further notice. There are two major topics I want to discuss, one per section:

Domain-constrained comparisons and "domain check override"
: I hope this part of the discussion will persuade you that domains really are types.

Data value atomicity and first normal form
: And I hope this part will persuade you that those types can be arbitrarily complex.

Domain-Constrained Comparisons

Everyone knows (or should know!) that, in the relational model, two values can be tested for equality only if they come from the same domain. In the case of suppliers and parts, for example, the following comparison—which might be part of a WHERE clause—is obviously valid:

```
SP.SNO = S.SNO          /* OK     */
```

By contrast, the following one is not:

```
SP.PNO = S.SNO          /* not OK */
```

* SQL does have a varying-length character-string type, called VARCHAR, but even there a *maximum* length has to be specified.

The reason is that part numbers and supplier numbers are different kinds of things, and they correspond to different domains. So the general idea is that the database management system (DBMS)* should reject any attempt to perform any relational operation—join, union, divide, or whatever—that calls, explicitly or implicitly, for a comparison between values from different domains. For example, here's an SQL query where the user is trying to find suppliers who supply no parts:

```
SELECT S.SNO, S.SNAME, S.STATUS, S.CITY
FROM   S
WHERE  NOT EXISTS
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.PNO = S.SNO )       /* not OK */
```

(There's no terminating semicolon because this is an expression, not a statement. See Exercise 2-24 at the end of the chapter.)

As the comment says, this query is *not* OK. The reason is that, in the last line, the user presumably meant to say WHERE SP.SNO = S.SNO, but by mistake—probably just a slip of the typing fingers—he or she said WHERE SP.*PNO* = S.SNO instead. And, given that we're indeed talking about a simple typo (probably), it would be a friendly act on the part of the DBMS to interrupt at this point, highlight the error, and ask if the user would like to correct it before proceeding.

Now, I don't know any commercial product that actually behaves in the way I've just suggested; in today's products, depending on how you've set up the database, either the query will simply fail or it'll give the wrong answer. Well...not exactly the *wrong* answer, perhaps, but the right answer to the wrong question. (Does that make you feel any better?)

To repeat, therefore, the DBMS should reject a comparison like SP.PNO = S.SNO if it isn't valid. Codd proposed, however, that there should be a way for the user to force the DBMS to go ahead and do the comparison anyway, even if it isn't valid, on the grounds that sometimes the user will know more than the DBMS does. Now, it's a little hard for me to do justice to this proposal, because I don't agree with it—but let me try.

Suppose it's your job to design a database involving, say, customers and suppliers, and you decide therefore to have a domain of customer numbers and another domain of supplier numbers. You build your database that way and load it, and everything works just fine for a year or two. Then, one day, one of your users comes along with a query you never heard before, namely: "Are any of our customers also suppliers to us?" Observe that this is a perfectly reasonable query; observe too that it *might* involve a comparison between a customer number and a supplier number (a cross-domain

* By the way, what's the difference between a DBMS and a database? (This isn't an idle question, because the industry very commonly uses the term *database* when it means either some DBMS product, such as Oracle, or the particular copy of such a product that happens to be installed on a particular computer. The problem is, if you call the DBMS a database, what do you call the database?)

comparison), to see if they're equal. And if it does, well, certainly the system mustn't prevent you from doing it; certainly the system mustn't prevent you from asking a reasonable query.

In light of the foregoing, Codd proposed what he called *domain check override* (DCO) versions of certain of his algebraic operators. A DCO version of join, for example, would perform the join even if the joining attributes were defined on different domains. In SQL terms, we might imagine this proposal being realized by means of a new clause, IGNORE DOMAIN CHECKS, that could be included in an SQL query, as here:

```
SELECT ...
FROM   ...
WHERE  CUSTNO = SNO
IGNORE DOMAIN CHECKS
```

And this new clause would be separately authorizable—most users wouldn't be allowed to use it at all; perhaps only the database administrator (DBA) would be allowed to use it.

Before analyzing the DCO idea in detail, I want to look at a simpler example. Consider the following two queries:

```
SELECT ...                 | SELECT ...
FROM   P, SP               | FROM   P, SP
WHERE  P.WEIGHT = SP.QTY   | WHERE  P.WEIGHT - SP.QTY = 0
```

Assuming, reasonably enough, that weights and quantities are defined on different domains, the query on the left is clearly invalid. But what about the one on the right? According to Codd, that one's valid! In his book *The Relational Model for Database Management Version 2* (Addison-Wesley, 1990), he says that in such a situation "the DBMS [merely] checks that the basic data types are the same"; in the case at hand, the "basic data types" are all numbers, loosely speaking, and so the check succeeds.

To me, this conclusion seems unreasonable. The semantics of an expression should not depend on the arbitrary choice of syntax we use to formulate it! Thus, I believe the expressions P.WEIGHT = SP.QTY and P.WEIGHT - SP.QTY = 0 must either both be valid or both be invalid; the suggestion that they have different semantics is unacceptable. So it seems to me there's something strange about Codd-style domain checks in the first place, before we even get to "domain check override." (In essence, in fact, Codd-style domain checks apply only in the very special case where both comparands are specified as relational attributes and not as anything else, such as an operational expression like P.WEIGHT - SP.QTY.)

Let's look at some even simpler examples. Consider the following comparisons (each of which might appear as part of an SQL WHERE clause, for example):

```
S.SNO = 'X4'        P.PNO = 'X4'        S.SNO = P.PNO
```

I hope you agree that it's at least possible that the first two are valid and the third not. But if so, then I hope you also agree there's something strange going on; apparently, we can

have three values *a*, *b*, and *c* such that *a* = *c* is true and *b* = *c* is true, but as for *a* = *b*... well, we can't even do the comparison, let alone have it come out true! So what *is* going on?

I return now to the fact that attributes S.SNO and P.PNO are defined on domains SNO and PNO, respectively, and to my claim that domains are really types; in fact, I said in the introduction that domains SNO and PNO in particular were *user-defined* types. Now, it's likely that both types are physically represented in terms of the built-in type CHAR—but physical representations are part of the implementation, not the model; they're irrelevant to the user, and in fact they're hidden from the user (or should be), as we saw in Chapter 1. In particular, the operators that apply to supplier numbers and part numbers are the operators defined in connection with those types, not the operators that happen to be defined in connection with type CHAR. For example, we can concatenate two character strings, but we probably can't concatenate two supplier numbers (we could do this latter only if concatenation was an operator defined in connection with type SNO).

Now, when we define a type, one operator we must define is what's called a *selector* operator, which allows us to select, or specify, an arbitrary value of the type in question.* The selector for type SNO, for example (which, as we'll see in Chapter 6, will probably also be called SNO), allows us to select the particular SNO value that has some specified CHAR representation. Here's an example:

```
SNO('S1')
```

This expression is an invocation of the SNO selector, and it returns a certain supplier number: namely, the one that's conceptually represented by the character string value 'S1'. Likewise, the expression:

```
PNO('P1')
```

is an invocation of the PNO selector, and it returns a certain part number: namely, the one that's conceptually represented by the character string value 'P1'. As you can see, therefore, the SNO and PNO selectors effectively work by *converting* a certain CHAR value to a certain SNO value and a certain PNO value, respectively.

Returning now to the comparison S.SNO = 'X4': What happens here is that the system notices that the left and right comparands are of different types (SNO and CHAR, to be specific). Since they're of different types, they certainly can't be equal. However, the system also knows there's an operator—the SNO selector—that effectively performs CHAR-to-SNO conversions. So it can invoke that operator *implicitly* to convert the right comparand to a supplier number, thereby effectively replacing the original comparison with this one:

```
S.SNO = SNO('X4')
```

* This observation is valid regardless of whether we're in an SQL context (as in the present discussion) or otherwise. I omit the details of what's involved in defining selectors in SQL, since they're a little messy—but I'll assume, here and throughout this book, that such operators have indeed been defined.

Now we're comparing two supplier numbers, which is legitimate.

In the same kind of way, the system effectively replaces the comparison P.PNO = 'X4' with this one:

```
P.PNO = PNO('X4')
```

But in the case of the comparison S.SNO = P.PNO, there's no conversion operator known to the system (at least, let's assume not) that will convert a supplier number to a part number or the other way around, and so the comparison fails on a *type error:* the comparands are of different types, and there's no way to make them be of the same type.

> *Terminology:* Implicit conversion is often called *coercion* in the literature. In the first example, therefore, the character string 'X4' is *coerced* to type SNO; in the second it's coerced to type PNO.

To continue with the example, another operator you must define when you define a type like SNO or PNO is what's called, generically, a *THE_* operator, which effectively converts a given SNO or PNO value to the character string (or whatever else it is) that's used to represent it.* Assume for the sake of the example, not unreasonably, that the THE_ operators for types SNO and PNO are both called THE_CHAR. Then, if we really did want to compare S.SNO and P.PNO, the only sense I can make of that requirement is that we want to see if the character-string representations are the same, which we might do like this:

```
THE_CHAR ( S.SNO ) = THE_CHAR ( P.PNO )
```

In other words: convert the supplier number to a string, convert the part number to a string, and compare the two strings.

As I'm sure you can see, the mechanism I've sketched above effectively provides both (a) the domain checking we want, in the first place, and (b) a way of overriding that checking when we want, in the second place. Moreover, it does all this in a clean, fully orthogonal, non ad hoc manner. By contrast, "domain check override" doesn't really do the job; in fact, it doesn't really make sense at all, because it confuses types and representations (as noted previously, types are a model concept, representations are an implementation concept).

Now, you might have realized that what I'm really talking about is here is what's known in language circles as *strong typing*. Different writers have slightly different definitions for this term, but basically it means that (a) everything—in particular, every value and every variable—has a type, and (b) whenever we try to perform some operation, the system

* Again this observation is valid regardless of whether we're in an SQL context or some other context, though SQL doesn't use the terminology of "THE_ operators" as such. (Actually, it doesn't use the terminology of "selectors" either.)

checks that the operands are of the right types for the operation in question.* Observe too that this mechanism works for any operation, not just for the comparison operations we've been discussing; the emphasis on comparison operations in discussions of domain checking is sanctified by historical usage but is in fact misplaced. For example, consider the following expressions:

```
P.WEIGHT * SP.QTY
P.WEIGHT + SP.QTY
```

The first of these is probably valid (it yields another weight: namely, the total weight of the pertinent shipment). The second, by contrast, is probably not valid (what could it mean to add a weight and a quantity?).

Data Value Atomicity

I hope the previous section succeeded in convincing you that domains are indeed types, no more and no less. Now I want to turn to the issue of *data value atomicity* and the related notion of *first normal form* (1NF for short). In Chapter 1, I said that 1NF meant that every tuple in every relation contains just a single value (of the appropriate type, of course) in every attribute position—and it's usual to add that those "single values" are supposed to be *atomic*. But this latter requirement raises the obvious question: what does it mean for data to be atomic?

Well, on page 6 of the book mentioned earlier, Codd defines *atomic data* as data that "cannot be decomposed into smaller pieces by the DBMS (excluding certain special functions)." But even if we ignore that parenthetical exclusion, this definition is a trifle puzzling, and not very precise. For example, what about character strings? Are character strings atomic? Every product I know provides several operators on such strings—LIKE, SUBSTR (substring), "||" (concatenate), and so on—that clearly rely on the fact that character strings in general *can* be decomposed by the DBMS. So are those strings atomic? What do you think?

Here are some other examples of values whose atomicity is at least open to question and yet we would certainly want to be able to include as attribute values in tuples in relations:

- Integers, which might be regarded as being decomposable into their prime factors (I know this isn't the kind of decomposability we usually consider in this context—I'm just trying to show that the notion of decomposability itself is open to a variety of interpretations)

* Or, possibly, are coercible to those right types. For reasons not directly connected with the current topic, however, I would argue that all type conversions should be explicit. Coercions are a well-known source of error.

- Fixed-point numbers, which might be regarded as being decomposable into integer and fractional parts
- Dates and times, which might be regarded as being decomposable into year/month/day and hour/minute second components, respectively

Now I'd like to move on to a potentially more startling example. Refer to Figure 2-1. Relation R1 in that figure is a reduced version of the shipments relation from our running example; it shows that certain suppliers supply certain parts, and it contains one tuple for each legitimate SNO-PNO combination. For the sake of the example, let's agree that supplier numbers and part numbers are indeed "atomic"; then we can presumably agree that R1, at least, is in 1NF.

R1

SNO	PNO
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

R2

SNO	PNO
S2	P1, P2
S3	P2
S4	P2, P4, P5

R3

SNO	PNO_SET
S2	{P1, P2}
S3	{P2}
S4	{P2, P4, P5}

FIGURE 2-1. *Relations R1, R2, and R3*

Now suppose we replace R1 by R2, which shows that certain suppliers supply certain *groups* of parts (attribute PNO in R2 is what some would call *multivalued*, and values of that attribute are groups of part numbers). Then most people would surely say that R2 is not in 1NF; in fact, it looks like an example of "repeating groups," and repeating groups are the one thing that almost everybody agrees 1NF is supposed to prohibit (because such groups are obviously not atomic, right?).

Well, let's assume for the sake of the argument that R2 isn't in 1NF. But suppose we now replace R2 by R3. Then I claim that *R3 is in 1NF!** For consider:

- First, note that—of course deliberately—I've renamed the attribute PNO_SET, and I've shown the groups of part numbers that are PNO_SET values enclosed in set braces "{" and "}", to emphasize the fact that each such group is indeed a single value: a set value, to be sure, but a set is still, at a certain level of abstraction, a single value.
- Second (and regardless of what you might think of my first argument), the fact is that a set like {P2,P4,P5} is no more and no less decomposable by the DBMS than a character string is. Like character strings, sets do have some inner structure; as with character strings, however, it's convenient to ignore that structure for certain purposes. In other words, if character strings are compatible with the requirements of 1NF—that is, if character strings are atomic—then sets must be, too.

* I don't claim it's well designed—indeed, it probably isn't—but that's a separate issue. I'm concerned here with what's legal, not with questions of good design. The design of R3 is legal.

The real point I'm getting at here is that the notion of atomicity *has no absolute meaning;* it just depends on what we want to do with the data. Sometimes we want to deal with an entire set of part numbers as a single thing, and sometimes we want to deal with individual part numbers within that set—but then we're descending to a lower level of detail (a lower level of abstraction). The following analogy might help. In physics (which after all is where the terminology of atomicity comes from) the situation is exactly parallel: sometimes we want to think about individual physical atoms as indivisible things, and sometimes we want to think about the protons, neutrons, and electrons that go to make up those atoms. What's more, protons and neutrons, at least, aren't really indivisible, either—they contain a variety of "subsubatomic" particles called *quarks.* And so on, possibly.

Let's return for a moment to relation R3. In Figure 2-1, I showed PNO_SET values as general sets. But it would be more useful in practice if they were, more specifically, relations (see Figure 2-2, where I've changed the attribute name to PNO_REL). Why would it be more useful? Because relations, not general sets, are what the relational model is all about.* As a consequence, the full power of the relational algebra immediately becomes available for the relations in question—they can be restricted, projected, joined, and so on. By contrast, if we use general sets instead of relations, then we need to introduce new operators (set union, set intersection, and so on) for dealing with those sets. Much better to get as much mileage as we can out of the operators we already have!

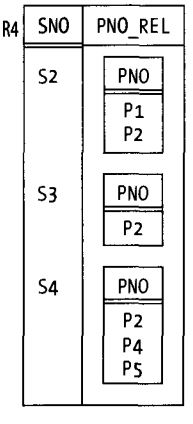


FIGURE 2-2. *Relation R4 (a revised version of R3)*

Attribute PNO_REL in Figure 2-2 is an example of a *relation-valued attribute* (RVA). Of course, the underlying domain is relation-valued too (that is, the values it's made up of are relations). I'll have more to say about RVAs in Chapters 5 and 7; here let me just note that SQL doesn't support them. (More precisely, it doesn't support what would be its analog of

* In case you're wondering, the crucial difference is that general sets can contain anything, whereas relations contain tuples specifically. Note, however, that a relation certainly resembles a general set in that it too can be regarded as a single value.

RVAs, *table-valued columns*—though oddly enough it does support both (a) columns whose values are arrays and (b) columns whose values are "multisets of rows." A *multiset*, also known as a *bag*, is like a set except that it permits duplicates. Columns whose values are multisets of rows thus do look a bit like "table-valued columns" in some respects; however, they aren't table-valued columns, because the values they contain can't be operated upon by means of SQL's regular table operators.)

Now, I chose the foregoing example deliberately, for its shock value. After all, relations with RVAs do look rather like relations with repeating groups, and you've probably always heard that repeating groups are a no-no in the relational world. But I could have used any number of different examples to make my point: I could have shown attributes (and therefore domains) that contained arrays; or bags; or lists; or photographs; or audio or video recordings; or X rays; or fingerprints; or XML documents; or any other kind of value, "atomic" or "nonatomic," that you might care to think of. Attributes, and therefore domains, can contain *anything* (any *values*, that is). All of which goes a long way, incidentally, toward explaining why a true "object/relational" system would be nothing more nor less than a true relational system—which is to say, a system that supports the relational model, with all that such support entails. After all, the whole point of an "object/relational" system is precisely that we can have attribute values in relations that are of arbitrary complexity. Perhaps a better way to say it is this: a proper object/relational system is just a relational system with proper type support—which just means it's a proper *relational* system, no more and no less.

So What's a Type?

From this point forward I'll switch to the term *type* in preference to *domain*. What is a type, exactly? In essence, it's *a named, finite set of values**—all possible values of some specific kind: for example, all possible integers, or all possible character strings, or all possible supplier numbers, or all possible XML documents, or all possible relations with a certain heading (and so on). Moreover:

- Every value is of some type—in fact, of exactly one type, except possibly if type inheritance is supported, a concept that's beyond the scope of this book. Note, therefore, that types are *disjoint* or *nonoverlapping*. (To elaborate briefly: as one reviewer said, surely types WarmBloodedAnimal and FourLeggedAnimal overlap? Indeed they do; but what I'm saying is that if types overlap, then for a variety of reasons we're getting into the realm of type inheritance—in fact, into *multiple* inheritance. Since those reasons, and indeed the whole topic of inheritance, are independent of the context we're in, be it relational or something else, I'm not going to discuss them in this book.)
- Every variable, every attribute, every operator that returns a result, and every parameter of every operator is declared to be of some type. And to say that (for example)

* Finite because we're dealing with computers, which are finite by definition. Also, note that qualifier *named;* types with different names are different types.

variable V is declared to be of type T means, precisely, that every value v that can legally be assigned to V is itself of type T.

- Every expression denotes some value and is therefore of some type: namely, the type of the value in question, which is to say the type of the value returned by the outermost operator in the expression (where by "outermost" I mean the operator that's executed last). For example, the type of the expression $(A+B)*(X-Y)$ is the declared type of the operator "*", whatever that happens to be.

The fact that parameters in particular are declared to be of some type touches on an issue that I've mentioned but haven't properly discussed yet: namely, the fact that every type has an associated set of operators for operating on values and variables of the type in question. For example, integers have the usual arithmetic operators; dates and times have special calendar arithmetic operators; XML documents have what are called "XPath" operators; relations have the operators of the relational algebra; and *every* type has the operators of assignment (":=") and equality comparison ("="). Thus, any system that provides proper type support—and "proper type support" here certainly includes allowing users to define their own types—must provide a way for users to define their own operators, too, because types without operators are useless.

It's important to understand also that values and variables of a given type can be operated upon *solely* by means of the operators defined for that type. For example, in the case of the system-defined type INTEGER:

- The system provides an assignment operator ":=" for assigning integer values to integer variables.
- It also provides comparison operators "=", "<", and so on, for comparing integer values.
- It also provides arithmetic operators "+", "*", and so on, for performing arithmetic on integer values.
- It does *not* provide string operators "||" (concatenate), SUBSTR (substring), and so on, for performing string operations on integer values; in other words, string operations on integer values are not supported.

By contrast, in the case of the user-defined type SNO, we would certainly define assignment and comparison operators (":=", "=", "≠", possibly ">", and so on); however, we probably wouldn't define operators "+", "*", and so on, which would mean that arithmetic on supplier numbers wouldn't be supported (why would we ever want to add or multiply two supplier numbers?).

From everything I've said so far, then, it should be clear that defining a new type involves at least all of the following:

1. Specifying a name for that type (obviously enough).
2. Specifying the values that make up that type. I'll discuss this aspect in more detail in Chapter 6.

3. Specifying the hidden physical representation for values of that type. As noted earlier, this is an implementation issue, not a model issue, and I won't discuss it further in this book.
4. Specifying the operators that apply to values and variables of that type (see below).
5. For those operators that return a result, specifying the type of that result (again, see below).

Observe that points 4 and 5 taken together imply that the system knows precisely which expressions are legal, as well as the type of the result for each such legal expression.

By way of example, suppose we have a user-defined type POINT, representing geometric points in two-dimensional space. Here then is the **Tutorial D** definition* for an operator called REFLECT which, given a point P with cartesian coordinates (*x,y*), returns the "reflected" or "inverse" point with cartesian coordinates (*-x,-y*):

```
1 OPERATOR REFLECT ( P POINT ) RETURNS POINT ;
2    RETURN ( POINT ( - THE_X ( P ) , - THE_Y ( P ) ) ) ;
3 END OPERATOR ;
```

Explanation:

- Line 1 shows that the operator is called REFLECT, takes a single parameter P of type POINT, and returns a result also of type POINT (so the declared type of the operator is POINT).
- Line 2 is the operator implementation code. It consists of a single RETURN statement. The value to be returned is, of course, a point, and it's obtained by invoking the POINT *selector* operator; that invocation has two arguments, corresponding to the X and Y coordinates of the point in question. Each of those arguments involves a *THE_ operator* invocation; those invocations yield the X and Y coordinates of the point argument corresponding to parameter P, and negating those coordinates leads us to the desired result.
- Line 3 marks the end of the definition.

Now, for the most part, I've been talking so far about user-defined types specifically. For a system-defined or built-in type, similar considerations apply, of course, but in this case the definitions are furnished by the system instead of by some user. For example, if INTEGER is a built-in type, then it's the system that defines the name, specifies legal integers, defines the hidden representation, and defines the corresponding operators. Of course, to someone who merely makes use of some user-defined type that's been defined by somebody else, that type looks just like a system-defined type anyway; indeed, in many ways that's the whole object of the exercise.

* I could have used SQL, but operator definitions in SQL involve a number of details that I don't want to get into here.

I don't propose to go into much more detail regarding type and operator definitions, because they aren't specifically relational topics, for the most part.

Scalar Versus Nonscalar Types

It's usual to think of types as being either *scalar* or *nonscalar*. Loosely, a type is scalar if it has no user-visible components and nonscalar otherwise—and the values, variables, attributes, operators, parameters, and expressions of some type *T* are scalar or nonscalar according as type *T* itself is scalar or nonscalar. For example:

- Type INTEGER is a scalar type; hence, values, variables, and so on of type INTEGER are also all scalar, meaning they have no user-visible components.
- Tuple and relation types are nonscalar—the pertinent user-visible components being, of course, the corresponding attributes—and hence tuple and relation values, variables, and so on are also all nonscalar.

That said, I must now stress the point that these notions are quite informal. Indeed, we've already seen that the concept of *atomicity* has no absolute meaning, and "scalarness" is just the concept of atomicity by another name. Thus, the relational model nowhere *formally* relies on the scalar versus nonscalar distinction. In this book, however, I do rely on it informally; to be specific, I use the term *scalar* in connection with types that are neither tuple nor relation types, and the term *nonscalar* in connection with types that *are* either tuple or relation types.

Let's look at an example. Here's a **Tutorial D** definition for the base relvar S (suppliers):

```
1 VAR S BASE
2     RELATION { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
3     KEY { SNO } ;
```

Explanation:

- The keyword VAR in line 1 means this is a variable definition; the keyword BASE means the variable is a base relvar specifically.
- Line 2 specifies the type of this variable. The keyword RELATION shows it's a relation type; the rest of the line specifies the set of attributes that make up the corresponding heading (where, as you'll recall from Chapter 1, an attribute is an attribute-name:type-name pair). The type is, of course, a nonscalar type. No significance attaches to the order in which the attributes are specified.
- Line 3 defines {SNO} to be a candidate key for this relvar.

In fact, the example also illustrates another point—namely, that the type:

```
RELATION { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
```

is an example of a *generated* type. In general, a generated type is one that's obtained by invoking some *type generator* (in the example, the type generator is RELATION). You can

think of a type generator as a special kind of operator; it's special because (a) it returns a type instead of (for example) a scalar value, and (b) it's invoked at compile time instead of run time. For instance, most programming languages support a type generator called ARRAY, which lets users define a variety of specific array types. For the purposes of this book, however, the only type generators we need to consider are TUPLE and, of course, RELATION. Here's an example involving the TUPLE type generator:

```
VAR SINGLE_SUPPLIER
    TUPLE { STATUS INTEGER, SNO SNO, CITY CHAR, SNAME NAME } ;
```

The value of variable SINGLE_SUPPLIER at any given time is a tuple with the same heading as that of relvar S. (I've specified the attributes in a different sequence deliberately, just to show that the sequence doesn't matter.) Thus, we might imagine a code fragment that, first, extracts a one-tuple relation (perhaps the relation containing just the tuple for supplier S1) from the current value of relvar S; then extracts the single tuple from that one-tuple relation; and, finally, assigns that tuple to the variable SINGLE_SUPPLIER. In **Tutorial D**:

```
SINGLE_SUPPLIER := TUPLE FROM ( S WHERE SNO = SNO('S1') ) ;
```

By the way, note carefully that a tuple *t* and a relation *r* that contains just that tuple *t* aren't the same thing. In particular, they're of different types—*t* is of some tuple type and *r* is of some relation type (though of course they do have the same heading).

> **NOTE**
> **I don't want you to misunderstand me here. While a variable like SINGLE_SUPPLIER might well be needed in some application program that accesses the suppliers-and-parts database, I'm *not* saying that such a variable can appear inside the database itself. A relational database contains variables of exactly one kind—namely, *relation* variables (relvars); that is, relvars are the *only* kind of variable allowed in a relational database. I'll revisit this point in Chapter 8, in connection with what's called *The Information Principle*.**

There's just one more thing I want to say about tuple and relation types: even though such types do obviously have user-visible components (namely, their attributes), there's no suggestion that those components have to be physically stored as such. In fact, the physical representation of values of such types should be hidden from the user, just as it is for scalar types.

Summary

It's a very common misconception that the relational model deals only with rather simple types: numbers, strings, perhaps dates and times, and not much else. In this chapter, I've tried to show that this is indeed a misconception. Rather, relations can have attributes of

any type whatsoever—the relational model nowhere prescribes what those types must be, and in fact they can be as complex as we like (except as noted in just a moment). In other words, the question as to what types are supported is orthogonal to the question of support for the relational model itself. Or (less precisely but more catchily): *types are orthogonal to tables.*

I also remind you that the foregoing state of affairs in no way violates the requirements of first normal form. First normal form just means that every tuple in every relation contains a single value, of the appropriate type, in every attribute position. Now that we know those types can be anything, we also know that all relations are in first normal form by definition.

Finally, I mentioned in the introduction to this chapter that there are certain important exceptions to the rule that relational attributes can be of any type whatsoever. In fact, there are two. The first—which I'll simplify just slightly for present purposes—is that if relation *r* is of type *T*, then no attribute of *r* can itself be of type *T* (think about it!). The second is that no relation in the database can have an attribute of any *pointer* type. As you probably know, prerelational databases were full of pointers, and access to such databases involved a lot of pointer-chasing: a fact that made application programming error-prone and direct end-user access impossible. Codd wanted to get away from such problems in his relational model, and of course he succeeded.

Exercises

Exercise 2-1. What's a type? What's the difference between a domain and a type?

Exercise 2-2. What do you understand by the term *selector?*

Exercise 2-3. What's a *THE_ operator?*

Exercise 2-4. Physical representations are always hidden from the user: true or false?

Exercise 2-5. Elaborate on the following: argument versus parameter; database versus DBMS; generated type versus nongenerated type; scalar versus nonscalar; type versus representation; user-defined type versus system-defined type; user-defined operator versus system-defined operator.

Exercise 2-6. What do you understand by the term *coercion?* Why is coercion a bad idea?

Exercise 2-7. Why doesn't "domain check override" make sense?

Exercise 2-8. What's a type generator?

Exercise 2-9. Define *first normal form.*

Exercise 2-10. Let *X* be an expression. What's the type of *X?* What's the significance of the fact that *X is* of some type?

Exercise 2-11. Using the definition of the REFLECT operator in the body of the chapter as a pattern, define a **Tutorial D** operator that, given an integer, returns the cube of that integer.

Exercise 2-12. Use **Tutorial D** to define an 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.

Exercise 2-13. Give an example of a relation type. Distinguish between relation types, relation values, and relation variables.

Exercise 2-14. Use SQL or **Tutorial D** (or both) to define relvars P and SP from the suppliers-and-parts database. If you give both SQL and **Tutorial D** definitions, identify as many differences between them as you can. What's the significance of the fact that relvar P (for example) is of a certain relation type?

Exercise 2-15. Given the types specified in the introductory section for attributes in the suppliers-and-parts database, which of the following scalar expressions are valid? For those that are, state the type of the result; for the others, show an expression that will achieve what appears to be the desired effect.

```
a. CITY = 'London'
b. SNAME || PNAME
c. QTY * 100
d. QTY + 100
e. STATUS + 5
f. 'ABC' < CITY
g. COLOR = CITY
h. CITY || 'burg'
```

Exercise 2-16. It's sometimes suggested that types are really variables, in a sense. For example, 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.

Exercise 2-17. A type is a set of values, and the empty set is a legitimate set; thus, we might define an *empty* type to be a type where the set in question is empty. Can you think of any uses for such a type?

Exercise 2-18. In the body of the chapter, I said the equality comparison operator "=" applies to every type (though I didn't spell out the semantics, which are that if *v1* and *v2* are values of the same type, then *v1* = *v2* evaluates to TRUE if and only if *v1* and *v2* are the very same value). As explained in more detail in Chapter 8, however, SQL doesn't require "=" to apply to every type, nor does it prescribe the semantics in all of the cases where it does apply. What are the implications of this state of affairs?

Exercise 2-19. Following on from the previous exercise, we can say that *v1* = *v2* evaluates to TRUE if and only if executing some operator *Op* on *v1* and executing that same operator *Op* on *v2* always has exactly the same effect, for all possible operators *Op*. But this is another precept that SQL violates. Can you think of any examples of such violation? What are the implications?

Exercise 2-20. Why are pointers excluded from the relational model?

Exercise 2-21. *The Assignment Principle*—which is very simple, but fundamental—states that after assignment of the value *v* to the variable *V*, the comparison *V* = *v* evaluates to TRUE. Yet again, however, this is a precept that SQL violates (fairly ubiquitously, in fact). Can you think of any examples of such violation? What are the implications?

Exercise 2-22. Do you think that types "belong to" databases, in the same sense that relvars do?

Exercise 2-23. In the section "Domain-Constrained Comparisons," I showed an SQL SELECT - FROM - WHERE expression that contained another such expression nested inside it (a "subquery"). Now, each of the SELECT clauses in the overall expression could have been replaced by the arguably simpler form SELECT * ("SELECT star"). But "SELECT *" suffers from certain problems, which is why I usually—not invariably—avoid using it in this book. Identify as many of those problems as you can. Can you think of any other constructs in SQL that suffer from similar problems?

Exercise 2-24. In the first example of an SQL SELECT expression in this chapter, I pointed out that there was no terminating semicolon because the SELECT expression was an expression and not a statement. But what's the difference?

CHAPTER THREE

Tuples and Relations

From the first two chapters you should have gained a pretty good understanding of what tuples and relations are, at least from an informal point of view. Now I want to define those concepts more precisely, and I want to explore some of the consequences of those precise definitions. Perhaps I should warn you that the definitions can look a little daunting, but that's not unusual with formal definitions; the tuple and relation concepts themselves are quite straightforward, once you've struggled through the formalism, and you should be ready to do that by now because the terminology, at least, should be reasonably familiar to you.

What's a Tuple?

Is this a tuple?

SNO SNO	SNAME NAME	STATUS INTEGER	CITY CHAR
S1	Smith	20	London

Well, no, of course it isn't—it's a *picture* of a tuple, not a tuple as such (and note that for once I've included the type names as well as the attribute names in that picture). As we saw in Chapter 1, there's a difference between a thing and a picture of a thing, and that

difference can be very important. For example, tuples have no left-to-right ordering to their attributes, and so the following is an equally good (or bad?) picture of the very same tuple:

STATUS INTEGER	SNAME NAME	CITY CHAR	SNO SNO
20	Smith	London	S1

Thus, while I'll certainly be making use of pictures like these in the sections to follow, please keep in mind that they're *only* pictures, and they can sometimes suggest some things that aren't true.

With that caveat out of the way, I can now say exactly what a tuple is:

> *Definition:* Let *T1, T2, ..., Tn* ($n \geq 0$) be type names, not necessarily all distinct. Associate with each *Ti* a distinct attribute name, *Ai;* each of the *n* attribute-name:type-name combinations that results is an *attribute*. Associate with each attribute a value *vi* of type *Ti;* each of the *n* attribute:value combinations that results is a *component*. The set of all *n* components thus defined, *t* say, is a *tuple value* (or just a *tuple* for short) over the attributes *A1, A2, ..., An*. The value *n* is the *degree* of *t;* a tuple of degree one is *unary*, a tuple of degree two is *binary*, a tuple of degree three is *ternary*, ..., and more generally a tuple of degree *n* is *n*-ary. The set of all *n* attributes is the *heading* of *t*.

For example, with reference to either of the earlier pictures (of our usual tuple for supplier S1), we have:

Degree
Four. The heading is also said to have degree four.

Type names
SNO, NAME, INTEGER, and CHAR.

Corresponding attribute names
SNO, SNAME, STATUS, and CITY.

Corresponding attribute values
SNO('S1'), NAME('Smith'), 20, and 'London'.

NOTE

I showed most of these attribute values in simplified form in the pictures (and I'll continue to make use of similar simplifications in such pictures throughout this book). However, it's strictly incorrect to say that, for example, the SNO value in the tuple we're talking about is just 'S1' or (even sloppier) just S1. A value of type SNO is a value of type SNO, not a value of type CHAR!—and the expression SNO('S1') is a *literal* of type SNO. (More precisely, it's *an invocation of the SNO selector*, as we saw in Chapter 2. A literal is, precisely, a selector invocation in which the arguments are themselves all literals in turn.)

Heading

The easiest thing to do here is show another picture:

SNO	SNO	SNAME	NAME	STATUS	INTEGER	CITY	CHAR

Of course, this picture represents a set, and the order of attributes is arbitrary. Here's another picture of the same heading:

STATUS	INTEGER	SNAME	NAME	CITY	CHAR	SNO	SNO

Exercise: How many different pictures of this same general nature could we draw to represent this heading? (*Answer:* 4 * 3 * 2 * 1 = 24.)

Now, a tuple is a value; like all values, therefore, it has a type (as we already know from Chapter 2), and that type, like all types, has a name. In **Tutorial D**, such names take the form TUPLE{*H*}, where {*H*} is the heading. In our example, the name is:

```
TUPLE { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
```

(but the order in which the attributes is specified is arbitrary).

To repeat, a tuple is a value. Like all values, therefore, it must be returned by some selector invocation (a tuple selector invocation, of course, if the value is a tuple). Here's a tuple selector invocation for our example (**Tutorial D** again):

```
TUPLE { SNO SNO('S1'), SNAME NAME('Smith'),
                       STATUS 20, CITY 'London' }
```

(where the order in which the components is specified is arbitrary). Observe that in **Tutorial D** each component is specified as just an attribute-name:attribute-value pair; the attribute type is omitted because it can always be inferred from the type of the expression denoting the attribute value.

NOTE

A remark on syntax: As you can see, the keyword TUPLE does double duty in **Tutorial D**—it's used in connection with both tuple selector invocations and tuple type names. An analogous remark applies to the keyword RELATION (see the upcoming section "What's a Relation?").

Finally, a word about SQL. Of course, SQL supports rows, not tuples; in particular, it supports *row types*, a *row type constructor*, and *row value constructors*, which are somewhat analogous to tuple types, the TUPLE type generator, and tuple selectors, respectively. But these analogies are loose at best. For example, this SQL expression:

```
ROW ( 1, 2 )
```

which is an example of a row value constructor, (a) has components that are unnamed and (b) doesn't represent the same row as the row value constructor ROW (2,1). (There

are other differences, too, between SQL's row support and our tuple support, but they're beyond the scope of the present discussion.)

NOTE

The keyword ROW in an SQL row value constructor is optional.

Some Important Consequences

Now I want to highlight some important consequences of the foregoing definitions. The first is: *no tuple ever contains any nulls*. The reason is that, by definition, every tuple contains a value (of the appropriate type) for each of its attributes, and we saw in Chapter 1 that nulls aren't values.* Of course, if no tuple ever contains any nulls, then no relation does either, *a fortiori*; so right away we have at least a formal reason for rejecting the concept of nulls—but I'll give some more pragmatic reasons as well, in the section "Why Nulls Are Prohibited," later in this chapter.

The next consequence is: *every subset of a tuple is a tuple and every subset of a heading is a heading*. For example, given our usual tuple for supplier S1, what we might call "the {SNO,CITY} value" within that tuple is itself another tuple:

SNO SNO	CITY CHAR
S1	London

Its heading is as indicated, and its type is:

```
TUPLE { SNO SNO, CITY CHAR }
```

In the same way, the following is a tuple too (its type is TUPLE {SNO SNO}):

SNO SNO
S1

So if we want to access the actual attribute *value*—SNO('S1') in the example—we have to *extract* it somehow from its containing tuple. **Tutorial D** uses syntax of the form SNO FROM *t* for this purpose (where *t* is any expression that denotes a tuple with an SNO attribute). SQL uses dot qualification: *t*.SNO.

NOTE

We saw in Chapter 2 that a tuple *t* and a relation *r* that contains just that tuple *t* aren't the same thing. Analogously, a value *v* and a tuple *t* that contains just that value *v* aren't the same thing, either; in particular, they're of different types.

* Despite the fact that SQL often (though not always) refers to them explicitly as null *values*.

Now, I'm sure you know that the empty set is a subset of every set. It follows that a tuple with an empty heading, and therefore an empty set of components, is a valid tuple!—though it's a little hard to draw a picture of such a tuple on paper, and I'm not even going to try. A tuple with an empty heading has type TUPLE{} (it has no components); indeed, we sometimes refer to it explicitly as a *0-tuple*, to emphasize the fact that it is of degree zero. We also sometimes call it an *empty tuple*. Now, you might think such a tuple is unlikely to be of much use in practice; in fact, it turns out, perhaps rather surprisingly, to be of crucial importance. I'll have more to say about it in the section "TABLE_DUM and TABLE_DEE," later in this chapter.

The last "important consequence" I want to discuss here has to do with the notion of *tuple equality*. (Recall from Chapter 2 that the "=" comparison operator is defined for every type, and tuple types are no exception.) Basically, of course, two tuples are equal if and only if they're the very same tuple (just as, for example, two integers are equal if and only if they're the very same integer). But it's worth spelling out the semantics of tuple equality explicitly, since so much in the relational model depends on it—for example, candidate keys, foreign keys, and most of the operators of the relational algebra are all defined in terms of it. Here then is a precise definition:

> *Definition*: Tuples *t1* and *t2* are *equal* if and only if they have the same attributes *A1*, *A2*, ..., *An*—in other words, they're of the same type—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*.

Also (this might seem obvious, but it needs to be said), two tuples are *duplicates* if and only if they're equal.

By the way, it's an immediate consequence of this definition that all 0-tuples are duplicates of one another. For this reason, we're within our rights if we talk in terms of *the* 0-tuple instead of *a* 0-tuple, and indeed we usually do.

Observe finally that the comparison operators "<" and ">" don't apply to tuples (see Exercise 3-11 at the end of the chapter).

What's a Relation?

I'll use our usual suppliers relation as a basis for examples in this section. Here's a picture:

SNO SNO	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

And here's a definition:

> *Definition:* Let {*H*} be a tuple heading and let *t1*, *t2*, ..., *tm* (*m* ≥ 0) be distinct tuples with heading {*H*}. The combination, *r* say, of {*H*} and the set of tuples {*t1*, *t2*, ..., *tm*} is

a *relation value* (or just a *relation* for short) over the attributes *A1, A2, ..., An*, where *A1, A2, ..., An* are the attributes in {*H*}. The *heading* of *r* is {*H*}; *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 the set of tuples {*t1, t2, ..., tm*}. The value *m* is the *cardinality* of *r*.

I'll leave it as an exercise to interpret the suppliers relation in terms of the foregoing definition. However, I will at least explain why we call such things relations. Basically, each tuple in a relation represents an *n*-ary relationship, in the ordinary natural-language sense, among a set of *n* values (one value for each tuple attribute), and the full set of tuples in a given relation represents the full set of such relationships that happen to exist at some given time—and, mathematically speaking, that's a relation. Thus, the "explanation" often heard, to the effect that the relational model is so called because it lets us "relate one table to another," though accurate in a kind of secondary sense, really misses the basic point. The relational model is so called because it deals with certain abstractions that we can think of as "tables" but are known, formally, as relations in mathematics.

Now, a relation, like a tuple, is itself a value and has a type, and that type has a name. In **Tutorial D**, such names take the form RELATION{*H*}, where {*H*} is the heading. Here's an example:

```
RELATION { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
```

(The order in which the attributes are specified is arbitrary.) Also, every relation value is returned by some relation selector invocation—for example:

```
RELATION {
   TUPLE { SNO SNO('S1'), SNAME NAME('Smith'),
                          STATUS 20, CITY 'London' } ,
   TUPLE { SNO SNO('S2'), SNAME NAME('Jones'),
                          STATUS 10, CITY 'Paris'  } ,
   TUPLE { SNO SNO('S3'), SNAME NAME('Blake'),
                          STATUS 30, CITY 'Paris'  } ,
   TUPLE { SNO SNO('S4'), SNAME NAME('Clark'),
                          STATUS 20, CITY 'London' } ,
   TUPLE { SNO SNO('S5'), SNAME NAME('Adams'),
                          STATUS 30, CITY 'Athens' } }
```

The order in which the tuples are specified is arbitrary.

What support does SQL provide for the foregoing ideas? The answer is: not much. It doesn't really have anything analogous to the concept of a relation type at all; rather, an SQL table is considered to consist of *rows* (a *multiset* or *bag* of rows, to be precise) that are of a certain *row* type. (It does have something it calls *named table types*, but this construct is very different from our relation type, and I'm not going to discuss it here.) It follows that SQL doesn't really have anything analogous to the RELATION type generator, either. It does, however, have something called a *table value constructor* that's akin, somewhat, to a relation selector. Here's an example:

```
VALUES ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 )
```

This expression evaluates to a table with four—not three!—rows and two columns (which have no names).

Further Important Consequences

Most of the properties of relations I talked about in Chapter 1 are direct consequences of the definitions in the previous section, but there are a couple I didn't call out explicitly before, and I want to elaborate on some of the others.

The first point I didn't mention before is that *every subset of a body is a body* (loosely, every subset of a relation is a relation). In particular, therefore, since the empty set is a subset of every set, a relation can have a body that consists of an empty set of tuples (and we call such a relation an *empty relation*). For example, suppose there are no shipments right now. In this case, relvar SP will have as its current value the empty shipments relation, which we might draw like this:*

SNO	PNO	QTY

Note that, given any particular relation type, there's exactly one empty relation of that type—but empty relations of different types aren't the same thing, precisely because they're of different types. For example, the empty suppliers relation isn't equal to the empty parts relation; their bodies are equal but their headings aren't.

The second point I deliberately didn't mention before is this. Recall from Chapter 1 that, while a relation can be pictured as a table, a relation *isn't* a table. (To say it one more time, a picture of a thing is not the same as the thing.) Of course, it can be convenient to *think* of a relation as a table; after all, tables are "user-friendly"; indeed, it's the fact that we can think of relations, informally, as tables—sometimes more explicitly as *flat* or *two-dimensional* tables—that makes relational systems intuitively easy to understand and use, and makes it intuitively easy to reason about the way such systems behave. In other words, it's a very nice property of the relational model that its basic data structure, the relation, has such an intuitively attractive pictorial representation.

Unfortunately, many people seem to have been blinded by that attractive pictorial representation into thinking that *relations as such* are "flat" or "two-dimensional." But they're not. Rather, if relation *r* has *n* attributes, *then each tuple in r represents a point in a certain n-dimensional space* (and the relation overall represents a set of such points). For example, each of the five tuples appearing in our usual sample value for the suppliers relvar S represents a certain point in a certain 4-dimensional space, and the relation overall can thus be said to be 4-dimensional. Thus, relations are *n*-dimensional, not

* I now revert to the convention by which we omit the type names from a heading in informal contexts. Throughout the remainder of this book, in fact, I'll feel free to regard headings as either including type names or excluding them—whichever best suits my purpose at the time.

two-dimensional!* As I've written elsewhere (in quite a few places, in fact): *let's all vow never to say "flat relations" ever again.*

Now I turn to the issues I want to elaborate on. There are three of them: duplicate tuples, nulls, and relations with no attributes. I'll discuss each in its own section.

Why Duplicate Tuples Are Prohibited

There are numerous practical arguments in support of the position that duplicate tuples ("duplicates" for short) should be prohibited. Here I have room for just one—but I think it's a powerful one. However, it does rely on certain notions I haven't discussed yet in this book, so I need to make a couple of preliminary assumptions:

- I assume you know that relational DBMSs include a component called the *optimizer*, whose job is to try to figure out the best way to implement user queries and the like (where "best" basically means best-performing).
- I assume you also know that one of the things optimizers do is what's sometimes called *query rewrite*. Query rewrite is the process of transforming some relational expression *X1*—representing some user query, say—into another such expression *X2*, such that *X1* and *X2* are guaranteed to produce the same result when evaluated, but where *X2* has better performance characteristics than *X1* (at least, we hope it does).

Now I can present my argument. The fundamental point I want to make is that certain expression transformations, and hence certain optimizations, that are valid in a relational context are *not* valid in the presence of duplicates. By way of example, consider the (nonrelational) database shown in Figure 3-1.

P

PNO	PNAME
P1	Screw
P1	Screw
P1	Screw
P2	Screw

SP

SNO	PNO
S1	P1
S1	P1
S1	P2

FIGURE 3-1. *A nonrelational database, with duplicates*

Before going any further, perhaps I should ask the question: what does it mean to have three <P1,Screw> rows in table P and not two, or four, or seventeen?† It must mean something, for if it means nothing, then why are the duplicates there in the first place? As I once heard Ted Codd say, "If something is true, saying it twice doesn't make it any more true."

* Indeed, I think it could be argued that one reason we hear so much about the need for "multidimensional databases" (for decision support applications in particular) is precisely because so many people think relations aren't multidimensional.

† I use table terminology in this section because the things we're talking about are certainly neither relations nor relvars; in particular, they have no keys (observe that there's no double underlining in Figure 3-1).

So I have to assume there's some meaning attached to the duplication, even though that meaning, whatever it is, is hardly very explicit. (I note in passing, therefore, that duplicates contravene one of the original objectives of the relational model: *explicitness*. The meaning of the data should be as obvious and explicit as possible, since we're supposed to be talking about a shared database. The presence of duplicates implies that part of the meaning is hidden.) In other words, given that duplicates do have some meaning, there are presumably going to be business decisions made on the basis of the fact that, for example, there are three <P1,Screw> rows in table P and not two or four or seventeen. If not, then why are the duplicates there in the first place?

Now consider the following query: "Get part numbers for parts that either are screws or are supplied by supplier S1, or both." Here are some candidate SQL formulations for this query, together with the output produced in each case:

```
1  SELECT P.PNO
   FROM   P
   WHERE  P.PNAME = NAME('Screw')
   OR     P.PNO IN
        ( SELECT SP.PNO
          FROM   SP
          WHERE  SP.SNO = SNO('S1') )

   Result: P1 * 3, P2 * 1.

2  SELECT SP.PNO
   FROM   SP
   WHERE  SP.SNO = SNO('S1')
   OR     SP.PNO IN
        ( SELECT P.PNO
          FROM   P
          WHERE  P.PNAME = NAME('Screw') )

   Result: P1 * 2, P2 * 1.

3  SELECT P.PNO
   FROM   P, SP
   WHERE  ( SP.SNO = SNO('S1') AND
            SP.PNO = P.PNO )
   OR     P.PNAME = NAME('Screw')

   Result: P1 * 9, P2 * 3.

4  SELECT SP.PNO
   FROM   P, SP
   WHERE  ( SP.SNO = SNO('S1') AND
            SP.PNO = P.PNO )
   OR     P.PNAME = NAME('Screw')

   Result: P1 * 8, P2 * 4.

5  SELECT P.PNO
   FROM   P
   WHERE  P.PNAME = NAME('Screw')
```

```
UNION  ALL
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = SNO('S1')
```

Result: P1 * 5, P2 * 2.

6
```
SELECT DISTINCT P.PNO
FROM   P
WHERE  P.PNAME = NAME('Screw')
UNION  ALL
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = SNO('S1')
```

Result: P1 * 3, P2 * 2.

7
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = NAME('Screw')
UNION  ALL
SELECT DISTINCT SP.PNO
FROM   SP
WHERE  SP.SNO = SNO('S1')
```

Result: P1 * 4, P2 * 2.

8
```
SELECT DISTINCT P.PNO
FROM   P
WHERE  P.PNAME = NAME('Screw')
OR     P.PNO IN
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.SNO = SNO('S1') )
```

Result: P1 * 1, P2 * 1.

9
```
SELECT DISTINCT SP.PNO
FROM   SP
WHERE  SP.SNO = SNO('S1')
OR     SP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.PNAME = NAME('Screw') )
```

Result: P1 * 1, P2 * 1.

10
```
SELECT P.PNO
FROM   P
GROUP  BY P.PNO, P.PNAME
HAVING P.PNAME = NAME('Screw')
OR     P.PNO IN
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.SNO = SNO('S1') )
```

Result: P1 * 1, P2 * 1.

```
11 SELECT P.PNO
   FROM   P, SP
   GROUP  BY P.PNO, P.PNAME, SP.SNO, SP.PNO
   HAVING ( SP.SNO = SNO('S1') AND
            SP.PNO = P.PNO )
   OR     P.PNAME = NAME('Screw')

   Result: P1 * 2, P2 * 2.

12 SELECT P.PNO
   FROM   P
   WHERE  P.PNAME = NAME('Screw')
   UNION
   SELECT SP.PNO
   FROM   SP
   WHERE  SP.SNO = SNO('S1')

   Result: P1 * 1, P2 * 1.
```

NOTE

Actually, certain of the foregoing formulations—which?—are a little suspect, because they assume that every screw is supplied by at least one supplier. But this fact has no material effect on the argument that follows.

The obvious first point is that the twelve different formulations produce nine different results—different, that is, with respect to their *degree of duplication*. (I make no claim, incidentally, that the twelve different formulations and the nine different results are the only ones possible; indeed, they aren't, in general.) Thus, if the user really cares about duplicates, he or she needs to be *extremely* careful in formulating the query in such a way as to obtain exactly the desired result.

Furthermore, of course, analogous remarks apply to the system itself: because different formulations can produce different results, the optimizer too has to be extremely careful in its task of expression transformation. For example, the optimizer isn't free to transform, say, formulation 1 into formulation 3 or the other way around, even if it would like to. In other words, duplicate rows act as a significant *optimization inhibitor*. Here are some implications of this fact:

- The optimizer code itself is harder to write, harder to maintain, and probably more buggy—all of which combines to make the product simultaneously more expensive and less reliable, as well as late in delivery to the marketplace.
- System performance is likely to be worse than it might be.
- Users are going to have to get involved in performance issues. To be more specific, they're going to have to spend time and effort in figuring out how to formulate a given query in order to get the best performance—a state of affairs the relational model was expressly meant to avoid!

Using Select Distinct

At this point in the original draft, I said that if you find the discipline of always specifying DISTINCT annoying, don't complain to me—complain to the SQL vendors instead. But my reviewers reacted with almost unanimous horror to my suggestion that you should always specify DISTINCT. One wrote: "Those who really know SQL well will be shocked at the thought of coding SELECT DISTINCT by default." Well, I'd like to suggest, politely, that (a) those who are "shocked at the thought" probably know the *implementations* well, not SQL, and (b) their shock is probably due to their recognition that those implementations do such a poor job of optimizing away unnecessary DISTINCTs. If I write SELECT DISTINCT S.SNO FROM S ..., that DISTINCT can safely be ignored. If I write EXISTS (SELECT DISTINCT ...) or IN (SELECT DISTINCT ...), that DISTINCT can safely be ignored. If I write SELECT DISTINCT SP.SNO FROM SP ... GROUP BY SP.SNO, that DISTINCT can safely be ignored. If I write SELECT DISTINCT ... UNION SELECT DISTINCT ..., those DISTINCTs can safely be ignored. And so on. Why should I, as a user, have to devote time and effort to figuring out whether some DISTINCT is going to be a performance hit and whether it's logically safe to omit it, and to remembering all of the details of SQL's inconsistent rules for when duplicates are automatically eliminated and when they're not?

Well, I could go on. However, I decided—against my own better judgment, but in the interests of maintaining good relations (with my reviewers, I mean)—*not* to follow my own advice in the remainder of this book but only to request duplicate elimination explicitly when it seemed logically necessary to do so. It wasn't always easy to decide when that was, either. But at least now I can add my voice to those complaining to the vendors, I suppose.

The fact that duplicates serve as an optimization inhibitor is particularly frustrating in view of the fact that, in most cases, users probably *don't* care how many duplicates appear in the result. In other words:

- Different formulations produce different results.
- But the differences are probably irrelevant from the user's point of view.
- *But* the optimizer is unaware of this latter fact and is therefore prevented, unnecessarily, from performing the transformations it might like to perform.

On the basis of examples like the foregoing, I'm tempted to conclude among other things that you should always ensure that query results contain no duplicates—for example, by always specifying DISTINCT in your SQL queries—and thus simply forget about the whole problem. And if you follow this advice, of course, then there's no good reason for allowing duplicates in the first place....

There's much more that could be said on this issue, but I have room for just four more points. First, the alternative to SELECT DISTINCT in SQL is SELECT ALL—and SELECT ALL is unfortunately the default. The trouble is, however, SELECT DISTINCT might take longer to execute than SELECT ALL, even if the DISTINCT is effectively a "no op." I don't want to discuss this one any further, except to note that the reason SQL systems are typically unable to optimize properly over duplicate elimination is their lack of knowledge of *key inheritance* (that is, their inability to figure out the keys for the result of an arbitrary relational expression).

Second, you might object, not unreasonably, that base tables in practice never do include duplicates, and my example therefore intuitively fails. True enough; but the trouble is, SQL can *generate* duplicates in query results. Indeed, different formulations of the same query can produce results with different degrees of duplication, as we've already seen, even if the input tables themselves have no duplicates at all. For example, consider the following formulations of the

query "Get supplier numbers for suppliers who supply at least one part" on the usual suppliers-and-parts database:

```
SELECT S.SNO
FROM   S
WHERE  S.SNO IN
     ( SELECT SP.SNO
       FROM   SP )
```

```
SELECT S.SNO
FROM   S, SP
WHERE  S.SNO = SP.SNO
```

(Given our usual sample data values, what results do these two expressions produce?) So if you don't want to think of the tables in Figure 3-1 as base tables specifically, that's fine: just take them to be the output from previous queries, and the rest of the analysis goes through unchanged.

Third, there's another at least psychological argument against duplicates that I think is quite persuasive (thanks to Jonathan Gennick for this one): if, in accordance with the *n*-dimensional perspective on relations introduced in the previous section, you think of a table as a plot of points in some *n*-dimensional space, then duplicate rows clearly don't add anything; they simply amount to plotting the same point twice.

My last point is this. Suppose table *T* does permit duplicates. Then we can't tell the difference between "genuine" duplicates in *T* and duplicates that arise from errors in data entry on *T!* For example, what happens if the person responsible for data entry unintentionally—that is, *by mistake*—enters the very same row into *T* twice? (Easily done, by the way.) Is there a way to delete just the "second" row and not the "first"? Note that we presumably do want to delete that "second" row, since it shouldn't have been entered in the first place.

Why Nulls Are Prohibited

The opening paragraph from the previous section applies equally well here (with just one tiny text substitution), so I'll basically just repeat it: There are numerous practical arguments in support of the position that nulls should be prohibited. Here I have room for just one—but I think it's a powerful one. But it does rely on certain notions I haven't discussed yet in this book, so I need to make a couple of preliminary assumptions:

- I assume you know that any comparison in which either or both of the comparands is null evaluates to the UNKNOWN truth value instead of TRUE or FALSE. The justification for this state of affairs is the intended interpretation of null as *value unknown:* if the value of *A* is unknown, then obviously it's unknown whether, for example, $A > B$, regardless of the value of *B* (even—perhaps especially—if the value of *B* is unknown as well). Incidentally, it's this fact that's the source of the term *three-valued logic* (3VL): the notion of nulls inevitably leads us into a logic in which there are three truth values

instead of the usual two. (The relational model, by contrast, is based on conventional *two*-valued logic, 2VL.)

- I assume you also know the 3VL truth tables for the familiar operators NOT, AND, and OR (T = TRUE, F = FALSE, U = UNKNOWN):

p	NOT *p*
T	F
U	U
F	T

p	*q*	*p* AND *q*
T	T	T
T	U	U
T	F	F
U	T	U
U	U	U
U	F	F
F	T	F
F	U	F
F	F	F

p	*q*	*p* OR *q*
T	T	T
T	U	T
T	F	T
U	T	T
U	U	U
U	F	U
F	T	T
F	U	U
F	F	F

Observe in particular that NOT returns UNKNOWN if its input is UNKNOWN; AND returns UNKNOWN if one input is UNKNOWN and the other is either UNKNOWN or TRUE; OR returns UNKNOWN if one input is UNKNOWN and the other is either UNKNOWN or FALSE.

Now I can present my argument. The fundamental point I want to make is that certain boolean expressions—and therefore certain queries—produce results that are correct according to three-valued logic but *not* correct in the real world. By way of example, consider the (nonrelational) database shown in Figure 3-2, in which "the CITY is null" for part P1. Note carefully that the empty space in that figure, in the place where the CITY value for part P1 ought to be, stands for *nothing at all*; conceptually, there's nothing—not even a string of blanks or an empty string—in that position (which means the "tuple" for part P1 isn't really a tuple, a point I'll come back to near the end of this section).

S

SNO	CITY
S1	London

P

PNO	CITY
P1	

FIGURE 3-2. *A nonrelational database, with a null*

Consider now the following (admittedly rather contrived) query on the database of Figure 3-2: "Get SNO-PNO pairs where either the supplier and part cities are different or the part city isn't Paris (or both)." Here's the obvious SQL formulation of this query:*

```
SELECT S.SNO, P.PNO
FROM   S, P
WHERE  S.CITY <> P.CITY
OR     P.CITY <> 'Paris'
```

* As an exercise, show the result of this query given our usual sample data values (see Figure 1-3 in Chapter 1). *Note:* The symbol "<>" is SQL syntax for "≠" (and I'll add for the record that the symbols "<=" and ">=" are SQL syntax for "≤" and "≥", respectively, as you probably know).

Let's focus on the boolean expression in the WHERE clause (parentheses added for clarity):

```
( S.CITY <> P.CITY ) OR ( P.CITY <> 'Paris' )
```

For the only data we have in the database, this expression evaluates to UNKNOWN OR UNKNOWN, which reduces to just UNKNOWN. Now, SQL queries retrieve data for which the expression in the WHERE clause evaluates to TRUE, not to FALSE and not to UNKNOWN; in the example, therefore, nothing is retrieved at all.

But of course part P1 does have *some* corresponding city in the real world; in other words, "the null CITY" for part P1 does stand for some real value, say *xyz*. Obviously, either *xyz* is Paris or it isn't. If it is, then the expression:

```
( S.CITY <> P.CITY ) OR ( P.CITY <> 'Paris' )
```

becomes (for the only data we have):

```
( 'London' <> 'Paris' ) OR ( 'Paris' <> 'Paris' )
```

This expression evaluates to TRUE, because the first term evaluates to TRUE. Alternatively, if *xyz* isn't Paris, the expression becomes (again, for the only data we have):

```
( 'London' <> xyz ) OR ( xyz <> 'Paris' )
```

This expression also evaluates to TRUE, because the second term evaluates to TRUE. Thus, the boolean expression is always TRUE in the real world, and the query should return the pair S1-P1, *regardless of what real value the null stands for*. In other words, the result that's correct according to the logic (that is, 3VL) and the result that's correct in the real world are different!

By way of another example, consider the following query (I didn't lead with this example because it's even more contrived than the previous one, but in some ways it makes the point with still more force):

```
SELECT P.PNO
FROM   P
WHERE  P.CITY = P.CITY
```

The real-world answer here is obviously the set of part numbers currently appearing in P (in other words, the set containing just part number P1, given the sample data shown in Figure 3-2). SQL, however, will return no part numbers at all.

To sum up: if you have any nulls in your database, you're getting wrong answers to some of your queries. What's more, you have no way of knowing, in general, just which queries you're getting wrong answers to; *all* results become suspect. *You can never trust the answers you get from a database with nulls.* In my opinion, this state of affairs is a complete showstopper.

As with the business of duplicates in the previous section, there's much more that could be said on this issue, but I just want to close with a review of the *formal* argument against nulls. Recall that, by definition, a null isn't a value. It follows that:*

- A "type" that contains a null isn't a type (because types contain values).
- A "tuple" that contains a null isn't a tuple (because tuples contain values).
- A "relation" that contains a null isn't a relation (because relations contain tuples, and tuples don't contain nulls).

The net is that if nulls are present, then we're certainly not talking about the relational model (I don't know what we *are* talking about, but it's not the relational model); the entire edifice crumbles, and all bets are off.

TABLE_DUM and TABLE_DEE

The previous two sections were perhaps a little depressing; this section, by contrast, is much more upbeat. Recall from the section "Some Important Consequences" that the empty set is a subset of every set, and hence that there's such a thing as the empty tuple (also called the 0-tuple), and of course it has an empty heading. What I didn't point out before is that—obviously enough—a relation too might have an empty heading (a heading is a set of attributes, and there's no reason why that set shouldn't be empty). Such a relation is of type RELATION{}, and its degree is zero. It's also sometimes said to be 0-ary (or *nullary*), just as, for example, a relation of degree two is said to be binary. (By the way, *nullary* here has nothing to do with nulls; nulls are still bad news, but nullary relations are very good news indeed. However, I prefer to avoid the term *nullary*, precisely because it does sound as if it had something to do with nulls.)

Let *r* be a relation of degree zero, then. How many such relations are there? The answer: just two. First, of course, *r* might be empty (meaning it contains no tuples)—remember there's exactly one empty relation of any given type. Second, if *r* isn't empty, then the tuples it contains must all be 0-tuples. But there's only one 0-tuple!—equivalently, all 0-tuples are duplicates of one another—and so *r* cannot possibly contain more than one of them. So there are indeed just two relations with no attributes: one with just one tuple, and one with no tuples at all. For fairly obvious reasons, I'm not going to try drawing pictures of these relations; in fact, this is one place where the idea of thinking of relations as tables really does break down.

Now, you might well be thinking: "So what? Why on earth would I ever want a relation that has no attributes at all?" Even if they're mathematically respectable (which they are), surely they're of no *practical* significance? In fact, however, it turns out they're of very great practical significance indeed: so much so, that we have pet names for them—

* Remarks analogous to those that follow, though possibly a little less severe, might be made in connection with "relations" with duplicates, or top-to-bottom tuple ordering, or left-to-right attribute ordering.

we call them TABLE_DUM and TABLE_DEE, or DUM and DEE for short (DUM is the empty one, DEE is the one with one tuple).* And the reason they're so significant is their *meanings*, which are FALSE (or NO) for DUM and TRUE (or YES) for DEE. They have the most fundamental meanings of all.

> **NOTE**
> I'll discuss the whole notion of what relations in general mean in the next chapter.

By the way, a good way to remember which is which is this: YES and DEE both have an "E"; NO and DUM don't.

I haven't covered enough in this book yet to show concrete examples of DUM and DEE in action, as it were, but we'll see plenty of examples of their use in the pages ahead. Here I'll just mention one point that should make at least intuitive sense at this early juncture: DEE in particular plays a role in the relational algebra that's analogous to the role played by zero in conventional arithmetic. And we all know how important zero is; in fact, it's hard to imagine an arithmetic without zero (the ancient Romans tried, but it didn't get them very far). Well, it should be equally hard to imagine a relational algebra without TABLE_DEE.

Summary

In this chapter I've presented precise definitions for the fundamental concepts *tuple* and *relation*. As I said in the introductory section, those definitions can be a little daunting at first, but I hope you were able to make sense of them after having read the first two chapters. I also discussed tuple and relation *types*, and showed some examples of tuple and relation *selector invocations*. And I discussed certain important consequences of the definitions:

- No tuple, and therefore no relation, ever contains any nulls.
- Every subset of a tuple is a tuple, every subset of a heading is a heading, and every subset of a body is a body (and those subsets can always be empty).
- Two tuples are equal if and only if they're the very same tuple.

Other consequences already covered in earlier chapters include:

- Every tuple contains exactly one value for each of its attributes, and relations are therefore always in first normal form.
- The attributes of a tuple, and therefore of a relation, are unordered, left to right.

* Let me say a little more about these pet names. First, for the benefit of non–English speakers, I should explain that they're basically just wordplay on Tweedledum and Tweedledee, who were originally characters in a children's nursery rhyme and were subsequently incorporated into Lewis Carroll's *Through the Looking-Glass*. Second, the names are a little unfortunate, given that these two relations are precisely the ones that can't reasonably be depicted as tables! But we've been using those names for so long now that we're probably not going to change them.

- The tuples of a relation are also unordered, top to bottom.
- Relations never contain any duplicate tuples.

I also gave some important pragmatic arguments in favor of prohibiting both duplicates and nulls. And I finished up by briefly introducing the very important degree-zero relations TABLE_DUM and TABLE_DEE; DUM means NO or FALSE, and DEE means YES or TRUE.

Exercises

Exercise 3-1. Define as precisely as you can the terms *attribute, body, cardinality, degree, heading, relation, relvar,* and *tuple*. Also, every relation and every relvar is of some (relation) type; define the term *relation type* as precisely as you can too.

Exercise 3-2. What's a literal?

Exercise 3-3. State as precisely as you can what it means for two tuples to be equal.

Exercise 3-4. We saw in the section "What's a Tuple?" that it's strictly incorrect to say that, for example, S1 or even 'S1' is a supplier number ("a value of type SNO is a value of type SNO, not a value of type CHAR"). As a consequence, Figure 1-3 in Chapter 1 is pretty sloppy, inasmuch as it pretends that it *is* correct to think of, for example, S1 as a supplier number. Show the correct way of referring to the various scalar values in that figure, given the types specified for attributes in the suppliers-and-parts database in the introductory section in Chapter 2.

Exercise 3-5. Write **Tutorial D** tuple selector invocations for a typical tuple from (a) the parts relvar, (b) the shipments relvar. Also show SQL's counterparts, if any, to those selector invocations.

Exercise 3-6. Write a typical **Tutorial D** relation selector invocation. Also show SQL's counterpart, if any, to that selector invocation.

Exercise 3-7. Does the concept of a tuple literal make any sense? What about a relation literal?

Exercise 3-8. (This is essentially a repeat of Exercise 1-8 from Chapter 1, but you should be able to give a more comprehensive answer now.) There are many differences between a relation and a table. List as many as you can.

Exercise 3-9. The attributes of a tuple can be of any type whatsoever (well, almost; can you think of any exceptions?). Give an example of (a) a tuple with a tuple-valued attribute, (b) a tuple with a relation-valued attribute (RVA).

Exercise 3-10. Tuple equality was defined precisely in the body of the chapter—but what about relation equality? (I'll discuss the question of relation comparisons in general in Chapter 5, but you might like to take a shot at defining relation equality for yourself here.)

Exercise 3-11. Why don't the comparison operators "<" and ">" apply to tuples? Note that, by contrast, they do apply to SQL rows. How can this be? (*Hint:* What's the difference between SQL's rules for row equality and the relational rules for tuple equality?)

Exercise 3-12. Give an example of a relation with (a) one RVA, (b) two RVAs. Also give two more relations that represent the same information as those relations but do not involve RVAs. Then give an example of a relation with an RVA such that there's no relation without an RVA that represents precisely the same information. (You might prefer to come back to this exercise after reading more about RVAs in Chapter 5.)

Exercise 3-13. It's sometimes suggested that a relation (or relvar, rather) is really just a traditional computer file, with tuples instead of records and attributes instead of fields. Discuss.

Exercise 3-14. Explain the relations TABLE_DUM and TABLE_DEE in your own words. Does SQL support them?

Exercise 3-15. The following is a legitimate example of an SQL row value constructor:

```
ROW ( 1, NULL )
```

Is the row it represents null or not?

Exercise 3-16. "Duplicates are a good idea in databases because duplicates occur naturally in the real world. For example, all pennies are duplicates of one another." How would you respond to this argument?

Exercise 3-17. Why are the databases of Figures 3-1 and 3-2 nonrelational?

Exercise 3-18. Do you think nulls occur naturally in the real world?

Exercise 3-19. Nulls were originally proposed as a solution to the problem of missing information. Now, it's true that information is often missing in the real world (think of such things as "Speaker to be announced" or "Present address unknown," for example). Therefore, if nulls are prohibited, how *should* we deal with missing information inside our database systems? (As I'm sure you noticed, I didn't answer this question in the body of the chapter. This exercise might best serve as a basis for group discussion. I'll have a little more to say on the subject in Chapter 7.)

Exercise 3-20. TABLE_DEE means TRUE and TABLE_DUM means FALSE. Do these facts mean we could dispense with the usual BOOLEAN data type? Also, DEE and DUM are relations, not relvars. Do you think it would ever make sense to define a *relvar* of degree zero?

Exercise 3-21. We saw in the body of the chapter that this SQL expression:

```
VALUES ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 )
```

denotes a table with four rows. What does the following SQL expression denote?

```
VALUES ( ( 1, 2 ), ( 2, 1 ), ( 1, 1 ), ( 1, 2 ) )
```

CHAPTER FOUR

Relation Variables

We saw in Chapter 1 that a relation variable (relvar for short) is a variable whose permitted values are relations, and that it's specifically relvars, not relations, that are the target for INSERT, DELETE, and UPDATE operations. We also saw that INSERT, DELETE, and UPDATE are all just shorthand for certain relational assignments. I remind you too that (a) if *R* is a relvar and *r* is a relation to be assigned to *R*, then *R* and *r* must be of the same relation type, and (b) the terms *heading, body, attribute, tuple, cardinality,* and *degree,* formally defined in Chapter 3 for relations, can all be interpreted in the obvious way to apply to relvars as well. Now it's time to take a closer look at these matters. As a basis for examples, I'll use the following **Tutorial D** definitions for the base relvars in the suppliers-and-parts database:

```
VAR S BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
    KEY { SNO } ;

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

VAR SP BASE RELATION
  { SNO SNO, PNO PNO, QTY QTY }
    KEY { SNO, PNO }
```

```
FOREIGN KEY { SNO } REFERENCES S
FOREIGN KEY { PNO } REFERENCES P ;
```

Updating Is Set-at-a-Time

The first point I want to stress is that, regardless of what syntax we use to express it, relational assignment is a *set-level* operation. (In fact, *all* operations in the relational model are set-level, as we'll see in Chapter 5.) Thus, INSERT inserts a set of tuples into the target relvar; DELETE deletes a set of tuples from the target relvar; and UPDATE updates a set of tuples in the target relvar. Now, it's true that we often talk in terms of (for example) updating some individual tuple as such, but you need to understand that:

- Such talk really means the set of tuples we're updating just happens to have cardinality one.
- What's more, updating a set of tuples of cardinality one sometimes isn't possible anyway.

For example, suppose relvar S is subject to the integrity constraint (see Chapter 6) that suppliers S1 and S4 are always in the same city. Then any "single-tuple UPDATE" that tries to change the city for just one of those two suppliers will necessarily fail. Instead, we must update them both at the same time, perhaps like this (SQL):

```
UPDATE S
SET    CITY = 'New York'
WHERE  S.SNO = SNO('S1') OR S.SNO = SNO('S4') ;
```

What's being updated here is, obviously enough, a set of two tuples.

> **NOTE**
> Here, for interest, is the same update expressed in **Tutorial D** (it looks very similar, as you can see):
>
> ```
> UPDATE S WHERE SNO = SNO('S1') OR SNO = SNO('S4')
>  ( CITY := 'New York' ) ;
> ```

One consequence of the foregoing is that there's nothing in the relational model that resembles SQL's "positioned updates" (that is, UPDATE or DELETE "WHERE CURRENT OF cursor"), because those operations are tuple-level, not set-level, by definition. They do happen to work in today's products, most of the time, but that's because those products aren't very good at supporting integrity constraints. If the products were to improve in that regard, those "positioned updates" might *not* work any more; that is, applications that succeed today might fail tomorrow—not a very desirable state of affairs, it seems to me.

Now I need to 'fess up to something. The fact is, to talk as I've been doing of "updating a tuple"—or set of tuples, rather—is very imprecise (not to say sloppy) anyway. If *V* is subject to update, then *V* must be a *variable* by definition, not a value, and tuples, like relations, are *values* and can't be updated, again by definition. What we really mean when we talk of (for example) updating tuple *t1* to *t2*, within some relvar *R*, is that we're

replacing tuple *t1* in *R* by another tuple *t2*. And that kind of talk is still sloppy! What we *really* mean is that we're replacing the relation *r1* that's the original value of *R* by another relation *r2*. And what exactly is relation *r2* here? Well, let *s1* and *s2* be relations containing just tuple *t1* and tuple *t2*, respectively; then *r2* is (*r1* MINUS *s1*) UNION *s2*. In other words, "updating tuple *t1* to *t2* in relvar *R*" can be thought of as first deleting *t1* and then inserting *t2*—if (despite everything I've been saying) I might be permitted to talk in terms of deleting and inserting individual tuples in this loose fashion.

In the same kind of way, it doesn't really make sense to talk in terms of "updating attribute *A* within tuple *t*"—or within relation *r*, or even within relvar *R*. Of course, we do it anyway, because it's convenient (it saves a lot of circumlocution); but it's like that business of user-friendly terminology I discussed in Chapter 1—it's OK to do it only if we all understand that such talk is only an approximation to the truth, and indeed it tends to obscure the essence of what's really going on.

More on Candidate Keys

I explained the basic idea of candidate keys in Chapter 1, but now I want to make the concept more precise. Here's a definition:

Definition: Let *K* be a subset of the heading of relvar *R*. Then *K* is a *candidate key* (or just *key* for short) for *R* if and only if it possesses both of the following properties:

Uniqueness
No possible value for *R* contains two distinct tuples with the same value for *K*.

Irreducibility
No proper subset of *K* has the uniqueness property.

> **NOTE**
> In accordance with usual practice, throughout this book I take statements of the form "*B* is a subset of *A*" and "*A* is a superset of *B*" to include the possibility that *A* and *B* might be equal. If I want to exclude that possibility, I'll talk explicitly in terms of *proper* subsets and supersets.

Now, the uniqueness property is self-explanatory, but I need to say a bit more about the irreducibility property. Consider relvar S and the set of attributes {SNO,CITY}—let's call it SK—which is certainly a subset of the heading of S that has the uniqueness property (no relation that's a possible S value ever has two distinct tuples with the same SK value). But it doesn't have the irreducibility property, because we could discard the CITY attribute and what's left, the set {SNO}, would still have the uniqueness property. So we don't regard SK as a key, because it's "too big." By contrast, {SNO} *is* irreducible, and it's a key.

Why do we want keys to be irreducible? One reason is that if we were to specify a "key" that was *not* irreducible, the DBMS wouldn't be able to enforce the uniqueness constraint

properly. For example, suppose we lied and told the DBMS that {SNO,CITY} was a key. Then it couldn't enforce the constraint that supplier numbers are "globally" unique; instead, it could enforce only the weaker constraint that supplier numbers are "locally" unique, in the sense that they're unique within a given city. So this is one reason (not the only one) why we require keys not to include any attributes that aren't needed for unique identification purposes.

Now, all of the relvars we've seen so far have had just one key. Here, by contrast, are some with two or more (they're meant to be self-explanatory). Note the overlapping nature of the keys in the second and third examples.

```
VAR TAX_BRACKET BASE RELATION
  { LOW MONEY, HIGH MONEY, PERCENTAGE INTEGER }
    KEY { LOW }
    KEY { HIGH }
    KEY { PERCENTAGE } ;

VAR ROSTER BASE RELATION
  { DAY DAY_OF_WEEK, TIME TIME_OF_DAY, GATE GATE, PILOT NAME }
    KEY { DAY, TIME, GATE }
    KEY { DAY, TIME, PILOT } ;

VAR MARRIAGE BASE RELATION
  { SPOUSE_A NAME, SPOUSE_B NAME, DATE_OF_MARRIAGE DATE }
    /* assume no polygamy and no couple marrying */
    /* each other more than once ...             */
    KEY { SPOUSE_A, DATE_OF_MARRIAGE }
    KEY { DATE_OF_MARRIAGE, SPOUSE_B }
    KEY { SPOUSE_B, SPOUSE_A } ;
```

I'll close this section with a few miscellaneous points. First, note that the key concept applies to relvars, not relations. Why? Because to say something is a key is to say a certain integrity constraint is in effect—a certain uniqueness constraint, to be specific—and integrity constraints apply to variables, not values. (Integrity constraints constrain updates, and updates apply to variables, not values. See Chapter 6 for further discussion.)

Second, if *R* is a relvar, then *R* certainly does have at least one key. The reason is that every possible value of *R* is a relation and therefore contains no duplicate tuples, by definition; at the very least, therefore, the combination of all of the attributes of *R* certainly has the uniqueness property.* Thus, either that combination also has the irreducibility property or there's some proper subset of that combination that does. Either way, there's *something* that's both unique and irreducible.

Third, note that key values are *tuples*. In the case of relvar S, for example, with its sole key {SNO}, the value of that key for some specific tuple—say, that for supplier S1—is:

```
TUPLE { SNO SNO('S1') }
```

* We can't say the same for SQL tables, of course—SQL tables allow duplicate rows and so might have no key at all.

(Recall from Chapter 3 that every subset of a tuple is a tuple in turn.) Of course, in practice we would usually say, informally, that the key value in this example is just S1—or SNO('S1'), rather—but it really isn't.

Following on from the previous point: it should now be clear that the key concept, like so much else in the relational model, relies on the fundamental concept of *tuple equality*. That is, in order to enforce the uniqueness constraint, we need to be able to tell when two key values are equal, and that's precisely a matter of tuple equality—even when, as in the case of relvar S, the tuples in question are of degree one and "look like" simple scalar values.

My final point has to do with the notion of *functional dependency*. I don't want to get into a lot of detail regarding that concept here—I'll do that in Chapter 7—but you're probably familiar with it anyway; all I want to do here is call your attention to the following. Let *K* be a key for relvar *R*, and let *A* be any attribute of *R*. Then *R* necessarily satisfies the functional dependency:

```
K → A
```

To elaborate briefly: in general, the functional dependency $K \rightarrow A$ means that whenever two tuples of *R* have the same value for *K*, they also have the same value for *A*. But if two tuples have the same value for *K*, where *K* is a key, then by definition they're the very same tuple!—and so they *must* have the same value for *A*. In other words, loosely: we always have "functional dependency arrows" out of keys to everything else in the relvar. Again, I'll revisit these ideas in Chapter 7.

More on Foreign Keys

I explained the basic idea of foreign keys in Chapter 1, but here's a precise definition (note the reliance on tuple equality once again):

> *Definition:* Let *R1* and *R2* be relvars, not necessarily distinct, and let *K* be a key for *R1*. Let *FK* be a subset of the heading of *R2* that, possibly after some attribute renaming, involves exactly the same attributes as *K*. Then *FK* is a *foreign key* if and only if, at all times, every tuple in *R2* has an *FK* value that is equal to the *K* value in some (necessarily unique) tuple in *R1* at the time in question.

As we know, in the suppliers-and-parts database, {SNO} and {PNO} are foreign keys in relvar SP, referencing the sole candidate key—in fact, the primary key—in relvar S and relvar P, respectively. Here's another example:

```
VAR EMP BASE RELATION
  { ENO ENO, ..., MNO ENO, ... }
    KEY { ENO }
    FOREIGN KEY { RENAME ( MNO AS ENO ) } REFERENCES EMP ;
```

Attribute MNO here denotes the employee number of the manager of the employee identified by ENO; thus, the "referencing relvar" (*R2* in the definition) and the

"referenced relvar" (*R1* in the definition) in this example are one and the same. For example, the EMP tuple for employee E3 might include an MNO value of E2, which constitutes a reference to the EMP tuple for employee E2. But foreign key values, like candidate key values, are tuples; conceptually, therefore, we have to rename the MNO attribute in the foreign key specification, in order for the tuple equality comparison to be valid. (What tuple equality comparison? The one that's implicit in the process of checking the foreign key constraint—recall that tuples must certainly be of the same type if they're to be tested for equality, and "same type" means they must have the same attribute names.)

As an aside, I should mention that the relational model as originally formulated required foreign keys to match not just some candidate key but, very specifically, the *primary* key in the referenced relvar. However, I gave my reasons in Chapter 1 for not insisting that some candidate key always be chosen and made primary; accordingly, therefore, I don't want to insist that foreign keys always match primary keys specifically. (I agree with SQL on this one.)

Now, SQL supports not just foreign keys as such but also certain associated *referential actions* such as CASCADE (which can be specified as part of either an ON DELETE clause or an ON UPDATE clause). For example, the CREATE TABLE statement for shipments might include the following:

```
FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ON DELETE CASCADE
```

Given this specification, an attempt to delete a specific supplier will cascade to delete all shipments for that supplier as well. I mention this point for the following reasons:

- First, such specifications might be useful in practice, but they aren't part of the relational model as such.
- But that's not necessarily a problem! The relational model is the foundation of the database field, but it's *only* the foundation. There's no reason why additional features shouldn't be built on top of, or alongside, that foundation—just so long as those additions don't violate the prescriptions of the model, of course (and are in the spirit of the model and can be shown to be useful, I suppose I should add). To elaborate:

  a. Type theory provides the most obvious example. We saw in Chapter 2 that "types are orthogonal to tables," but we also saw that full and proper type support in relational systems is highly desirable, to say the very least.

  b. By way of a second example, the relational model has almost nothing to say about recovery and concurrency controls, but this fact obviously doesn't mean that relational systems shouldn't provide such controls. (Actually it could be argued that the relational model does say something about such matters implicitly, because it relies on the DBMS to implement updates properly and not to lose data—but it doesn't prescribe anything specific.)

One final remark to close this section: I've discussed foreign keys because they're of considerable pragmatic importance, and also because they're part of the model as originally defined. But I think I should stress the point that they aren't truly fundamental—they're really just shorthand for certain integrity constraints that are commonly required in practice, as we'll see in Chapter 6.* (In fact, much the same could be said for candidate keys as well, but in that case the practical benefits of providing a shorthand are overwhelming.)

More on Views

A view, also called a virtual relvar, is a relvar that doesn't have separate existence in its own right but looks to the user as if it did. Here's a definition:

> *Definition:* A *view* or *virtual relvar* V is a relvar whose value at time t is the result of evaluating a certain relational expression at that time t. The expression in question is specified when V is defined and must mention at least one relvar.

The following are a couple of examples, "London suppliers" and "non-London suppliers" (**Tutorial D** on the left, SQL on the right):

```
VAR LS VIRTUAL                    CREATE VIEW LS AS
( S WHERE CITY = 'London' ) ;     ( SELECT S.*
WHERE  S.SNO IN                     FROM   S
                                    WHERE  S.CITY = 'London' ) ;

VAR NLS VIRTUAL                   CREATE VIEW NLS AS
( S WHERE CITY ≠ 'London' ) ;     ( SELECT S.*
                                    FROM   S
                                    WHERE  S.CITY <> 'London' ) ;

VAR NLS VIRTUAL                   CREATE VIEW NLS AS
( S WHERE CITY ≠ 'London' ) ;     ( SELECT S.*
                                    FROM   S
                                    WHERE  S.CITY <> 'London' ) ;
```

The parentheses in all of these examples are unnecessary but not wrong. I include them for clarity.

View Retrievals

To repeat, views are meant to look to the user as if they had their own separate existence; in other words, they're supposed to "look and feel" just like base relvars so far as the user is concerned. In particular, the user should be able to operate on views as if they were base relvars, and the DBMS should be able to map those user operations into suitable

* Precisely for this reason, in fact, explicit support for them is currently omitted from **Tutorial D**. However, I'm sure an "industrial-strength" version of that language would support them, and I'm taking the liberty in this book of pretending such support is already there.

operations on the base relvars in terms of which the views are ultimately defined. I say "ultimately" here because (of course) one thing we can do, if views really do behave just like base relvars, is define further views on top of them, as in this SQL example:

```
CREATE VIEW LS_STATUS
  AS ( SELECT LS.SNO, LS.STATUS
       FROM   LS ) ;
```

Mapping read-only operations is straightforward. For example, suppose we issue this SQL query on view LS:

```
SELECT LS.SNO
FROM   LS
WHERE  LS.STATUS > 10
```

First, the DBMS replaces the reference to the view in the FROM clause by the expression that defines that view, yielding:

```
SELECT LS.SNO
FROM ( SELECT S.*
       FROM   S
       WHERE  S.CITY = 'London' ) AS LS
WHERE  LS.STATUS > 10
```

This expression can now be simplified to:

```
SELECT S.SNO
FROM   S
WHERE  S.CITY = 'London'
AND    S.STATUS > 10
```

The reason the foregoing process works is precisely because of the *closure* property of the relational algebra. Closure implies, among other things, that wherever we're allowed to have the *name* of something—for example, in a query—we can always have a more general expression that evaluates to a thing of the appropriate type.* In the FROM clause, for example, we can have an SQL table name; thus we can also have a more general SQL table expression, and that's why we're allowed to substitute the expression that defines the view LS for the name LS in the example.

By the way, it's worth mentioning that the foregoing process didn't always work in early versions of SQL (to be specific, in versions of the SQL standard prior to 1992), because those early versions failed to support closure properly. As a result, certain innocuous-looking queries against certain innocuous-looking tables (actually views) failed—and failed, moreover, in ways that were hard to explain. Here's a simple example:

```
CREATE VIEW V
  AS ( SELECT S.CITY, SUM ( S.STATUS ) AS ST
       FROM   S
       GROUP  BY S.CITY ) ;
```

* I would much have preferred to use the more formal term *object* in this sentence in place of the very vague term *thing*, but *object* has become a loaded term in computing contexts.

```
SELECT V.CITY
FROM   V
WHERE  V.ST > 25
```

This example failed in the SQL standard prior to 1992. And although the standard has now been fixed, it doesn't follow that all of the products have! And indeed there's at least one major product that still hasn't, at the time of writing (early 2005).

View Updates

I turn now to update operations. Before I get into specifics, I want to look at the London and non-London supplier views again (and now I'll switch to **Tutorial D**):

```
VAR LS  VIRTUAL ( S WHERE CITY = 'London' ) ;

VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' ) ;
```

The important point here is as follows: instead of S being the base relvar and LS and NLS views, *LS and NLS could be base relvars and S could be a view,* like this:

```
VAR LS BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
    KEY { SNO } ;

VAR NLS BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
    KEY { SNO } ;

VAR S VIRTUAL ( LS UNION NLS ) ;
```

> **NOTE**
> In order to achieve complete equivalence, we would also have to specify certain constraints—in particular, constraints to the effect that every CITY value in LS is London and every CITY value in NLS is *not* London—but I omit such details here. See Chapter 6 for further discussion of such matters.

The message of this example is that, to a very large extent, *which relvars are base and which virtual is arbitrary*—from which it follows that there must be no arbitrary and unnecessary distinctions between base and virtual relvars. This state of affairs is referred to as *The Principle of Interchangeability* (of base and virtual relvars). Here are some implications:

- Like base relvars, views are subject to integrity constraints. (We usually think of integrity constraints as applying just to base relvars, but *The Principle of Interchangeability* shows that this position isn't really tenable.)
- In particular, views have candidate keys (and so I should perhaps have included some key specifications in my view examples prior to this point; **Tutorial D** permits such specifications, but SQL doesn't). They might also have foreign keys, and foreign keys might refer to them.

- I didn't mention this point in Chapter 1, but the "entity integrity" rule is supposed to apply specifically to base relvars, not views. It thereby violates *The Principle of Interchangeability*. (Of course, I reject that rule anyway, because it has to do with nulls.)
- Many SQL products, and the SQL standard, provide some kind of "row ID" feature. If that feature applies to base tables and not to views—which in practice is quite likely—it violates *The Principle of Interchangeability*.* Of course, row IDs as such aren't part of the relational model, but that fact in itself doesn't mean they shouldn't be supported. But I observe as an important aside that if those row IDs are regarded as some kind of *object* ID in the object-oriented sense (as they are, most unfortunately, in the SQL standard, as well as in most of the major SQL products), then they're definitely prohibited! Object IDs are effectively *pointers*, and the relational model explicitly prohibits pointers.

And, to revert to the main point of this discussion, we *must* be able to update views—because if we can't, then that fact in itself constitutes the clearest possible violation of *The Principle of Interchangeability*.

As you probably know, SQL's support for this requirement is quite weak, both in the standard and in the major commercial products. It does at least typically include support for updates on views defined as simple restrictions and/or projections of a single underlying base relvar (though even here there are some problems). For example, consider the following view (which is essentially identical to one we saw in Chapter 1):

```
CREATE VIEW SST_PARIS
  AS ( SELECT S.SNO, S.STATUS
       FROM   S
       WHERE  S.CITY = 'Paris' ) ;
```

This view is a projection of a restriction of base table S, and so we might, for example, be able to perform the following DELETE on it:

```
DELETE
FROM   SST_PARIS
WHERE  SST_PARIS.STATUS > 15 ;
```

This DELETE maps to:

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

But few products provide support for updating views that are much more sophisticated than this one.

Unfortunately, I'm now straying into an area in which there's still a certain amount of controversy. My own opinion is that the view updating problem has largely been solved (that is, the theory exists); however, not everybody agrees with me, and in any case a

* It might violate *The Information Principle*, too (see Chapter 8).

detailed discussion of the subject requires rather more background than it's possible to include in a book of this nature. Thus, I'm afraid the best I can do here is refer you to another book—*Databases, Types, and the Relational Model: The Third Manifesto*, Third Edition (Addison-Wesley, 2006), by C. J. Date and Hugh Darwen, where the subject is examined in depth—if you want more specifics.

Miscellaneous Points

There are a few more things I need to say in order to finish up this section. First of all, it's well known, but worth mentioning anyway, that views serve two rather different purposes:

- The user who actually defines view *V* is, obviously, aware of the expression *X* in terms of which *V* is defined. That user can use the name *V* wherever the expression *X* is intended, but such uses are basically just shorthand.
- By contrast, a user who's merely informed that *V* exists and is available for use is supposed (at least ideally) *not* to be aware of the expression *X;* to that user, in fact, *V* is supposed to look and feel just like a base relvar, as we've already seen. And it's this second use of views that's the really important one, and the one I've been concentrating on, tacitly, throughout this section so far.

Second, when I explained what a view was at the beginning of this section, I said the relational expression that defined the view had to mention at least one relvar. Why? Because if it doesn't, the "virtual relvar" won't be a relvar at all!—I mean, it won't be a *variable*, and it certainly won't be updatable. Instead, it'll be a *relation constant*, or what we might call a "relcon." For example (to invent some syntax on the fly):

```
CONST PERIODIC_TABLE ( RELATION {
      TUPLE { ELEMENT 'Hydrogen', SYMBOL 'H' , ATOMICNO  1 },
      TUPLE { ELEMENT 'Helium'  , SYMBOL 'He', ATOMICNO  2 },
      ...
      TUPLE { ELEMENT 'Uranium' , SYMBOL 'U' , ATOMICNO 92 }
                                } ) ;
```

While it certainly might be desirable to provide some kind of "relcon" functionality along the foregoing lines, I don't think we should think of such things as relvars. I don't think it helps the cause of understanding to pretend that constants are variables.

Third, an unfortunate terminological clash is arising as I write, certainly in the academic world, and to some extent in the commercial world also. Recall from Chapter 1 that a view can be thought of as a derived relvar. Well, there's another kind of derived relvar, too, called a *snapshot*. As the name might suggest, a snapshot, although it's derived, is real, not virtual—meaning it's represented not just by its definition in terms of other relvars but also, at least conceptually, by its own separate copy of the data. For example (to invent some syntax again):

```
VAR LSS SNAPSHOT ( S WHERE CITY = 'London' )
    REFRESH EVERY DAY ;
```

Defining a snapshot is just like executing a query, except that:

- The result of the query is saved in the database under the specified name (LSS in the example) as a read-only relvar (read-only, that is, apart from the periodic refresh; see the next bullet item).
- Periodically (EVERY DAY in the example) the snapshot is refreshed—its current value is discarded, the query is executed again, and the result of that new execution becomes the new snapshot value.

In the example, therefore, snapshot LSS represents the data as it was at most 24 hours ago.

Snapshots are important in data warehouses, distributed systems, and many other contexts. In all cases, the rationale is that many applications can tolerate, or might even require, data "as of" some particular point in time. Reporting and accounting applications are a case in point; such applications typically require the data to be frozen at an appropriate moment (for example, at the end of an accounting period), and snapshots allow such freezing to occur without locking out other applications.

So far, so good. The problem is that snapshots have come to be known (at least in some circles) not as snapshots at all but as *materialized views*. But snapshots *aren't* views! Indeed, the whole point about views, at least so far as the relational model is concerned, is that they *aren't* materialized; as we've seen, operations on views are implemented by mapping them into suitable operations on the underlying base relvars. Thus, "materialized view" is simply a contradiction in terms. Worse yet, the unqualified term *view* is now often taken to mean a "materialized view" specifically—again, at least in some circles—and so we're in danger of no longer having a good term to mean a view in the original sense. In this book, of course, I do use the term *view* in its original sense, but be warned that it doesn't always have that meaning elsewhere. *Caveat lector.*

Relvars and Predicates

Now we come to what in many ways is the most important part of this chapter. The essence of it is this: there's another way to think about relvars. I mean, most people think of relvars as if they were just files in the traditional computing sense—rather abstract files, perhaps (maybe *disciplined* would be a better word than abstract), but files nonetheless. But there's a different way to look at them, a way that I believe can lead to a much deeper understanding of what's really going on. It goes like this.

Consider the suppliers relvar S. Like all relvars, that relvar is supposed to represent some portion of the real world. In fact, I can be more precise: the heading of that relvar represents a certain *predicate*, meaning it's a kind of generic statement about some portion of the real world (it's generic because it's *parameterized*, as I'll explain in a moment). The predicate in question looks like this:

> Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY.

This predicate is the *intended interpretation*—in other words, the *meaning*, also called the *intension* (note the spelling)—for relvar S.

You can think of a predicate as a *truth-valued function*. Like all functions, it has a set of parameters, it returns a result when it's invoked, and (because it's truth-valued) that result is either TRUE or FALSE. In the case of the predicate just shown, for example, the parameters are SNO, SNAME, STATUS, and CITY (corresponding to the attributes of the relvar, of course), and they stand for values of the applicable types (SNO, NAME, INTEGER, and CHAR, respectively). When we invoke the function—when we *instantiate the predicate*, as the logicians say—we substitute arguments for the parameters. Suppose we substitute the arguments S1, Smith, 20, and London, respectively. Then we obtain the following *proposition:*

> Supplier S1 is under contract, is named Smith, has status 20, and is located in city London.

In general, a proposition in logic is a statement that's unconditionally either true or false. Here are a couple of examples:

> Edward Abbey wrote *The Monkey Wrench Gang*.

> William Shakespeare wrote *The Monkey Wrench Gang*.

The first of these is true and the second false. Don't fall into the common trap of thinking propositions must always be true ones! However, the ones I'm talking about in connection with relvars *are* supposed to be true ones specifically, as I now explain:

- First, *every* relvar has an associated predicate, called the *relvar predicate* for the relvar in question.
- Let relvar *R* have predicate *P*. Then every tuple *t* appearing in *R* at some given time can be regarded as representing a certain proposition *p*, derived by invoking (or instantiating) *P* at that time with the attribute values from *t* as arguments.
- And (very important!) we assume by convention that each proposition *p* that's obtained in this manner evaluates to TRUE.

Hence, given our usual sample value for relvar S, we assume the following propositions all evaluate to TRUE:

> Supplier S1 is under contract, is named Smith, has status 20, and is located in city London.

> Supplier S2 is under contract, is named Jones, has status 10, and is located in city Paris.

> Supplier S3 is under contract, is named Blake, has status 30, and is located in city Paris.

And so on. What's more, we go further: if a certain tuple plausibly could appear in some relvar at some time but in fact doesn't, then we assume the corresponding proposition is

false at that time (in other words, we adopt what's called the *Closed World Assumption*). For example, the tuple:

```
TUPLE { SNO SNO('S6'), SNAME NAME('Lopez'),
                       STATUS 30, CITY 'Madrid' }
```

is—let's agree—a plausible supplier tuple but doesn't appear in relvar S at this time, and so we assume *it's not the case that* the following proposition is true at this time:

> Supplier S6 is under contract, is named Lopez, has status 30, and is located in city Madrid.

In other words, a given relvar contains, at any given time, *all* and *only* the tuples that represent true propositions (true instantiations of the predicate) at that time.

> *More terminology:* Again, let *P* be the relvar predicate or intension for relvar *R*, and let the value of *R* at some given time be relation *r*. Then *r*—or the body of *r*, to be more precise—constitutes the *extension* of *P* at that time. Note, therefore, that the extension varies over time, but the intension does not.

Relational Expressions

The ideas I've been discussing in this section all extend in a natural way to apply to arbitrary relational expressions. For example, consider this expression, which represents the projection of suppliers on all attributes but CITY:

```
S { SNO, SNAME, STATUS }
```

The result contains all tuples of the form:

```
TUPLE { SNO s, SNAME n, STATUS t }
```

such that a tuple of the form:

```
TUPLE { SNO s, SNAME n, STATUS t, CITY c }
```

currently appears in S for some CITY value *c*. In other words, the result represents the current extension of a predicate that looks like this:

> There exists some city CITY such that supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY.

Observe that this predicate has just three parameters and the corresponding relation (the projection of suppliers on all but CITY) has just three attributes—CITY is *not* a parameter but what the logicians instead call a *bound variable*, owing to the fact that it's "quantified" by the phrase *there exists some city*. (See Appendix A for further explanation of bound variables and quantifiers.) A possibly clearer way of making the same point—that the predicate has just three parameters, not four—is to observe that the predicate in question is logically equivalent to this one:

Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in some city (*in other words, somewhere, but we don't know where*).

It follows from all of the above that views in particular represent certain predicates. For example, if view SST is defined as follows:

```
VAR SST VIRTUAL ( S { SNO, SNAME, STATUS } ) ;
```

then the relvar predicate for SST is precisely:

Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in some city.

There's one last point I want to make here about predicates and propositions. I've said a predicate has a set of parameters. As usual, however, that set might be empty—and if it is, the predicate becomes a proposition! (Certainly it's a statement that's unconditionally either true or false.) In other words, a proposition is a *degenerate* predicate; all propositions are predicates, but most predicates aren't propositions.

More on Relations Versus Types

Chapter 2 was called *Relations Versus Types*. However, I wasn't in a position in that chapter to explain the most important difference between those two concepts—but now I am, and I will.

I've shown that the database at any given time can be thought of as a collection of true propositions: for example, the proposition *Supplier S1 is under contract, is named Smith, has status 20, and is located in city London*. More specifically, I've shown that the argument values appearing in such a proposition (S1, Smith, 20, and London, in the example) are, precisely, the attribute values from the corresponding tuple, and of course each such attribute value is a value of the associated type. It follows that:

Types are sets of things we can talk about;
relations are (true) statements about those things.

In other words, types give us our vocabulary—the things we can talk about—and relations give us the ability to say things about the things we can talk about. (There's a nice analogy here that might help: *Types are to relations as nouns are to sentences.*) For example, if we limit our attention to suppliers only, for simplicity, we see that:

- The things we can talk about are supplier numbers, names, integers, and character strings—and nothing else.
- The things we can say are things of the form "The supplier with the specified supplier number is under contract, has the specified name, has the status denoted by the specified integer, and is located in the city denoted by the specified character string"—and nothing else. (Nothing else, that is, except for things that are *logically implied* by things

we can say. For example, given the things we already know we can say about supplier S1, we can also say things like *Supplier S1 is under contract, is named Smith, has status 20, and is located in some city*—where the city is left unspecified. And if you're thinking that what I've just told you is very reminiscent of, and probably has some deep connection to, relational projection as well, you'd be absolutely right.)

The foregoing state of affairs has at least three important corollaries. To be specific, in order to represent "some portion of the real world" (as I put it in the previous section):

1. Types and relations are both *necessary*—without types, we have nothing to talk about; without relations, we can't say anything.
2. Types and relations taken together are *sufficient*, as well as necessary—we don't need anything else, logically speaking. (Well, we do need relvars, in order to reflect the fact that the real world changes over time, but we don't need them to represent the situation at any given time.)
3. Types and relations *aren't the same thing*. Beware of anyone who tries to pretend they are! In fact, pretending a type is just a special kind of relation is precisely what certain commercial products try to do (though they don't usually talk in such terms)—and I hope it's clear that any product that's founded on such a logical error is doomed to eventual failure. The products I have in mind aren't relational products, of course; typically, they're products that support objects in the object-oriented sense, or products that try somehow to marry such objects and SQL tables. (In fact, at least one of the products I have in mind has indeed already failed.) Further details are beyond the scope of this book, however.

I'd like to wind up this section by offering a slightly more formal perspective on some of what I've been saying. I've said a database can be thought of as a collection of true propositions. In fact, a database, together with the operators that apply to the propositions represented in that database (or to sets of such propositions, rather), is *a logical system*. And when I say "a logical system," I mean a formal system—like euclidean geometry, for example—that has *axioms* ("given truths") and *rules of inference* by which we can prove *theorems* ("derived truths") from those axioms. Indeed, it was Codd's very great insight, when he first invented the relational model back in 1969, that (despite the name) a database isn't really just a collection of *data;* rather, it's a collection of *facts*, or in other words true propositions. Those propositions—the given ones, that is, which is to say the ones represented by the base relvars—are the axioms of the logical system under discussion. The inference rules are essentially the rules by which new propositions can be derived from the given ones; in other words, they're the rules that tell us how to apply the operators of the relational algebra. Thus, when the system evaluates some relational expression (in particular, when it responds to some query), it's really deriving new truths from given ones; in effect, it's proving a theorem!

Once we understand the foregoing, we can see that the whole apparatus of formal logic becomes available for use in attacking "the database problem." In other words, questions such as:

- What should the database look like to the user?
- What should integrity constraints look like?
- What should the query language look like?
- How can we best implement queries?
- More generally, how can we best evaluate database expressions?
- How should results be presented to the user?
- How should we design the database in the first place?

(and others like them) all become, in effect, questions in logic that are susceptible to logical treatment and can be given logical answers.

Of course, it goes without saying that the relational model supports the foregoing perception very directly—which is why, in my opinion, that model is rock solid, and "right," and will endure. It's also why, again in my opinion, other "data models" are simply not in the same ballpark. Indeed, I seriously question whether those other "models" deserve to be called models at all, in the same sense that the relational model can be called a model. Certainly most of them are ad hoc to a degree, instead of being firmly founded, as the relational model is, in set theory and predicate logic. I'll expand on these issues in Chapter 8.

Summary

The most important part of this chapter is the section having to do with predicates (along with the subsequent section, which discusses the related issue of relations versus types). Basically, every relvar *R* has an associated predicate *P* (the *relvar predicate* for *R*); *P* is the *intended interpretation* or *intension* for *R*, and it doesn't change over time. And if the current value of *R* is relation *r*, then *r* represents the *current extension* of *P*; it consists of a set of tuples, each of which represents a true proposition, or in other words a true instantiation of *P*. The extension does change over time. Thus, a database together with its operators can be seen as a *logical system*.

Here are some other important points from the body of the chapter:

- Only relvars are updatable; talk of "updating a tuple" or "updating an attribute" is convenient but sloppy. Updating is always set-at-a-time.
- Every relvar has at least one (candidate) key. Keys have the properties of uniqueness and irreducibility. Key values are tuples.
- Some relvars have foreign keys. SQL supports certain associated "referential actions," such as CASCADE; such actions might well be useful in practice, but they aren't part of the relational model. What's more, foreign keys themselves aren't truly fundamental, either.
- Operations on views are implemented by mapping them into appropriate operations on the underlying base relvars. (The mapping process, which basically works because of

closure, is straightforward for read-only operations but less so for update operations.) *The Principle of Interchangeability* states that there must be no arbitrary and unnecessary distinctions between base and virtual relvars.

- "Types are to relations as nouns are to sentences."
- Types and relations are both necessary and sufficient to represent any data we like. (At the logical level, that is; of course, other constructs are useful at the physical level, as we all know, but that's because the objectives are different at that level. The physical level is deliberately beyond the purview of the relational model.)

Exercises

Exercise 4-1. Explain in your own words why remarks like (for example) "This UPDATE operation updates the status for suppliers in London" aren't very precise. Give a replacement for that remark that's as precise as you can make it.

Exercise 4-2. Why are SQL's "positioned update" operations a bad idea?

Exercise 4-3. Give definitions for SQL analogs of the TAX_BRACKET, ROSTER, and MARRIAGE relvars from the section "More on Candidate Keys."

Exercise 4-4. Why doesn't it make much sense to say a relation has a key?

Exercise 4-5. In the body of the chapter, I gave one reason why key irreducibility is a good idea. Can you think of any others?

Exercise 4-6. "Key values are not scalars but tuples." Explain this remark.

Exercise 4-7. Let relvar *R* be of degree *n*. What's the maximum number of keys *R* can have?

Exercise 4-8. Relvar EMP from the section "More on Foreign Keys" is an example of what's sometimes called a *self-referencing* relvar. Invent some sample data for that relvar. Does this example lead inevitably to a requirement for null support? (*Answer:* No, but it does serve to show how seductive the nulls idea can be.) What can be done in this example if nulls are prohibited?

Exercise 4-9. SQL has nothing analogous to **Tutorial D**'s renaming option in its foreign key specifications. Why not?

Exercise 4-10. Can you think of a situation in which two relvars *R1* and *R2* might each have a foreign key referencing the other?

Exercise 4-11. Investigate any SQL product available to you. What referential actions does that product support? Which ones do you think are useful? Can you think of any others that the product doesn't support but might be useful?

Exercise 4-12. The relational model has nothing to say about triggered procedures (often known simply as *triggers*). Is this omission a problem? If so, why? If not, why not? Do you think triggered procedures are necessary? Or desirable?

Exercise 4-13. Let view LSSP be defined as follows (SQL):

```
CREATE VIEW LSSP
  AS ( SELECT S.SNO, S.SNAME, S.STATUS, SP.PNO, SP.QTY
       FROM   S, SP
       WHERE  S.SNO = SP.SNO
       AND    S.CITY = 'London' ) ;
```

Here's a query on this view:

```
SELECT DISTINCT LSSP.STATUS, LSSP.QTY
FROM   LSSP
WHERE  LSSP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'London' )
```

What might the query that's actually executed on the underlying base relvars look like?

Exercise 4-14. What key(s) does view LSSP from the previous exercise have?

Exercise 4-15. Investigate any SQL product available to you. Are there any apparently legitimate queries on views that fail in that product? If so, state as precisely as you can which ones they are. What justification does the vendor offer for failing to provide full support? (Note that this exercise asks about queries only, not updates.)

Exercise 4-16. Investigate any SQL product available to you. What view updates does that product support? Be as precise as you can in your answer.

Exercise 4-17. Using either the suppliers-and-parts database or any database you happen to be familiar with, give some further examples to illustrate the point that which relvars are base and which virtual is largely arbitrary.

Exercise 4-18. Investigate any SQL product available to you. In what ways—there will be some!—does that product violate *The Principle of Interchangeability?*

Exercise 4-19. Distinguish between views and snapshots. Does SQL support snapshots? Does any product that you're aware of?

Exercise 4-20. What's a "materialized view"? Why is the term deprecated?

Exercise 4-21. Define the terms *proposition* and *predicate*. Give examples.

Exercise 4-22. State the predicates for relvars P and SP from the suppliers-and-parts database.

Exercise 4-23. What do you understand by the terms *intension* and *extension?*

Exercise 4-24. Let *DB* be any database you happen to be familiar with and let *R* be any relvar in *DB*. What's the predicate for *R*? *Note:* The point of this exercise is to get you to apply the fundamental ideas discussed in the body of this chapter to your own data, in an attempt to get you thinking about data in general in such terms. Obviously the exercise has no unique right answer.

Exercise 4-25. Consider views LS and NLS from the section "More on Views." What are the corresponding relvar predicates? Would it make any difference if they weren't views but base relvars instead?

Exercise 4-26. What's the predicate for view LSSP from Exercise 4-13?

Exercise 4-27. Explain the *Closed World Assumption.*

Exercise 4-28. A key is a set of attributes, and the empty set is a legitimate set; thus, we could define an *empty* key to be a key where the set of attributes is empty. Can you think of any uses for such a key?

Exercise 4-29. What's the predicate for a relvar of degree zero? (Does this question even make sense? Justify your answer.)

Exercise 4-30. Every relvar has some relation as its value. Is the converse true? That is, is every relation a value of some relvar?

CHAPTER FIVE

Relational Algebra

I'll begin this chapter by briefly reviewing a few important points from Chapter 1. First, I remind you that each algebraic operator takes at least one relation as input and produces another relation as output. Second, I remind you that the fact that the output is the same kind of thing as the input—they're all relations—is the *closure* property of the algebra, and it's that property that lets us write nested relational expressions. Third, I gave outline descriptions in Chapter 1 of what I called "the original eight operators" (restrict, project, product, intersect, union, difference, join, and divide); now I want to define those operators, as well as a number of others, much more carefully. Before I can do so, however, I need to make a few general points:

- First, the operators are *generic*: they apply, in effect, to *all possible relations*. For example, we don't need one specific join operator to join employees and departments and another, different, join operator to join suppliers and shipments. (Incidentally, do you think an analogous observation applies to object-oriented systems?)
- Second, the operators are *read-only*: they "read" their operands and return a result, but they don't update anything. In other words, they operate on *relations*, not *relvars*.
- Of course, the previous point doesn't mean that relational expressions can't refer to relvars. For example, *R* UNION *S*, where *R* and *S* are relvar names, is certainly a valid

relational expression in **Tutorial D**. In the context of such an expression, however, a relvar name doesn't denote the corresponding relvar as such; rather, it denotes the relation that happens to be the current value of that relvar at that time. In other words, we can certainly use a relvar name to denote a relation operand, and such a *relvar reference* thus constitutes a valid relational expression*—but in principle we could equally well denote the very same operand by means of an appropriate relation literal instead. (An analogy might help clarify this point. Suppose N is a variable of type INTEGER, and at time *t* it has the value 3. Then N+2 is certainly a valid expression, but at time *t* it means exactly the same as 3+2, no more and no less.)

- Last, given that the operators of the algebra are all read-only, it follows that INSERT, DELETE, and UPDATE (and relational assignment), though they *are* relational operators, aren't part of the algebra as such.

I also need to say something about the design of **Tutorial D**, because its support for the algebra differs in certain significant respects from that of SQL (perhaps it would be more accurate to say that its support for the algebra is—deliberately, of course—much more direct than that of SQL). The overriding point is that, in operations such as UNION or JOIN that need some correspondence to be established between operand attributes, **Tutorial D** does so by requiring the attributes in question *to have the same names* (as well as, necessarily, the same types). For example, here's a **Tutorial D** expression for the join of parts and suppliers on cities:

```
P JOIN S
```

By definition, this join is performed on the basis of part and supplier cities, P and S having just the CITY attribute in common. Here by contrast is the same operation in SQL (note the last line in particular):

```
SELECT P.PNO, P.PNAME, P.COLOR, P.WEIGHT, P.CITY,
       S.SNO, S.SNAME, S.STATUS
FROM   P,S
WHERE  P.CITY = S.CITY
```

Note: Actually, this example can be formulated in many different ways in SQL, and here are three more. As you'll observe, the second and third are a little closer to the spirit of **Tutorial D** (note in particular that the result of the join in those two formulations has a column called just CITY and *no* columns called either S.CITY or P.CITY):

```
SELECT P.PNO, P.PNAME, P.COLOR, P.WEIGHT, P.CITY,
       S.SNO, S.SNAME, S.STATUS
FROM   P JOIN S
ON     P.CITY = S.CITY
```

* Not in SQL, though! For example, if *R* and *S* are SQL table names, we typically can't write things like *R* UNION *S*; instead, we have to write something like SELECT *R*.* FROM *R* UNION SELECT *S*.* FROM *S*.

```
SELECT P.PNO, P.PNAME, P.COLOR, P.WEIGHT, CITY,
       S.SNO, S.SNAME, S.STATUS
FROM   P JOIN S
USING  ( CITY )

SELECT P.PNO, P.PNAME, P.COLOR, P.WEIGHT, CITY,
       S.SNO, S.SNAME, S.STATUS
FROM   P NATURAL JOIN S
```

I chose the particular formulation I did partly because it was the only one supported in SQL as originally defined and partly, and more importantly, because it allows me to make a number of additional points concerning differences between SQL and the algebra (at least as realized in **Tutorial D**):

- SQL permits (and sometimes requires) dot-qualified names. **Tutorial D** doesn't.
- **Tutorial D** sometimes needs to rename attributes in order to avoid what would otherwise be naming clashes or mismatches. SQL usually doesn't (though it does support an analog of the RENAME operator that **Tutorial D** uses for the purpose, as we'll see in the next section).
- Partly as a consequence of the previous point, **Tutorial D** has no need for SQL's "correlation names" (in effect, it replaces SQL's correlation-name concept by the idea that attributes occasionally need to be renamed in order to "disambiguate" what would otherwise be a syntactically invalid expression).
- In addition to either explicitly or implicitly supporting certain features of the relational algebra, SQL also explicitly supports certain features of the relational *calculus* (EXISTS is a case in point—see Appendix A). **Tutorial D** doesn't. One consequence of this difference is that SQL tends to be a rather redundant language, in that it often provides numerous different ways of formulating the same query (a fact that can have serious negative implications for the optimizer).*
- SQL requires most queries to conform to its SELECT - FROM - WHERE template. **Tutorial D** has no analogous requirement. I'll have more to say on this particular issue in the section "Extend and Summarize," later in this chapter.

In what follows, I'll show most examples in both **Tutorial D** and SQL.

More on Closure

When I say the output from each algebraic operation is another relation, I hope it's clear that I'm talking *from a conceptual point of view*. I don't mean the system actually has to

* I once wrote a paper on this topic called "Fifty Ways to Quote Your Query" (*www.dbpd.com*, July 1998), in which I showed that even a query as simple as "Get names of suppliers who supply part P2" can be expressed in well over 50 different ways in SQL.

materialize that output in its entirety. For example, consider the following expression (a restriction of a join—**Tutorial D** on the left, SQL on the right):

```
( P JOIN S )
WHERE PNAME > SNAME
```

```
SELECT P.*,
       S.SNO, S.SNAME, S.STATUS
FROM   P, S
WHERE  P.CITY = S.CITY
AND    P.PNAME > S.SNAME
```

Clearly, as soon as a given tuple of the join is formed, the system can test that tuple right away against the restriction condition PNAME > SNAME (P.PNAME > S.SNAME in the SQL version) to see if it belongs in the final output and discard it if not. Thus, the intermediate result that's the output from the join might never exist as a fully materialized relation in its own right at all. (In fact, of course, the system tries very hard *not* to materialize intermediate results in their entirety, for obvious performance reasons.)

The foregoing example raises another important point, however. Consider the boolean expression PNAME > SNAME in the **Tutorial D** version. That expression applies, conceptually, to the result of P JOIN S, and the attribute names PNAME and SNAME in that expression therefore refer to attributes of that result—*not* to the attributes of those names in relvars P and S. But how do we know the result includes any such attributes? What *is* the heading of that result? More generally, how do we know what the heading is for the result of *any* algebraic operation? Clearly, what we need is a set of *inference rules*—to be more specific, *relation type inference rules*—such that if we know the type(s) of the input relation(s) for any given operation, we can infer the type of the output relation from that operation. In the case of join, for example, those rules say the output from P JOIN S is of this type:

```
RELATION { PNO PNO, PNAME NAME, COLOR COLOR, WEIGHT WEIGHT,
           CITY CHAR, SNO SNO, SNAME NAME, STATUS INTEGER }
```

In fact, for join, the heading of the output is the *union* of the headings of the inputs—where by *union* I mean the regular set-theoretic union, not the special relational union I'll be discussing later in this chapter. In other words, the output has all of the attributes of the inputs, except that common attributes—just CITY in the example—appear once, not twice. Of course, those attributes don't have any left-to-right order, so I could equally well say that (for example) the type of the result of P JOIN S is this:

```
RELATION { SNO SNO, PNO PNO, SNAME NAME, PNAME NAME, CITY CHAR,
           STATUS INTEGER, WEIGHT WEIGHT, COLOR COLOR }
```

Note that type inference rules of some kind are definitely needed in order to support the closure property fully. Closure says that every result is a relation, and relations have not just a body but also a heading; thus, every result must have a proper relational heading as well as a proper relational body.

The RENAME operator mentioned in this chapter's introduction is needed in large part to support the relational model's type inference rules. RENAME takes one relation as input and returns another relation as output; the output relation is identical to the input

relation, except that one of its attributes has a different name. For example (**Tutorial D** on the left and SQL on the right):

```
S RENAME ( CITY AS SCITY )  | SELECT S.SNO, S.SNAME, S.STATUS,
                            |        S.CITY AS SCITY
                            | FROM   S
```

Given our usual sample values, the result looks like this:

SNO	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

(I won't usually bother to show results explicitly in this chapter unless I think the particular operator I'm talking about might be unfamiliar to you, as in the case at hand.)

NOTE

Important: The foregoing example does *not* change relvar S in the database! RENAME isn't like SQL's ALTER TABLE; the RENAME invocation is only an expression (just as, for example, P JOIN S or N+2 are only expressions), and like any expression it simply denotes a certain value. What's more, of course, since it *is* an expression, not a statement or "command," it can be nested inside other expressions. We'll see plenty of examples later.

So how does SQL handle this business of specifying result attribute names (or column names, rather)? The answer is: not very well. First of all, we saw in Chapter 3 that SQL doesn't really have a notion of "relation type" anyway (it has row types instead). Second, it can produce results with columns that effectively have no name at all (for example, consider SELECT DISTINCT P.WEIGHT * 2 FROM P). Third, it can also produce results with duplicate column names (for example, consider SELECT DISTINCT P.CITY, S.CITY FROM P, S). Finally, let's take another look at the example from the beginning of this section:

```
( P JOIN S )          | SELECT P.*,
WHERE PNAME > SNAME   |        S.SNO, S.SNAME, S.STATUS
                      | FROM   P, S
                      | WHERE  P.CITY = S.CITY
                      | AND    P.PNAME > S.SNAME
```

As you can see, the counterpart to **Tutorial D**'s PNAME > SNAME in the SQL version is P.PNAME > S.SNAME—which is curious, because that expression is supposed to apply to the result of the FROM clause, and relvars P and S certainly aren't part of that result! Indeed, it's quite difficult to explain how something like P.PNAME in the WHERE and

SELECT clauses (and possibly elsewhere in the overall expression) can make any sense at all in terms of the result of the FROM clause. The SQL standard does explain it, but the machinations it has to go through in order to do so are *much* more complicated than **Tutorial D**'s rules—so much so that I won't even try to explain them here, but will simply rely on the fact that they can be explained if necessary. I justify the omission by appealing to the fact that this book is supposed to be about the relational model, not SQL.

Now I'd like to go on to describe some other algebraic operators. Please note that I'm not trying to be exhaustive in what follows; I won't be covering "all known operators," and I won't even define all of the operators I do cover in full generality. In most cases, in fact, I'll just give a careful but somewhat informal definition and show some simple examples.

The Original Operators

This section gives definitions for the original set of relational operators defined by Codd: restrict, project, join, intersect, union, difference, product, and divide.

Restrict

Let *bx* be a boolean expression involving zero or more attribute names, such that all of the attributes mentioned are attributes of the same relation *r*.* Then the *restriction* of *r* according to *bx*:

```
r WHERE bx
```

is a relation with a heading the same as that of *r* and a body consisting of all tuples of *r* for which *bx* evaluates to TRUE. For example:

```
S WHERE CITY = 'Paris'  |  SELECT S.*
                        |  FROM   S
                        |  WHERE  S.CITY = 'Paris'
```

As an aside, I remark that restrict is sometimes called *select;* I prefer not to use this term, however, because of the potential confusion with SQL's SELECT.

Project

Let relation *r* have attributes *X, Y, ..., Z* (and possibly others). Then the *projection* of *r* on *X, Y, ..., Z*:

```
r { X, Y, ..., Z }
```

* The boolean expression is subject to certain other limitations as well. See Exercise 5-6 at the end of the chapter.

is a relation with (a) a heading derived from the heading of r by removing all attributes not mentioned in the set $\{X,Y,\ldots,Z\}$ and (b) a body consisting of all tuples $\{X\,x,Y\,y,\ldots,Z\,z\}$ such that a tuple appears in r with X value x, Y value y, ..., and Z value z. For example:

```
S { SNAME, CITY, STATUS } | SELECT DISTINCT S.SNAME,
                          |        S.CITY, S.STATUS
                          | FROM   S
```

To repeat, the result is a relation; thus, "duplicates are eliminated," to use the common phrase, and that DISTINCT in the SQL formulation is really needed, therefore.

By the way, **Tutorial D** also allows a projection to be expressed in terms of the attributes to be discarded instead of the ones to be kept. Thus, for example, the **Tutorial D** expressions:

```
S { SNAME, CITY, STATUS }
```

and:

```
S { ALL BUT SNO }
```

are equivalent. This feature can save a lot of writing (think of projecting a relation of degree 100 on 99 of its attributes). An analogous remark applies to SUMMARIZE (BY form only) and GROUP (see the later sections "Extend and Summarize" and "Group and Ungroup," respectively).

In concrete syntax, it turns out to be convenient to assign high precedence to the projection operator. In **Tutorial D**, for example, we take this expression:

```
S JOIN P { PNO, CITY }
```

to mean:

```
S JOIN ( P { PNO, CITY } )
```

and not:

```
( S JOIN P ) { PNO, CITY }
```

As an exercise, show the difference between these two interpretations, given our usual sample data.

Join

Let relations r and s have attributes $X1, X2, \ldots, Xm, Y1, Y2, \ldots, Yn$ and $Y1, Y2, \ldots, Yn, Z1, Z2, \ldots, Zp$, respectively; in other words, the Y's (n of them) are the common attributes, the X's (m of them) are the other attributes of r, and the Z's (p of them) are the other attributes of s. We can assume without loss of generality that none of the X's has the same name as any of the Z's, thanks to the availability of RENAME. Now let the X's taken

together be denoted just X, and similarly for the Y's and the Z's. Then the *natural join** (*join* for short) of r and s:

```
r JOIN s
```

is a relation with (a) a heading that is the (set-theoretic) union of the headings of r and s and (b) a body consisting of the set of all tuples t such that t is the (set-theoretic) union of a tuple appearing in r and a tuple appearing in s. In other words, the heading is $\{X,Y,Z\}$ and the body consists of all tuples $\{X\,x,Y\,y,Z\,z\}$ such that a tuple appears in r with X value x and Y value y and a tuple appears in s with Y value y and Z value z. Here again is the example from the introductory section:

```
P JOIN S | SELECT P.PNO, P.PNAME, P.COLOR, P.WEIGHT, P.CITY,
         |        S.SNO, S.SNAME, S.STATUS
         | FROM   P, S
         | WHERE  P.CITY = S.CITY
```

Now, I'm quite sure you were already familiar with what I've said so far regarding join, but you might not be so familiar with the following points. First of all, I remind you that the SQL standard allows the join of parts and suppliers on cities to be expressed in an alternative style that's a little closer to that of **Tutorial D** (and this time I deliberately replace that long list of column references in the SELECT clause by a simple "*"):

```
SELECT *
FROM   P NATURAL JOIN S
```

However, not all SQL products actually support this syntax.

Second, and more important, *intersection* is a special case of join. To be specific, if $m = p = 0$ (meaning there are no X's and no Z's, and r and s are thus of the same type), r JOIN s degenerates to r INTERSECT s (see the next subsection "Intersect").

Third (also important), *cartesian product* is a special case of join, too: if $n = 0$, meaning there are no Y's and r and s thus have no common attributes at all, r JOIN s degenerates to r TIMES s (see the later subsection "Cartesian Product"). Why? Because (a) the set of attributes common to r and s in this case is the *empty* set of attributes; (b) every possible tuple has the same value for the empty set of attributes (namely, the 0-tuple); thus, (c) every tuple in r joins to every tuple in s, and so we get the cartesian product as stated.

Fourth, it turns out in practice that many queries that require join really require an extended form of that operator called *semijoin*. (You might not have heard of semijoin before, but in fact it's quite important.) Here's the definition: the *semijoin* of r with s is the join of r and s, projected back on to the attributes of r. By way of example, consider the query "Get suppliers who supply at least one part":

* I'll discuss other kinds of joins in the next section.

```
S SEMIJOIN SP | SELECT DISTINCT S.*
              | FROM   S, SP
              | WHERE  S.SNO = SP.SNO
```

As you can see, this query can be thought of as asking for just those suppliers that match at least one shipment. **Tutorial D** therefore provides a more user-friendly spelling for SEMIJOIN that allows the query to be expressed thus:

```
S MATCHING SP
```

Now observe what happens to *r* JOIN *s* if $p = 0$ (meaning the heading of *s* is a subset of that of *r*—equivalently, every attribute of *s* is an attribute of *r*). As I hope you can see, *r* JOIN *s* degenerates to *r* MATCHING *s* in this case. Likewise, if $m = 0$, *r* JOIN *s* degenerates to *s* MATCHING *r* (note that *r* MATCHING *s* and *s* MATCHING *r* are different, in general).

Fifth, join is fundamentally a *dyadic* operator, meaning it takes two operands. However, it's possible, and useful, to support an *n*-adic or prefix version of the operator (and **Tutorial D** does), according to which we can write expressions of the form:

```
JOIN { r, s, ..., w }
```

to join any number of relations *r*, *s*, ..., *w*. For example, the join of parts and suppliers could alternatively be expressed as follows:

```
JOIN { P, S }
```

What's more, we can use this syntax to ask for "joins" of just a single relation, or even of no relations at all! The join of a single relation, JOIN{*r*}, is just *r* itself; this case is perhaps not of much practical importance. Perhaps surprisingly, however, the join of no relations at all, JOIN{}, is very important indeed!—and the result is TABLE_DEE. (Recall that TABLE_DEE is the unique relation with no attributes and one tuple.) Why is the result TABLE_DEE? Well, consider the following:

- In ordinary arithmetic, 0 is the *identity* with respect to "+"; that is, for all numbers *x*, the expressions $x + 0$ and $0 + x$ are both identically equal to *x*. As a consequence, *the sum of no numbers is 0.* (To see that this claim is reasonable, consider a piece of code that computes the sum of *n* numbers by initializing the sum to 0 and then iterating over those *n* numbers. What happens if $n = 0$?)
- In like manner, 1 is the identity with respect to "*"; that is, for all numbers *x*, the expressions $x * 1$ and $1 * x$ are both identically equal to *x*. As a consequence, the product of no numbers is 1.
- In the relational algebra, *TABLE_DEE is the identity with respect to JOIN;* that is, the join of any relation *r* with TABLE_DEE is simply *r*. As a consequence, the join of no relations is TABLE_DEE.

If you're having difficulty with this idea, don't worry about it too much for now. But if you come back to reread this section later, I suggest you try to convince yourself then that *r* JOIN TABLE_DEE and TABLE_DEE JOIN *r* are indeed both identically equal to *r*. It might help if I point out that the joins in question are actually cartesian products (right?).

Intersect

Let relations *r* and *s* be of the same type. Then the *intersection* of those relations, *r* INTERSECT *s*, is a relation of the same type, with a body consisting of all tuples *t* such that *t* appears in both *r* and *s*. For example:

```
S { CITY }        | SELECT DISTINCT S.CITY
  INTERSECT       | FROM   S
    P { CITY }    | INTERSECT
                  | SELECT DISTINCT P.CITY
                  | FROM   P
```

(Actually we don't need those DISTINCTs in the SQL version, but I prefer to include them for explicitness. See the upcoming discussion of union.)

As we've already seen, intersect is really just a special case of join. **Tutorial D** and SQL both support it, however, if only for psychological reasons. **Tutorial D** also supports an *n*-adic or prefix form, but I'll skip the details here.

Union

Again, let relations *r* and *s* be of the same type. Then the *union* of those relations, *r* UNION *s*, is a relation of the same type, with a body consisting of all tuples *t* such that *t* appears in *r* or *s* or both. For example:

```
S { CITY } UNION P { CITY } | SELECT DISTINCT S.CITY
                            | FROM   S
                            | UNION  DISTINCT
                            | SELECT DISTINCT P.CITY
                            | FROM   P
```

As with projection, it's worth noting explicitly in connection with union that "duplicates are eliminated." In the SQL version, however, that DISTINCT after the keyword UNION is *not* strictly needed; although UNION provides the same options as SELECT does (DISTINCT versus ALL), the default for UNION is DISTINCT, not ALL. (It's the other way around for SELECT, as you'll recall from Chapter 3.) As a consequence, the DISTINCTs following the two SELECTs aren't needed, either, though of course they're not wrong. Analogous remarks apply to intersection (also to difference, which we'll get to in a moment).

Tutorial D also supports *disjoint union* (D_UNION), a version of union that requires its operands to be disjoint, which means they have no tuples in common. For example:

```
S { CITY } D_UNION P { CITY }
```

Given our usual sample data, this expression will produce a runtime error because supplier cities and part cities aren't disjoint. An SQL counterpart might look like this:

```
SELECT *
FROM ( SELECT S.CITY FROM S
```

```
        UNION
        SELECT P.CITY FROM P ) AS POINTLESS
WHERE   NOT EXISTS
      ( SELECT S.CITY FROM S
        INTERSECT
        SELECT P.CITY FROM P )
```

There's a difference, though: if supplier cities and part cities aren't disjoint, the SQL expression won't fail at run time, but will simply return an empty result. I should also mention that not all SQL products allow subqueries (as they're called) to be nested inside the FROM clause in the manner shown, though the SQL standard does.

> **NOTE**
> The correlation name *POINTLESS* in the foregoing expression is indeed pointless—notice that it's never referenced!—but it's required by the standard's syntax rules.

Tutorial D also supports *n*-adic or prefix forms of both UNION and D_UNION. I'll skip the details here.

Difference

Again let relations *r* and *s* be of the same type. Then the *difference* between those relations, *r* MINUS *s* (in that order), is a relation of the same type, with a body consisting of all tuples *t* such that *t* appears in *r* and not *s*. For example:

```
S { CITY } MINUS P { CITY } | SELECT S.CITY
                            | FROM   S
                            | EXCEPT
                            | SELECT P.CITY
                            | FROM   P
```

Recall now that there's an operator related to join called semijoin. Well, it turns out there's a *semidifference* operator, too, but in this case the operators aren't simply "related"—rather, regular difference is *a special case* of semidifference ("all differences are semidifferences," you might say). And if semijoin is in some ways more important than join, a similar remark applies here also, but with even more force—in practice, most queries that require difference really require semidifference. Here's the definition: the *semidifference* between *r* and *s* is the difference between *r* and *r* MATCHING *s;* that is, *r* SEMIMINUS *s* is equivalent to *r* MINUS (*r* MATCHING *s*). By way of example, consider the query "Get suppliers who supply no parts at all":

```
S SEMIMINUS SP | SELECT S.*
               | FROM   S
               | EXCEPT
               | SELECT S.*
               | FROM   S, SP
               | WHERE  S.SNO = SP.SNO
```

Tutorial D also provides a more user-friendly spelling that allows this query to be expressed thus:

```
S NOT MATCHING SP
```

To see that regular difference is a special case of semidifference, consider what happens if *r* and *s* are of the same type (I'll leave the details as an exercise).

Cartesian Product

I include this operator mainly for completeness; as we've seen, it's really just a special case of join, and as a matter of fact **Tutorial D** doesn't support it directly—but let's suppose it did, just for the sake of discussion. Clearly, then, the operands must have no common attribute names (what would happen otherwise?), and the result heading is the set-theoretic union of the input headings. Here's an example:

```
( S RENAME ( CITY AS SCITY ) )
TIMES
( P RENAME ( CITY AS PCITY ) )
```

SQL analog:

```
SELECT S.SNO, S.SNAME, S.STATUS, S.CITY AS SCITY
       P.PNO, P.PNAME, P.COLOR, P.WEIGHT, P.CITY AS PCITY
FROM   S, P
```

Divide

As promised in the introduction to this chapter, I'll give a precise definition of this operator here, but for several reasons I don't want to discuss it in much detail. The most important of those reasons is that any query that can be formulated in terms of divide can alternatively, and I think more simply, be formulated in terms of *relational comparisons* (I'll discuss relational comparisons later in this chapter). You might therefore want to skip the following discussion, at least on a first reading.

Another reason I don't want to get into too much detail is that there are several different "divide" operators anyway! That is, there are, unfortunately, several different operators all called "divide," and I certainly don't want to explain all of them. Instead, I'll limit my attention here to the original and simplest one, which can be defined as follows. Let relations *r* and *s* be such that the heading of *s* is a subset of the heading of *r*. Then the *division* of *r* by *s*, *r* DIVIDEBY *s*,* is shorthand for the following:

```
r { X } MINUS ( ( r { X } TIMES s ) MINUS r ) { X }
```

X here is the (set-theoretic) difference between the heading of *r* and that of *s*. Thus, for example, the expression:

```
SP { SNO, PNO } DIVIDEBY P { PNO }
```

* **Tutorial D** doesn't directly support "the original and simplest" divide, and this is thus not valid **Tutorial D** syntax.

(given our usual sample data values) yields:

SNO
S1

(loosely, supplier numbers for suppliers who supply all parts; I'll explain the reason for that qualifier "loosely" in the section "Relational Comparisons," later in this chapter). Here's an SQL analog:

```
SELECT DISTINCT SPX.SNO
FROM   SP AS SPX
WHERE  NOT EXISTS
     ( SELECT P.PNO
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT SPY.SNO
              FROM   SP AS SPY
              WHERE  SPY.SNO = SPX.SNO
              AND    SPY.PNO = P.PNO ) )
```

By the way, have you ever wondered why divide is called divide? The reason is that if *r* and *s* are relations with no common attributes, and we form the cartesian product *r* TIMES *s* and then divide the result by *s*, we get back to *r*;* in other words, cartesian product and divide are inverses of each other, in a sense.

Which Operators Are Primitive?

As I've more or less said already, not all of the operators we've been discussing are primitive—several of them can be defined in terms of others. One possible primitive set (not the only one) is that consisting of restrict, project, join, union, and semidifference. *Note:* You might be surprised not to see rename in this list. In fact, however, rename isn't primitive, though I haven't covered enough groundwork yet to show why not. What this example shows, however, is that there's a difference between being primitive and being useful! I certainly wouldn't want to be without our useful rename operator, even if it isn't primitive.

Evaluating SQL Expressions

In addition to natural join, Codd originally defined an operator he called *theta*-join, where *theta* denoted any of the usual scalar comparison operators ("=", "≠", "<", and so on). The operator isn't primitive; in fact, it's equivalent to a restriction of a cartesian product. Here by way of example is a **Tutorial D** formulation of the "not equals"–join of suppliers and parts on cities (so theta here is "≠"):

```
( ( S RENAME ( CITY AS SCITY )
  TIMES
```

* So long as *s* isn't empty. What happens if it is?

```
  ( P RENAME ( CITY AS PCITY ) )
WHERE SCITY ≠ PCITY
```

Note that the result has two "city" attributes, SCITY and PCITY. An SQL analog:

```
SELECT S.SNO, S.SNAME, S.STATUS, S.CITY AS SCITY,
       P.PNO, P.PNAME, P.COLOR, P.WEIGHT, P.CITY AS PCITY
FROM   S, P
WHERE  S.CITY <> P.CITY
```

We can think of this SQL expression as being implemented in three steps, as follows:

1. The FROM clause is executed and yields the cartesian product of tables S and P. (If we were doing this relationally, we would have to rename the city attributes *before* that cartesian product is computed. SQL gets away with renaming them afterward because its tables have a left-to-right ordering to their columns, meaning it can distinguish the two city columns by their ordinal position. For simplicity, let's ignore this detail.)
2. Next, the WHERE clause is executed and yields a restriction of that cartesian product by eliminating rows in which the two city values are equal.*
3. Finally, the SELECT clause is executed and yields a projection of that restriction on the columns specified in the SELECT clause. (Of course, the SELECT clause is doing some renaming as well in this particular example—and later we'll see that it also provides functionality similar to that of the relational *extend* operator. What's more, it generally doesn't eliminate duplicates, unless DISTINCT is specified. But I want to ignore all of these details too, for simplicity.)

At least to a first approximation, then, the FROM clause corresponds to a cartesian product, the WHERE clause to a restriction, and the SELECT clause to a projection, and the overall SELECT - FROM - WHERE expression represents a projection of a restriction of a cartesian product. It follows that I've just given a loose, but reasonably formal, definition of the *semantics* of SQL's SELECT - FROM - WHERE expressions; equivalently, I've given a *conceptual algorithm* for evaluating such expressions. Of course, there's no implication that the implementation has to use exactly that algorithm in order to implement such expressions; *au contraire*, it can use any algorithm it likes, just so long as whatever algorithm it does use is guaranteed to give the same result as the conceptual one. And there are often good reasons—usually performance reasons—for using a different algorithm, thereby (for example) executing the clauses in a different order or otherwise rewriting the original query. However, the implementation is free to do such things *only if it can be proved that the algorithm it does use is logically equivalent to the conceptual one*. Indeed, one way to characterize the job of the system optimizer is to find an algorithm that's guaranteed to be equivalent to the conceptual one but performs better.

* If *theta* had been "=" instead of "≠", Step 2 would have been: Restrict the cartesian product by *retaining* just the rows in which the two city values are equal—in which case we would now have formed what's called the *equijoin* of suppliers and parts on cities. In other words, an equijoin is a theta-join for which theta is "=". *Exercise:* What's the difference between an equijoin and a natural join?

Extend and Summarize

As you might have noticed, the algebra as I've described it so far has no conventional computational capabilities. Now, SQL does; for example, we can write queries along the lines of SELECT A+B AS ... or SELECT SUM(C) AS ... (and so on). However, as soon as we write that "+" or that SUM, we've gone beyond the bounds of the relational algebra as originally defined. So we need to add some new operators to the algebra in order to provide this kind of functionality. That's what extend and summarize are for. Loosely, extend supports computation "across tuples," and summarize supports computation "down attributes." Let's take a closer look.

Extend

I'll start with an example, since this operator might be new to you. Suppose part weights are given in pounds, and we want to see those weights in grams. There are 454 grams to a pound, and so we can write:

```
EXTEND P ADD                  | SELECT P.*,
( WEIGHT * 454 AS GMWT )      |      ( P.WEIGHT * 454 ) AS GMWT
                              | FROM   P
```

Given our usual sample values, the result looks like this:

PNO	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

NOTE

Important: Relvar P is *not* changed in the database! EXTEND is *not* an SQL-style ALTER TABLE; the EXTEND expression is just an expression, and like any expression it simply denotes a value.

To continue with the example, consider now the query "Get part number and gram weight for parts with gram weight greater than 7000 grams." Here's a **Tutorial D** formulation:

```
( ( EXTEND P
      ADD ( WEIGHT * 454 AS GMWT ) )
          WHERE GMWT > 7000.0 ) { PNO, GMWT }
```

SQL analog:

```
SELECT P.*, ( P.WEIGHT * 454 ) AS GMWT
FROM   P
WHERE  ( P.WEIGHT * 454 ) > 7000.0
```

As you can see, the expression P.WEIGHT * 454 appears twice in the SQL formulation, and we have to hope the implementation will be smart enough to recognize that it need evaluate that expression just once per tuple instead of twice. In **Tutorial D**, by contrast, the expression appears only once.

The problem this example illustrates is that SQL's SELECT - FROM - WHERE template is simply too rigid. What we need to do, as the **Tutorial D** formulation makes clear, is perform a restriction of an extension; in SQL terms, we need to apply the WHERE clause to the result of the SELECT clause, as it were. But the SELECT - FROM - WHERE template forces the WHERE clause to apply to the result of the *FROM* clause, not the SELECT clause. To put it another way: in many respects, it's the whole point of the algebra that (thanks to closure) relational operations can be combined in arbitrary ways; but SQL's SELECT - FROM - WHERE template effectively means that queries *must* be expressed as a cartesian product,* followed by a restrict, followed by some combination of project and/or extend and/or rename—and many queries just don't fit this pattern.

Incidentally, you might be wondering why I didn't formulate the SQL query like this:

```
SELECT P.*, ( P.WEIGHT * 454 ) AS GMWT
FROM   P
WHERE  GMWT > 7000.0
```

(The change is in the last line.) The reason is that GMWT is the name of a column of *the final result;* table P has no such column, the WHERE clause thus makes no sense, and the expression fails on a syntax error.

Actually, the SQL standard does allow the query under discussion to be formulated in a style that's a little closer to that of **Tutorial D**:

```
SELECT TEMP.PNO, TEMP.GMWT
FROM ( SELECT P.PNO, ( P.WEIGHT * 454 ) AS GMWT
       FROM P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

As noted earlier, however, not all SQL products allow nested subqueries to appear in the FROM clause in this manner.

Here now is a definition. Let *r* be a relation. Then the *extension:*

```
EXTEND r ADD ( exp AS X )
```

is a relation with (a) heading equal to the heading of *r* extended with attribute *X*, and (b) body consisting of all tuples *t* such that *t* is a tuple of *r* extended with a value for attribute *X* that is computed by evaluating *exp* on that tuple of *r*. Relation *r* must not have an attribute called *X*, and *exp* must not refer to *X*. Observe that the result has

* It's true—thanks to, among other things, the ability to nest subqueries in the FROM clause—that the operands to that cartesian product aren't limited to being named tables (base tables or views), but the FROM clause as such still basically means cartesian product.

cardinality equal to that of *r* and degree equal to that of *r* plus one. The type of *X* in that result is the type of *exp*.

Summarize

Again I'll start with an example (the query is "For each supplier, get the supplier number and a count of the number of parts that supplier supplies"):

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( ) AS P_COUNT )
```

Given our usual sample values, the result looks like this:

SNO	P_COUNT
S1	6
S2	2
S3	1
S4	3
S5	0

Note the tuple for supplier S5 in particular; the PER specification indicates that the summarizing is to be done "per the projection of S on SNO," which means it produces a result with five tuples (one for each tuple in that projection). By contrast, the following SQL expression (which might be thought to be equivalent to the **Tutorial D** formulation):

```
SELECT SP.SNO, COUNT(*) AS P_COUNT
FROM   SP
GROUP  BY SP.SNO
```

produces a result containing tuples for suppliers S1, S2, S3, and S4 only. The reason is, of course, that it extracts supplier numbers from SP, and supplier S5 doesn't appear in SP at all. An SQL expression that *is* equivalent to the **Tutorial D** formulation is:

```
SELECT S.SNO, TEMP.P_COUNT
FROM   S, LATERAL ( SELECT COUNT(*) AS P_COUNT
                    FROM   SP
                    WHERE  SP.SNO = S.SNO ) AS TEMP
```

As I've already pointed out, however, not all SQL products support this kind of expression.

> **NOTE**
> **The standard requires the keyword LATERAL here because the subquery refers "laterally" to another element in the same FROM clause.**

Here now is the definition. Let *r* and *s* be relations such that *s* is of the same type as some projection of *r*, and let the attributes of *s* be *A1, A2, ..., An*. Then the *summarization:*

```
SUMMARIZE r PER ( s ) ADD ( summary AS X )
```

is a relation with (a) heading equal to the heading of *s* extended with attribute *X*, and (b) body consisting of all tuples *t* such that *t* is a tuple of *s* extended with a value for attribute *X*. That *X* value is computed by evaluating *summary* over all tuples of *r* that

have the same value for attributes *A1, A2, ..., An* as tuple *t* does. Relation *s* must not have an attribute called *X*, and *summary* must not refer to *X*. Observe that the result has cardinality equal to that of *s* and degree equal to that of *s* plus one. The type of *X* in that result is the type of *summary*.

What "summaries" are supported? Well, the set is open-ended but certainly includes the usual COUNT, SUM, AVG, MAX, and MIN. Here's an example involving MAX and MIN:

```
SUMMARIZE SP PER ( SP { SNO } ) ADD ( MAX ( QTY ) AS MAXQ ,
                                     MIN ( QTY ) AS MINQ )
```

This example illustrates two further points:

- It's possible to perform "multiple summarizations" within a single SUMMARIZE. (I didn't mention the point earlier, but analogous remarks apply to RENAME and EXTEND as well.)
- The PER operand in this example isn't just "of the same type as" some projection of the relation to be summarized, it actually is such a projection. In such cases the expression can be simplified slightly, as here:

  ```
  SUMMARIZE SP BY { SNO } ADD ( MAX ( QTY ) AS MAXQ ,
                                MIN ( QTY ) AS MINQ )
  ```

Other legal "summaries" include COUNTD, SUMD, and AVGD (where "D" stands for "distinct" and means "eliminate redundant duplicate values before summarizing"); AND, OR, and XOR (for attributes of type BOOLEAN); INTERSECT, UNION, and D_UNION (for relation-valued attributes); and so on.

By the way, COUNT and the rest here are *not* aggregate operators, though most of them do have the same names as aggregate operators (SQL confuses these two notions, with unfortunate results). An aggregate operator invocation is a scalar expression, and it returns a scalar value.* Here's an example:

```
VAR N INTEGER ;

N := COUNT ( S WHERE CITY = 'London' ) ;
```

Summaries, by contrast, are merely operands to SUMMARIZE invocations; they have no meaning outside that context, and in fact can't be used outside that context.

Here's one more SUMMARIZE example ("How many suppliers are there in London?"):

```
SUMMARIZE ( S WHERE CITY = 'London' ) ADD ( COUNT ( ) AS N )
```

Again, this example illustrates two points:

- This SUMMARIZE has no PER specification. By default, therefore, the summarizing is done *per TABLE_DEE*—that is, the expression shown is shorthand for:

* Nonscalar aggregate operators can be defined too, but they're beyond the scope of this book.

```
SUMMARIZE ( S WHERE CITY = 'London' )
          PER ( TABLE_DEE ) ADD ( COUNT ( ) AS N )
```

I remind you again that TABLE_DEE is the relation that has no attributes and one tuple (and is thus certainly of the same type as "some projection of" every possible relation!—namely, the projection of the relation in question on the empty set of attributes). The output from this SUMMARIZE therefore has *one* attribute and one tuple, and given our usual sample data values it looks like this:*

N
2

- I said a moment ago that aggregate operator invocations and summaries were different things, and you might be wondering what the difference is between the SUMMARIZE example under discussion and the COUNT operator invocation we saw a few paragraphs back:

  ```
  COUNT ( S WHERE CITY = 'London' )
  ```

  The overriding difference is, of course, that SUMMARIZE returns a relation and aggregate operators return a scalar. For further discussion, see Exercise 5-11 at the end of this chapter.

Group and Ungroup

Recall from Chapter 2 that relations with relation-valued attributes (RVAs for short) are legal. Figure 5-1 shows relations R1 and R4 from Figures 2-1 and 2-2 in that chapter; R4 has an RVA and R1 doesn't, but the two relations clearly convey the same information.

R1

SNO	PNO
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

R4

SNO	PNO_REL
S2	PNO P1 P2
S3	PNO P2
S4	PNO P2 P4 P5

FIGURE 5-1. *Relations R1 and R4 from Figures 2-1 and 2-2 in Chapter 2*

* The fact that the picture involves no double underlining is *not* a mistake. See Exercise 4-28 in Chapter 4.

Of course, we need a way to map between relations without RVAs and relations with them, and that's the purpose of the GROUP and UNGROUP operators. I don't want to go into a lot of detail on those operators here; let me just say that, given relations R1 and R4 as shown in Figure 5-1, respectively, the expression:

```
R1 GROUP ( { PNO } AS PNO_REL )
```

will produce R4, and the expression:

```
R4 UNGROUP ( PNO_REL )
```

will produce R1. SQL has no direct counterpart to these operators.

By the way: if R4 includes exactly one tuple for supplier number S*x*, say, and if the PNO_REL value in that tuple is empty, then the result of the foregoing UNGROUP will contain no tuple at all for supplier number S*x*. For further details, I refer you to my book *An Introduction to Database Systems,* Eighth Edition (Addison-Wesley, 2004) or the book *Databases, Types, and the Relational Model: The Third Manifesto,* Third Edition (Addison-Wesley, 2006), by Hugh Darwen and myself.

And by the way again: you might be wondering what operations on relations with RVAs look like. Well, operations on *any* relation, when they refer to some attribute *A* of that relation of type *T*, say, typically involve what we might call "suboperations" on values of that attribute *A* that are, precisely, operations that have been defined in connection with that type *T*. So if *T* is a relation type, those "suboperations" are operations that have been defined in connection with relation types—which is to say, they're essentially the relational operations (join and the rest) described in the present chapter! See Exercises 5-27 through 5-30 at the end of this chapter.

Expression Transformation

I've now covered all of the algebraic operators I want to discuss in any detail, but there are some related topics that merit attention in this chapter. The first is the possibility of transforming a given relational expression into another, logically equivalent expression. I mentioned this possibility in Chapter 3, in the section "Why Duplicate Tuples Are Prohibited," where I explained that such transformations are one of the things the optimizer does; in fact, such transformations constitute one of the two great ideas at the heart of relational optimization (the other, beyond the scope of this book, is the use of "database statistics" to do what's called *cost-based optimizing*). In this section, I want to say a little more about the process of expression transformation (or *query rewrite,* as it's sometimes called). I'll start with a trivial example. Consider the following expression (the query is "Get suppliers who supply part P2, together with the corresponding quantities"):

```
( ( S JOIN SP ) WHERE PNO = PNO('P2') ) { ALL BUT PNO }
```

Suppose there are 100 suppliers and 1,000,000 shipments, of which 500 are for part P2. If the expression is simply evaluated by brute force, as it were, without any optimization at all, the sequence of events is:

1. *Join S and SP:* This step involves reading the 100 supplier tuples; reading the 1,000,000 shipment tuples 100 times each, once for each of the 100 suppliers; constructing an intermediate result consisting of 1,000,000 tuples; and writing those 1,000,000 tuples back out to the disk. (I'm assuming for simplicity that tuples are physically stored as such, and I'm also assuming I can take "number of tuple I/O's" as a reasonable measure of performance. Neither of these assumptions is realistic, but this fact doesn't materially affect my argument.)
2. *Restrict the result of Step 1:* This step involves reading 1,000,000 tuples but produces a result containing only 500 tuples, which I'll assume can be kept in main memory.
3. *Project the result of Step 2:* This step involves no tuple I/O's at all, so we can ignore it.

The following procedure is equivalent to the one just described, in the sense that it produces the same final result, but is obviously much more efficient:

1. *Restrict SP to just the tuples for part P2:* This step involves reading 1,000,000 shipment tuples but produces a result containing only 500 tuples, which can be kept in main memory.
2. *Join S and the result of Step 1:* This step involves reading 100 supplier tuples (once only, not once per P2 shipment, because all the P2 shipments are in memory). The result contains 500 tuples (still in main memory).
3. *Project the result of Step 2:* Again we can ignore this step.

The first of these two procedures involves a total of 102,000,100 tuple I/O's, whereas the second involves only 1,000,100; thus, it's clear that the second procedure is over 100 times faster than the first. It's also clear that we'd like the implementation to use the second procedure rather than the first! If it does, then it is (in effect) transforming the original expression:

```
( S JOIN SP ) WHERE PNO = PNO('P2')
```

(I'm ignoring the final projection now, since it isn't really relevant to the argument) into the expression:

```
S JOIN ( SP WHERE PNO = PNO('P2') )
```

These two expressions are logically equivalent, but they have very different performance characteristics, as we've seen. If the system is presented with the first expression, therefore, we'd like it to transform it into the second before evaluating it—and of course it can. The point is that the relational algebra, being a high-level formalism, is subject to various formal *transformation laws*; for example, there's a law that says a join followed by a restriction can be transformed into a restriction followed by a join (I was using that particular law in the example). *And a good optimizer will know those laws,* and will apply them—because, of course, the performance of a query shouldn't depend on the specific syntax used to express that query in the first place. (Of course, it's an immediate consequence of the fact that not all of the algebraic operators are primitive that certain expressions can be transformed into others—for example, an expression involving

intersect can be transformed into one involving join instead—but there's much more to it than that, as we'll quickly see.)

Now, there are many possible transformation laws, and this isn't the place for an exhaustive discussion. All I want to do is highlight a few important cases and key points. First, the law mentioned in the previous paragraph is actually a special case of a more general law, called the *distributive* law. In general, the monadic operator *f distributes* over the dyadic operator *g* if $f(g(a,b)) = g(f(a),f(b))$ for all *a* and *b*. In ordinary arithmetic, for example, SQRT (square root) distributes over multiplication, because:

```
SQRT ( a * b ) = SQRT ( a ) * SQRT ( b )
```

for all *a* and *b* (take *f* as SQRT and *g* as "*"); thus, a numeric expression optimizer can always replace either of these expressions by the other when doing numeric expression transformation. As a counterexample, SQRT does *not* distribute over addition, because the square root of $a+b$ is not equal to the sum of the square roots of *a* and *b*, in general.

In relational algebra, restrict distributes over intersect, union, and difference. It also distributes over join, provided the restriction condition consists at its most complex of the AND of two separate conditions, one for each of the two join operands. In the case of the example discussed above, this requirement was satisfied—in fact, the restriction condition was very simple and applied to just one of the operands—and so we were able to use the distributive law to replace the expression with a more efficient equivalent. The net effect was that we were able to "do the restriction early." Doing restrictions early is almost always a good idea, because it serves to reduce the number of tuples to be scanned in the next operation in sequence, and probably reduces the number of tuples in the output from that operation too.

Here are some other specific cases of the distributive law, this time involving projection. First, project distributes over intersect and union, though not difference. Second, it also distributes over join, so long as all of the joining attributes are included in the projection. These laws can be used to "do projections early," which again is usually a good idea, for reasons similar to those given earlier for restrictions.

Two more important general laws are the laws of *commutativity* and *associativity*:

- The dyadic operator *g* is *commutative* if $g(a,b) = g(b,a)$ for all *a* and *b*. In ordinary arithmetic, for example, addition and multiplication are commutative, but division and subtraction aren't. In relational algebra, intersect, union, and join are all commutative,* but difference and division aren't. So, for example, if a query involves a join of two relations *r* and *s*, the commutative law tells us it doesn't matter which of *r* and *s* is taken as the "outer" relation and which the "inner." The system is therefore free to choose (say) the smaller relation as the "outer" one in computing the join.

* Strictly speaking, join is *not* commutative in SQL, because SQL attaches significance to the left-to-right ordering of columns, and *r* JOIN *s* and *s* JOIN *r* thus differ in their result column orderings.

- The dyadic operator g is *associative* if $g(a,g(b,c)) = g(g(a,b),c)$ for all a, b, c. In arithmetic, addition and multiplication are associative, but subtraction and division aren't. In relational algebra, intersect, union, and join are all associative, but difference and division aren't. So, for example, if a query involves a join of three relations r, s, and u, the associative and commutative laws together tell us we can join the relations pairwise in any order we like. (They also tell us it's legitimate to define an n-adic or prefix version of the operator, as **Tutorial D** does.) The system is thus free to decide which of the various possible sequences is most efficient.

Observe that all of these transformations can be performed without any regard for either actual data values or actual storage structures (indexes and the like) in the database as physically stored. In other words, such transformations represent optimizations that are virtually guaranteed to be good, regardless of what the database looks like physically.

Relational Comparisons

In Chapter 2, I mentioned the fact that the equality comparison operator "=" applies to every type. In particular, therefore, it applies to *relation* types; that is, given two relations r and s of the same relation type T, we must at least be able to test whether those two relations are equal. Other comparisons might be useful, too; for example, we might want to test whether r is a subset of s (meaning every tuple in r is also in s), or whether r is a *proper* subset of s (meaning every tuple in r is also in s, but s contains at least one tuple that isn't in r).

Now, I must immediately explain that these operators aren't relational operators as such—that is, they're not part of the relational algebra—because their result is a truth value, not a relation. But it's convenient to discuss them in this chapter, and I will. Here's a simple example:

```
S { CITY } = P { CITY }
```

The left comparand is the projection of suppliers on CITY, the right comparand is the projection of parts on CITY, and the comparison returns TRUE if these two projections are equal, FALSE otherwise. In other words, the comparison (which is, of course, a boolean expression) means: "Is the set of supplier cities equal to the set of part cities?"

Here's another example:

```
S { SNO } ⊃ SP { SNO }
```

Explanation: The symbol "⊃" here means "is a proper superset of." The meaning of this comparison (considerably paraphrased) is: "Are there any suppliers who supply no parts at all?"

Other useful relational comparison operators include "⊇" ("is a superset of"), "⊆" ("is a subset of"), and "⊂" ("is a proper subset of").

The obvious place where relational comparisons are useful is in connection with restrictions.* Let's look at some examples. Consider first the query "Get suppliers who supply all parts." Here's a possible formulation:

```
WITH ( SP RENAME ( SNO AS X ) ) AS R :
     S WHERE ( R WHERE X = SNO ) { PNO } = P { PNO }
```

Explanation: The expression WITH ... AS R is effectively equivalent to an assignment that assigns the value of the expression SP RENAME (SNO AS X) to some temporary, system-generated relvar R; after that assignment, the value of R is a relation that's identical to the current value of relvar SP, except that attribute SNO has been renamed as X.† (The purpose of introducing the names R and X is simply to avoid a naming conflict that would subsequently arise otherwise.) Then, for a given supplier S*x*, say, in relvar S, the expression:

```
( R WHERE X = SNO ) { PNO }
```

evaluates to a relation with one attribute (PNO), giving part numbers for all parts supplied by supplier S*x*. Note in particular that if supplier S*x* supplies no parts, that relation will contain no tuples. Finally, that degree-one relation is tested for equality with the relation that's the projection of P on PNO. Clearly, that test will give TRUE if and only if the set of part numbers for parts supplied by S*x* is equal to the set of part numbers in relvar P. The overall result thus contains precisely those tuples from relvar S that represent suppliers who supply all of the parts mentioned in relvar P.

Of course, we can write the entire query as a single expression if we like:

```
S WHERE ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO ) { PNO } = P { PNO }
```

But using WITH to introduce names for the results of subexpressions often helps to simplify the job of formulating complicated queries. Here's a definition: if *rx* is a relational expression that mentions some relvar *R*, then WITH *ry* AS *R: rx*, where *ry* is another relational expression, is also a relational expression. Of course, *rx* might be a WITH expression, too, like this:

```
WITH ry AS R1 : WITH rz AS R2 : rx
```

This latter expression can be abbreviated to just:

```
WITH ry AS R1, rz AS R2 : rx
```

We'll see plenty of examples of this abbreviated form later.

By the way, the foregoing example ("Get suppliers who supply all parts") is very similar to one that's often used to illustrate the use of the relational divide operator. To be specific, the expression:

```
SP { SNO, PNO } DIVIDEBY P { PNO }
```

* But see Exercise 5-26 at the end of the chapter.

† SQL supports WITH too, though it writes the operands the other way around: WITH *name* AS *expression*.

(which I gave as an example of divide in the earlier section "The Original Operators") is often characterized as a relational formulation of the query "Get supplier numbers for suppliers who supply all parts." But it isn't! Rather, it is a relational formulation of the query "Get supplier numbers for suppliers who *supply at least one part and in fact* supply all parts." (If you're wondering what the logical difference is here, see Exercise 5-25 at the end of this chapter.) In my opinion, the formulation involving a relational comparison is not only easier to understand than the divide formulation—it also has the advantage of being *correct*.

Here's another example. The query is: "Get pairs of supplier numbers, S*x* and S*y* say, such that S*x* and S*y* supply exactly the same set of parts each." This query is very difficult without relational comparisons! Here it is:

```
WITH ( SP RENAME ( SNO AS SX ) ) { SX, PNO } AS R1 ,
     ( SP RENAME ( SNO AS SY ) ) { SY, PNO } AS R2 ,
     R1 { SX } AS R3 ,
     R2 { SY } AS R4 ,
     ( R1 JOIN R4 ) AS R5 ,
     ( R2 JOIN R3 ) AS R6 ,
     ( R1 JOIN R2 ) AS R7 ,
     ( R3 JOIN R4 ) AS R8 ,
     SP { PNO } AS R9 ,
     ( R8 JOIN R9 ) AS R10 ,
     ( R10 MINUS R7 ) AS R11 ,
     ( R6 JOIN R11 ) AS R12 ,
     ( R5 JOIN R11 ) AS R13 ,
     R12 { SX, SY } AS R14 ,
     R13 { SX, SY } AS R15 ,
     ( R14 UNION R15 ) AS R16 ,
     R7 { SX, SY } AS R17 :
R17 MINUS R16
```

But with relational comparisons it's fairly straightforward:

```
WITH ( S RENAME ( SNO AS SX ) ) { SX } AS RX ,
     ( S RENAME ( SNO AS SY ) ) { SY } AS RY :
( RX JOIN RY ) WHERE ( SP WHERE SNO = SX ) { PNO } =
                     ( SP WHERE SNO = SY ) { PNO }
```

As an aside, I remark that appending "AND SX < SY" to the WHERE clause here would produce a slightly tidier result; to be specific, it would (a) eliminate pairs of the form (S*x*,S*x*) and (b) ensure that the pairs (S*x*,S*y*) and (S*y*,S*x*) don't both appear.

One particular comparison that's often needed in practice is an "=" comparison between a given relation *r* and an empty relation of the same type—in other words, a test to see whether *r* is empty. So let me introduce the following shorthand:

```
IS_EMPTY ( rx )
```

This expression is defined to return TRUE if the relation *r* denoted by the relational expression *rx* is empty and FALSE otherwise. I'll be relying heavily on this construct in the next chapter.

Another common requirement is to be able to test whether a given tuple *t* appears in a given relation *r:*

```
t ∈ rx
```

This expression is defined to be shorthand for the relational comparison:

```
RELATION { t } ⊆ rx
```

The left operand here is a relation selector invocation, and it returns a relation containing just the specified tuple *t*. The comparison thus returns TRUE if *t* appears in the relation *r* denoted by the relational expression *rx* and FALSE otherwise ("∈" is the *set membership* operator; the expression $t \in r$ can be read as "*t* [appears] in *r*"). In fact, as you've probably realized, the "∈" operator is essentially SQL's IN operator—except that the left operand of SQL's IN is usually a scalar, not a row, which means there's some kind of coercion going on. Be that as it may, SQL certainly doesn't support any relational comparisons apart from this one rather special one. (Well, it does support NOT IN; so does **Tutorial D**, in the form of "∉".)

More on Relational Assignment

The assignment operator ":=" resembles the equality comparison operator "=" in that it applies to every type, and relation types are no exception. Relational assignment in particular resembles "=" and the other comparison operators from the previous section in another respect as well: it isn't part of the relational algebra. Why not? Because its target must be, very specifically, a relvar, not a relation (relvars aren't part of the relational algebra either—there's no notion of updating in the relational algebra as such, and "variable" *means* "updatable"). Nevertheless, I want to say a little more about relational assignment in this chapter; to be specific, I want to examine the INSERT, DELETE, and UPDATE shorthands a little more closely. As you know, these operators *are* just shorthand for certain relational assignments—but now I'm in a position to explain just what the "longhand" versions look like, in terms of appropriate algebraic operators. I'll do this by showing some simple examples.

First, relational assignment in general looks like this:

```
R := rx
```

R here is a relvar and *rx* is a relational expression of the same type as *R*. The effect is to assign the relation *r* that's denoted by the expression *rx* to the relvar *R* (and I remind you from the exercises in Chapter 2 that after the assignment, the boolean expression $R = r$ is required to evaluate to TRUE: *The Assignment Principle*). Here's a simple example:

```
S := RELATION { TUPLE { SNO SNO('S1'), SNAME NAME('Smith'),
                        STATUS 20, CITY 'London' } ,
                TUPLE { SNO SNO('S2'), SNAME NAME('Jones'),
                        STATUS 10, CITY 'Paris'  } ,
                TUPLE { SNO SNO('S3'), SNAME NAME('Blake'),
                        STATUS 30, CITY 'Paris'  } ,
```

```
                TUPLE { SNO SNO('S4'), SNAME NAME('Clark'),
                        STATUS 20, CITY 'London' } ,
                TUPLE { SNO SNO('S5'), SNAME NAME('Adams'),
                        STATUS 30, CITY 'Athens' } } ;
```

(As we saw in Chapter 3, the expression on the right here is another relation selector invocation; in fact, it's a relation literal.)

Now I can turn to INSERT, DELETE, and UPDATE. Let relvar PQ be defined as follows:

```
VAR PQ BASE RELATION { PNO PNO, QTY QTY } KEY { PNO } ;
```

Here's a possible INSERT on this relvar:

```
INSERT PQ ( SUMMARIZE SP PER ( P { PNO } )
                      ADD ( SUM ( QTY ) AS QTY ) ) ;
```

And here's the "longhand" assignment equivalent:

```
PQ := PQ UNION ( SUMMARIZE SP PER ( P { PNO } )
                           ADD ( SUM ( QTY ) AS QTY ) ) ;
```

In other words, the INSERT works by forming the union, *pq* say, of the old value of relvar PQ and the relation denoted by the SUMMARIZE invocation, and then assigning that relation *pq* to relvar PQ. (By the way, I'm assuming here that it's not an error to insert a tuple that already exists in the target. If it is, that UNION in the expansion should be replaced by D_UNION.)

Next, a DELETE example:

```
DELETE S WHERE CITY = 'Athens' ;
```

Longhand equivalent:

```
S := S WHERE NOT ( CITY = 'Athens' ) ;
```

Finally, an UPDATE example:

```
UPDATE P WHERE CITY = 'London'
       ( WEIGHT := 2 * WEIGHT , CITY := 'Oslo' ) ;
```

Longhand equivalent:

```
P := WITH ( P WHERE CITY = 'London' ) AS R1 ,
          ( EXTEND R1 ADD ( 2 * WEIGHT AS NEW_WEIGHT,
                            'Oslo' AS NEW_CITY ) ) AS R2 ,
          R2 { ALL BUT WEIGHT, CITY } AS R3 :
     R3 RENAME ( NEW_WEIGHT AS WEIGHT, NEW_CITY AS CITY ) ;
```

This one needs a little more explanation. First, R1 is the set of tuples to be updated (loosely speaking—see the section "Updating Is Set-at-a-Time" in Chapter 4). Next, we extend each tuple in R1 with the applicable new WEIGHT and CITY values; that's R2. Then we throw away the old WEIGHT and CITY values (that's R3). Finally, we rename NEW_WEIGHT and NEW_CITY as WEIGHT and CITY, respectively, and assign the resulting relation to relvar P. Notice the use of the "multiple" forms of EXTEND and RENAME in this example.

The ORDER BY Operator

The last topic I want to address in this chapter is ORDER BY. ORDER BY is yet another operator that's not part of the relational algebra; as I pointed out in Chapter 1, in fact, it isn't a relational operator at all, because it produces a result that isn't a relation (it does take a relation as input, but it produces something else—namely, a *sequence of tuples*—as output). Of course, I'm not saying ORDER BY isn't useful; however, I *am* saying it can't legally or sensibly appear in a relational expression.* By definition, therefore, the following expressions, though legal, aren't relational expressions as such:

```
S MATCHING SP
  ORDER ( ASC SNO )
```

```
SELECT DISTINCT S.*
FROM   S, SP
WHERE  S.SNO = SP.SNO
ORDER  BY S.SNO ASC
```

That said, I'd like to point out that for a couple of reasons ORDER BY (just ORDER, in **Tutorial D**) is actually a rather strange operator. First, it effectively works by sorting tuples into some specified sequence—and yet "<" and ">" aren't defined for tuples (see Exercise 3-11 in Chapter 3). Second, it's not a *function*. All of the operators of the relational algebra—in fact, all read-only operators, in the usual sense of that term—*are* functions, meaning there's always just one possible output for any given input. By contrast, ORDER BY can produce several different outputs from the same input. As an illustration of this point, consider the effect of the operation ORDER BY CITY on our usual suppliers relation. Clearly, this operation can return any of four distinct results, corresponding to the following sequences (I'll show just the supplier numbers, for simplicity):

```
S5, S1, S4, S2, S3

S5, S4, S1, S2, S3

S5, S1, S4, S3, S2

S5, S4, S1, S3, S2
```

A note on SQL: It would be remiss of me not to mention in passing that most of the algebraic operators have analogs in SQL that also aren't functions. This state of affairs is due to the fact that, as indicated in the exercises in Chapter 2, SQL allows the comparison *v1* = *v2* to evaluate to TRUE even if *v1* and *v2* are distinct. For example, consider the character strings 'Paris' and 'Paris ', respectively (note the trailing space in the latter); these values are clearly distinct, and yet SQL sometimes regards them as equal. As a

* In particular, therefore, it can't appear in a view definition—despite the fact that at least one well-known product allows it to!

consequence, certain queries give what the standard calls "possibly nondeterministic results." Here's a simple example:

```
SELECT DISTINCT S.CITY FROM S
```

If one supplier has CITY value 'Paris' and another 'Paris ', then the result will include either or both of 'Paris' and 'Paris ', but which result we get might be undefined. We could even legitimately get one result on one day and another on another, even if the database hasn't changed at all in the interim. You might like to meditate on some of the implications of this state of affairs.

Summary

Almost inevitably, this has been the longest chapter in the book. Even so, I haven't covered "all known" algebraic operators, nor have I explained every last detail of the ones I did cover; but I hope I *have* covered enough to give you a good sense of what the algebra is all about. To review briefly, here's a list of the operators I did discuss: rename, restrict, project (including the ALL BUT form), join, semijoin (MATCHING), intersection, union, disjoint union (D_UNION), difference, semidifference (NOT MATCHING), product, divide, theta-join (including equijoin), extend, summarize, group, and ungroup. I also discussed certain nonalgebraic operators: relational comparisons, INSERT, DELETE, and UPDATE (and relational assignment), and ORDER BY. Other topics I covered include:

- *Closure:* I explained the need for a set of relation type inference rules, so that we always know the type of the result of any given relational expression.
- *Primitive operators:* I showed that many of the operators are in fact shorthand for certain combinations of others. In particular, intersect and product are special cases of join, and difference is a special case of semidifference.
- *Join:* I also showed that—thanks to the fact that join is both commutative and associative—it was possible to define an n-adic or prefix version of the operator (where n is any integer greater than or equal to zero). The join of no relations at all is TABLE_DEE.
- *SQL expressions:* I gave a conceptual algorithm for evaluating SQL SELECT - FROM - WHERE expressions. I also suggested that the SELECT - FROM - WHERE template is sometimes too rigid.
- *Summaries versus aggregate operators:* I stressed the point (without getting into too much detail) that these are logically distinct constructs.
- *Expression transformation:* I briefly explained the distributive, commutative, and associative laws and described their role in optimization ("query rewrite").
- *WITH:* I discussed the use of WITH in simplifying expression formulation by allowing names to be introduced for the results of subexpressions.

Exercises

Exercise 5-1. From a relational perspective, what's wrong (if anything) with the following SQL expressions?

```
a. SELECT * FROM S, SP
b. SELECT S.SNO, S.CITY FROM S
c. SELECT SP.SNO, SP.PNO, 2 * SP.QTY FROM SP
d. SELECT P.PNO FROM P
e. SELECT S.SNO FROM S, SP
f. SELECT S.SNO FROM S ORDER BY S.CITY DESC
```

Exercise 5-2. Closure is important in the relational model for the same kind of reason that numeric closure is important in ordinary arithmetic. In arithmetic, however, there's one situation where the closure property breaks down, in a sense—namely, division by zero. Is there any analogous situation in the relational algebra?

Exercise 5-3. Given the usual suppliers-and-parts database, what's the value of the expression JOIN{S,SP,P}? What's the corresponding predicate? *Warning:* There's a trap here; in fact, some might even argue that the trap is such that it makes join (as I've defined it in the body of this chapter) a rather dangerous operation. What do you think of that argument?

Exercise 5-4. Why do you think the project operator is so called?

Exercise 5-5. Given our usual sample values for the suppliers-and-parts database, what values do the following expressions denote? In each case, give an informal interpretation of the expression in natural language.

```
a. S SEMIJOIN ( SP WHERE PNO = PNO('P2') )
b. P NOT MATCHING ( SP WHERE SNO = SNO('S2') )
c. S { CITY } MINUS P { CITY }
d. ( S { SNO, CITY } JOIN P { PNO, CITY } ) { ALL BUT CITY }
e. JOIN { ( S RENAME ( CITY AS SC ) ) { SC } ,
          ( P RENAME ( CITY AS PC ) ) { PC } }
```

Exercise 5-6. Union, intersection, product, and join are all both commutative and associative. Verify these claims. Also verify that semijoin is associative but not commutative.

Exercise 5-7. In what circumstances (if any) are *r* SEMIJOIN *s* and *s* SEMIJOIN *r* equivalent?

Exercise 5-8. Which of the algebraic operators (if any) have a definition that doesn't rely on tuple equality?

Exercise 5-9. The SQL FROM clause FROM *t1*, *t2*, ..., *tn* (where each *ti* is an expression denoting a table) returns the cartesian product of its arguments. But what if $n = 1$? What's the cartesian product of just one table? And by the way, what's the product of *t1* and *t2* if *t1* and *t2* both contain duplicate rows?

Exercise 5-10. Show that rename isn't primitive.

Exercise 5-11. Distinguish between a "summary" and an aggregate operator. What do the SQL analogs of these constructs look like?

Exercise 5-12. Give an expression involving EXTEND instead of SUMMARIZE that's logically equivalent to the following:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( ) AS NP )
```

Exercise 5-13. Given our usual sample values for the suppliers-and-parts database, what values do the following expressions denote? In each case, give an informal interpretation of the expression in natural language.

```
a. EXTEND S ADD ( 'Supplier' AS TAG )
b. EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY AS SHIPWT )
c. EXTEND P ADD ( WEIGHT * 454 AS GMWT, WEIGHT * 16 AS OZWT )
d. EXTEND S
      ADD ( COUNT ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO )
          AS NP )
e. SUMMARIZE S BY { CITY } ADD ( AVG ( STATUS ) AS AVG_STATUS )
```

Exercise 5-14. Which 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 )
```

Exercise 5-15. Simplifying just slightly, an aggregate operator invocation in **Tutorial D** takes the 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 SNO = SNO('S1'), QTY )
SUM ( ( SP WHERE SNO = SNO('S1') ) { QTY } )
```

Note the difference between these two expressions: the first gives the total of *all* shipment quantities for supplier S1, and the second gives the total of all *distinct* shipment quantities for supplier S1. How does the scheme just outlined differ from its SQL counterpart?

Exercise 5-16. In **Tutorial D**, if the argument to an aggregate operator invocation happens to be an empty set, COUNT returns zero and so does SUM; MAX and MIN return, respectively, the lowest and the highest value of the applicable type; AND and OR return TRUE and FALSE, respectively; and AVG raises an exception (I ignore **Tutorial D**'s other aggregate operators here deliberately). What does SQL do in these circumstances and why?

Exercise 5-17. Let relation R4 from Figure 5-1 denote the current value of some relvar *R*. If R4 has the meaning described in Chapter 2, give the predicate for that relvar *R*.

Exercise 5-18. Let *r* be the relation denoted by the following expression:

```
SP GROUP ( { } AS X )
```

What does *r* look like, given our usual sample value for SP? Also, what does the following expression yield?

```
r UNGROUP ( X )
```

Exercise 5-19. Without using the IS_EMPTY shorthand, write a **Tutorial D** expression that returns TRUE if the current value of the parts relvar P is empty and FALSE otherwise. Also show an SQL analog of that expression.

Exercise 5-20. Write **Tutorial D** and/or SQL expressions for the following queries on the suppliers-and-parts database:

a. Get all shipments.

b. Get supplier numbers for suppliers who supply part P1.

c. Get suppliers with status in the range 15 to 25 inclusive.

d. Get part numbers for parts supplied by a supplier in London.

e. Get part numbers for parts not supplied by any supplier in London.

f. Get city names for cities in which at least two suppliers are located.

g. Get all pairs of part numbers such that some supplier supplies both of the indicated parts.

h. Get the total number of parts supplied by supplier S1.

i. Get supplier numbers for suppliers with a status lower than that of supplier S1.

j. Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.

k. Get part numbers for parts supplied by all suppliers in London.

l. Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

m. Get suppliers who supply at least all parts supplied by supplier S2.

n. Get supplier numbers for suppliers who supply at least all parts supplied by at least one supplier who supplies at least one London part.

Exercise 5-21. Prove the following statements (making them more precise, where necessary):

a. A sequence of restrictions against a given relation can be transformed into a single restriction.

b. A sequence of projections against a given relation can be transformed into a single projection.

c. A restriction of a projection can be transformed into a projection of a restriction.

Exercise 5-22. Union is said to be *idempotent*, because *r* UNION *r* is identically equal to *r* for all *r*. (Is this true in SQL?) As you might expect, idempotence can be useful in expression transformation. Which other relational operators (if any) are idempotent?

Exercise 5-23. Let *r* be a relation. What does the expression *r*{} mean? What does it return?

Exercise 5-24. The following boolean expression:

```
x > y AND y > 3
```

(which might be part of a query) is clearly equivalent to—and can therefore be transformed into—the following:

```
x > y AND y > 3 AND x > 3
```

The equivalence is based on the fact that the comparison operator ">" is *transitive*. The transformation is worth making if *x* and *y* are from different relations, because it enables the system to perform an additional restriction (using $x > 3$) before doing the greater-than join implied by $x > y$. As we saw in the body of the chapter, doing restrictions early is generally a good idea; having the system *infer* additional "early" restrictions, as here, is also a good idea. Do you know of any commercial products that actually perform this kind of optimization?

Exercise 5-25. Consider the following expressions:

```
a. WITH ( P WHERE COLOR = 'Purple' ) AS PP ,
        ( SP RENAME ( SNO AS X ) ) AS T :
   S WHERE ( T WHERE X = SNO ) { PNO } ⊇ PP { PNO }

b. WITH ( P WHERE COLOR = 'Purple' ) AS PP :
   S JOIN ( SP { SNO, PNO } DIVIDEBY PP { PNO } )
```

These are both attempts at formulating the query "Get suppliers who supply every purple part." Given our usual sample data values, show the result returned in each case. Which result (and which formulation), if either, do you regard as correct? Justify your answer.

Exercise 5-26. Consider the restriction *r* WHERE *bx*. In the body of the chapter, I said that every attribute mentioned in the restriction condition *bx* must be an attribute of *r*, but I also mentioned that *bx* was subject to certain further limitations. One of those limitations* is that, strictly speaking, *bx* is supposed to consist of just a single term of the form *x Op y*, where each of *x* and *y* is either an attribute of *r* or a literal and *Op* is a comparison operator that makes sense for *x* and *y*. But a real language will allow WHERE clauses to contain boolean expressions of arbitrary complexity, just as long as the only "free variables" (see Appendix A) in that expression are attributes of *r*. Show that it's legitimate to extend the definition of restrict in such ways—that is, show that such extensions are really just shorthand for something we already know is legitimate.

* Another is that *bx* mustn't involve any quantifiers, nor must it be equivalent to an expression that does (see Appendix A).

Exercise 5-27. Here are two simple expressions involving relation R4 from Figure 5-1 in the body of the chapter. What queries do they represent?

```
( R4 WHERE TUPLE { PNO PNO('P2') } ∈ PNO_REL ) { SNO }

( ( R4 WHERE SNO = SNO('S2') ) UNGROUP ( PNO_REL ) ) { PNO }
```

Exercise 5-28. Given our usual sample values for the suppliers-and-parts database, what does the following expression denote?

```
EXTEND S
   ADD ( ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO ) { PNO }
      AS PNO_REL )
```

Exercise 5-29. Let the relation returned by the expression in the previous exercise be kept as a relvar called SSP. What do the following updates do?

```
INSERT SSP RELATION
  { TUPLE { SNO SNO('S6'),
            PNO_REL RELATION { TUPLE { PNO PNO('P5') } } } } ;

UPDATE SSP WHERE SNO = SNO('S2')
  ( INSERT PNO_REL RELATION { TUPLE { PNO PNO('P5') } } ) ;
```

Exercise 5-30. Using relvar SSP from the previous exercise, write expressions for the following queries:

- Get pairs of supplier numbers for suppliers who supply exactly the same set of parts.
- Get pairs of part numbers for parts supplied by exactly the same set of supliers.

Exercise 5-31. A *quota query* is a query that specifies a desired limit, or quota, on the cardinality of the result: for example, the query "Get the three heaviest parts," for which the specified quota is three. Give **Tutorial D** and SQL formulations of this query.

CHAPTER SIX

Integrity Constraints

I've touched on the issue of integrity constraints at many points in preceding chap-ters, but it's time to get more specific. Here first is a rough definition, repeated from Chapter 1: an *integrity constraint* (*constraint* for short) is basically just a boolean expression that must evaluate to TRUE. Constraints are so called because they constrain the values that can legally appear in some particular context. The ones we're interested in fall into two broad categories, *type* constraints and *database* constraints; in essence, a type constraint defines the values that constitute a given type, and a database constraint defines the values that can appear in a given database.

By the way, it's worth noting right away that constraints in general can be regarded as a formal version of what some people call *business rules*. I'll touch on this point again in the next chapter.

Type Constraints

As we saw in Chapter 2, one of the things we have to do when we define a type is specify the values that make up that type. Here's an example:

```
TYPE QTY
     POSSREP QPR
```

```
3            { Q INTEGER
4                  CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

Explanation:

- Line 1 just says we're defining a type called QTY ("quantities").
- Line 2 says that quantities have a *possible representation* called QPR. Now, *physical* representations are always hidden from the user, as we know from Chapter 2. However, **Tutorial D** requires every TYPE statement to include at least one POSSREP specification,* indicating that values of the type in question can "possibly be represented" in some specific way; and unlike physical representations, possible representations—which we usually abbreviate to just *possreps*—are definitely visible to the user (in the example, users definitely know that quantities have a possrep called QPR). Note very carefully, however, that there's absolutely no suggestion that the specified possible representation is the same as the physical representation, whatever that happens to be; it might be or it might not, but either way it makes no difference to the user.
- Line 3 says the possrep QPR has a single component, called Q, which is of type INTEGER; in other words, values of type QTY can "possibly be represented" by integers (and users are aware of this fact).
- Finally, line 4 specifies that those integers must lie in the range 0 to 5000 inclusive—and it's that fact that constitutes the *type constraint* for type QTY; in other words, valid quantities are precisely those that can possibly be represented by integers in the specified range.

Here's a slightly more complicated example:

```
TYPE POINT
     POSSREP CARTESIAN { X NUMERIC, Y NUMERIC
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 } ;
```

Type POINT denotes points in two-dimensional space; it has a possrep CARTESIAN with two numeric components called X and Y (corresponding, presumably, to cartesian coordinates), and there's a type constraint that says, in effect, that the only points we're interested in are those that lie on or inside a circle with center the origin and radius 100.

Selectors and THE_ Operators

Before I continue with my discussion of type constraints as such, I'd like to digress for a few moments; the QTY and POINT examples raise a number of issues that I need to address somewhere, and here's as good a place as any.

Recall from Chapter 2 that user-defined types like QTY and POINT have associated selector and THE_ operators. Well, those operators are intimately related to the possrep notion; in fact, selector operators correspond one-to-one to possreps, and THE_ operators correspond one-to-one to possrep components. Here are some examples:

* There are some minor exceptions to this rule that need not concern us here.

`QPR ( 250 )`

This expression is a selector invocation for type QTY. The selector has the same name, QPR, as the sole possrep for that type; it takes an argument that corresponds to, and is of the same type as, the sole component of that possrep, and it returns a quantity (that is, a value of type QTY). *Note:* In practice, possreps often have the same name as the associated type—I used different names in the QTY example to make it clear there's a logical difference between the possrep and the type, but it would be much more usual not to. In fact, there's a syntax rule that says we can omit the possrep name from the TYPE statement entirely if we want to, in which case it defaults to the associated type name. So let's change the QTY type definition accordingly:

```
TYPE QTY POSSREP { Q INTEGER CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

Now the possrep and the corresponding selector are both called QTY, and the selector invocation shown earlier becomes just QTY(250)—which is the style I've been using for selectors throughout this book prior to this point. I'll assume this revised definition for type QTY for the rest of this chapter.

`QTY ( A + B )`

The argument to a QTY selector invocation can be specified as an arbitrarily complex expression (just so long as it's of type INTEGER, of course). If that expression is a literal, as it was in the previous example, the selector invocation is a literal in turn; in other words, a literal is a special case of a selector invocation, as we already know from Chapter 3.

`THE_Q ( QZ )`

This expression is a THE_ operator invocation for type QTY. The operator is named THE_Q because Q is the name of the sole component of the sole possrep for type QTY; it takes an argument (specified as an arbitrarily complex expression) of type QTY; and it returns the integer that's the Q component of the possrep for that argument.

Now let's redefine type POINT to make its possrep have the same name as the type (for simplicity, let's also drop the type constraint, at least for the time being):

```
TYPE POINT POSSREP { X NUMERIC, Y NUMERIC } ;
```

To continue with the examples:

`POINT ( 5.7, -3.9 )`

This is a POINT selector invocation (actually a POINT literal).

`THE_X ( P )`

This expression returns the NUMERIC value that's the X coordinate of the cartesian possible representation of the point that's contained in the variable P. (That variable must be of type POINT, of course.)

By the way, POINT is a good example of a type for which we might want to define more than one possrep—for example:

```
TYPE POINT POSSREP CARTESIAN { X NUMERIC, Y NUMERIC }
           POSSREP POLAR { R NUMERIC, THETA NUMERIC } ;
```

The two possreps here reflect the fact that points in two-dimensional space can indeed "possibly be represented" by either cartesian or polar coordinates. Each possrep has two components, both of type NUMERIC and both user-visible. Continuing with the examples:

`POLAR ( H, K )`
: This is a POLAR selector invocation; it returns a value of type POINT.

`THE_THETA ( P )`
: This expression returns the NUMERIC value that's the THETA coordinate of the polar possible representation of the point that's contained in the variable P (which again must be of type POINT).

Type Constraints Continued

Now let's get back to type constraints as such. Suppose I had defined type QTY as follows, with no explicit constraint:

```
TYPE QTY POSSREP { Q INTEGER } ;
```

This definition is defined to be shorthand for this one:

```
TYPE QTY POSSREP { Q INTEGER CONSTRAINT TRUE } ;
```

With either of these definitions, anything that can possibly be represented by an integer would be a legitimate QTY value (and so type QTY would still be subject to a certain constraint, albeit a fairly weak one). In other words, the specified possrep imposes a kind of *a priori* type constraint, and the CONSTRAINT specification if present imposes an additional constraint, over and above that *a priori* one. Informally, however, we usually take the term "type constraint" to refer to what's stated in the CONSTRAINT specification.

One important issue I've ducked so far is the question of when type constraints are checked. In fact, they're checked *whenever some selector is invoked.* Assume again that values of type QTY are subject to the constraint that they must be possibly representable as integers in the range 0 to 5000. Then the expression QTY(250) is an invocation of the QTY selector, and that invocation succeeds. By contrast, the expression QTY(6000) is also such an invocation, but it fails. In general, in fact, we can *never* tolerate an expression that's supposed to denote a value of some type *T* but in fact doesn't; indeed, "a value of type *T*" that's not a value of type *T* is a contradiction in terms. As a consequence, no variable—in particular, no relvar—can ever be assigned a value that's not of the right type.

A note on SQL: You will have noticed that all of my examples in this section have been expressed in **Tutorial D**, not SQL. That's because, believe it or not, SQL doesn't support type constraints at all—except for the rather trivial *a priori* ones, of course.* In other words, although SQL would let you define type QTY (for example) and specify that

* It doesn't support possreps, either. As for selectors and THE_ operators, the picture is too complicated to describe in detail here; suffice it to say that analogs of those operators are usually available, though they're not always provided "automatically" as they are in **Tutorial D**.

quantities must be representable as integers, it would not let you say those integers must lie in a certain range. For such reasons among others, I don't want to consider SQL user-defined types in detail in this book. However, I will at least show SQL counterparts to the **Tutorial D** QTY and POINT definitions that we've already seen:

```
CREATE TYPE QTY AS INTEGER FINAL ;

CREATE TYPE POINT AS ( X NUMERIC, Y NUMERIC ) NOT FINAL ;
```

One last point to close this section: defining anything to be of some particular type imposes a constraint on that thing, of course. In particular, defining attribute QTY of relvar SP (for example) to be of type QTY imposes the constraint that no tuple in relvar SP will ever contain a value in the QTY position that fails to satisfy the QTY type constraint. (This is an example of what's sometimes called an *attribute constraint*.)

Database Constraints

To recap, a database constraint constrains the values that can appear in a given database. In **Tutorial D**, such constraints are specified by means of a CONSTRAINT statement (or by some shorthand that's effectively equivalent to such a statement); in SQL, they're specified by means of a CREATE ASSERTION statement (or, again, by some equivalent shorthand). I don't want to get into details of those shorthands—at least, not yet—because they're really just a matter of syntax; for now, let's stay with the "longhand" forms. I'll begin with a series of examples (**Tutorial D** on the left and SQL on the right, as usual).

Example 1

Status values must be in the range 1 to 100 inclusive:

```
CONSTRAINT C1 IS_EMPTY
( S WHERE STATUS < 1
    OR    STATUS > 100 ) ;
```

```
CREATE ASSERTION C1 CHECK
( NOT EXISTS
( SELECT S.* FROM S
  WHERE  S.STATUS < 1
  OR     S.STATUS > 100 ) ) ;
```

This constraint involves just a single attribute of a single relvar; as a consequence, it can be checked for a given supplier tuple by examining just that tuple in isolation—there's no need to look at any other tuples in the relvar or any other relvars in the database. (For this reason, such constraints are sometimes referred to, informally, as *tuple constraints*.)

Example 2

Suppliers in London must have status 20:

```
CONSTRAINT C2 IS_EMPTY
( S WHERE CITY = 'London'
    AND   STATUS ≠ 20 ) ;
```

```
CREATE ASSERTION C2 CHECK
( NOT EXISTS
( SELECT S.* FROM S
  WHERE  S.CITY = 'London'
  AND    S.STATUS <> 20 ) ) ;
```

This constraint involves two distinct attributes (of the same relvar), but it's still the case, as it was with constraint C1, that it can be checked for a given supplier tuple by examining just that tuple in isolation.

Example 3

No two tuples in relvar S have the same supplier number (in other words, {SNO} is a key for that relvar):*

```
CONSTRAINT C3                 | CREATE ASSERTION C3 CHECK
   COUNT ( S ) =              |  ( UNIQUE ( SELECT S.SNO
   COUNT ( S { SNO } ) ;      |            FROM   S ) ) ;
```

Like constraints C1 and C2, this constraint still involves just one relvar; however, it clearly can't be checked for a given supplier tuple by examining just that tuple in isolation. Of course, it's very unlikely in practice that constraint C3 would be specified in longhand as shown—some kind of explicit KEY shorthand is obviously to be preferred. I give the longhand form merely to make the point that such shorthands are indeed, in the final analysis, just shorthands.

By the way, the SQL formulation of constraint C3 needs a word of explanation. UNIQUE is an SQL operator that returns TRUE if and only if every row within its argument table is distinct; the UNIQUE invocation in the constraint thus returns TRUE if and only if no two rows in table S have the same supplier number. But note that if I had followed my preferred discipline of always specifying DISTINCT, thus:

```
UNIQUE ( SELECT DISTINCT S.SNO FROM S )
```

then the UNIQUE couldn't possibly return FALSE! In fact, UNIQUE makes no sense from a relational point of view; it's needed in SQL only because SQL tables aren't relations, in general.

Here for interest is an SQL formulation of constraint C3 that more closely resembles the **Tutorial D** formulation:

```
CREATE ASSERTION C3 CHECK
 ( ( SELECT COUNT ( * ) FROM S ) =
   ( SELECT COUNT ( SNO ) FROM S ) ) ;
```

Example 4

No supplier with status less than 20 can supply part P6:

```
CONSTRAINT C4 IS_EMPTY        | CREATE ASSERTION C4 CHECK
( ( S JOIN SP )               | ( NOT EXISTS
  WHERE STATUS < 20           | ( SELECT *
```

* Or a *super*key, rather. A superkey is a superset of a key. See Chapter 7 for further discussion.

```
AND   PNO = PNO('P6') ) ;
```

```
FROM   S, SP
WHERE  S.SNO = SP.SNO
AND    S.STATUS < 20
AND    SP.PNO = PNO('P6') ) ) ;
```

This constraint involves—in fact, interrelates—two distinct relvars, S and SP; in general, of course, a database constraint might involve or interrelate any number of distinct relvars.

> **NOTE**
> *Terminology:* A constraint that involves just a single relvar is known, informally, as a *relvar constraint* (sometimes a *single*-relvar constraint, for emphasis). A constraint that involves two or more distinct relvars is known, informally, as a *multi-relvar constraint.*

Example 5

Every supplier number in relvar SP must appear in relvar S:

```
CONSTRAINT C5
   SP { SNO } ⊆ S { SNO } ;
```

```
CREATE ASSERTION C5 CHECK
( NOT EXISTS
   ( SELECT SP.SNO
     FROM   SP
     EXCEPT
     SELECT S.SNO
     FROM   S ) ) ;
```

As you can see, the **Tutorial D** formulation of this contraint involves a relational comparison. SQL doesn't support relational comparisons, however, and so we have to indulge in some circumlocution in the SQL formulation. Of course, given that {SNO} is a key—in fact, *the* key—for relvar S, it's clear that constraint C5 is basically just the foreign key constraint from SP to S. The usual FOREIGN KEY syntax can thus be regarded as shorthand for constraints like C5.

So when are database constraints checked? Conventional wisdom has it that single-relvar constraint checking is *immediate* (meaning it's done whenever the relvar in question is updated), while multi-relvar constraint checking is *deferred* to end-of-transaction ("commit time"). I want to argue, however, that *all* database constraint checking must be immediate, and deferred checking—which is supported in the SQL standard, and indeed in some SQL products—is a logical mistake. In order to explain this unorthodox view, I need to digress for a moment to discuss transactions.

Transactions

Transaction theory is a large topic in its own right. As mentioned in Chapter 4, however, it doesn't have much to do with the relational model as such (at least, not directly), and

for that reason I don't want to discuss it in detail here. In any case, you're a database professional, and I'm sure you're familiar with basic transaction concepts. The standard reference—highly recommended, by the way—is *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter (Morgan Kaufmann, 1993). All I want to do here is briefly review the so-called *ACID properties* of transactions.

ACID is an acronym; it stands for *atomicity - consistency - isolation - durability*. Here are brief explanations:

Atomicity
: Transactions are "all or nothing."

Consistency
: Any given transaction transforms a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points.*

Isolation
: Any given transaction's updates are concealed from all other transactions, until such time as the given transaction commits.

Durability
: Once a given transaction commits, its updates survive in the database, even if there's a subsequent system crash.

Now, one argument in favor of transactions has always been that they're supposed to act as "a unit of integrity" (that's what the consistency property is all about). But I don't believe that argument; as I've more or less said already, I believe that *statements* have to be that unit, and that database constraints must therefore be satisfied *at statement boundaries*. The section immediately following gives my justification for this position.

Why Database Constraint Checking Must Be Immediate

I have at least five reasons for taking the position that database constraints must be satisfied at statement boundaries. The first and biggest one is as follows. As we know from Chapter 4, a database can be regarded as a collection of propositions, propositions that we believe are true. And if that collection is ever allowed to include any inconsistencies, then *all bets are off*. As I'll show in the section "Constraints and Predicates," later in this chapter, we can never trust the answers we get from an inconsistent database! While it might be true, thanks to the isolation property, that no more than one transaction ever sees any particular inconsistency, the fact remains that that particular transaction does see the inconsistency and can therefore produce wrong answers.

* A database state is *consistent* if and only if it satisfies all defined constraints (consistency is just another word for integrity in this context).

Now, I think this first argument is strong enough to stand on its own, but I'll give the other four arguments as well, for purposes of reference if nothing else. Second, then, I don't agree that any given inconsistency can be seen by only one transaction, anyway; that is, I don't believe in the isolation property. Part of the problem here is that the word *isolation* doesn't mean the same in the world of transactions as it does in ordinary English—in particular, it doesn't mean that transactions can't communicate with one another. For if transaction *T1* produces some result, in the database or elsewhere, that's subsequently read by transaction *T2*, then *T1* and *T2* aren't truly isolated from each other (and this remark applies regardless of whether *T1* and *T2* run concurrently or otherwise). In particular, therefore, if (a) *T1* sees an inconsistent state of the database and therefore produces an incorrect result, and (b) that result is then seen by *T2*, then (c) the inconsistency seen by *T1* has effectively been propagated to *T2*. In other words, it can't be guaranteed that a given inconsistency, if permitted, *will* be seen by just one transaction, anyway.

Third, we surely don't want every program (or other "code unit") to have to cater for the possibility that the database might be inconsistent when it's invoked. There's a severe loss of orthogonality if a program that assumes consistency can't be used safely while constraint checking is deferred. In other words, I want to be able to specify code units independently of whether they're to be executed as a transaction as such or just as part of a transaction. (In fact, I'd like support for *nested transactions*, but that's a topic for another day.)

Fourth, *The Principle of Interchangeability* (of base relvars and views—see Chapter 4) implies that the very same constraint might be a single-relvar constraint with one design for the database and a multi-relvar constraint with another. For example, recall these two views from Chapter 4:

```
VAR LS  VIRTUAL ( S WHERE CITY = 'London' ) ;

VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' ) ;
```

These views satisfy the constraint that no supplier number appears in both. However, there's no need to state that constraint explicitly, because it's implied by the single-relvar constraint that {SNO} is a key for relvar S (along with the fact that every supplier has exactly one city, which is implicit in the design of relvar S). But suppose we made LS and NLS base relvars and defined their union as a view called S. Then the constraint would have to be stated explicitly:

```
CONSTRAINT C6 IS_EMPTY
   ( LS { SNO } JOIN
        NLS { SNO } ) ;
```

```
CREATE ASSERTION C6 CHECK
  ( NOT EXISTS
    ( SELECT *
      FROM   LS, NLS
      WHERE  LS.SNO = NLS.SNO ) ) ;
```

Now what was previously a single-relvar constraint has become a multi-relvar constraint instead. Thus, if we agree that single-relvar constraints must be checked immediately, we must surely agree that multi-relvar constraints must be checked immediately as well.

Fifth and last, there's an optimization technique called *semantic optimization* (it involves expression transformation, but I deliberately didn't discuss it in the section of that name in Chapter 5). For example, consider the expression (SP JOIN S){PNO}. Clearly, the join here is a foreign-to-matching-primary-key join; as a consequence, every SP tuple does join to some S tuple and therefore contributes a part number to the overall result. So there's no need to do the join!—the expression can be simplified to just SP{PNO}. The point to note, however, is that this transformation is valid *only* because of the semantics of the situation; in general, each operand to a join will include some tuples that have no counterpart in the other operand and so don't contribute to the overall result, and transformations such as the one just shown therefore won't be valid. In the case at hand, however, every SP tuple *must* have a counterpart in S, because of the integrity constraint—actually a foreign key constraint—that says that every shipment must have a supplier, and so the transformation is valid after all. A transformation that's valid only because a certain integrity constraint is in effect is called a *semantic* transformation, and the resulting optimization is called a semantic optimization.

In principle, any constraint whatsoever can be used in semantic optimization (we're not limited to foreign key constraints). For example, suppose the suppliers-and-parts database is subject to the constraint "All red parts must be stored in London," and consider the query:

> Get suppliers who supply only red parts and are located in the same city as at least one of the parts they supply.

This is a fairly complex query. But thanks to the integrity constraint, we see that it can be transformed—transformed by the optimizer, I mean, not by the user—into this much simpler one:

> Get London suppliers who supply only red parts.

We could easily be talking about several orders of magnitude improvement in performance here. And so, while few products do much in the way of semantic optimization at the time of writing (as far as I know), I certainly expect them to do more in the future, because the payoff is so dramatic.

To get back to the main thread of the discussion, I now observe that if a given constraint is to be usable in semantic optimization, that constraint must be satisfied *at all times* (more precisely, at statement boundaries), not just at transaction boundaries. As we've seen, semantic optimization means using constraints to simplify queries in order to improve performance. Clearly, then, if some constraint is violated at some time, then any simplification based on that constraint won't be valid at that time, and query results based on that simplification will be wrong at that time (in general).

NOTE

Of course, we could adopt the weaker position that "deferred constraints" (meaning constraints for which the checking is deferred) just can't be used in semantic optimization—but I think such a position would effectively just mean we've shot ourselves in the foot, that's all.

In sum, then, database constraints must be satisfied—that is, they must evaluate to TRUE, given the values currently appearing in the database—*at statement boundaries* (or, very informally, "at semicolons"); in other words, they must be checked at the end of any statement that might cause them to be violated. If any such check fails, the effects on the database of the offending statement are undone and an exception is raised.

But Doesn't Some Checking Have to Be Deferred?

As I've said, the conventional wisdom is that multi-relvar constraint checking, at least, has to be deferred to commit time (the arguments of the previous section notwithstanding). By way of example, suppose the suppliers-and-parts database is subject to the following constraint:

```
CONSTRAINT C7
    COUNT ( ( S WHERE SNO = SNO('S1') ) { CITY }
            UNION
            ( P WHERE PNO = PNO('P1') ) { CITY } ) < 2 ;
```

This constraint says that supplier S1 and part P1 must never be in different cities. To elaborate: if relvars S and P contain tuples for supplier S1 and part P1, respectively, then those tuples must contain the same CITY value (if they didn't, the COUNT invocation would return the value two). However, it's legal for relvar S to contain no tuple for S1, or relvar P to contain no tuple for P1, or both (in which case the COUNT invocation will return either one or zero). Given our usual sample values, then, each of the following SQL UPDATEs will fail under immediate checking:*

```
UPDATE S SET CITY = 'Paris' WHERE SNO = SNO('S1') ;

UPDATE P SET CITY = 'Paris' WHERE PNO = PNO('P1') ;
```

Now, the conventional solution to this problem is to defer the checking and to bundle up the two UPDATEs into a transaction, like this:

```
BEGIN TRANSACTION ;
UPDATE S SET CITY = 'Paris' WHERE SNO = SNO('S1') ;
UPDATE P SET CITY = 'Paris' WHERE PNO = PNO('P1') ;
COMMIT ;
```

* I show those UPDATEs in SQL precisely because checking *is* immediate in **Tutorial D** and the conventional solution I'll be discussing in a moment therefore doesn't work in **Tutorial D**.

In this conventional solution, the constraint is checked at commit time, and the database is inconsistent between the two UPDATEs. In particular, if the transaction were to ask the question "Are supplier S1 and part P1 in different cities?" between the two UPDATEs (and if we assume that tuples for S1 and P1 do exist), it would get the answer *yes*.

Multiple Assignment

A better solution to the foregoing problem is to support a *multiple* form of assignment, which allows any number of individual assignments to be performed "simultaneously." For example (switching back now to **Tutorial D**):

```
UPDATE S WHERE SNO = SNO('S1') ( CITY := 'Paris' ) ,
UPDATE P WHERE PNO = PNO('P1') ( CITY := 'Paris' ) ;
```

Explanation: First, note the comma separator, which means the two UPDATEs are part of the same overall statement. Second, UPDATE is really assignment, of course, and the foregoing "double UPDATE" is thus just shorthand for a double assignment of the following general form:

```
S := ... , P := ... ;
```

This double assignment assigns one value to relvar S and another to relvar P, all as part of the same overall operation. In general, the semantics of multiple assignment are as follows:

- First, all of the expressions on the right sides are evaluated.
- Second, all of the constituent assignments are then executed in sequence as written.*

Observe that, precisely because the expressions on the right sides are evaluated before any constituent assignment is executed, none of those constituent assignments can depend on the result of any other. Moreover, since multiple assignment is considered to be a single operation, no integrity checking is performed "in the middle of" any such assignment. (Indeed, this fact is the major rationale for supporting multiple assignment in the first place.) In the example, therefore, the double assignment succeeds where the two separate single assignments failed.

Note in particular in the example that there's now no way for the transaction to see an inconsistent state of the database between the two UPDATEs, because the notion of "between the two UPDATEs" now has no meaning. Note further that there's now no need for deferred checking at all.

By the way, SQL does have some support for multiple assignment; in fact, it's had *some* support for many years. First, referential actions such as CASCADE imply, in effect, that a single DELETE or UPDATE statement can cause several tables to be updated "at the same time," as part of a single operation. Second, the ability to update (for example) certain

* This definition requires some slight refinement in the case where two or more of the individual assignments specify the same target. The details are beyond the scope of this book.

join views implies the same thing. Third, FETCH INTO and SELECT INTO are both multiple assignment operations, of a kind. Fourth, SQL:2003 introduced a multiple-assignment form of the SET statement. And so on (this isn't an exhaustive list). However, the one kind of multiple assignment that SQL doesn't currently support is an explicit assignment to several different tables (precisely the case illustrated by the foregoing example, of course).

One last point: please understand that support for multiple assignment doesn't mean we can discard support for transactions; transactions are still necessary for recovery and concurrency purposes, at least. All I'm saying is that transactions aren't the "unit of integrity" they're usually supposed to be.

Constraints and Predicates

Recall from Chapter 4 that the relvar predicate for a relvar is *the intended interpretation*—loosely, the *meaning*—for that relvar. For example, the predicate for relvar S looks like this:

> Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY.

In an ideal world, therefore, this predicate would serve as the criterion for acceptability of updates on relvar S—that is, it would dictate whether a given INSERT or DELETE or UPDATE operation on that relvar can be accepted. But this goal is clearly unachievable:

- For one thing, the system can't know what it means for a "supplier" to be "under contract" or to be "located" somewhere; to repeat, these are matters of interpretation. For example, if the supplier number S1 and the city name London happen to appear together in the same tuple, then the user can interpret that fact to mean that supplier S1 is located in London,* but there's no way the system can do anything analogous.
- For another, even if the system could know what it means for a supplier to be under contract or to be located somewhere, it still couldn't know *a priori* whether what the user tells it is true! If the user asserts to the system, by means of some update, that there's a supplier S6 named Lopez with status 30 and city Madrid, there's no way for the system to know whether that assertion is true. All the system can do is check that the user's assertion doesn't violate any integrity constraints. Assuming it doesn't, then the system will accept the user's assertion *and will treat it as true from that point forward* (until such time as the user tells the system, by executing another update, that it isn't true anymore).

* Or that supplier S1 *used to be* located in London, or that supplier S1 *has an office* in London, or that supplier S1 *doesn't* have an office in London, or any of an infinite number of other possible interpretations (corresponding, of course, to an infinite number of possible predicates).

Thus, the *pragmatic* "criterion for acceptability of updates," as opposed to the ideal one, is not the predicate but the corresponding set of constraints, which might thus be regarded as the system's approximation to the predicate.* Equivalently:

The system can't enforce truth, only consistency.

In other words: the system can't guarantee that the database contains only true propositions—it can guarantee only that it doesn't contain anything that violates any constraint (meaning it contains no inconsistencies). Sadly, truth and consistency aren't the same thing. To be specific:

- If the database contains only true propositions, then it's consistent, but the converse isn't necessarily so.
- If the database is inconsistent, then it contains at least one false proposition, but the converse isn't necessarily so.

More succinctly, *correct* implies *consistent* (but not the other way around), and *inconsistent* implies *incorrect* (but not the other way around)—where to say that the database is *correct* is to say it faithfully reflects the true state of affairs in the real world, no more and no less.

Now let me try to pin down these notions a little more precisely. Let *R* be a base relvar—I'll get to views in just a moment—and let *C1*, *C2*, ..., *Cn* be all of the defined database constraints, single- or multi-relvar, that mention *R*. Assume for simplicity that each *Ci* is just a boolean expression (in other words, let's ignore the constraint names, for simplicity). Then the boolean expression:

```
( C1 ) AND ( C2 ) AND ... AND ( Cn ) AND TRUE
```

is *the total relvar constraint* for relvar *R* (but I'll refer to it for the purposes of this book as just *the* constraint for *R*). Note that final "AND TRUE," by the way; the implication is that if no constraints are defined for a given relvar, the default is just TRUE.

Now let *RC* be "the" relvar constraint for relvar *R*. Clearly, *R* must never be allowed to have a value that causes *RC* to evaluate to FALSE. This state of affairs is the motivation for (the first version of) what I like to call *The Golden Rule*:

No update operation must ever cause any relvar constraint to evaluate to FALSE.

Now let *V* be a view. Then *V* too has a "total relvar constraint" (which I'll usually abbreviate to just *the* constraint for *V*, for simplicity), derived in an obvious manner from the constraints that apply to the relvars in terms of which *V* is defined. For example, let *SC* be the total constraint for base relvar S. Then the total constraint for view LS ("London suppliers") is:

```
( SC ) AND ( CITY = 'London' )
```

* Precisely for this reason, I've elsewhere used the term *internal predicate* to refer to constraints and the term *external predicate* to refer to what I'm here calling the relvar predicate.

Now let *DB* be a database, and let *DB* contain relvars *R1, R2, ..., Rn* (only). Let the constraints for those relvars be *RC1, RC2, ..., RCn*, respectively. Then *the total database constraint* for *DB, DBC* say—which I'll refer to for the purposes of this book as just *the* constraint for *DB*—is the AND of all of those relvar constraints:

```
( RC1 ) AND ( RC2 ) AND ... AND ( RCn )
```

And here's a correspondingly extended (in fact, the final) version of *The Golden Rule*:

> No update operation must ever cause any database constraint to evaluate to FALSE.

Observe in particular that—in accordance with my position that all integrity checking must be immediate—the rule talks in terms of update operations, not transactions.

Now I can take care of a piece of unfinished business. I've said we can never trust the answers we get from an inconsistent database. Here's the proof. As we know, a database can be regarded as a collection of propositions. Suppose that collection is inconsistent; that is, suppose it implies that both *p* and NOT *p* are true, where *p* is some proposition. Now let *q* be any arbitrary proposition. Then:

- From the truth of *p*, we can infer the truth of *p* OR *q*.
- From the truth of *p* OR *q* and the truth of NOT *p*, we can infer the truth of *q*.

But *q* was arbitrary! It follows that any proposition whatsoever (even obviously false ones like 1 = 0) can be shown to be true in an inconsistent system.

Miscellaneous Issues

In this section, I want to address a few integrity-related issues that don't fit very well in any of the preceding sections.

First, since it's basically a boolean expression that must evaluate to TRUE, it follows that from a formal perspective a constraint is a *proposition*. Here for example is constraint C1 from the earlier section "Database Constraints":

```
CONSTRAINT C1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Given a particular value for relvar S, the boolean expression:

```
IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 )
```

(which might be thought of, loosely, as the constraint proper) is certainly either true or false, unconditionally, and that's the definition of what it means to be a proposition (see Chapter 4).

Second, suppose relvar S already contains a tuple that violates constraint C1 when the CONSTRAINT statement just shown is executed. Then that execution must fail, of course. More generally, whenever we try to define a new database constraint, the system must first check to see whether that constraint is satisfied by the database at that time. If it

isn't, the constraint must be rejected; otherwise, it must be accepted and enforced from that point forward.

Third, recall from Chapter 1 that the relational model includes what I called a "generic" integrity rule: namely, the *referential integrity* rule (I deliberately ignore the entity integrity rule). But it should be clear that the referential integrity rule is different in kind from the constraints we've been examining in this chapter. It's really a *meta*constraint, in a sense; it says, for example, that in the specific database containing relvars S, P, and SP, there must be certain specific constraints (foreign key constraints) between SP and S and between SP and P—because if there aren't, then that database might violate the referential integrity metaconstraint. Likewise, in the specific database containing relvars EMP and DEPT (see Chapter 1), there must be a specific foreign key constraint between EMP and DEPT, because if there isn't, then again that database might violate the referential integrity metaconstraint.

Fourth, I never mentioned the point explicitly, but I trust it's obvious that we want constraints to be stated *declaratively*. Although the SQL standard does include fairly extensive support for declarative constraints, at least some of the major SQL products don't; instead, they assume you'll use *triggered procedures*—also known as just *triggers*—to enforce integrity. (The standard includes explicit support for triggers, too.) As I pointed out in Chapter 1, however, declarative solutions are always to be preferred over procedural ones, if they're available. Also, declarative constraints in particular open the door to the possibility of doing semantic optimization, which triggers don't.

Another issue I didn't mention previously is the possibility of supporting *transition constraints*. A transition constraint is a constraint on the legal transitions that variables of some kind (relvars in particular) can make from one value to another. For example, a person's marital status can change from "never married" to "married" but not the other way around. Here's an example ("No supplier's status must ever decrease"):

```
CONSTRAINT C8 IS_EMPTY
   ( ( ( S' { SNO, STATUS } RENAME ( STATUS AS STATUS' ) )
       JOIN
       ( S { SNO, STATUS } ) )
     WHERE STATUS' > STATUS ) ;
```

Explanation: I'm adopting the convention that a primed relvar name such as S' refers to the indicated relvar as it was *prior to the update under consideration*. Constraint C8 thus says: "If we join the old value of S and the new one and restrict the result to just those tuples where the old status is greater than the new one, the final result must be empty." (Since the join is on SNO, any tuple in the join for which the old status is greater than the new one would represent a supplier whose status had decreased.)

Last, I hope you agree from everything we've covered in this chapter that constraints are *vital*—and yet they seem to be very poorly supported in commercial products; indeed, they seem to be underappreciated at best, if not completely misunderstood. The emphasis in the commercial world always seems to be on *performance, performance, performance;*

other objectives, such as ease of use, data independence, and in particular integrity, seem so often to be sacrificed to—or at best to take a back seat to—that overriding goal.* But what's the point of a system performing well if we can't be sure the information we're getting from it is correct? Frankly, I don't care how fast a system runs if I don't feel I can trust it to give me the right answers to my queries.

Summary

I've discussed two basic kinds of constraints, type constraints and database constraints. A type constraint defines the set of values that constitute a given type; in **Tutorial D**, it's specified as part of the definition of the type in question, and it's expressed in terms of a *possrep* (possible representation) for that type. The possrep itself imposes an *a priori* constraint on the type (and SQL doesn't support any type constraints apart from such *a priori* ones). Type constraints are checked as part of the execution of selector operators. As a digression, I elaborated on the notion of selectors and the related notion of THE_ operators, both of which are intimately related to the possrep notion.

Database constraints constrain the values that can appear in a given database (if they apply to just one relvar, they're sometimes referred to, informally, as *relvar* constraints). They're specified by means of a CONSTRAINT statement in **Tutorial D** or a CREATE ASSERTION statement in SQL, and they're supposed to be checked "at semicolons" (though they might not be, in SQL). Checking constraints at semicolons means checking them at the end of any statement that might cause them to be violated—which basically means *relational assignments,* since relational assignment is fundamentally the only operation that can update the database. I gave a series of arguments for rejecting the conventional wisdom that multi-relvar constraints, at least, need not be checked until commit time (and in passing I briefly discussed the notion of *semantic optimization*). I also introduced the important idea of *multiple assignment.*

Next, I showed that "the" relvar constraint for relvar *R* might be regarded as the system's approximation to the relvar predicate for *R;* in particular, it serves as the criterion for acceptability of updates on *R*. "The system can't enforce truth, only consistency." *The Golden Rule* says:

> No update operation must ever cause any database constraint to evaluate to FALSE.

I concluded by claiming that integrity constraints are absolutely vital. To me, in fact, they're what database systems are all about. To say it again: I don't care how fast your system runs if I can't trust the answers it gives me.

* I don't mean to suggest that proper support for integrity checking implies bad performance; in fact, I believe quite strongly that it should *improve* performance. All I mean is that there tends to be a huge emphasis in vendor development effort on performance issues, to the exclusion of other matters such as data integrity.

Exercises

Exercise 6-1. Define the terms *type constraint* and *database constraint.* When are such constraints checked? What happens if the check fails?

Exercise 6-2. State *The Golden Rule.*

Exercise 6-3. What do you understand by the terms *attribute constraint; tuple constraint; relvar constraint; "the" database constraint; "the" relvar constraint; single-relvar constraint; multi-relvar constraint?*

Exercise 6-4. Distinguish between possible and physical representations.

Exercise 6-5. Explain as carefully as you can (a) what a selector is and (b) what a THE_ operator is.

Exercise 6-6. Suppose the only legal CITY values are London, Paris, Rome, Athens, Oslo, Stockholm, Madrid, and Amsterdam. Define a type called CITY that satisfies this constraint. Is there a way to impose the same constraint without an explicit CITY type? If so, compare and contrast the different approaches.

Exercise 6-7. Throughout this book I assume that SNO is a user-defined type. Give a definition for this type. Assume that the only legal supplier numbers are ones that can be represented by a character string of at least two characters, of which the first is an "S" and the remainder denote a decimal integer in the range 1 to 9999.

Exercise 6-8. A *line segment* is a straight line connecting two points in the euclidean plane. Give a corresponding type definition.

Exercise 6-9. Can you think of a type for which we might want to specify two different possreps? If there are two or more possreps for the same type, does it make sense for each to include a type constraint?

Exercise 6-10. Can you think of an example of a type for which different possreps have different numbers of components?

Exercise 6-11. Which operations might cause constraints C1–C8 from the body of the chapter to be violated?

Exercise 6-12. Constraint C1 (for example) had the property that it could be checked for a given tuple by examining just that tuple in isolation; constraint C4 (for example) did not. What is it, formally, that accounts for this difference? What's the pragmatic significance, if any, of this difference?

Exercise 6-13. Can you give a **Tutorial D** database constraint that's *exactly* equivalent to the specification KEY{SNO} for relvar S?

Exercise 6-14. Give an SQL formulation of constraint C7 from the body of the chapter.

Exercise 6-15. Give an SQL formulation of constraint C8 from the body of the chapter.

Exercise 6-16. Using **Tutorial D** and/or SQL, write constraints for the suppliers-and-parts database to express the following requirements:

a. All red parts must weigh less than 50 pounds.

b. Every London supplier must supply part P2.

c. No two suppliers can be located in the same city.

d. At most one supplier can be located in Athens at any one time.

e. There must exist at least one London supplier.

f. At least one red part must weigh less than 50 pounds.

g. The average supplier status must be at least 10.

h. No shipment can have a quantity more than double the average of all such quantities.

i. No supplier with maximum status can be located in the same city as any supplier with minimum status.

j. Every part must be located in a city in which there is at least one supplier.

k. Every part must be located in a city in which there is at least one supplier of that part.

l. Suppliers in London must supply more different kinds of parts than suppliers in Paris.

m. Suppliers in London must supply more parts in total than suppliers in Paris.

In each case, state which operations might cause the constraint to be violated.

Exercise 6-17. In a footnote in the section "Constraints and Predicates," I said that if the values S1 and London appeared together in some tuple, then it might mean (among many other possible interpretations) that supplier S1 doesn't have an office in London. Actually, this particular interpretation is extremely unlikely. Why? (*Hint:* Remember the Closed World Assumption.)

Exercise 6-18. Suppose no "cascade delete" rule is stated declaratively for suppliers and shipments. Write a **Tutorial D** statement that will delete some specified supplier and all shipments for that supplier in a single operation.

Exercise 6-19. Using the syntax sketched for transition constraints in the section "Miscellaneous Issues," write transition constraints to express the following requirements:

a. Suppliers in Athens can move only to London or Paris, and suppliers in London can move only to Paris.

b. The total shipment quantity for a given part can never decrease.

c. The total shipment quantity for a given supplier cannot be reduced in a single update to less than half its current value. (What do you think the qualification "in a single update" means here? Why is it important? *Is* it important?)

Exercise 6-20. Distinguish between correctness and consistency.

Exercise 6-21. Do you think the following is or should be a valid **Tutorial D** TYPE statement?

```
TYPE TTT POSSREP { ... CONSTRAINT FALSE } ;
```

Exercise 6-22. Do you think the following is or should be a valid **Tutorial D** CONSTRAINT statement?

```
CONSTRAINT FFF FALSE ;
```

Exercise 6-23. Investigate any SQL product that might be available to you. What semantic optimization does it support, if any?

Exercise 6-24. Why do you think SQL fails to support type constraints? What are the consequences of this state of affairs?

Exercise 6-25. The discussion in this chapter of types in general, and type constraints in particular, tacitly assumed that types were all (a) scalar and (b) user-defined. Do the concepts discussed apply equally well to nonscalar types and/or system-defined types?

CHAPTER SEVEN

Database Design Theory

The goal of data independence has the direct implication that logical and physical database design are different disciplines: logical design is concerned with what the database looks like to the user, and physical design is concerned with how the logical design maps to physical storage. The primary focus of this chapter is on logical design, and I'll use the unqualified term *design* to mean logical design specifically, until further notice.

One point I want to stress right away is this. Recall that "the" relvar constraint for relvar *R* can be regarded as the system's approximation to the predicate for *R*; recall too that the predicate for *R* is the intended interpretation, or meaning, for *R*. It follows that constraints and predicates are highly relevant to the business of logical design! Indeed, logical design is, in essence, precisely a process of pinning down the predicates as carefully as possible and then mapping those predicates to relvars and constraints. Of course, those predicates are necessarily somewhat informal (they're what some people like to call "business rules"); by contrast, the relvar and constraint definitions are necessarily formal.

Incidentally, the foregoing state of affairs explains why I'm not much of a fan of entity/relationship (E/R) modeling and similar pictorial methodologies. The problem with E/R diagrams and similar pictures is that they're completely incapable of representing all but a few rather specialized constraints. Thus, although it might be OK to use such diagrams to

explicate the overall design at a high level of abstraction, it's misleading, and in some respects quite dangerous, to think of such a diagram as actually *being* the design in its entirety. *Au contraire:* the design is the relvars, which the diagrams do show, *together with the constraints,* which they don't.

There's another general point I need to make up front. Recall from Chapter 4 that views are supposed to look and feel just like base relvars (I don't mean views defined as mere shorthands, I mean views that insulate the user from the "real" database in some way). In general, in fact, any given user interacts not with a database that contains only base relvars (the "real" database), but rather with what might be called a *user database* that contains some mixture of base relvars and views. Of course, that user database is supposed to look and feel like the real database to that user...and so it follows that all of the design principles to be discussed in this chapter apply equally well to such user databases, not just to the real database.

I feel compelled to make one further introductory remark. Several reviewers of earlier drafts of this chapter seemed to assume that what I was trying to do was teach elementary database design. But I wasn't. You're a database professional, so you're supposed to be familiar with design basics already. So the purpose of the chapter is not to explain the design process as it's actually carried out in practice; rather, the purpose is to reinforce certain aspects of design that you already know, by looking at them from a possibly unfamiliar perspective, and to explore certain other aspects that you might not already know. I don't plan to spend a lot of time covering what should be familiar territory. Thus, for example, I deliberately won't go into a lot of detail on second and third normal forms, because they're part of conventional design wisdom and shouldn't need any elaboration in a book of this nature (in any case, they're not all that important in themselves except as a stepping-stone to Boyce/Codd normal form, which I *will* discuss in this chapter).

The Place of Design Theory

Design theory as such isn't part of the relational model; rather, it's a separate theory in its own right that builds on top of that model. (It's appropriate to think of it as part of relational theory overall, but it's not, to repeat, part of the model as such.) However, it does rely on certain fundamental notions—for example, the operators projection and join—that *are* part of the model.

And another thing: the design theory I'm talking about doesn't really tell you how to do design! Rather, it tells you what goes wrong if you don't design the database in the "obvious" way. Consider suppliers and parts, for example. The obvious design is the one I've been assuming in this book all along; I mean, it's "obvious" that three relvars are necessary, that attribute STATUS belongs in relvar S, that attribute COLOR belongs in relvar P, that attribute QTY belongs in relvar SP, and so on. But why exactly are these things obvious? Well, suppose we tried a different design; for example, suppose we moved the STATUS attribute out of relvar S and into relvar SP (intuitively the wrong

place for it, since status has to do with suppliers, not shipments). Figure 7-1 shows a sample value for this revised shipments relvar (which I'll call STP to avoid confusion).

STP	SNO	STATUS	PNO	QTY
	S1	20	P1	300
	S1	20	P2	200
	S1	20	P3	400
	S1	20	P4	200
	S1	20	P5	100
	S1	20	P6	100
	S2	10	P1	300
	S2	10	P2	400
	S3	30	P2	200
	S4	20	P2	200
	S4	20	P4	300
	S4	20	P5	400

FIGURE 7-1. *Relvar STP—sample value*

A glance at the figure is sufficient to show what's wrong with this design: it's *redundant*, in the sense that every tuple for supplier S1 tells us S1 has status 20, every tuple for supplier S2 tells us S2 has status 10, and so on. And design theory tells us that not designing the database in the obvious way will lead to such redundancy, and it also tells us the consequences of such redundancy. In other words, design theory is basically all about reducing redundancy, as we'll soon see. For such reasons, design theory has been characterized—perhaps a little unkindly—as *a good source of bad examples*. What's more, it has also been criticized on the grounds that it's all just common sense anyway. I'll come back to this criticism in the next section.

To put a more positive spin on matters, design theory can be useful in checking that designs produced via some other methodology don't violate any formal design principles. Then again...the sad fact is, while those formal design principles do constitute the scientific part of the design discipline, there are numerous aspects of design that they simply don't address at all. *Database design is still largely subjective in nature;* the formal principles I'm going to describe in this chapter represent the one small piece of science in what's otherwise a mostly artistic endeavor.

So I want to consider the scientific part of design. To be specific, I want to examine two broad topics, *normalization* and *orthogonality*. Now, I assume you already know a lot about the first of these, at least. In particular, I assume you know that:

- There are several different *normal forms* (first, second, third, and so on).
- Loosely speaking, if relvar R is in $(n+1)$st normal form, then it's certainly in nth normal form.
- It's possible for a relvar to be in nth normal form and not in $(n+1)$st normal form.
- The higher the normal form the better, from a design point of view.
- These ideas all rely on certain *dependencies* (in this context, just another term for integrity constraints).

I'd like to elaborate briefly on the last of these points. I've said that constraints in general are highly relevant to the design process. It turns out, however, that the particular constraints we're talking about here—the so-called dependencies—enjoy certain formal properties that constraints in general don't (so far as we know). I can't get into this issue very deeply here; however, the basic point is that it's possible to define certain *inference rules* for such dependencies, and it's the existence of those inference rules that make it possible to develop the design theory that I'm going to describe.

To repeat, I assume you already know something about these matters. As noted in the previous section, however, I want to focus on aspects of the subject that you might not be so familiar with; I want to highlight the more important parts and downplay the others, and more generally I want to look at the whole subject from a perspective that might be a little different from what you're used to.

Functional Dependencies and Boyce/Codd Normal Form

It's well known that the notions of *second normal form* (2NF), *third normal form* (3NF), and *Boyce/Codd normal form* (BCNF) all depend on the notion of *functional dependency*. Here's a precise definition:

> *Definition:* Let A and B be subsets of the heading of relvar R. Then relvar R satisfies the *functional dependency* (FD) $A \rightarrow B$ if and only if, in every relation that's a legal value for R, whenever two tuples have the same value for A, they also have the same value for B.

The FD $A \rightarrow B$ is read as "B is functionally dependent on A," or "A functionally determines B," or, more simply, just "A arrow B."

By way of example, suppose there's an integrity constraint to the effect that if two suppliers are in the same city, then they must have the same status (see Figure 7-2, where I've changed the status for supplier S2 from 10 to 30 in order to conform to this hypothetical new constraint). Then the FD:

```
{ CITY } → { STATUS }
```

is satisfied by this revised form—let's call it RS—of the suppliers relvar S. Note the braces, by the way; I use braces to stress the point that both sides of the FD are *sets* of attributes, even when (as in the example) the sets in question involve just a single attribute.

RS

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

← *note the change*

FIGURE 7-2. *Revised suppliers relvar RS—sample value*

As the example indicates, the fact that some relvar *R* satisfies a given FD constitutes a database constraint in the sense of the previous chapter; more precisely, it constitutes a (single-relvar) constraint on that relvar *R*. For instance, the FD in the example is equivalent to the following **Tutorial D** constraint:

```
CONSTRAINT RSC COUNT ( RS { CITY } ) =
               COUNT ( RS { CITY, STATUS } ) ;
```

By the way, here's a useful thing to remember: if relvar *R* satisfies the FD $A \rightarrow B$, it necessarily satisfies the FD $A' \rightarrow B'$ for all supersets A' of A and all subsets B' of B. In other words, you can always add attributes to the left side or subtract them from the right side, and what you get will still be a valid FD.

At this point I need to introduce a couple of terms. The first is *superkey*. Basically, a superkey is a superset of a key (not necessarily a proper superset, of course); equivalently (with reference to the formal definition of *key* from Chapter 4), a subset *SK* of the heading of relvar *R* is a superkey for *R* if and only if it possesses the uniqueness property but not necessarily the irreducibility property. Thus, every key is a superkey, but most superkeys aren't keys; for example, {SNO,CITY} is a superkey for relvar S but not a key. Observe in particular that the heading of relvar *R* is always a superkey for *R*.

NOTE

Important: If *SK* is a superkey for *R* and *A* is any subset of the heading of *R*, *R* necessarily satisfies the FD $SK \rightarrow A$—because if two tuples of *R* have the same value for *SK*, then by definition they're the very same tuple, and so they *obviously* have the same value for *A*. (I did touch on this point in Chapter 4, but there I talked in terms of keys, not superkeys.)

The other new term is *trivial FD*. Basically, an FD is *trivial* if there's no way it can possibly be violated. For example, the following FDs are all trivially satisfied by any relvar that includes attributes called SNO, STATUS, and CITY:

```
{ CITY, STATUS } → { CITY }
{ SNO, CITY }    → { CITY }
{ CITY }         → { CITY }
{ SNO }          → { SNO }
```

In the first case, for instance, if two tuples have the same value for CITY and STATUS, they certainly have the same value for CITY. In fact, an FD is trivial if and only if the left side is a superset of the right side (again, not necessarily a proper superset). Of course, we don't usually think about trivial FDs when we're doing database design, because they're, well, trivial; but when we're trying to be formal and precise about these matters, we need to take *all* FDs into account, trivial ones as well as nontrivial.

Having pinned down the notion of FD precisely, I can now say that Boyce/Codd normal form (BCNF) is *the* normal form with respect to FDs—and, of course, I can also define it precisely:

> *Definition:* Relvar R is in BCNF if and only if, for every nontrivial FD $A \rightarrow B$ satisfied by R, A is a superkey for R.

In other words, in a BCNF relvar, the only FDs are either trivial ones (we can't get rid of those, obviously) or "arrows out of superkeys" (we can't get rid of those, either). Or as some people like to say: *Every fact is a fact about the key, the whole key, and nothing but the key*—though I must immediately add that this informal characterization, intuitively pleasing though it is, isn't really accurate, because it assumes among other things that there's just one key.

I need to elaborate slightly on the previous paragraph. When I talk about "getting rid of" some FD, I fear I'm being a little sloppy once again. For example, the revised suppliers relvar RS of Figure 7-2 satisfies the FD {SNO} → {STATUS}; but if we decompose it—as we're going to do in a moment—into relvars SNC and CS (where SNC has attributes SNO, SNAME, and CITY, and CS has attributes CITY and STATUS), that FD "disappears," in a sense, and thus we have indeed "gotten rid of it." But what does it mean to say the FD has disappeared? What's happened is that it's become a multi-relvar constraint (that is, a constraint that involves two or more relvars).* So the constraint certainly still exists—it just isn't an FD any more. Similar remarks apply to all of my uses of the phrase "get rid of" in this chapter.

Finally, as I assume you know, the normalization discipline says that if relvar R is not in BCNF, it should be decomposed into smaller ones that are (where "smaller" means, basically, having fewer attributes). For example:

- Relvar STP (see Figure 7-1) satisfies the FD {SNO} → {STATUS}, which is neither trivial nor "an arrow out of a superkey"—{SNO} isn't a superkey for STP—and the relvar is thus not in BCNF (and of course it suffers from redundancy, as we saw earlier). So we decompose it into relvars SP and SS, say, where SP has attributes SNO, PNO, and QTY (as usual) and SS has attributes SNO and STATUS. (As an exercise, show sample values for relvars SP and SS corresponding to the STP value in Figure 7-1; convince yourself that SP and SS are in BCNF and that the decomposition eliminates the redundancy.)
- Similarly, relvar RS (see Figure 7-2) satisfies the FD {CITY} → {STATUS} and should therefore be decomposed into, say, SNC (with attributes SNO, SNAME, and CITY) and CS (with attributes CITY and STATUS). (As an exercise, show sample values for SNC and CS corresponding to the RS value in Figure 7-2; convince yourself that SNC and CS are in BCNF and that the decomposition eliminates the redundancy.)

* It's also, in this particular example, an *implied* constraint (as it was before, in fact, since it was implied by the constraint KEY{SNO} for RS). To be specific, it's implied by the combination of constraints KEY{SNO} (for SNC), KEY{CITY} (for CS), and the foreign key constraint from SNC to CS.

Nonloss Decomposition

We know that if some relvar isn't in BCNF, it should be decomposed into smaller ones that are. Of course, it's important that the decomposition be *nonloss* (also called *lossless*): we must be able to get back to where we came from—the decomposition mustn't lose any information. Consider relvar RS once again (see Figure 7-2), with its FD {CITY} → {STATUS}. Suppose we were to decompose that relvar, not as before into relvars SNC and CS, but instead into relvars *SNS* and CS as illustrated in Figure 7-3. (Relvar CS is the same in both decompositions, but SNS has attributes SNO, SNAME, and STATUS instead of SNO, SNAME, and CITY.) Then I hope it's clear that (a) SNS and CS are both in BCNF, but (b) the decomposition is not nonloss but "lossy"—for example, we can't tell whether supplier S2 is in Paris or Athens, and so we've lost information.

SNS

SNO	SNAME	STATUS
S1	Smith	20
S2	Jones	30
S3	Blake	30
S4	Clark	20
S5	Adams	30

CS

CITY	STATUS
London	20
Paris	30
Athens	30

FIGURE 7-3. *Relvars SNS and CS—sample values*

What exactly is it that makes some decompositions nonloss and others lossy? Well, note that the decomposition process is, formally, *a process of taking projections;* all of the "smaller" relvars in all of our examples so far have been projections of the original relvar. In other words, the decomposition operator is, precisely, the projection operator of relational algebra.

> **NOTE**
>
> I'm being sloppy again. Like all of the algebraic operators, projection really applies to relations, not relvars. But we often say things like *relvar CS is a projection of relvar RS* when what we really mean is *the relation that's the value of relvar CS at any given time is a projection of the relation that's the value of relvar RS at that time.* I hope that's clear!

Onward. When we say a certain decomposition is nonloss, what we really mean is that *if we join the projections together again, we get back to the original relvar.* Observe in particular that, with reference to Figure 7-3, relvar RS is *not* equal to the join of its projections SNS and CS, and that's why the decomposition is lossy. With reference to Figure 7-2, by contrast, it *is* equal to the join of its projections SNC and CS; that decomposition is indeed nonloss.

To say it again, then, the decomposition operator is projection and the recomposition operator is join. And the formal question that lies at the heart of normalization theory is this:

> Let *R* be a relvar and let *R1, R2, ..., Rn* be projections of *R*. What conditions must be satisfied in order for *R* to be equal to the join of those projections?

An important, though partial, answer to this question was provided by Ian Heath in 1971 when he proved the following theorem:

> Let *A*, *B*, and *C* be subsets of the heading of relvar *R* such that the (set-theoretic) union of *A*, *B*, and *C* is equal to that heading. Let *AB* denote the (set-theoretic) union of *A* and *B*, and similarly for *AC*. If *R* satisfies the FD $A \rightarrow B$, then *R* is equal to the join of its projections on *AB* and *AC*.

By way of example, consider relvar RS once again (Figure 7-2). That relvar satisfies the FD {CITY} → {STATUS}. Thus, taking *A* as {CITY}, *B* as {STATUS}, and *C* as {SNO,SNAME}, Heath's theorem tells us that RS can be nonloss-decomposed into its projections on {CITY,STATUS} and {CITY,SNO,SNAME}—as indeed we already know.

NOTE

In case you're wondering why I said Heath's theorem provides only a partial answer to the original question, let me explain in terms of the foregoing example. Basically, the theorem does tell us that the decomposition of Figure 7-2 is nonloss; however, it doesn't tell us that the decomposition of Figure 7-3 is lossy. That is, it gives a *sufficient* condition, but not a *necessary* one, for a decomposition to be nonloss. (A stronger form of Heath's theorem, giving both necessary and sufficient conditions, was proved by Ron Fagin in 1977, but the details are beyond the scope of the present discussion. See Exercise 7.18 at the end of the chapter.)

As an aside, I remark that in the paper in which he proved his theorem, Heath also gave a definition of what he called "third" normal form that was in fact a definition of BCNF. Since that definition predated Boyce and Codd's own definition by some three years, it seems to me that BCNF ought by rights to be called *Heath* normal form. But it isn't.

One last point: it follows from the discussions of this subsection that the constraint I showed earlier for relvar RS:

```
CONSTRAINT RSC COUNT ( RS { CITY } ) =
               COUNT ( RS { CITY, STATUS } ) ;
```

could alternatively be expressed thus:

```
CONSTRAINT RSC
   RS = JOIN { RS { SNO, SNAME, CITY }, RS { CITY, STATUS } } ;
```

("At all times, relvar RS is equal to the join of its projections on {SNO,SNAME,CITY} and {CITY,STATUS}"; I'm using the prefix version of JOIN here.)

But Isn't It All Just Common Sense?

I noted earlier that normalization theory has been criticized on the grounds that it's all basically just common sense. Consider relvar STP again, for example (see Figure 7-1).

That relvar is *obviously* badly designed; the redundancies are obvious, the consequences are obvious too, and any competent human designer would "naturally" decompose that relvar into its projections SP and SS as previously discussed, even if that designer had no knowledge of BCNF whatsoever. But what does "naturally" mean here? What *principles* is the designer applying in opting for that "natural" design?

The answer is: they're exactly the principles of normalization. That is, competent designers already have those principles in their brain, as it were, even if they've never studied them formally and can't put a name to them. So yes, the principles *are* common sense—but they're *formalized* common sense. (Common sense might be common, but it's not so easy to say exactly what it is!) What normalization theory does is state in a precise way what certain aspects of common sense consist of. In my opinion, that's the real achievement of normalization theory: it formalizes certain commonsense principles, thereby opening the door to the possibility of mechanizing those principles (that is, incorporating them into mechanical design tools). Critics of normalization usually miss this point; they claim, quite rightly, that the ideas are really just common sense, but they typically don't realize that it's a significant achievement to state what common sense means in a precise and formal way.

1NF, 2NF, 3NF

Normal forms below BCNF are mostly of historical interest; as noted in the introductory section, in fact, I don't even want to bother to give the definitions here. I'll just remind you that *all* relvars are at least in 1NF,* even ones with relation-valued attributes (RVAs). From a design point of view, however, relvars with RVAs are usually—though not invariably—contraindicated. Of course, this doesn't mean you should never have RVAs (in particular, there's no problem with query results that include RVAs); it just means we don't usually want RVAs "designed into the database," as it were (and we can always eliminate them, thanks to the availability of the UNGROUP operator of relational algebra). I don't want to get into a lot of detail on this issue here; let me just say that relvars with RVAs look very like the hierarchies found in older, nonrelational systems like IMS, and all of the old problems that used to arise with hierarchies therefore raise their head again. Here for purposes of reference is a brief list of some of those problems:

- The fundamental problem is that hierarchies are asymmetric. Thus, though they might make some tasks "easy," they certainly make others difficult. (See Exercises 5-27, 5-29, and 5-30 at the end of Chapter 5 for some illustrations of this point.)
- Queries are therefore asymmetric too, as well as being more complicated than their symmetric counterparts.
- The same goes for integrity constraints.
- The same goes for updates, but more so.

* In earlier chapters I said it was *relations*, not relvars, that were always in 1NF, but no harm is done if we extend the term to apply to relvars as well.

- There's no guidance as to how to choose the "best" hierarchy.
- Even "natural" hierarchies like organization charts are still best represented, usually, by nonhierarchic designs.

To repeat, however, RVAs can occasionally be OK, even in base relvars. See Exercise 7-14 at the end of the chapter.

Join Dependencies and Fifth Normal Form

Fifth normal form (5NF) is—in a certain special sense which I'll explain later in this section—"the final normal form." In fact, just as BCNF is *the* normal form with respect to functional dependencies, so fifth normal form is *the* normal form with respect to what are called *join* dependencies:

> *Definition:* Let *A, B, ..., Z* be subsets of the heading of relvar *R*. Then *R* satisfies the *join dependency* (JD)
>
> ```
> ⋆ { A, B, ..., Z }
> ```
>
> if and only if every relation that's a legal value for *R* is equal to the join of its projections on *A, B, ..., Z*.

The JD ⋆{*A,B,...,Z*} is read as "star *A, B, ..., Z*." Points arising from this definition:

- It's immediate that *R* can be nonloss-decomposed into its projections on *A, B, ..., Z* if and only if it satisfies the JD ⋆{*A,B,...,Z*}.
- It's also immediate that every FD is a JD because (as we know from the previous section) if *R* satisfies a certain FD, then it can be nonloss-decomposed into certain projections (in other words, it satisfies a certain JD).

As an example of this latter point, consider relvar RS once again (Figure 7-2). That relvar satisfies the FD {CITY} → {STATUS} and can therefore be nonloss-decomposed into its projections SNC (on SNO, SNAME, and CITY) and CS (on CITY and STATUS). It follows that relvar RS satisfies the JD ⋆{SNC,CS}—if you'll allow me to use the names SNC and CS, just for the moment, to refer to the applicable subsets of the heading of that relvar as well as to the projections as such.

Now, we saw in the previous section that there are always "arrows out of superkeys"; that is, certain functional dependencies are implied by superkeys, and we can never get rid of them. More generally, in fact, certain *join* dependencies are implied by superkeys, and we can never get rid of those, either. To be specific, the JD ⋆{*A,B,...,Z*} is *implied by superkeys* if and only if each of *A, B, ..., Z* is a superkey for the pertinent relvar *R*. For example, consider our usual suppliers relvar S. The fact that {SNO} is a superkey (actually a key) for that relvar implies among other things that the relvar satisfies this JD:

```
⋆ { SN, SS, SC }
```

where SN is {SNO,SNAME}, SS is {SNO,STATUS}, and SC is {SNO,CITY} (note that each of these is a superkey for S). And it's certainly true that we could nonloss-decompose S, if we wanted to, into its projections on SN, SS, and SC—though whether we would actually want to is another matter.

We also saw in the previous section that certain FDs are trivial. As you're probably expecting by now, certain JDs are trivial too. To be specific, the JD $\star\{A,B,\ldots,Z\}$ is *trivial* if and only if at least one of A, B, ..., Z is equal to the entire heading of the pertinent relvar R. For example, here's one of the many trivial JDs that relvar S satisfies:

```
★ { S, SN, SS, SC }
```

I'm using the name S here, just for the moment, to refer to the set of all attributes—the heading—of relvar S (corresponding, of course, to the *identity* projection of the relvar S). I hope it's obvious that any relvar can always be nonloss-decomposed into a given set of projections if one of the projections in that set is the pertinent identity projection. (Though it's a bit of a stretch to talk about "decomposition" in such a situation, because one of the projections in that "decomposition" is identical to the original relvar; I mean, there's not much "decomposing" going on here!)

Having pinned down the notion of JD precisely, I can now give a precise definition of 5NF:

> *Definition:* Relvar R is in 5NF if and only if every nontrivial JD satisfied by R is implied by the superkeys of R.

In other words, the only JDs satisfied by a 5NF relvar are ones we can't get rid of; it's if a relvar satisfies any *other* JDs that it's not in 5NF (and therefore suffers from redundancy problems), and so probably needs to be decomposed.

The Significance of 5NF

I'm sure you noticed that I didn't show an example in the foregoing discussion of a relvar that was in BCNF but not in 5NF (and so could be nonloss-decomposed to adavantage). The reason I didn't is this: while JDs that aren't just simple FDs do exist, (a) those JDs tend to be unusual in practice, and (b) they also tend to be a little complicated, more or less by definition. Because they're complicated, I decided not to give an example right away (I'll give one in the next subsection, however); because they're unusual, they aren't so important anyway from a practical point of view. Let me elaborate.

First of all, if you're a database designer, you certainly do need to know about JDs and 5NF; they're tools in your toolkit, as it were, and (other things being equal) you should generally try to ensure that all of the relvars in your database are in 5NF. But most relvars (not all) that occur in practice, if they're at least in BCNF, are in 5NF as well; that is, it's quite rare in practice to find a relvar that's in BCNF and not also in 5NF. Indeed, there's a theorem that addresses this issue:

> Let R be a BCNF relvar and let R have no composite keys (that is, no keys consisting of two or more attributes). Then R is in 5NF.

This theorem is quite useful. What it says is that if you can get to BCNF (which is easy enough) and there aren't any composite keys in your BCNF relvar (which is often but not always the case), you don't have to worry about the complexities of general JDs and 5NF—you know without having to think about the matter any further that the relvar simply *is* in 5NF.

As an aside, I remark in the interests of accuracy that the foregoing theorem actually applies to 3NF, not BCNF; that is, it really says a *3NF* relvar with no composite keys is necessarily in 5NF. But every BCNF relvar is in 3NF, and in any case BCNF is much more important than 3NF, pragmatically speaking.

So 5NF as a concept is perhaps not all that important from a practical point of view. But it's *very* important from a theoretical one, because (as I said at the beginning of this section) it's "the final normal form" and—what amounts to the same thing—it's *the* normal form with respect to general join dependencies. For if relvar *R* is in 5NF, the only nontrivial JDs are ones implied by superkeys. Hence, the only nonloss decompositions are ones in which every projection is on the attributes of some superkey; in other words, every such projection includes some key of *R*. As a consequence, the corresponding "recomposition" joins are all one-to-one, and no redundancies are or can be eliminated by the decomposition.

Let me put this point another way. To say that relvar *R* is in 5NF is to say that further nonloss decomposition of *R* into projections, while it might be possible, certainly won't eliminate any redundancies. *Note very carefully, however, that to say that R is in 5NF is not to say that R is redundancy-free.* There are many kinds of redundancy that projection as such is powerless to remove—which is an illustration of the point I made earlier, in the section "The Place of Design Theory," to the effect that there are numerous issues that design theory simply doesn't address. By way of example, consider Figure 7-4, which shows a sample value for a relvar, SPJ, that's in 5NF and yet suffers from redundancy. For example, the fact that supplier S2 supplies part P3 appears several times; so does the fact that part P3 is supplied to project J4—JNO stands for *project number*—and so does the fact that project J1 is supplied by supplier S2. (The relvar predicate is *Supplier SNO supplies part PNO to project JNO in quantity QTY*, and the sole key is {SNO,PNO,JNO}.) The only nontrivial join dependency satisfied by this relvar is this *functional* dependency:*

```
{ SNO, PNO, JNO } → { QTY }
```

which is an "arrow out of a superkey." In other words, QTY depends on all three of SNO, PNO, and JNO, and it can't appear in a relvar with anything less than all three. Hence, there's no nonloss decomposition that can remove the redundancies.

There are a few further points I need to make here. First, I didn't mention the point previously, but you probably know that 5NF is always achievable; that is, it's always possible to decompose a non-5NF relvar into 5NF projections.

* Equivalently, using the name SPJQ to refer to the set of all attributes of SPJ, the relvar satisfies the trivial JD ★{SPJQ}. (Recall from Chapter 5 that the join of a single relation *r* is just *r* itself.)

SPJ

SNO	PNO	JNO	QTY
S1	P1	J1	200
S1	P3	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	J1	100

FIGURE 7-4. *The 5NF relvar SPJ—sample value*

Second, every 5NF relvar is in BCNF, of course; so to say that *R* is in BCNF certainly doesn't preclude the possibility that *R* is in 5NF as well. Informally, however, it's very common to interpret statements to the effect that *R* is in BCNF as meaning that *R* is in BCNF *and not in any higher normal form.* I have *not* followed this practice in this chapter (and will continue not to do so).

Third, because it's "the final normal form," 5NF is sometimes called *projection/join* normal form (PJ/NF), to stress the point that it's *the* normal form so long as we limit ourselves to projection as the decomposition operator and join as the recomposition operator. But I should immediately add that it's possible to consider other operators and therefore, possibly, other normal forms. In particular, it's possible, and desirable, to define (a) generalized versions of the projection and join operators, and hence (b) a generalized form of join dependency, and hence (c) a new "sixth" normal form, 6NF. It turns out that these developments are particularly important in connection with support for temporal data, and they're discussed in detail in the book *Temporal Data and the Relational Model* by Hugh Darwen, Nikos Lorentzos, and myself (Morgan Kaufmann, 2003). However, all I want to do here is give a definition of 6NF that works for "regular" (that is, nontemporal) relvars. Here it is:

Definition: Relvar *R* is in 6NF if and only if it satisfies no nontrivial JDs at all.

Equivalently, a "regular" relvar is in 6NF if and only if it consists of a single key, plus at most one additional attribute. Our usual shipments relvar SP is in 6NF, as is relvar SPJ (see Figure 7-4); by contrast, our usual suppliers and parts relvars S and P are in 5NF but not 6NF.

NOTE

A 6NF relvar is sometimes said to be *irreducible,* because it can't be nonloss-decomposed via projection at all. Any 6NF relvar is necessarily in 5NF.

To close this subsection, observe that it follows from all of the above that any relvar that's "all key" or consists of a key plus one additional attribute, since it's in 6NF, is certainly in BCNF. However, it does *not* follow that such relvars are always well designed! For example, if relvar RS (see Figure 7-2) satisfies the FD {CITY} → {STATUS}, the projection

of RS on {SNO,STATUS} is in BCNF—in fact, it's in 6NF—but it certainly isn't well designed. (See the discussion of *dependency preservation* in the section "Two Cheers for Normalization," later, for a more detailed explanation.)

More on 5NF

Consider Figure 7-5, which shows a sample value for a simplified version of relvar SPJ from the previous subsection. Suppose that simplified version satisfies the join dependency ⋆{SP,PJ,SJ}, where SP, PJ, and SJ stand for {SNO,PNO}, {PNO,JNO}, and {SNO,JNO}, respectively. What does that JD mean from an intuitive point of view? The answer is as follows:

SPJ

SNO	PNO	JNO
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

FIGURE 7-5. *Simplified relvar SPJ—sample value*

- The JD means the relvar is equal to the join of, and so can be nonloss-decomposed into, its projections SP, PJ, and SJ. (Note that now I'm using the names SP, PJ, and SJ to refer to the projections as such, instead of to the corresponding subsets of the heading of relvar SPJ; I hope this kind of punning on my part doesn't confuse you.)
- It follows that the following constraint is satisfied:

  IF <*s*,*p*> ∈ SP AND <*p*,*j*> ∈ PJ AND <*s*,*j*> ∈ SJ THEN <*s*,*p*,*j*> ∈ SPJ

  because if <*s*,*p*>, <*p*,*j*>, and <*s*,*j*> appear in SP, PJ, and SJ, respectively, then <*s*,*p*,*j*> certainly appears in the join of SP, PJ, and SJ, and that join is supposed to be equal to SPJ (that's what the JD says). Given the sample value of Figure 7-5, for example, the tuples <S1,P1>, <P1,J1>, and <S1,J1> appear in SP, PJ, and SJ, respectively, and the tuple <S1,P1,J1> appears in SPJ. (I'm using what I hope is a self-explanatory shorthand notation for tuples, and I remind you that the symbol "∈" can be read as "appears in.")
- Now, the tuple <*s*,*p*> obviously appears in SP if and only if the tuple <*s*,*p*,*z*> appears in SPJ for some *z*. Likewise, the tuple <*p*,*j*> appears in PJ if and only if the tuple <*x*,*p*,*j*> appears in SPJ for some *x*, and the tuple <*s*,*j*> appears in SJ if and only if the tuple <*s*,*y*,*j*> appears in SPJ for some *y*. So the foregoing constraint is logically equivalent to this one:

  IF for some *x*, *y*, *z* <*s*,*p*,*z*> ∈ SPJ AND
  <*x*,*p*,*j*> ∈ SPJ AND
  <*s*,*y*,*j*> ∈ SPJ
  THEN <*s*,*p*,*j*> ∈ SPJ

  With reference to Figure 7-5, for example, the tuples <S1,P1,J2>, <S2,P1,J1>, and <S1,P2,J1> all appear in SPJ, and therefore so does the tuple <S1,P1,J1>.

So the original JD is equivalent to the foregoing constraint. But what does that constraint mean in real-world terms? Well, here's a concrete illustration. Suppose relvar SPJ contains tuples that tell us that all three of the following are true propositions:

1. Smith supplies monkey wrenches to some project.
2. Somebody supplies monkey wrenches to the Manhattan project.
3. Something is supplied to the Manhattan project by Smith.

Then the JD says the relvar must contain a tuple that tells us that the following is a true proposition too:

4. Smith supplies monkey wrenches to the Manhattan project.

Now, propositions 1, 2, and 3 together would normally *not* imply proposition 4. If we know only that propositions 1, 2, and 3 are true, then we know that Smith supplies monkey wrenches to *some* project (say, project *z*), that *some* supplier (say, supplier *x*) supplies monkey wrenches to the Manhattan project, and that Smith supplies *some* part (say, part *y*) to the Manhattan project—but we cannot validly infer that *x* is Smith or *y* is monkey wrenches or *z* is the Manhattan project. False inferences such as this one are examples of what's sometimes called *the connection trap*. In the case at hand, however, the existence of the JD tells us *there is no trap*; that is, we *can* validly infer proposition 4 from propositions 1, 2, and 3 in this particular case.

Observe now the *cyclic nature* of the constraint ("IF *s* is connected to *p* and *p* is connected to *j* and *j* is connected back to *s* again, THEN *s* and *p* and *j* must all be directly connected, in the sense that they must all appear together in the same tuple"). It's precisely if such a cyclic constraint occurs that we might have a relvar that's in BCNF and not in 5NF. In my experience, however, such cyclic constraints are very rare in practice—which is why I said in the previous subsection that I don't think they're very important from a practical point of view.

I'll close this section with a brief remark on *fourth* normal form (4NF). In the subsection "The Significance of 5NF," I said that if you're a database designer, you need to know about JDs and 5NF. In fact, you also need to know about *multi-valued* dependencies (MVDs) and fourth normal form. However, I mention these concepts for completeness only; like 2NF and 3NF, they're mainly of historical interest. I'll just note for the record that:

- An MVD is a JD that involves no more than two projections (in practice, usually exactly two).
- A relvar is in 4NF if and only if every nontrivial MVD it satisfies is implied by some superkey.

Details of what it means for an MVD to be trivial or implied by some superkey are beyond the scope of this discussion (see Exercise 7-19 at the end of the chapter)—but let me at least point out that it follows from these definitions that repeated nonloss-decomposition

into exactly two projections is sufficient to take us at least as far as 4NF. By contrast, the JD in the previous subsection involved three projections, as I'm sure you noticed. In fact, we can say that in order to reach 5NF, decomposition into n projections (where $n > 2$) is necessary only if the relvar in question satisfies an n-way cyclic constraint: equivalently, only if it satisfies a JD involving n projections and not one involving fewer.

Two Cheers for Normalization

Normalization is far from being a panacea, as we can easily see by considering what its goals are and how well it measures up against them. Here are those goals:

- To achieve a design that's a "good" representation of the real world—one that's intuitively easy to understand and is a good basis for future growth
- To reduce redundancy
- Thereby to avoid certain update anomalies
- To simplify the statement and enforcement of certain integrity constraints

I'll consider each in turn.

Good representation of the real world: Normalization does well on this one. I have no criticisms here.

Reduce redundancy: Normalization is a good start on this problem, but it's only a start. For one thing, it's a process of nonloss decomposition, and (as we've seen) not all redundancies can be removed by nonloss decomposition; indeed, there are some kinds of redundancy, not discussed in this chapter so far, that normalization simply doesn't address at all. For another thing, the objective of reducing redundancy can conflict with another objective, also not previously discussed—namely, the objective of *dependency preservation*. Let me explain. Consider the following relvar (attribute ZIP denotes ZIP Code or postcode):

```
ADDR { STREET, CITY, STATE, ZIP }
```

Assume for the sake of the example that this relvar satisfies the following FDs:

```
{ STREET, CITY, STATE } → { ZIP }
{ ZIP }                 → { CITY, STATE }
```

The second of these FDs implies that the relvar isn't in BCNF. But if we apply Heath's theorem and decompose it into BCNF projections as follows:

```
ZCS { ZIP, CITY, STATE }
    KEY { ZIP }

ZS { ZIP, STREET }
   KEY { ZIP, STREET }
```

then the FD {STREET,CITY,STATE} → {ZIP}, which was certainly satisfied by the original relvar, "disappears"! (It's satisfied by the join of ZCS and ZS but, obviously enough, not by either of those projections alone.) As a consequence, relvars ZCS and ZS can't be independently updated. For example, suppose those projections currently have values as shown in Figure 7-6; then an attempt to insert the tuple <10111,Broadway> into ZS will violate the "missing" FD. However, this fact can't be determined without examining the projection ZCS as well as the projection ZS. For precisely this kind of reason, the dependency preservation objective says: *Don't split dependencies across projections.* However, the ADDR example shows that, sadly, this objective and the objective of decomposing into BCNF projections can sometimes be in conflict.

ZCS

ZIP	CITY	STATE
10003	New York	NY
10111	New York	NY

ZS

ZIP	STREET
10003	Broadway

FIGURE 7-6. *Projections ZCS and ZS—sample values*

Avoid update anomalies: This point is effectively just the previous one ("reduce redundancy") by another name. It's well known that less than fully normalized designs can be subject to certain update anomalies, precisely because of the redundancies they entail. In relvar STP, for example (see Figure 7-1 once again), supplier S1 might be shown as having status 20 in one tuple and status 25 in another. (Of course, this "update anomaly" can arise only if a less than perfect job is being done on integrity. Perhaps a better way to characterize the update anomaly issue is this: The constraints needed to prevent such anomalies are easier to state, and might be easier to enforce, if the design is fully normalized than they would be if it isn't. See the next paragraph.)

Simplify statement and enforcement of constraints: It's clear as a general observation that some constraints imply others. As a trivial example, if quantities must be less than or equal to 5000, they must certainly be less than or equal to 6000 (speaking a little loosely). Now, if constraint *A* implies constraint *B*, then stating and enforcing *A* will effectively state and enforce *B* "automatically" (indeed, *B* won't actually need to be stated at all). And normalization to 5NF gives a very simple way of stating and enforcing certain important constraints: basically, all we have to do is define keys and enforce their uniqueness—which we're going to do anyway—and then all JDs (and all MVDs and all FDs) will effectively be stated and enforced automatically, because they'll all be implied by those keys. So normalization does a pretty good job in this area, too.

On the other hand...here are some more reasons why normalization is no panacea. First, JDs aren't the only kind of constraint, and normalization doesn't help with any others. Second, given a particular set of relvars, there'll often be several possible decompositions into 5NF projections, and there's little or no formal guidance available to tell us which one to choose in such cases. Third, there are many design issues that normalization simply doesn't address at all. For example, what is it that tells us there should be just one

suppliers relvar instead of one for London suppliers, one for Paris suppliers, and so on? It certainly isn't normalization as classically understood.

That said, I must make it clear that I don't want my comments in this section to be seen as an attack. I believe firmly that anything less than a fully normalized design is *strongly contraindicated.* In fact, I want to close this section with an argument—a *logical* argument, and one you might not have seen before—in support of the position that you should *"denormalize" only as a last resort.* That is, you should back off from a fully normalized design only if all other strategies for improving performance have somehow failed to meet requirements. By the way, note that I'm going along here with the usual assumption that normalization has performance implications. So it does, in current SQL products; but this is another topic I want to come back to later (see the section "Some Remarks on Physical Design"). Anyway, here's the argument.

We all know that denormalization is bad for update (logically bad, I mean; it makes updates harder to formulate, and it can jeopardize the integrity of the database as well). What doesn't seem to be so widely known is that denormalization can be bad for retrieval too; that is, it can make certain queries harder to formulate (equivalently, it can make them easier to formulate incorrectly—meaning, if they run, that you're getting answers that might be "correct" in themselves but are answers to the wrong questions). Let me illustrate. Take another look at relvar RS (Figure 7-2), with its FD {CITY} → {STATUS}, and consider the query "Get the average city status." Given the sample values in Figure 7-2, the status values for Athens, London, and Paris are 30, 20, and 30, respectively, and so the average is 26.667 (to three decimal places). Here are some attempts at formulating this query in SQL:

```
1  SELECT AVG ( RS.STATUS ) AS RESULT
   FROM  RS
```

Result (incorrect): 26. The problem here is that London's status and Paris's status have both been counted twice. Perhaps we need a DISTINCT inside the AVG invocation? Let's try that:

```
2  SELECT AVG ( DISTINCT RS.STATUS ) AS RESULT
   FROM  RS
```

Result (incorrect): 25. No, it's distinct *cities* we need to examine, not distinct status values. We can do that by grouping:

```
3  SELECT RS.CITY, AVG ( RS.STATUS ) AS RESULT
   FROM   RS
   GROUP  BY RS.CITY
```

Result (incorrect): <Athens,30>, <London,20>, <Paris,30>. This formulation gives average status *per city,* not the overall average. Perhaps what we want is the average of the averages?

```
4  SELECT RS.CITY, AVG ( AVG ( RS.STATUS ) ) AS RESULT
   FROM   RS
   GROUP  BY RS.CITY
```

Result: Syntax error. The SQL standard quite rightly* doesn't allow invocations of what it calls "set functions" such as AVG to be nested in this manner. One more attempt:

```
5  SELECT AVG ( TEMP.STATUS ) AS RESULT
   FROM ( SELECT DISTINCT RS.CITY, RS.STATUS
          FROM   RS ) AS TEMP
```

Result (correct at last): 26.667. As I pointed out in Chapter 5, however, not all SQL products allow nested subqueries to appear in the FROM clause in this manner.

That's the end of normalization (for the time being, at any rate); now I want to switch to a topic that's almost certainly less familiar to you, *orthogonality*, which constitutes another little piece of science in this overall business of database design.

Orthogonality

Figure 7-7 shows sample values for a possible but clearly bad design for suppliers: relvar SA is suppliers in Paris, and relvar SB is suppliers who either aren't in Paris or have status 30. As you can see, the design leads to redundancy—to be specific, the tuple for supplier S3 appears in both relvars—and as usual such redundancies give rise to update anomalies. (Redundancy of any kind always has the potential to cause update anomalies.)

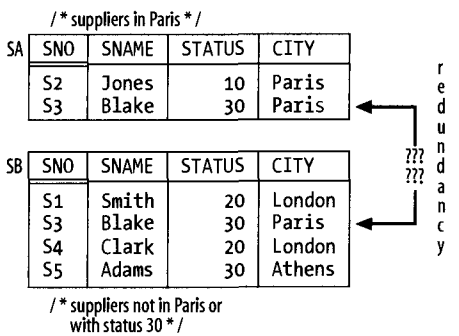


FIGURE 7-7. *Relvars SA and SB—sample values*

* I say "quite rightly" only because we're in the SQL context specifically; a more orthodox language would certainly let us nest such invocations. Let me explain. Consider the SQL expression SELECT SUM(SP.QTY) AS SQ FROM SP WHERE SP.QTY > 100 (I deliberately switch to a different example). The SUM argument here is really SP.QTY FROM SP WHERE SP.QTY > 100, and a more orthodox language would therefore enclose that whole expression in parentheses. But SQL doesn't. As a consequence, an expression of the form AVG(SUM(QTY)) has to be illegal, because SQL can't figure out which portions of the rest of the surrounding expression are part of the AVG argument and which are part of the SUM argument.

By the way, note that the tuple for supplier S3 *must* appear in both relvars. For suppose it appeared in SB but not SA, say. From SA, then, the Closed World Assumption would allow us to infer that it's not the case that supplier S3 is in Paris. But SB tells us that supplier S3 *is* in Paris. Thus, we would have a contradiction on our hands, and the database would be inconsistent.

Well, the problem with the design of Figure 7-7 is obvious: it's precisely the fact that the very same tuple can appear in two distinct relvars—meaning, more precisely, that it's possible for that tuple to appear in both relvars without violating any constraint for either. So an obvious rule is:

> *The Principle of Orthogonal Design (first version):* No two distinct relvars in the same database should be such that their relvar constraints permit the same tuple to appear in both.

The term *orthogonal* here derives from the fact that what the principle effectively says is that relvars should all be independent of one another—which they won't be, if their constraints "overlap," as it were.

Now, it should be clear that two relvars can't possibly violate the foregoing principle if they're of different types, and so you might be thinking the principle isn't worth much. After all, it isn't very usual for a database to contain two or more relvars of the same type. But consider Figure 7-8, which shows another possible but clearly bad design for suppliers. While there's no way in that design for the same tuple to appear in both relvars, it *is* possible for a tuple in SX and a tuple in SY to have the same projection on {SNO,SNAME}*—and that fact leads to redundancy and update anomalies again. So we need to extend the design principle accordingly:

> *The Principle of Orthogonal Design (second and final version):* Let *A* and *B* be distinct relvars in the same database. Then there must not exist nonloss decompositions of *A* and *B* into (say) *A1, A2, ..., Am* and *B1, B2, ..., Bn*, respectively, such that the relvar constraints for some projection *Ai* in the set *A1, A2, ..., Am* and some projection *Bj* in the set *B1, B2, ..., Bn* permit the same tuple to appear in both.

The term *nonloss decomposition* here refers to nonloss decomposition in the usual normalization sense.

SX

SNO	SNAME	STATUS
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

SY

SNO	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Paris
S4	Clark	London
S5	Adams	Athens

FIGURE 7-8. *Relvars SX and SY—sample values*

* Projection is a relational operator, of course, but it clearly makes sense to define a version of the operator that works for tuples instead of relations and thereby to talk of a projection of some tuple. A similar remark applies to several other relational operators, too.

Several points arise from the foregoing discussion and definition:

- The second version of the principle subsumes the previous version, because one "nonloss decomposition" that's always available for any relvar *R* is the one that consists of just the identity projection (the projection of *R* on all of its attributes).
- Like the principles of normalization, *The Principle of Orthogonal Design* is basically just common sense—but (again like normalization) it's *formalized* common sense.
- The goal of orthogonal design is to reduce redundancy and thereby to avoid update anomalies (again like normalization). In fact, orthogonality complements normalization, in the sense that—loosely speaking—normalization reduces redundancy *within* relvars, while orthogonality reduces redundancy *across* relvars.
- In fact, orthogonality complements normalization in another way, too. Again consider the decomposition of relvar S into its projections SX and SY, as illustrated in Figure 7-8. I now observe that that decomposition satisfies *all* of the usual normalization principles! Both projections are in 5NF; the decomposition is nonloss; dependencies are preserved; and both projections are needed to reconstruct the original relvar S.* It's orthogonality, not normalization, that tells us the decomposition is bad.
- Orthogonality might be common sense, but it's often flouted in practice. That is, designs like this one, from a financial database, are encountered quite frequently:

```
ACTIVITIES_2001 { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2002 { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2003 { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2004 { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2005 { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL }
...
```

  A better design involves just a single relvar:

```
ACTIVITIES { ENTRYNO, DESCRIPTION, AMOUNT, NEW_BAL, YEAR }
```

NOTE

Of course, one reason why such multi-relvar designs occur in practice is because there are often good reasons for partitioning the data at the physical level, and the system in question is such that the partitioning therefore has to show through to the logical level as well. But reasons in favor of a certain physical design aren't good reasons in favor of a bad logical design.

* I said earlier that a decomposition is nonloss if, when we join the projections back together, we get back to the original relvar. That's true, but it's not quite enough; in practice, we surely want to impose the extra requirement that every projection is needed in that join. For example, we probably wouldn't consider the decomposition of relvar S into its projections on {SNO}, {SNO,STATUS}, and {SNO,SNAME,CITY} as a sensible design—yet S is certainly equal to the join of those three projections.

- If *A* and *B* are relvars of the same type, adherence to the orthogonal design principle implies that:

  `A INTERSECT B` : *is always empty*
  `A UNION B` : *is always a disjoint union*
  `A MINUS B` : *is always equal to A*

- Suppose we decide for some reason to decompose our usual suppliers relvar into a set of restrictions. Then orthogonality tells us those restrictions should be *pairwise disjoint,* in the sense that no tuple can ever appear in more than one of them. (Also, of course, the union of those restrictions must give us back the original relvar.) Such a decomposition is said to be an *orthogonal decomposition.*

Some Remarks on Physical Design

The relational model has nothing to say about physical design. But there are still some things that can usefully be said about physical design in a relational context—things that are at least implied by the model, even though they aren't stated explicitly (and even though the details of physical design are, of necessity, somewhat DBMS-specific and vary from system to system).

The first point is that *physical design should follow logical design.* That is, the "right" approach is to do a clean logical design first, and then, as a follow-on step, to map that logical design into whatever physical structures the target DBMS happens to support. Equivalently, the physical design should be derived from the logical design and not the other way around. Ideally, in fact, the system should be able to derive an optimal physical design for itself, without any need for human involvement at all. (This goal isn't as far-fetched as it might sound. I'll say a little more about it later in this section.)

As for my second point: we saw in Chapter 1 that one reason for excluding physical issues of all kinds from the relational model was to give implementers the freedom to implement the model in any way they liked—and here, I think, the widespread lack of understanding of the model has really hurt us. Certainly most SQL products have failed to live up to the model's full potential in this respect; in those products, what the user sees and what's physically stored are essentially the same thing. In other words, what's physically stored is effectively just a *direct image* of what the user logically sees, as Figure 7-9 suggests. (I realize these remarks are oversimplified, but they're true enough for present purposes.)

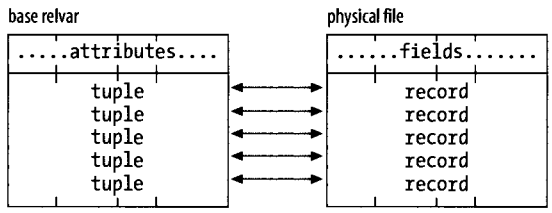


FIGURE 7-9. *Direct-image implementation (deprecated)*

Now, there are many things wrong with this direct-image style of implementation, far too many to discuss in detail here. But the overriding point is that *it provides almost no data independence:* if we have to change the physical design (for performance reasons, of course), we have to change the logical design too. In particular, it accounts for the argument, so often heard, to the effect that we have to "denormalize for performance." In principle, logical design has absolutely nothing to do with performance at all; but if the logical design maps one-to-one to the physical design... well, the conclusion is obvious. Surely we can do better than this. Relational advocates have argued for years that the relational model doesn't have to be implemented this way. And indeed it doesn't; all being well, a brand-new implementation technology is due to appear soon that addresses all of the problems of the direct-image scheme. That technology is called *The TransRelational™ Model.* Since it *is* an implementation technology, the details are beyond the scope of this book; you can find a preliminary description in my book *An Introduction to Database Systems,* Eighth Edition (Addison-Wesley, 2004). All I want to do here is point out a few desirable consequences of having an implementation that does keep the logical and physical levels rigidly and properly separate.

First, we would never need to "denormalize for performance" at all (at the logical level, I mean); all relvars could be in 5NF, or even 6NF, without any performance penalty. The logical design really would have no performance implications at all.

Second, 6NF offers a basis for a truly relational way of dealing with the problem of missing information (I mean, a way that doesn't involve nulls and three-valued logic). If you use nulls, you're effectively making the database state explicitly that there's something you don't know. But if you don't know something, it's much better to say nothing at all! To quote Wittgenstein: *Wovon man nicht reden kann, darüber muss man schweigen* ("Whereof one cannot speak, thereon one must remain silent"). For example, suppose for simplicity that there are just two suppliers right now, S1 and S2, and we know the status for S1 but not for S2. A 6NF design for this situation might look as shown in Figure 7-10.

SN

SNO	SNAME
S1	Smith
S2	Jones

SS

SNO	STATUS
S1	20

SC

SNO	CITY
S1	London
S2	Paris

FIGURE 7-10. *The status for supplier S2 is unknown*

Of course, there's a lot more that could and should be said about this approach to missing information, but this isn't the place. Here I just want to stress the point that with this design, we don't have a "tuple" showing supplier S2's status as "null"—we don't have a tuple showing supplier S2's status *at all.*

Finally, in the kind of system I'm sketching here, it really would be possible for the system to derive the optimal physical design from the logical design automatically, with little or no involvement on the part of any human designer. Space considerations among other things mean I can't provide evidence here to support this claim, but I stand by it.

Summary

The main focus of this chapter has been on logical database design theory, by which I mean, essentially, normalization and orthogonality (the scientific part of the design discipline). The point is that logical design, unlike physical design, is or should be quite DBMS-independent, and as we've seen there are some solid theoretical principles that can usefully be applied to the problem.

One point I didn't call out explicitly in the body of the chapter is that logical design should generally aim to be application-independent, too, as well as DBMS-independent. The aim is to produce a design that concentrates on what the data *means*, rather than on how it will be *used*—and I emphasized the significance of constraints and predicates ("business rules") in this connection. The database is supposed to be a faithful representation of the semantics of the situation, and constraints are what represent semantics. Abstractly, then, the logical design process goes like this:

1. Pin down the relvar predicates as carefully as possible.
2. Map the output from Step 1 into relvars and constraints. Of course, some of those constraints will be FDs, MVDs, or JDs in particular.

Incidentally, one reason application independence is desirable is that we never know all of the uses to which the data will be put. It follows that we want a design that will be robust, one that won't be invalidated by new processing requirements.

Much of design theory has to do with *reducing redundancy*: normalization reduces redundancy within relvars, orthogonality reduces it across relvars. My discussion of normalization concentrated on BCNF and 5NF, which are *the* normal forms with respect to FDs and JDs, respectively. (However, I did at least mention other normal forms, including 6NF in particular.) I pointed out that normalization makes certain constraints easier to state (and perhaps enforce); equivalently—this is something else I didn't say earlier—it makes more single-tuple updates logically acceptable than would otherwise be the case (because unnormalized designs imply redundancy, and redundancy implies that sometimes we have to update several things at once). I explained that normalization is really formalized common sense. I also gave a logical and possibly unfamiliar argument, having to do with retrieval rather than update, for not denormalizing; here let me add that although the cry is always "denormalize for performance," denormalization can actually be bad for performance (both retrieval and update performance, too), as you probably know. In fact, "denormalizing for performance" usually means improving the performance of one application at the expense of others.

I also described *The Principle of Orthogonal Design* (more formalized common sense), and I offered a few remarks on physical design. First, the physical design should be derived from the logical design and not the other way around. Second, it would be nice to get away from the currently ubiquitous direct-image style of implementation. Third, it would also be nice if physical design could be fully automated, and I held out some hope in this regard.

One last point: I want to stress that the principles of normalization and orthogonality are always, in a sense, optional. They aren't hard and fast rules, never to be broken. As we know, sometimes there are sound reasons for not normalizing "all the way" (sound *logical* reasons, I mean; I'm not talking about "denormalizing for performance"). Well, the same is true of orthogonality—although, just as a failure to normalize all the way implies redundancy and can lead to certain anomalies, so too can a failure to adhere to orthogonality. Even with the design theory I've been describing in this chapter, database design usually involves trade-offs and compromises.

Exercises

Exercise 7-1. Give definitions, as precisely as you can, of *functional dependency* and *join dependency*.

Exercise 7-2. List all of the FDs, trivial as well as nontrivial, satisfied by our usual shipments relvar SP.

Exercise 7-3. The concept of FD relies on the notion of tuple equality: true or false?

Exercise 7-4. Prove Heath's theorem. Prove also that the converse of that theorem isn't valid.

Exercise 7-5. Nonloss decomposition means a relvar is decomposed into projections in such a way that we can recover the original relvar by joining those projections back together again. In fact, if projections *r1* and *r2* of relation *r* are such that every attribute of *r* appears in at least one of *r1* and *r2*, then joining *r1* and *r2* will always produce every tuple of *r*. Prove this assertion. (It follows from the foregoing that the problem with a lossy decomposition is that the join produces additional, "spurious" tuples. Since we have no way in general of knowing which tuples in the result are spurious and which genuine, we've lost information.)

Exercise 7-6. What's a superkey? What does it mean to say an FD is implied by a superkey? What does it mean to say a JD is implied by a superkey?

Exercise 7-7. Keys are supposed to be unique and irreducible. Now, the system is obviously capable of enforcing uniqueness; but what about irreducibility?

Exercise 7-8. What's (a) a trivial FD, (b) a trivial JD? Is the former a special case of the latter?

Exercise 7-9. Let *R* be a relvar of degree *n*. What's the maximum number of FDs that *R* can possibly satisfy (trivial as well as nontrivial)?

Exercise 7-10. Given that *A* and *B* in the FD $A \rightarrow B$ are both *sets* of attributes, what happens if either is the *empty* set?

Exercise 7-11. Here's a predicate: on day *d* during period *p*, student *s* is attending lesson *l*, which is being taught by teacher *t* in classroom *c* (where *d* is a day of the week—Monday to Friday—and *p* is a period—1 to 8—within the day). Lessons are one period in duration and have a name that's unique with respect to all lessons taught in the

week. Design a set of BCNF relvars for this database. Are your relvars in 5NF? 6NF? What are the keys?

Exercise 7-12. Most of the examples of nonloss decomposition in the body of the chapter showed a relvar being decomposed into exactly two projections. Is it ever necessary to decompose into three or more?

Exercise 7-13. Many database designers recommend the use of *artifical* or *surrogate* keys in base relvars in place of what are sometimes called "natural" primary keys. For example, we might add an attribute—SPNO, say—to our usual shipments relvar (making sure it has the uniqueness property, of course) and then make {SPNO} a surrogate primary key for that relvar. (Note, however, that {SNO,PNO} would still be a key; it just wouldn't be the primary key any longer.) Thus, surrogate keys are keys in the usual relational sense, but (a) they always involve exactly one attribute and (b) their values serve *solely* as surrogates for the entities they stand for (that is, they serve merely to represent the fact that those entities exist—they carry absolutely no additional meaning or baggage of any kind). Ideally, those surrogate values would be system-generated, but whether they're system- or user-generated has nothing to do with the basic idea of surrogate keys as such. Are surrogate keys the same thing as tuple IDs? And do you think they're a good idea?

Exercise 7-14. (With acknowledgments to Hugh Darwen.) I decided to throw a party, so I drew up a list of people I wanted to invite and made some preliminary soundings. The response was good, but several people made their acceptance conditional on the acceptance of certain other invitees. For example, Bob and Cal both said they would come if Amy came; Hal said he would come if either Don and Eve both came or Fay came; Guy said he would come anyway; Joe said he would come if Bob and Amy both came; and so on. Design a database to show whose acceptance is based on whose.

Exercise 7-15. Design a database for the following. The entities to be represented are employees and programmers. Every programmer is an employee, but some employees aren't programmers. Employees have an employee number, name, and salary. Programmers have a (single) programming language skill. What difference would it make if programmers could have an arbitrary number of such skills?

Exercise 7-16. Let A, B, and C be subsets of the heading of relvar R such that the (set-theoretic) union of A, B, and C is equal to that heading. Let AB denote the (set-theoretic) union of A and B, and similarly for AC. Then R satisfies the *multi-valued dependencies* (MVDs):

$$A \rightarrow\rightarrow B$$
$$A \rightarrow\rightarrow C$$

(where $A \rightarrow\rightarrow B$ is pronounced "A double arrow B" or "A multi-determines B" or "B is multi-dependent on A," and similarly for $A \rightarrow\rightarrow C$) if and only if R satisfies the JD $\star\{AB,AC\}$. Show that if relvar R satisfies the MVDs $A \rightarrow\rightarrow B$ and $A \rightarrow\rightarrow C$, then it satisfies the property that if it includes the pair of tuples $<a,b1,c1>$ and $<a,b2,c2>$, then it also includes the pair of tuples $<a,b1,c2>$ and $<a,b2,c1>$.

Exercise 7-17. Show that if R satisfies the FD $A \rightarrow B$, it also satisfies the MVD $A \rightarrow\rightarrow B$.

Exercise 7-18. (Fagin's theorem.) Let R be as in Exercise 7-16. Show that R can be nonloss-decomposed into its projections on AB and AC if and only if it satisfies the MVDs $A \rightarrow\rightarrow B$ and $A \rightarrow\rightarrow C$.

Exercise 7-19. Show that if K is a key for R, then $K \rightarrow\rightarrow A$ is satisfied for all attributes A of R.

NOTE

Here is a convenient place to introduce some more definitions. Recall that R is in 4NF if and only if every nontrivial MVD it satisfies is implied by some superkey. The MVD $A \rightarrow\rightarrow B$ is *trivial* if and only if AB is equal to the heading of R or A is a superset of B; it's *implied by a superkey* if and only if A is a superkey.

Exercise 7-20. Give an example of a relvar that's in BCNF and not 4NF.

Exercise 7-21. Design a database for the following. The entities to be represented are sales representatives, sales areas, and products. Each representative is responsible for sales in one or more areas; each area has one or more responsible representatives. Each representative is responsible for sales of one or more products, and each product has one or more responsible representatives. Each product is sold in each area; however, no two representatives sell the same product in the same area. Each representative sells the same set of products in each area for which that representative is responsible.

Exercise 7-22. Write a **Tutorial D** CONSTRAINT statement to express the JD satisfied by relvar SPJ of Figure 7-5.

Exercise 7-23. (Modified version of Exercise 7-21.) Design a database for the following. The entities to be represented are sales representatives, sales areas, and products. Each representative is responsible for sales in one or more areas; each area has one or more responsible representatives. Each representative is responsible for sales of one or more products, and each product has one or more responsible representatives. Each product is sold in one or more areas, and each area has one or more products sold in it. Finally, if representative r is responsible for area a, product p is sold in area a, and representative r sells product p, then r sells p in a.

Exercise 7-24. Which of the following are true statements?

a. Every "all key" relvar is in BCNF.

b. Every "all key" relvar is in 5NF.

c. Every binary relvar is in BCNF.

Exercise 7-25. There's a lot of discussion in the industry at the time of writing about the possibility of *XML databases*. But XML documents are inherently hierarchic in nature. Do you think the criticisms of hierarchies in the body of the chapter apply to XML databases? Justify your answer.

Exercise 7-26. Draw E/R diagrams for the databases from Exercises 7-11, 7-14, 7-21, and 7-23. What do you conclude from this exercise?

Exercise 7-27. A certain database includes two base relvars that look like this:

```
FATHER_OF { X NAME, Y NAME } KEY { X }
MOTHER_OF { X NAME, Y NAME } KEY { X }
```

The predicates are *The father of X is Y* and *The mother of X is Y*, respectively. For simplicity, no constraints are defined, except for the two KEY constraints. Comment on this design.

Exercise 7-28. This chapter has been concerned with *data* design; in essence, I've discussed certain aspects of relational design, and there's a strong argument that you should do a clean relational design first, even if your target DBMS isn't relational at all. In the same kind of way, it's sometimes suggested that *transaction* or *query* design might be done by defining the transaction or query in terms of relational algebra first, and then mapping that definition into SQL (or whatever your target language is) as a follow-on activity afterward. What do you think of this strategy?

CHAPTER EIGHT

What Is the Relational Model?

You might think it odd for the final chapter in a book that's meant to explain the relational model to have as its title *What Is the Relational Model?* Aren't you supposed to know by now?

Well, yes, you are. But the point of this chapter is to serve as a kind of wrap-up for everything that's gone before; in particular, it's meant to give a precise definition, for purposes of future reference, of just what it is that constitutes the relational model. The trouble is, the definition I'll give is indeed precise: so precise, in fact, that I think it would have been pretty hard to understand if I'd given it in Chapter 1. (As Bertrand Russell once memorably said: *Writing can be either readable or precise, but not at the same time.*) Of course, I did give a definition in Chapter 1—a definition, that is, of what I there called "the original model"—but I frankly don't think that definition is even close to being good enough, for the following reasons among others:

- For starters, it was much too long and rambling. (Well, that was fair enough, given the intent of that preliminary chapter, but now I want a definition that's succinct as well as precise.)
- I don't really care for the idea that the model should be thought of as consisting of "structure plus integrity plus manipulation"; in some ways, in fact, I think it's actively misleading to think of it in such terms.

- "The original model" included a few things I'm not too comfortable with: for instance, nulls (of course!), the entity integrity rule, the idea of having to make one key primary, and the idea—which I didn't discuss at all in this book, because I don't agree with it—that domains and types are somehow different things. Regarding nulls, incidentally, I note that Codd first defined the relational model in 1969 and didn't introduce nulls until 1979; in other words, the model managed perfectly well (in my opinion, better) for some 10 years without any notion of nulls at all. What's more, early implementations managed perfectly well without them, too.
- The original version of the model also omitted a few things I now consider vital. For example, it excluded any mention—at least, any explicit mention—of all of the following: predicates, constraints (other than candidate and foreign key constraints), relation variables, relational comparisons, relation type inference and associated features, certain algebraic operators (especially rename, extend, summarize, semijoin, and semidifference), and the important relations TABLE_DUM and TABLE_DEE. (On the other hand, I think it could fairly be argued that these features at least weren't precluded by the original version of the model; it might even be argued in some cases that they were in fact included, in a kind of embryonic form. For example, it was certainly always intended that implementations should include support for constraints other than just candidate and foreign key constraints.)

Without further ado, then, let me give my own definition.

The Relational Model Defined

The relational model consists of five components:

1. An open-ended collection of *scalar types*, including in particular the type BOOLEAN (truth values)
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* operator for assigning relation values to such relation variables
5. An open-ended collection of generic *relational operators* for deriving relation values from other relation values

The following subsections elaborate on each of these components in turn.

Scalar Types

Scalar types can be either system-defined (built-in) or user-defined, in general; thus, a means must be available for users to define their own scalar types (this requirement is implied, partly, by the fact that the set of scalar types is open-ended). A means must therefore also be available for users to define their own scalar operators, since types

without operators are useless. The only *required* system-defined scalar type is type BOOLEAN—the most fundamental type of all—but a real system will surely support others as well (INTEGER, CHAR, and so on).

Support for type BOOLEAN implies support for the usual logical operators—NOT, AND, OR, and so on—as well as other operators (system- or user-defined) that return boolean values. In particular, the equality comparison operator "=" (equality comparison) *must* be available in connection with every type, nonscalar as well as scalar, because without it we couldn't even define the values that constitute the type in question. What's more, the model prescribes the semantics of that operator, too. To be specific, if *v1* and *v2* are values of the same type, *v1* = *v2* evaluates to TRUE if *v1* and *v2* are the very same value and FALSE otherwise.

By the way, SQL fails seriously on this requirement (that "=" be supported for every type, with prescribed semantics). For example:

- For the built-in types CHAR and VARCHAR, it's possible—in fact, common—for "=" to give TRUE even if the comparands are distinct.
- For the built-in type XML (and the built-in types BLOB and CLOB, in certain products), "=" isn't defined at all.
- For user-defined types, "=" can be defined only when "<" is defined as well, and for some types "<" makes no sense; and even when "=" *is* defined, the semantics are arbitrary, in the sense that they're left to the type definer.
- For tables, no comparison operators are defined at all.
- Last, it's quite common for "=" not to give TRUE even if the comparands are indistinguishable; in particular, this situation arises if both comparands are null.

One consequence of such deficiencies is that SQL violates *The Assignment Principle* (see the later section "Some Database Principles")—ubiquitously so, in fact.

Relation Types

The relation type generator allows users to specify individual relation types as desired: in particular, as the type for some relation variable or some relation-valued attribute (see Chapter 2 for further explanation). The intended interpretation for a given relation of a given type in a given context is as a set of true propositions; each such proposition constitutes an instantiation of some predicate that (a) corresponds to the relation heading and (b) is represented by a tuple in the relation body. If the context in question is some relvar—that is, if we're talking about the relation that happens to appear as the current value of some relvar—then the predicate in question is the relvar predicate for that relvar. If a tuple plausibly could appear in that relvar at some time but doesn't, the corresponding proposition is assumed to be false at that time.

Since the equality comparison operator "=" is available in connection with every type, it's available in connection with every relation type in particular.

Relation Variables

As noted in the previous subsection, a particularly important use for the relation type generator is in specifying the type of a relation variable, or relvar, when that relvar is defined. The *only* kind of variable permitted in a relational database is the relvar (in particular, scalar and tuple variables are prohibited, even though they're not prohibited—in fact, they're probably required—in programs that access such a database).

The statement that the database contains nothing but relvars is one possible formulation of what Codd originally called *The Information Principle*, though I don't think it's a formulation he ever used himself. Instead, he usually stated the principle like this:

> The entire information content of the database (at any given time) is represented in one and only one way: namely, as explicit values in attribute positions in tuples in relations.

I heard Codd refer to this principle on more than one occasion as the fundamental principle underlying the relational model. Why is it so important? The answer is bound up with the observations I made in Chapter 4 to the effect that, along with types, relations are both necessary and sufficient to represent any data whatsoever at the logical level. In other words, the relational model gives us exactly what we do need in this respect, and it doesn't give us anything we don't need.

I'd like to pursue this point a moment longer. In general, it's axiomatic that if we have *n* different ways of representing data, then we need *n* different sets of operators. For example, if we had arrays as well as relations, we'd need a full complement of array operators as well as a full complement of relational ones. If *n* is greater than one, therefore, we have more operators to implement, document, teach, learn, remember, and use. But those extra operators add complexity, not power! There's nothing useful that can be done if *n* is greater than one that can't be done if *n* equals one (and in the relational model, of course, *n* does equal one).

What's more, not only does the relational model give us just one construct, the relation itself, for representing data, but that construct is—to quote Codd himself (see the next section, "Objectives of the Relational Model")—*of spartan simplicity*: it has no ordering to its tuples, no ordering to its attributes, no duplicate tuples, no pointers, and (at least as far as I'm concerned) no nulls. Any contravention of these properties is tantamount to introducing another way of representing data, and therefore to introducing more operators as well. In fact, SQL is living proof of this observation; for example, SQL has eight different union operators,* while (as we know) the relational model has just one.

* By rights that eight ought really to be *twelve*—SQL's so-called "multiset union," which applies to the "multisets of rows" that are permitted as values within columns of tables, supports only two of the six options that are supported for its regular table union. To make matters worse, SQL doesn't support the true multiset union operator at all; the operator it calls "multiset union" corresponds to what's called "union plus" in the literature. See Donald E. Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition (Addison-Wesley, 1997) for further explanation.

As you can see, *The Information Principle* is certainly important—but it has to be said that its name hardly does it justice. Other names that have been proposed, mainly by Hugh Darwen or myself or both, include *The Principle of Uniform Representation* and *The Principle of Uniformity of Representation.* (This latter is clumsy, I admit, but at least it's accurate.)

Relational Assignment

Like the equality comparison operator "=", the assignment operator ":=" must be available in connection with every type, for without it we would have no way of assigning an arbitrary value to a variable of the type in question—and again, relation types are no exception to this rule. INSERT, DELETE, and UPDATE shorthands are permitted and indeed useful, but strictly speaking they're only shorthands. What's more, support for relational assignment must include support for *multiple* relational assignment in particular.

Relational Operators

The "generic relational operators" are the operators that make up the relational algebra, and they're therefore built-in (though there's no inherent reason why users shouldn't be allowed to define additional operators of their own, if desired).

Now, there seems to a widespread misconception concerning the purpose of the algebra. To be specific, many people seem to think it's meant just for writing queries—but it's not; rather, it's for *writing relational expressions.* Those expressions in turn serve many purposes, including but certainly not limited to query. Here are some other important ones:

- Defining views and snapshots
- Defining the set of tuples to be inserted into, deleted from, or updated in some relvar (or, more generally, defining the set of tuples to be assigned to some relvar)
- Defining constraints (though here the relational expression is just a subexpression of some boolean expression, frequently though not invariably an IS_EMPTY invocation)

And so on (this isn't an exhaustive list).

The algebra also serves as a kind of yardstick against which the expressive power of database languages can be measured. Essentially, a language is said to be *relationally complete* if and only if it's at least as powerful as the algebra, meaning its expressions permit the definition of every relation that can be defined by means of expressions of the algebra. Relational completeness is a basic measure of the expressive capability of a language; if a language is relationally complete, it means (among other things, and speaking a trifle loosely) that queries of arbitrary complexity can be formulated without having to resort to loops or recursion. In other words, it's relational completeness that allows end users—at least in principle, though possibly not in practice—to access the database directly, without having to go through the potential bottleneck of the IT department.

Objectives of the Relational Model

For purposes of reference if nothing else, it seems appropriate in this chapter to document Codd's own stated objectives in introducing his relational model. The following list is based on one he gave in his paper "Recent Investigations into Relational Data Base Systems" (an invited paper to the 1974 IFIP Congress), but I've edited it just slightly here.

1. To provide a high degree of data independence
2. To provide a community view of the data of spartan simplicity, so that a wide variety of users in an enterprise, ranging from the most computer-naïve to the most computer-sophisticated, can interact with a common model (while not prohibiting superimposed user views for specialized purposes)
3. To simplify the potentially formidable job of the DBA
4. To introduce a theoretical foundation (albeit modest) into database management (a field sadly lacking in solid principles and guidelines)
5. To merge the fact-retrieval and file-management fields in preparation for the addition at a later time of inferential services in the commercial world
6. To lift database application programming to a new level—a level in which sets (and more specifically relations) are treated as operands instead of being processed element by element

I'll leave it to you to judge to what extent you think the relational model meets these objectives. Myself, I think it does pretty well.

Some Database Principles

In Chapter 1, I said I was interested in principles, not products, and we've encountered several principles at various points in the book. Here I list them for purposes of reference:

The Information Principle

The database contains nothing but relvars; equivalently, the entire information content of the database (at any given time) is represented in one and only one way—namely, as explicit values in attribute positions in tuples in relations.

The Closed World Assumption

If tuple *t* could plausibly appear in relvar *R* at some given time but doesn't, then the proposition *p* corresponding to *t* is assumed to be false at that time.

The Principle of Interchangeability

There must be no arbitrary and unnecessary distinctions between base and virtual relvars.

The Principles of Normalization

- A non-5NF relvar should be decomposed into a set of 5NF projections.
- The decomposition should be nonloss.
- The decomposition should preserve dependencies.
- Every projection should be needed in the reconstruction process.
- Decomposition should stop as soon as all relvars are in 5NF. (This one isn't as firm as the first four; indeed, there are strong arguments in favor of going all the way to 6NF, if the system is such that it allows full normalization without imposing any serious penalties for doing so.)

The Principle of Orthogonal Design

Let *A* and *B* be distinct relvars in the same database. Then there must not exist nonloss decompositions of *A* and *B* such that the relvar constraints for some projection of *A* and some projection of *B* in those decompositions are such as to permit the same tuple to appear in both of those projections.

The Assignment Principle

After assignment of the value *v* to the variable *V*, the comparison $V = v$ must evaluate to TRUE.

The Golden Rule

No update operation must ever cause any database constraint to evaluate to FALSE.

The Principle of Identity of Indiscernibles

Let *a* and *b* be any two things (any two "entities," if you prefer); then, if there's no way whatsoever of distinguishing between *a* and *b*, there aren't two things but only one.

NOTE

I didn't mention *The Principle of Identity of Indiscernibles* earlier in this book, but I appealed to it tacitly on many occasions. It can alternatively be stated thus: *Every entity has its own unique identity.* In the relational model, such identities are represented in the same way as everything else—namely, by means of attribute values (see *The Information Principle,* earlier in this list)—and numerous benefits accrue from this simple fact.

The Relational Model Versus Others

To repeat something I said in Chapter 4, it's my opinion that the relational model is rock solid, and "right," and will endure. A hundred years from now, I fully expect database systems still to be based on Codd's relational model. Why? Because the foundations of that model—namely, set theory and predicate logic—are themselves rock solid in turn. Elements of predicate logic in particular go back well over 2,000 years, at least as far as Aristotle (384–322 BCE).

So what about other data models—the "object-oriented model," for example, or the "hierarchic model," or the CODASYL "network model," or the "semistructured model"?

In my view, these other models just aren't in the same ballpark. Indeed, I seriously question whether they deserve to be called models at all.* The hierarchic and network models in particular never really existed in the first place!—as abstract models, I mean, predating any implementations. Instead, they were invented *after the fact*; that is, commercial hierarchic and network products were built first, and the corresponding models were defined subsequently by a process of induction—here just a polite term for guesswork—from those products. As for the object-oriented and semistructured models, it's entirely possible that the same criticism applies, though it's hard to be sure. One problem is that there doesn't seem to be any consensus on what those models might consist of. It certainly can't be claimed, for example, that there's a unique, clearly defined, and universally accepted object-oriented model; and similar remarks apply to the semistructured model. (Actually, some people might claim there isn't a unique relational model, either! I'll deal with that argument a little later.)

Another important reason why I don't believe those other models really deserve to be called models at all is the following. First, I hope you agree it's undeniable that the relational model is indeed a model and is thus not, by definition, concerned with implementation issues. By contrast, the other models all fail, much of the time, to make a clear distinction between issues that truly are model issues and issues that are better regarded as implementation matters; at the very best, they muddy that distinction considerably (they're all much "closer to the metal," as it were). As a consequence, they're harder to use and understand, and they give implementers far less freedom—far less than the relational model does, I mean—to adopt inventive or creative approaches to questions of implementation.

So what of the claims to the effect that there are several "relational" models, too? For example, a recent (well, fairly recent) book† has a chapter titled "Different Relational Models," in which we find this:

> There is no such thing as *the* relational model for databases anymore [*sic*] than there is just one geometry.

And to bolster his argument, the author then goes to identify what he claims are six "different relational models."

Now, I wrote a response to these claims soon after I first encountered them. Here's an edited version of what I said at the time:

> Of course it's true there are several different geometries (euclidean, elliptic, hyperbolic, and so forth). But is the analogy a valid one? That is, do those "different relational models" differ in the same way those different geometries differ? It seems to

* Which is why I set them all in quotation marks. I'll drop the quotation marks from this point forward because I know how annoying they can be, but you should think of them as still being there in some virtual kind of sense.

† *Joe Celko's Data and Databases: Concepts in Practice* (Morgan Kaufmann, 1999).

me the answer to this question is *no*. Elliptic and hyperbolic geometries are often referred to, quite explicitly, as *noneuclidean* geometries; for the analogy to be valid, therefore, it would seem that at least five of those "six different relational models" would have to be *nonrelational* models, and hence, by definition, not "relational models" at all. (Actually, I would agree that several of those "six different relational models" are indeed not relational. But then it can hardly be claimed—at least, it can't be claimed *consistently*—that they're "different *relational* models.")

And I went on to say this (again somewhat edited here):

But I have to admit that Codd did revise his own definitions of what the relational model was, somewhat, throughout the 1970s and 1980s. One consequence of this fact is that critics have been able to accuse Codd in particular, and relational advocates in general, of "moving the goalposts" far too much. For example, Mike Stonebraker has written* that "one can think of four different versions" of the model:

- Version 1: Defined by the 1970 CACM paper
- Version 2: Defined by the 1981 Turing Award paper
- Version 3: Defined by Codd's 12 rules and scoring system
- Version 4: Defined by Codd's book

Let me interrupt myself briefly to explain the references here. They're all by Codd. The 1970 CACM paper is "A Relational Model of Data for Large Shared Data Banks," *CACM 13*, No. 6 (June 1970). The 1981 Turing Award paper is "Relational Database: A Practical Foundation for Productivity," *CACM 25*, No. 2 (February 1982). The 12 rules and the accompanying scoring system are described in Codd's *Computerworld* articles "Is Your DBMS Really Relational?" and "Does Your DBMS Run By The Rules?" (October 14th and October 21st, 1985). Finally, Codd's book is *The Relational Model For Database Management Version 2* (Addison-Wesley, 1990). Now back to my response:

Perhaps because we're a trifle sensitive to such criticisms, Hugh Darwen and I have tried to provide, in our book *Foundation for Future Database Systems: The Third Manifesto* (2nd edition, Addison-Wesley, 2000), our own careful statement of what we believe the relational model is (or ought to be!).† Indeed, we'd like our *Manifesto* to be seen in part as a definitive statement in this regard. I refer you to the book itself for the details; here just let me say that we see our contribution in this area as primarily one of dotting a few *i*'s and crossing a few *t*'s that Codd himself left undotted or uncrossed in his own original work. We most certainly don't want to be thought of as departing in any major respect from Codd's original vision; indeed, the whole of the *Manifesto* is very much in the spirit of Codd's ideas and continues along the path that he originally laid down.

* In his introduction to Chapter 1 ("The Roots"), *Readings in Database Systems*, Second Edition (Morgan Kaufmann, 1994).

† That book is now superseded by our book *Databases, Types, and the Relational Model: The Third Manifesto*, Third Edition (Addison-Wesley, 2006), but its broad message remains unchanged.

To all of the preceding points I'd now like to add another, which I think clearly refutes the author's original argument. I agree there are several different geometries. But the reason why those geometries are all different is this: *they start from different axioms*. By contrast, we've never changed the axioms for the relational model. We *have* made a number of changes over the years to the model itself—for example, we've added relational comparisons—but the axioms (which are basically those of classical predicate logic) have remained unchanged ever since Codd's first papers. Moreover, what changes have occurred have all been, in my view, evolutionary, not revolutionary, in nature. Thus, I really do claim there's only one relational model, even though it has evolved over time and will presumably continue to do so. As I said in Chapter 1, the model can be seen as a small branch of mathematics; as such, it grows over time as new theorems are proved and new results discovered.

So what are those evolutionary changes? Here are some of them:

- As already mentioned, we've added relational comparisons.
- We've clarified the importance of the logical difference between relations and relvars.
- We have a better understanding of the nature of relational algebra, including the relative significance of various operators and an appreciation of the importance of relations of degree zero, and we've identified certain new operators (for example, extend).
- We also have a better understanding of updating, including view updating in particular.
- We've clarified the concept of first normal form; as a consequence, we've embraced the concept of relation-valued attributes in particular.
- We have a better understanding of the fundamental significance of integrity constraints in general, and we have many good theoretical results regarding certain important special cases.
- We've clarified the nature of the relationship between the model and predicate logic.
- Finally, we have a clearer understanding of the relationship between the relational model and type theory (more specifically, we've clarified the nature of domains).

What Remains to Be Done?

All of the above is not to say we won't continue to make progress or there isn't still work to be done. In fact, I see at least four areas, somewhat interrelated, where developments are either under way or are needed: implementation, foundations, higher-level abstractions, and higher-level interfaces.

Implementation

In some ways the message of this book can be summed up very simply:

Let's implement the relational model!

I think it's clear from earlier chapters that it's being extremely charitable to describe SQL as a relational language, and hence that SQL products can be considered relational only to a first approximation. The truth is, the relational model has never been properly implemented in commercial form, and users have never really enjoyed the benefits that a truly relational product would bring. Indeed, that's one of the reasons why Hugh Darwen and I have been working for so long on *The Third Manifesto*. *The Third Manifesto*—the *Manifesto* for short—is a formal proposal for a solid foundation for future DBMSs. And it goes without saying that what it really does, in as careful and precise a manner as the authors are capable of, is define the relational model and spell out some of the implications of that definition. (It also goes into a great deal of detail on the impact of type theory on that model; in particular, it proposes a comprehensive model of type inheritance as a logical consequence of that type theory.)

So we'd really like to see the ideas of the *Manifesto* implemented properly in commercial form ("we" here meaning Darwen and myself). We believe such an implementation would serve as a solid basis on which to build so many other things—for example, "object/relational" DBMSs; spatiotemporal DBMSs; DBMSs used in connection with the World Wide Web; and "rule engines" (also known as "business logic servers"), which some people see as the next generation of general-purpose DBMS products. We further believe we would then have the right framework for supporting the other items that are suggested in the rest of this section as also being desirable. Personally, in fact, I would go further: I would suggest that trying to implement those items in any other kind of framework is likely to prove more difficult than doing it correctly. To quote the well-known mathematician Gregory Chudnovsky: "If you do it the stupid way, you will have to do it again" (from an article in *The New York Times*, December 24, 1997).

To repeat, I want the model to be properly implemented. And in the previous chapter, I tried to suggest that a promising new implementation technology called *The TransRelational™ Model* looks as if it might be well suited to that task. This possibility is under active investigation.

Foundations

There's still much interesting work to be done on theoretical foundations (it's certainly not the case that all of the foundation problems have been solved). Here are three examples:

- Let *rx* be some relational expression. By definition, the relation *r* denoted by *rx* satisfies a constraint *rc* that's derived from the constraints satisfied by the relations in terms of which *rx* is expressed. Can that constraint *rc* be computed?
- Can we inject more science into the database design process? In particular, can we come up with a precise characterization of the notion of redundancy?
- In the previous chapter I sketched an approach to the missing information problem based on 6NF. What are the implications of that approach?

Higher-Level Abstractions

One way we make progress in computer languages and applications is by *raising the level of abstraction*. For example, I pointed out in Chapter 4 that the familiar KEY and FOREIGN KEY specifications are really just shorthand for constraints that can be expressed more longwindedly using the general integrity features of any relationally complete language like **Tutorial D**. But those shorthands are *useful:* quite apart from the fact that they save us some writing, they also serve to raise the level of abstraction, by allowing us to talk in terms of certain bundles of concepts that naturally belong together. In a sense, they make it easier for us to see the forest as well as the trees.

By way of another illustration, consider the relational algebra. I showed in Chapter 5 that many of the operators of the algebra—including ones we use all the time (even if we don't realize it), such as semijoin—are really shorthand for certain combinations of other operators.* Indeed, there are other useful operators that I didn't discuss in that chapter at all, for space reasons, for which these remarks might be regarded as "even more true," in a sense. Again, what's really going on here is a raising of the level of abstraction (rather like macros raise the level of abstraction in a conventional programming language).

Raising the level of abstraction in the relational world can be regarded as a kind of building on top of the relational model; it doesn't change the model, but it does make it more directly useful for certain tasks. And one area where this approach looks as if it's going to prove really fruitful is temporal databases. In our book *Temporal Data and the Relational Model* (Morgan Kaufmann, 2003), Hugh Darwen, Nikos Lorentzos, and I—building on original work by Lorentzos—introduce *interval types* as a basis for supporting temporal data in a relational framework. For example, consider the "temporal relation" in Figure 8-1, which shows that certain suppliers supplied certain parts during certain intervals of time (you can read *d04* as "day 4," *d06* as "day 6," and so on; likewise, you can read [*d04*:*d06*] as "the interval from day 4 to day 6 inclusive," and so on). Attribute DURING in that relation is interval-valued.

SNO	PNO	DURING
S1	P1	[d04:d06]
S1	P1	[d09:d10]
S1	P3	[d05:d10]
S2	P1	[d02:d04]
S2	P1	[d08:d10]
S2	P2	[d03:d03]
S2	P2	[d09:d10]
S3	P1	[d02:d05]
S3	P2	[d08:d10]
S3	P3	[d08:d10]

FIGURE 8-1. *A relation with an interval attribute*

* As a matter of fact, Darwen and I show in our *Manifesto* that every algebraic operator can be expressed in terms of just two primitives, *remove* (which is basically project) and either *nand* or *nor*.

Support for interval attributes (and hence for temporal databases) involves, among other things, support for generalized versions of the regular algebraic operators. For reasons that aren't important here, we call those generalized operators U_ operators; thus, there's a *U_restrict* operator, a *U_join* operator, a *U_union* operator, and so on. But—and here comes the point—those *U_ operators* are all, in the last analysis, nothing but shorthand for certain combinations of regular algebraic operators. Once again, then, what's fundamentally going on is a raising of the level of abstraction.

Two more points on this topic. First, our relational approach to temporal data involves not just "U_" versions of the algebraic operators but also (a) "U_" keys and foreign keys, (b) "U_" comparison operators, and (c) "U_" versions of INSERT, DELETE, and UPDATE—but, again, all of these constructs turn out to be essentially just shorthand. Second, it also turns out that the *Manifesto's* type inheritance model has a crucial role to play in that temporal support—so once again we see an example of the interconnectedness of all of these issues.

Higher-Level Interfaces

There's another way in which we can build on top of the relational model, and that's by means of various kinds of applications that run above the relational interface and provide various specialized services. One example might be decision support; another might be data mining; another might be a natural-language frontend. For the users of such applications, the relational model obviously disappears under the covers, at least to some degree. (Though even if it does, and even if most users interact with the database only through some such frontend, I think database design and the like will still necessarily be based on solid relational principles.)

By the way, suppose it's your job to implement one of those frontend applications. Which would you prefer as a target—a relational DBMS or some other kind, say an object-oriented DBMS? And if you opt for the former, as I obviously think you should, which would you prefer—a DBMS that supports the relational model or one that supports SQL?

In case it's not clear, my point is this: we've come a long way from the early days when SQL was being touted as a language that end users could use for themselves, and I know many people will dismiss my numerous criticisms of SQL as mere carping for that very reason. Real users don't use it anyway, right? Only programmers use it. And in any case, much of the SQL code that's actually executed is never written by a human programmer at all but is generated by some frontend application. However, it seems to me that SQL is bad as a *target* language for all of the same reasons that it's bad as a *source* language. And it further seems to me, therefore, that my criticisms are still germane.

So What About SQL?

SQL is incapable of providing the kind of firm foundation we need for future growth and development. Instead, it's the relational model that has to provide that foundation. In *The Third Manifesto*, therefore, Darwen and I reject SQL as such; in its place, we argue that

some truly relational language like **Tutorial D** should be implemented as soon as possible. Of course, we aren't so naïve as to think that SQL will ever disappear. Rather, we hope that **Tutorial D**, or some other true relational language, will be sufficiently superior that it will become the database language of choice (by a process of natural selection), and SQL will become "the database language of last resort." In fact, we see a parallel with the world of programming languages, where COBOL has never disappeared (and never will); but COBOL has become "the programming language of last resort" for developing applications, because better alternatives exist. We see SQL as a kind of database COBOL, and we would like to see some other language become available as a better alternative to it.

Of course, we do realize that SQL databases and applications are going to be with us for a long time—to think otherwise would be quite unrealistic—and so we do have to pay some attention to the question of what to do about today's SQL legacy. The *Manifesto* therefore does include some specific proposals in this regard. In particular, it offers some suggestions for implementing SQL on top of a true relational language, so that existing SQL applications can continue to work. Detailed discussion of those proposals would be out of place here, however.

Summary

As noted in the introduction to this chapter, the purpose of this book has been to explain the relational model—primarily to people, especially database practitioners, who already know something about it but realize they need to know more. Since this is the final section of the final chapter in the entire book, it seems appropriate to review here what's been covered in the book as a whole. So here goes:

- Chapter 1, *Introduction*, set the stage by describing (not very formally) the original model. As I put it in that chapter, my primary purpose was to tell you what I hoped you knew already. But I also discussed the important logical differences between relations and relvars, and between values and variables in general, and between the model and its implementation. This last in particular led to a discussion of data independence.
- In Chapter 2, *Relations Versus Types*, I argued that domains were just types by another name and, further, that those types could be of arbitrary complexity. The question as to what types are supported is orthogonal to the question of support for the relational model itself ("Types are orthogonal to tables").
- Chapter 3, *Tuples and Relations*, gave precise definitions for these fundamental concepts and discussed several consequences of those definitions. It also gave some pragmatic arguments for prohibiting both duplicates and nulls, and it introduced the important special relations TABLE_DUM and TABLE_DEE.
- Chapter 4, *Relation Variables*, was where I first considered the crucial notion of relvar predicates. I showed that types and relations are both necessary and sufficient to represent any data we like at the logical level. I explained the details of candidate and foreign keys, and I took a much closer look at views (also known as virtual relvars).

- Chapter 5, *Relational Algebra*, was the longest chapter in the book. I stressed the importance of closure and introduced a set of relation type inference rules. I described many of the most important algebraic operators (especially join, which includes both intersection and product as special cases). I gave a conceptual algorithm for evaluating SQL SELECT - FROM - WHERE expressions. I briefly discussed the relevance of the algebra to optimization and to the update operators, and I described relational comparison operators.
- In Chapter 6, *Integrity Constraints*, I explained the two basic kinds of constraints, type constraints and database constraints. I also briefly discussed "possreps," selectors, and THE_ operators. Type constraints are checked as part of the execution of selector operators; database constraints are checked "at semicolons." I also discussed multiple assignment, *The Golden Rule*, and the difference between correctness and consistency. And I claimed that constraints were crucial ("I don't care how fast your system runs if I can't trust the answers it gives me").
- Chapter 7, *Database Design Theory*, concentrated on the theory of logical database design. That theory isn't really part of the relational model as such but uses that model as a base on which to build. I stressed the fact that logical design is intimately bound up with constraints and predicates. A large part of design theory has to do with reducing redundancy: normalization reduces redundancy within relvars, orthogonality reduces it across relvars. I discussed FDs, BCNF, JDs, and 5NF, and briefly mentioned 6NF, elaborating on the idea that "it's all really formalized common sense." Don't denormalize! I also discussed orthogonality, and I offered a few generic remarks on physical design as well.
- Finally, the present chapter, *What Is the Relational Model?*, has taken a look at some of the issues raised by the question in the title. I gave a terse, precise, and possibly not immediately understandable answer to that question, identifying five components of the model and elaborating on each in turn. I also listed Codd's objectives for his model. Next, I summarized the various principles—*The Information Principle* and the rest—that we encountered earlier in the text. I tried to show why other "models" aren't in the same ballpark as the relational model; as far as I can tell, in fact, the relational model is still the only one to have been defined before implementations were attempted, and it's still the only one that rigidly excludes all implementation considerations. Last, I discussed a variety of ways in which we might continue to move this whole field forward.

Exercises

These exercises are intended to serve as a review of the entire book, not just the present chapter. Some are repeats of exercises from earlier chapters.

Exercise 8-1. What exactly *is* the relational model? State as many differences as you can think of between SQL and the relational model. Why are SQL's departures from the relational model a bad thing?

Exercise 8-2. What's *The Information Principle?* How might "row IDs" violate it?

Exercise 8-3. What's a predicate? What's the connection between relations and predicates?

Exercise 8-4. Do you believe relations are "flat" or two-dimensional? Justify your answer.

Exercise 8-5. What's a join dependency? If relvar R satisfies the FD $A \rightarrow B$, what JD does it satisfy?

Exercise 8-6. What's the real difference between a domain and a relation?

Exercise 8-7. What's wrong with deferred integrity checking?

Exercise 8-8. What's *The Principle of Interchangeability?*

Exercise 8-9. What does first normal form really mean?

Exercise 8-10. What's "the final normal form"? In what sense is it final?

Exercise 8-11. What's the difference between a relation and a relvar?

Exercise 8-12. What's *The Principle of Orthogonal Design?*

Exercise 8-13. Would it make sense to define a relational comparison operator, "/" say, such that r / s gives TRUE if and only if r and s are disjoint (i.e., have no tuples in common)? Justify your answer.

Exercise 8-14. Why is ORDER BY not a relational operator?

Exercise 8-15. The difference between base relvars and views is that the former are physically stored and the latter aren't: true or false?

Exercise 8-16. Why doesn't the relational model permit nulls? Or duplicates?

Exercise 8-17. The relational model prescribes the data types that must be supported: true or false?

Exercise 8-18. What's the difference between a primary key and a candidate key?

Exercise 8-19. What's nonloss decomposition?

Exercise 8-20. Should it ever be necessary to denormalize?

Exercise 8-21. Product is a special case of join: true or false?

Exercise 8-22. What's a type constraint? When are type constraints checked?

Exercise 8-23. Can a relation have an attribute whose values are sets? Or arrays? Or relations?

Exercise 8-24. Why do SQL's updates through a cursor violate the relational model?

Exercise 8-25. Can a relation have no attributes at all?

Exercise 8-26. What does it mean to say that the relational algebra is closed? Why is such closure important?

Exercise 8-27. Any given relation r is identically equal to a certain restriction of r and a certain projection of r. Explain these observations.

Exercise 8-28. Why is "materialized view" a contradiction in terms?

Exercise 8-29. What's the difference between the relational model and an implementation thereof?

Exercise 8-30. What's the crucial logical difference between a relational database and any other kind of database? (*Hint*: Remember *The Information Principle*.)

Exercise 8-31. What's the difference between a true object/relational DBMS and a true relational DBMS?

Exercise 8-32. Does restrict distribute over union? Over minus?

Exercise 8-33. Does project distribute over union? Over minus?

Exercise 8-34. How does XML fit with the relational model?

Exercise 8-35. If you summarize an empty relation, what do you get?

Exercise 8-36. What's the Closed World Assumption?

Exercise 8-37. Define the operators semijoin and semiminus.

Exercise 8-38. Define (a) BCNF, (b) 5NF, and (c) 6NF. Did it ever occur to you that the last two—4NF too, come to that—are misnamed?

Exercise 8-39. Is join commutative? Associative? Idempotent?

Exercise 8-40. Every binary relvar is in BCNF: true or false?

Exercise 8-41. What's the identity with respect to join?

Exercise 8-42. What does it mean, formally or informally, to say an FD or JD is trivial?

Exercise 8-43. Does a key that involves no attributes at all make any sense? What about a foreign key?

Exercise 8-44. If a 3NF relvar has no keys that involve two or more attributes, that relvar is in 5NF: true or false?

Exercise 8-45. Give at least three reasons why the result of a SELECT statement in SQL isn't a relation, in general. You can assume we're talking about an interactive environment, meaning we're not limited to "singleton SELECTs" (that is, SELECTs that retrieve at most one row).

Exercise 8-46. Intersect is a special case of join: true or false?

Exercise 8-47. Suppose relation r is of degree three. How many distinct projections does r have?

Exercise 8-48. Suppose relvar R is of degree three. What's the maximum number of keys R can possibly have? What about FDs?

Exercise 8-49. Are (relational) cartesian product and (relational) division inverse operations? *Subsidiary question:* Why did I say, specifically, *relational* cartesian product?

Exercise 8-50. What's the join of n relations for $n = 3$? What about $n = 1$? And what about $n = 0$?

Exercise 8-51. How can we do relational comparisons in SQL?

Exercise 8-52. Does it make sense to declare keys for a view?

Exercise 8-53. (a) Let the FD $A \rightarrow B$ be satisfied by some relvar R. A and B are sets of attributes, of course; so what happens if either of those sets is empty? (b) Let K be a key for some relvar R. K is a set of attributes; so what happens if that set is empty?

Exercise 8-54. Which of the following identities are valid?

```
a. r INTERSECT s ≡ r MINUS ( r MINUS s )
b. r UNION ( r INTERSECT s ) ≡ r
c. r INTERSECT ( r UNION s ) ≡ r
d. r MATCHING r ≡ r
```

Exercise 8-55. What do you understand by the term "semantic optimization"?

Exercise 8-56. Why are relation-valued attributes usually contraindicated (at least in base relvars)?

Exercise 8-57. "TABLE_DEE plays a role in the relational algebra analogous to that played by 0 in ordinary arithmetic." Explain this remark.

Exercise 8-58. (a) Let *Op* be a monadic relational operator. What happens if the sole operand to *Op* is either TABLE_DUM or TABLE_DEE? (b) Let *Op* be a dyadic relational operator. What happens if each of the operands to *Op* is either TABLE_DUM or TABLE_DEE?

Exercise 8-59. "There's only one relational model." Either justify this claim or state clearly why you disagree.

Exercise 8-60. Why is it important for database professionals to be able to answer questions like these correctly?

APPENDIX A

A Little Bit of Logic

In Chapter 1, I mentioned that there's an alternative to the relational algebra called the relational calculus. Relational calculus is essentially an applied form of predicate calculus, tailored to the needs of relational databases. Queries and constraints, for example, can be formulated using relational calculus instead of relational algebra; indeed, sometimes it's easier to come up with a calculus formulation (on the other hand, sometimes the opposite is true). And it's desirable, in my opinion, for database professionals to have at least a nodding acquaintance with the basic concepts of the calculus as well as the algebra; hence this appendix, which offers a brief and very informal introduction to such matters.

By the way, it follows from the preceding paragraph that a relational language can be based on either the algebra or the calculus. So which is SQL based on? The answer, regrettably, is partly both and partly neither. When it was first designed, SQL was specifically intended to be different from both the algebra and the calculus; indeed, such a goal was the prime motivation for the introduction of the SQL "IN subquery" construct. As time went by, however, it turned out that certain features of both the algebra and the calculus were needed after all, and the language grew to accommodate them. The situation today is thus that some aspects of SQL are "algebra-like," some are "calculus-like," and some are neither.

Propositions

Recall from Chapter 4 that a *proposition* is a statement that's unequivocally either true or false. Given some specified set of propositions, we can combine propositions from that set in a variety of ways to form further propositions, using the *connectives* NOT, AND, OR, IF ... THEN ... (or IMPLIES), and IF AND ONLY IF (or BI-IMPLIES, or IS EQUIVALENT TO, or simply IFF). For example, here are some of the many propositions that can be formed from the set of propositions (a) Jupiter is a star, (b) Mars has two moons, and (c) Venus is between Earth and Mercury:

(Jupiter is a star) OR (Mars has two moons)
(Jupiter is a star) AND (Jupiter is a star)
(Venus is between Earth and Mercury) AND NOT (Jupiter is a star)
IF (Jupiter is a star) THEN (Mars has two moons)

Clearly, the connectives can be regarded as *logical operators*: they take propositions as their input and return another proposition as their output. (I've used parentheses in the examples to make the scope of the connectives clear. In practice, we adopt certain precedence rules that allow us to omit many of the parentheses that might otherwise be needed. Of course, it's never wrong to include them, even when they're logically unnecessary.)

A proposition that involves no connectives at all is called a *simple* proposition; a proposition that isn't simple is called *compound*. The truth value of a compound proposition can be determined from the truth values of the constituent simple propositions in accordance with the familiar truth tables:

p	NOT p
T	F
F	T

p q	p AND q	p OR q	p IMPLIES q	p IFF q
T T	T	T	T	T
T F	F	T	F	F
F T	F	T	T	F
F F	F	F	T	T

For space reasons I've abbreviated TRUE and FALSE to just T and F, respectively. By way of example, consider the following compound proposition:

IF (Mars has two moons) THEN (Venus is between Earth and Mercury)

This proposition is of the form IF p THEN q (or p IMPLIES q). Both constituent simple propositions evaluate to TRUE; the truth table for IMPLIES therefore shows us that the overall proposition evaluates to TRUE also. However, a point arises here that sometimes causes confusion. In the real world, the fact that Venus is between Earth and Mercury obviously has nothing to do with whether Mars has two moons. In logic, however, we simply *define* IF p THEN q to be true if q is true, regardless of whether p is true; in fact, if p is false, we define IF p THEN q to be true, regardless of the truth value of q. Thus, for example, IF (Jupiter is a star) THEN (Earth has two moons) is true! You can interpret this

state of affairs as saying, in effect, that if you'll believe a falsehood (p), then you'll believe anything (q).

> **NOTE**
> If this discussion leaves you with a slightly unpleasant taste in your mouth, you're not alone. But logicians much better than I have struggled with the problem of trying to justify the fact that IF p THEN q is true if p is false and q is true. The issue isn't worth fighting over for the purposes of this appendix.

It follows from the foregoing that the proposition p IMPLIES q (or IF p THEN q) is logically equivalent to the proposition (NOT p) or q—it evaluates to FALSE if and only if p evaluates to TRUE and q to FALSE. This equivalence illustrates the point that the connectives NOT, AND, OR, IMPLIES, and IFF aren't all primitive. In fact, all possible logical connectives involving either one or two propositions can be expressed in terms of suitable combinations of NOT and either AND or OR. (*Exercise:* Check this claim.) Perhaps even more remarkably, all such connectives can in fact be expressed in terms of just one primitive. Can you find it?

Predicates

Consider the following statements:

x is a star
x has two moons
x has n moons
x is between Earth and y
x is between y and z

These statements aren't propositions, because they aren't unequivocally either true or false. And the reason they aren't is that they involve *parameters* (also known as *placeholders* or *free variables*). For example, the statement "x is a star" involves the parameter x, and we obviously can't say whether it's true or false unless we're told what that x stands for (at which point we're no longer dealing with the given statement anyway but a different one instead—see the next paragraph).

Now, we can substitute *arguments* for the parameters and thereby obtain propositions from the statements. For example, if we substitute the argument *The Sun* for the parameter x in "x is a star," we obtain "The Sun is a star." And this statement is indeed a proposition because it's unequivocally either true or false (in fact, of course, it's true). But the original statement as such ("x is a star") is, to say it again, not itself a proposition. Rather, it's a *predicate*, which—as you'll recall from Chapter 4—is a truth-valued function; that is, it's a function that, when invoked, returns a truth value. Like all functions, a predicate has a set of parameters; when it's invoked, arguments are substituted for the parameters; substituting arguments for the parameters effectively converts the predicate

into a proposition; and we say the arguments *satisfy* the predicate if and only if that proposition is true. For example, the argument *The Sun* satisfies the predicate "*x* is a star," whereas the argument *The Moon* does not.

As an aside, I remind you from Chapter 4 that logicians speak not of invoking a predicate but rather of *instantiating* it. In fact, for reasons that need not concern us here, their concept of instantiation is slightly more general than that of the familiar notion of function invocation. However, I'll stay with the terminology of invocation in this appendix.

I also remind you from Chapter 4 that a proposition can be regarded as a degenerate predicate. To be precise, it's a predicate for which the corresponding set of parameters is empty (and the function thus always returns the same result, either TRUE or FALSE, every time it's invoked). In other words, all propositions are predicates, but most predicates aren't propositions.

Now consider the example "*x* has *n* moons." Obviously, this predicate involves two parameters, *x* and *n*. Substituting the arguments *Mars* for *x* and *2* for *n* yields a true proposition; substituting the arguments *Earth* for *x* and *2* for *n* yields a false one.

Predicates can conveniently be classified according to the cardinality of their set of parameters. That is, we often speak of an *n-place predicate*, meaning a predicate with exactly *n* parameters; for example, "*x* is between *y* and *z*" is a 3-place predicate, while "*x* has *n* moons" is an 2-place predicate. A proposition is a 0-place predicate.

> **NOTE**
> An *n*-place predicate is also called an *n-adic* predicate. If $n = 1$, the predicate is said to be monadic; if $n = 2$, it's said to be dyadic.

Given a set of predicates, we can combine predicates from that set in a variety of ways to form further predicates, using the logical connectives already discussed—NOT, AND, OR, and so forth (in other words, those connectives are logical operators that operate on predicates in general, not just on the special predicates that happen to be propositions). A predicate that involves no connectives is called *simple*; a predicate that isn't simple is called *compound*. Here's a trivial example of a compound predicate:

(*x* is a star) OR (*x* is between Earth and *y*)

This predicate is also *dyadic*—not because it involves two simple predicates, but because it involves two parameters, *x* and *y*.

Quantification

I showed in the previous section that one way to get a proposition from a predicate is to invoke it with an appropriate set of arguments. But there's another way, too, and that's

by means of *quantification*. Let $p(x)$ be a monadic predicate (I show the single parameter x explicitly, for clarity). Then:

- The expression:

  ```
  EXISTS x ( p ( x ) )
  ```

  is a proposition, and it means: "There exists at least one possible argument value a corresponding to the parameter x such that $p(a)$ evaluates to TRUE" (in other words, the argument value a satisfies predicate p). For example, if p is the predicate "x is a logician," then:

  ```
  EXISTS x ( x is a logician )
  ```

  is a proposition—one that evaluates to TRUE, as it happens (for example, take a to be Bertrand Russell).

- The expression:

  ```
  FORALL x ( p ( x ) )
  ```

  is a proposition, and it means: "All possible argument values a corresponding to the parameter x are such that $p(a)$ evaluates to TRUE" (in other words, all such argument values a satisfy predicate p). For example, if again p is the predicate "x is a logician," then:

  ```
  FORALL x ( x is a logician )
  ```

  is a proposition—one that evaluates to FALSE, as it happens (for example, take a to be George W. Bush).

Observe that it's sufficient to produce a single example to show the truth of the EXISTS proposition and a single counterexample to show the falsity of the FORALL proposition. Observe too in both cases that the parameter must be constrained to "range over" some set of permissible values (the set of all people, in the example). I'll come back to this point later, in the section "Database Constraints."

The term used in logic for the expressions EXISTS x and FORALL x in the foregoing discussion is *quantifiers* (the term derives from the verb *to quantify*, which simply means *to express as a quantity*—that is, to say how much of something there is or how many somethings there are). Quantifiers of the form EXISTS ... are said to be *existential;* quantifiers of the form FORALL ... are said to be *universal.* (And in logic texts, EXISTS is usually represented by a backward E and FORALL by an upside-down A. I use the keywords EXISTS and FORALL for typographical reasons—not to mention readability.)

By way of another example, let q be the dyadic predicate "x is taller than y." If we "quantify existentially over x," we obtain:

```
EXISTS x ( x is taller than y )
```

Now, this statement is not a proposition, because it isn't unequivocally either true or false; in fact, it's a monadic predicate—it has a single parameter, y. Suppose we invoke this predicate with argument Steve. We obtain:

```
EXISTS x ( x is taller than Steve )
```

This statement *is* a proposition (and if there exists at least one person—Arnold, say—who's taller than Steve, then it evaluates to TRUE, of course). But another way to obtain a proposition from the original predicate q is to quantify over *both* of the parameters x and y. For example:

```
EXISTS x ( EXISTS y ( x is taller than y ) )
```

This statement is indeed a proposition; it evaluates to FALSE only if nobody is taller than anybody else and to TRUE otherwise (think about it!).

There are several lessons to be learned from this example:

- To obtain a proposition from an n-adic predicate by quantification, it's necessary to quantify over *every* parameter. More generally, if we quantify over m parameters ($m \leq n$), then we obtain a k-adic predicate, where $k = n\text{-}m$.
- Let's focus on existential quantification only for the moment. Then there are apparently two different propositions we can obtain in the example by "quantifying over everything":

  ```
  EXISTS x ( EXISTS y ( x is taller than y ) )
  EXISTS y ( EXISTS x ( x is taller than y ) )
  ```

  It should be clear, however, that these two propositions both mean the same thing: "There exist two persons x and y such that x is taller than y." More generally, in fact, it's easy to see that a series of like quantifiers (all existential or all universal) can be written in any sequence we choose without changing the overall meaning of the predicate or proposition in question. We therefore allow unnecessary parentheses to be dropped, as here:

  ```
  EXISTS y EXISTS x ( x is taller than y )
  ```

  By contrast, with *unlike* quantifiers, the sequence matters, in general (see the next point).
- Of course, we can use either EXISTS or FORALL in each of the necessary quantifiers when "quantifying over everything." In the following example, therefore, there are six distinct propositions that can be obtained by fully quantifying, and I've listed them below. (Actually there are eight, but two of those eight can be eliminated by virtue of the previous point.) I've also shown a precise natural-language interpretation in each case. Note that those interpretations are all logically different! Please note too, however, that I've had to assume in connection with those interpretations that there do exist at least two people "in the universe," as it were. I'll come back to this assumption later (in a discussion of *empty ranges* in the section "More on Quantification").

  ```
  EXISTS x ( EXISTS y ( x is taller than y ) )
  ```

  Meaning: Somebody is taller than somebody else; TRUE, unless everybody is the same height.

  ```
  EXISTS x ( FORALL y ( x is taller than y ) )
  ```

  Meaning: Somebody is taller than everybody (that particular somebody included!); clearly FALSE.

```
FORALL x ( EXISTS y ( x is taller than y ) )
```
Meaning: Everybody is taller than somebody; clearly FALSE.

```
EXISTS y ( FORALL x ( x is taller than y ) )
```
Meaning: Somebody is shorter than everybody (that particular somebody included!); clearly FALSE.

```
FORALL y ( EXISTS x ( x is taller than y ) )
```
Meaning: Everybody is shorter than somebody; clearly FALSE.

```
FORALL x ( FORALL y ( x is taller than y ) )
```
Meaning: Everybody is taller than everybody; clearly FALSE.

- Finally (at the risk of belaboring the obvious), even though five out of six of the foregoing propositions all evaluate to the same truth value, FALSE, it doesn't follow that they all mean the same thing, and indeed they don't; in fact, no two of them do.

Free and Bound Variables

I said earlier that another term for what I've been calling parameters is *free variables*. Quantifying over a free variable converts it into a *bound variable*. For example, consider again the 2-place predicate *q* from the previous section:

```
x is taller than y
```

The parameters *x* and *y* here are free variables. If we now "quantify existentially" over *x*, we obtain:

```
EXISTS x ( x is taller than y )
```

Here, *y* is still free, but *x* is now bound. And in the "fully quantified" form:

```
EXISTS x EXISTS y ( x is taller than y )
```

x and *y* are both bound, and there are no free variables at all (the predicate has degenerated to a proposition).

Now, we already know that free variables correspond to parameters, in conventional programming terms. Bound variables, by contrast, don't have an exact counterpart in conventional programming terms at all; instead, they serve as a kind of *dummy*—they serve only to link the predicate inside the parentheses to the quantifier outside. For example, consider the simple predicate (actually a proposition):

```
EXISTS x ( x > 3 )
```

This proposition merely asserts that there exists some integer greater than three. (I'm assuming that *x* here is constrained to "range over" the set of integers. Again, this is a point I'll come back to in the section "Database Constraints.") *Note, therefore, that the meaning of the proposition would remain totally unchanged if the two x's were both replaced by some other variable y.* In other words, the proposition:

```
EXISTS y ( y > 3 )
```

is semantically identical to the one just shown.

Now consider the predicate:

```
EXISTS x ( x > 3 ) AND x < 0
```

Here there are three *x*'s, *but they don't all denote the same thing*. The first two are bound and can be replaced by, say, *y* without changing the overall meaning; but the third is free and cannot be replaced with impunity. Thus, of the following two predicates, the first is equivalent to the one just shown and the second is not:

```
EXISTS y ( y > 3 ) AND x < 0
EXISTS y ( y > 3 ) AND y < 0
```

To close this section, I remark that we can now (re)define a proposition to be a predicate in which all of the variables are bound.

More on Quantification

This section discusses a number of miscellaneous issues regarding quantification.

We Don't Need Both Quantifiers

You might already know that we don't actually need both EXISTS and FORALL. The reason is that any statement that can be expressed in terms of EXISTS can always be expressed in terms of FORALL instead and vice versa. By way of example, consider again the predicate:

```
EXISTS x ( x is taller than Steve )
```

("Somebody is taller than Steve"). Another way to say the same thing is:

```
NOT ( FORALL x ( NOT ( x is taller than Steve ) ) )
```

("It is not the case that nobody is taller than Steve"). More generally, in fact, the statement:

```
EXISTS x ( p ( x ) )
```

is logically equivalent to the following statement:

```
NOT ( FORALL x ( NOT ( p ( x ) ) )
```

(where the predicate *p* might legitimately involve other parameters in addition to *x*). Likewise, the statement:

```
FORALL x ( p ( x ) )
```

is logically equivalent to the statement:

```
NOT ( EXISTS x ( NOT ( p ( x ) ) )
```

(where, again, the predicate *p* might legitimately involve other parameters in addition to *x*).

It follows from the foregoing that a formal language need not explicitly support both EXISTS and FORALL. But it's very desirable to support both in practice. The reason is that some problems are "more naturally" formulated in terms of EXISTS and others are "more naturally" formulated in terms of FORALL. For example, SQL supports EXISTS but not FORALL; as a consequence, certain queries are quite awkward to formulate in SQL. For example, consider the query "Get suppliers who supply all parts," which can be expressed in logic quite simply as follows:

```
s WHERE FORALL p EXISTS sp ( s.SNO = sp.SNO AND sp.PNO = p.PNO )
```

("Get suppliers *s* such that, for all parts *p*, there exists a shipment *sp* linking that supplier *s* to that part *p*"; the variables *s*, *p*, and *sp* are to be understood as ranging over the current values of relvars S, P, and SP, respectively). In SQL, by contrast, the query looks something like this:

```
SELECT s.*
FROM   S AS s
WHERE  NOT EXISTS
     ( SELECT p.*
       FROM   P AS p
       WHERE  NOT EXISTS
            ( SELECT sp.*
              FROM   SP AS sp
              WHERE  s.SNO = sp.SNO
              AND    sp.PNO = p.PNO ) )
```

("Get suppliers *s* such that there does not exist a part *p* such that there does not exist a shipment *sp* linking that supplier *s* to that part *p*"). Well, *single* negation is bad enough (many users have difficulty with it); *double* negation, as in this SQL query, is much worse!

Empty Ranges

Consider again the fact that the statements:

```
EXISTS x ( p ( x ) )
```

and:

```
NOT ( FORALL x ( NOT ( p ( x ) ) )
```

are logically equivalent. As I've indicated several times already, the bound variable *x* in each of these statements must "range over" some set of permissible values. Suppose now that the set in question is empty (it might, for example, be the set of people over 50 feet tall). Then:

- The quantifier EXISTS *x* clearly evaluates to FALSE (because no such *x* exists).
- The statement EXISTS *x* (*p*(*x*))—"There exists an *x* that makes *p*(*x*) evaluate to TRUE"—thus evaluates to FALSE as well, *regardless of what p(x) happens to be*. For example, the statement "There exists a person over 50 feet tall who works for IBM" evaluates to FALSE (unsurprisingly).

- It follows that the negation NOT EXISTS x ($p(x)$), which is equivalent to the statement FORALL x (NOT ($p(x)$), evaluates to TRUE—again, regardless of what $p(x)$ happens to be. For example, the statement "All persons over 50 feet tall don't work for IBM"—or, more idiomatically, "No person over 50 feet tall works for IBM"—evaluates to TRUE (again unsurprisingly).
- But if the predicate $p(x)$ is arbitrary, then so is the predicate NOT ($p(x)$). And therefore we have the following possibly surprising result: the statement FORALL x (...) evaluates to TRUE if there are no x's, *regardless of what appears inside the parentheses.* For example, the statement "All persons over 50 feet tall do work for IBM" also evaluates to TRUE(!)—because, to say it again, there aren't any persons over 50 feet tall.

One implication of this state of affairs is that certain queries produce a result that you might not expect (if you don't know logic, that is). For example, the following query:

```
s WHERE FORALL p
      ( IF p.COLOR = 'Purple'
        THEN EXISTS sp ( s.SNO = sp.SNO AND sp.PNO = p.PNO ) )
```

("Get suppliers who supply all purple parts") will return all suppliers if there aren't any purple parts. *Exercise*: Show an SQL formulation of this query.

Defining EXISTS and FORALL

You might have already realized that EXISTS and FORALL can be defined as an *iterated OR* and an *iterated AND*, respectively. I'll consider EXISTS first. Let $p(x)$ be a predicate with a parameter x and let x range over the set $X = \{x1, x2, \ldots, xn\}$. Then:

```
EXISTS x ( p ( x ) )
```

is a predicate, and it's defined to be equivalent to (and hence shorthand for) the predicate:

```
FALSE OR p ( x1 ) OR p ( x2 ) OR ... OR p ( xn )
```

Observe in particular that this expression evaluates to FALSE if X is empty (as we already know). By way of example, let $p(x)$ be "x has a moon" and let X be the set {Mercury, Venus, Earth, Mars}. Then the predicate EXISTS x ($p(x)$) becomes "EXISTS x (x has a moon)," and it's shorthand for:

```
FALSE OR ( Mercury has a moon ) OR ( Venus has a moon )
      OR ( Earth has a moon )   OR ( Mars has a moon )
```

which evaluates to TRUE because (for example) "Mars has a moon" is true. Similarly:

```
FORALL x ( p ( x ) )
```

is a predicate, and it's defined to be equivalent to (and hence shorthand for) the predicate:

```
TRUE AND p ( x1 ) AND p ( x2 ) AND ... AND p ( xn )
```

And this expression evaluates to TRUE if X is empty (again, as we already know). By way of example, let $p(x)$ and X be as in the EXISTS example above. Then the predicate FORALL x ($p(x)$) becomes "FORALL x (x has a moon)," and it's shorthand for:

```
TRUE AND ( Mercury has a moon ) AND ( Venus has a moon )
     AND ( Earth has a moon )   AND ( Mars has a moon )
```

which evaluates to FALSE because (for example) "Venus has a moon" is false.

As an aside, let me remark that—as the examples clearly demonstrate—defining EXISTS and FORALL as iterated OR and AND, respectively, means that every predicate that involves quantification is equivalent to one that doesn't. Thus, you might be wondering, not without some justification, just what this business of quantification is really all about. Why all the fuss? The answer is as follows: we can define EXISTS and FORALL as iterated OR and AND *only because the sets we have to deal with are—thankfully—always finite* (because we're operating in the realm of computers, and computers in turn are finite, of course). In pure predicate logic, where there's no such restriction, those definitions aren't valid.

Perhaps I should add that, even though we're always dealing with finite sets and EXISTS and FORALL are thus merely shorthand, they're extremely *useful* shorthand! For my part, I certainly wouldn't want to have to formulate queries and the like purely in terms of AND and OR, without being able to use the quantifiers.

Other Kinds of Quantifiers

While it's true that EXISTS and FORALL are the most important quantifiers in practice, they aren't the only ones possible. There's no *a priori* reason, for example, why we shouldn't allow quantifiers of the form:

```
there exist at least three x's such that
```

or:

```
a majority of x's are such that
```

or:

```
an odd number of x's are such that
```

(and so on). One fairly important special case is *there exists exactly one x such that.* I'll use the keyword UNIQUE for this one. Here are some examples:

`UNIQUE x ( x is taller than Arnold )`
Meaning: Exactly one person is taller than Arnold; probably FALSE.

`UNIQUE x ( x has social security number y )`
Meaning: Exactly one person has social security number y (y is a parameter, of course). We can't assign a truth value to this example, because it's a predicate, not a proposition.

`FORALL y UNIQUE x ( x has social security number y )`
Meaning: Everybody has a unique social security number. (I'm assuming here that y ranges over the set of all social security numbers actually assigned, not all possible

ones. Incidentally, does this predicate—which is in fact a proposition, of course—evaluate to TRUE?)

As another exercise, what does the following predicate mean?

```
FORALL x UNIQUE y ( x has social security number y )
```

Database Constraints

Now let me show how the ideas I've been discussing can be used in the formulation of database constraints, using examples from Chapter 6. Please note that the formulations shown aren't the only ones possible, in general.

Example 1

Status values must be in the range 1 to 100 inclusive.

```
CONSTRAINT C1
   FORALL s ( IF s ∈ S THEN s.STATUS ≥ 0 AND s.STATUS ≤ 100 ) ;
```

("For all possible values *s*, if *s* is a tuple in relvar S, then the STATUS value within that tuple must be in the specified range"; note the dot-qualified attribute references, and recall from Chapter 5 that "*s* ∈ S" can be read as "*s* [is] in S").

Now, I'm sure you can see that many constraints are going to involve a proposition of the general form just illustrated:

```
FORALL t ( IF t ∈ R THEN p )
```

(where *p* is some predicate). We can simplify such propositions slightly by defining the following as a shorthand:

```
FORALL t ∈ R ( p )
```

The bound variable *t* here is said to *range over* (the current value of) relvar *R*, and the expression FORALL *t* ∈ *R* is said to be a *range-coupled* quantifier. Using this shorthand, constraint C1 becomes:

```
CONSTRAINT C1
   FORALL s ∈ S ( s.STATUS ≥ 0 AND s.STATUS ≤ 100 ) ;
```

In the same kind of way, it's helpful to define the expression:

```
EXISTS t ∈ R ( p )
```

to be shorthand for the following:

```
EXISTS t ( ( t ∈ R ) AND ( p ) )
```

In fact, we can (and I will) simplify matters still further, by allowing the ranges for bound variables to be defined external to the expressions that use those bound variables. For example:

```
s RANGES OVER { S }
```

Now I can reduce constraint C1 to just:*

```
CONSTRAINT C1
   FORALL s ( s.STATUS ≥ 0 AND s.STATUS ≤ 100 ) ;
```

For the remainder of this section (and the next), I'll assume the following range definitions are in effect:

```
s  RANGES OVER { S }
x  RANGES OVER { S }
y  RANGES OVER { S }
p  RANGES OVER { P }
sp RANGES OVER { SP }
```

I'll also introduce additional bound variables as and when necessary.

Example 2

Suppliers in London must have status 20.

```
CONSTRAINT C2
   FORALL s ( IF s.CITY = 'London' THEN s.STATUS = 20 ) ;
```

Example 3

Supplier numbers are unique.

```
CONSTRAINT C3
   FORALL x, y ( IF x.SNO = y.SNO THEN x.SNAME  = y.SNAME
                                  AND  x.STATUS = y.STATUS
                                  AND  x.CITY   = y.CITY ) ;
```

The quantifier FORALL *x,y* is shorthand for FORALL *x* FORALL *y*.

Example 4

No supplier with status less than 20 can supply part P6.

```
CONSTRAINT C4
   FORALL s ( IF s.STATUS < 20 THEN
      FORALL sp ( IF sp.SNO = s.SNO THEN
                     sp.PNO ≠ PNO('P6') ) ) ;
```

Example 5

Every shipment must have a supplier.

```
CONSTRAINT C5
   FORALL sp ( EXISTS s ( sp.SNO = s.SNO ) ) ;
```

Example 6

No supplier number appears in both relvar LS and relvar NLS.

```
CONSTRAINT C6
   FORALL m ∈ LS FORALL n ∈ NLS ( m.SNO ≠ n.SNO ) ;
```

* SQL allows a simple constraint like this one to be reduced still further by (in effect) eliding the quantifier. To be precise, we can specify a "CHECK constraint" as part of the definition of base table S that looks like this: CONSTRAINT C1 CHECK (STATUS >= 0 AND STATUS <= 100). We can even elide that "CONSTRAINT C1" as well, if we want.

Example 7

Supplier S1 and part P1 must never be in different cities.

```
CONSTRAINT C7
   NOT ( EXISTS s EXISTS p ( s.SNO = SNO('S1') AND
                             p.PNO = PNO('P1') AND
                             s.CITY ≠ p.CITY ) ) ;
```

I'll show a few more examples (not based on examples in Chapter 6) in order to illustrate some additional points.

Example 8

There must always be at least one supplier.

```
CONSTRAINT C8 EXISTS s ( TRUE ) ;
```

Example 9

All parts are either blue or green, and there exists at least one part.

```
CONSTRAINT C9
   FORALL p ( p.COLOR = 'Blue' OR p.COLOR = 'Green' ) AND
   EXISTS p ( TRUE ) ;
```

Note the use of two distinct bound variables here, both called *p*.

Example 10

All parts are either blue or green, or there exists at least one red part.

```
CONSTRAINT C10
   FORALL p ( p.COLOR = 'Blue' OR p.COLOR = 'Green' ) OR
   EXISTS p ( p.COLOR = 'Red' ) ;
```

Queries

In this section I'll show how the calculus notation can be used to formulate queries, using examples from Chapter 5. Again, please note that the formulations shown aren't the only ones possible, in general.

Example 1

Get suppliers in Paris.

```
s WHERE s.CITY = 'Paris'
```

Example 2

Get suppliers who supply at least one part.

```
s WHERE EXISTS sp ( sp.SNO = s.SNO )
```

Example 3

Get all supplier name/status/city combinations.

```
s.SNAME, s.STATUS, s.CITY
```

We could harmlessly add WHERE TRUE to the foregoing formulation.

Example 4

Get the join of suppliers and shipments.

```
s, sp.PNO, sp.QTY WHERE s.SNO = sp.SNO
```

Example 5

Get the union of supplier cities and part cities.

```
u RANGES OVER { s.CITY, p.CITY }
u
```

Bound variable *u* is defined to range over the union of supplier cities and part cities.

Example 6

Get suppliers who supply no parts at all.

```
s WHERE NOT ( EXISTS sp ( sp.SNO = s.SNO ) )
```

Example 7

Get parts with their weight in grams.

```
p, ( p.WEIGHT * 454 ) AS GMWT
```

Example 8

For each supplier, get the supplier number and a count of the number of parts that supplier supplies.

```
s.SNO, COUNT ( sp WHERE sp.PNO = s.SNO ) AS P_COUNT
```

Some Equivalences

I'll finish up this appendix with a few remarks on certain equivalences that might have already occurred to you. First, recall the IS_EMPTY operator, which I introduced in Chapter 5 and made much use of in Chapter 6. Well, if the system supports that operator, there's no logical need for it to support the quantifiers, thanks to the following equivalences:

```
EXISTS x ( p )  ≡  NOT ( IS_EMPTY ( x WHERE p ) )
```

and:

```
FORALL x ( p )  ≡  IS_EMPTY ( x WHERE NOT ( p ) )
```

In fact, the SQL support for quantifiers, such as it is, is based on exactly the foregoing equivalences. To be specific, SQL supports an EXISTS *operator*—not really a quantifier as such, because it doesn't involve any bound variables—that's defined as follows: the operator invocation EXISTS(*tx*), where *tx* is a table expression, returns TRUE if and only if the table *t* denoted by the expression *tx* is nonempty.* By contrast, SQL doesn't support a FORALL operator, because (a) we don't need both EXISTS and FORALL, and (b) more to the point, syntax of the form FORALL(*tx*) where *tx* is a table expression really makes

* There's a certain irony here, though. As we saw in Chapter 3, because SQL supports nulls, it's based on what's called *three-valued logic* (instead of the conventional two-valued logic the relational model is based on). In three-valued logic, the existential quantifier can return three different results: TRUE, FALSE, and what's usually called UNKNOWN (as indeed it is, in SQL). But SQL's EXISTS operator always returns TRUE or FALSE, never UNKNOWN. As a consequence, (a) SQL's EXISTS is not a faithful implementation of the existential quantifier of three-valued logic, and (b) once again, SQL queries sometimes return the wrong answer.

no sense. For example, what could an expression like FORALL (SELECT * FROM S) possibly mean?

In a similar fashion, we don't absolutely need the quantifiers anyway if the system supports the aggregate operator COUNT. This is because of the following equivalences:

```
EXISTS x ( p )  ≡  COUNT ( x WHERE p ) > 0
```

and:

```
FORALL x ( p )  ≡  COUNT ( x WHERE p ) = COUNT ( x )
```

Now, I'm certainly not a fan of the idea of replacing quantified expressions by expressions involving COUNT invocations—though sometimes we have to, if we're in a pure algebraic framework—but it would be wrong of me not to mention the possibility.

Summary

In the introduction to this appendix, I said it was my position that database professionals should have at least a nodding acquaintance with the basic concepts of relational calculus (or predicate logic—it comes to the same thing). I'd like to conclude by trying to justify that position.

My basic point is simply that a knowledge of logic helps you think precisely (and in our field, the importance of thinking precisely is surely paramount). In particular, it forces you to appreciate the significance of proper quantification. Natural language is so often imprecise; however, careful consideration of what quantification is needed allows you to pin down the meaning of what can otherwise be very imprecise natural-language statements. By way of example, you might like to meditate on *exactly* what Abraham Lincoln meant—or might have meant, or thought he might have meant, or might have thought he meant—when he famously said: "You can fool some of the people some of the time, and some of the people all the time, but you cannot fool all the people all of the time."

Of course, I'm well aware there are many people who disagree with me here; that is, there are many people who feel ordinary mortals shouldn't have to grapple with a subject as abstruse as logic seems to be. In effect, they claim that logic is just too difficult for most people to deal with. Now, that claim might be true in general (logic is a big subject), but you don't need to understand the whole of logic for the purpose at hand; in fact, I doubt whether you need much more than what I've covered in this appendix. And the benefits are so huge! I made essentially the same point in a 2004 article—"The Logic of Business Rules," *www.dbdebunk.com* (June 2004), on which much of the present appendix is based—and I'd like to quote the concluding remarks from that article here:

> Surely it's worth investing a little effort up front in becoming familiar with [the material in this appendix] in order to avoid the problems associated with [business

rules that are stated ambiguously]. Ambiguity in business rules leads to implementation delays at best or implementation errors at worst (possibly both). And such delays and errors certainly have costs associated with them, costs that are likely to outweigh those initial learning costs many times over. In other words, framing business rules properly is a serious matter, and it requires a certain level of technical competence.

As you can see, these remarks are set in the context of business rules specifically, but I think they're of wider applicability.

APPENDIX B

Suggestions for Further Reading

This appendix gives a short list of suggestions for further reading. I apologize for the fact that the majority of the references are to publications for which I'm either the author or a coauthor.... They're in chronological order by date of first publication.

Robert R. Stoll: *Sets, Logic, and Axiomatic Theories.* San Francisco, Calif.: W. H. Freeman and Company (1961).
The relational model is, of course, solidly founded on logic and set theory. This book provides a fairly formal but not too difficult introduction to these topics. See also the book by Hodges, later.

E. F. Codd: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19th, 1969); "A Relational Model of Data for Large Shared Data Banks," CACM 13, No. 6 (June 1970).
The 1969 paper was Codd's very first paper on the relational model; essentially it's a preliminary version of the 1970 paper, with a few interesting differences. That 1970 paper—which was republished in "Milestones of Research," *CACM 26*, No. 1 (January 1982) and elsewhere—was the first widely available paper on the subject. It's usually credited with being the seminal paper in the field, though that characterization is a little unfair to its 1969 predecessor. In my strong opinion, every database professional should read one or

both of these papers every year. *Exercise:* This latter sentence (not counting that initial "In my strong opinion," of course) is a business rule! Try casting it into the formalism of Appendix A.

C. J. Date: *An Introduction to Database Systems,* Eighth Edition. Boston, Mass.: Addison-Wesley (2004). The first edition of this book was published in 1975.
A college-level text on all aspects of database management. SQL discussions are at the SQL:1999 level, with a few comments on SQL:2003.

Patrick Hall, Peter Hitchcock, and Stephen 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).
This paper is perhaps a little "difficult," but I think it's important. In particular, the version of the relational algebra I've described in this book has its roots in this paper.

Wilfrid Hodges: *Logic.* London, England: Penguin Books (1977).
This book is a very gentle introduction to logic for the uninitiated.

C. J. Date and Hugh Darwen: *A Guide to the SQL Standard,* Fourth Edition. Reading, Mass.: Addison-Wesley (1997). The first edition of this book was published in 1987.
A complete tutorial reference and critical guide to the SQL standard as of 1997. Although that version of the standard has since been superseded, first by SQL:1999 and then by the current standard SQL:2003, just about everything in the book is still accurate and applicable today.

Jim Gray and Andreas Reuter: *Transaction Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann (1993).
The standard text on transaction management.

E. F. Codd and C. J. Date: "Much Ado about Nothing," in C. J. Date, *Relational Database Writings 1991-1994.* Reading, Mass.: Addison-Wesley (1995). The article was also published on the web site *www.dbdebunk.com* (February 2005).
Codd was perhaps the foremost advocate of nulls and 3VL as a basis for dealing with missing information (a curious state of affairs, you might think, given that nulls violate Codd's own *Information Principle*). This article contains the text of a debate between Codd and myself on the subject. It includes the following delightful remark: "Database management would be easier if missing values didn't exist" (Codd). *Note:* I've included this particular reference—out of a huge number of available publications on the topic—because it at least touches on most of the arguments on both sides of the issue.

Hugh Darwen: "What a Database *Really* Is: Predicates and Propositions," in C. J. Date, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).
A very approachable tutorial on relvar predicates and related matters.

C. J. Date and Hugh Darwen: ***Databases, Types, and the Relational Model: The Third Manifesto,*** **Third Edition. Reading, Mass.: Addison-Wesley (2006). The first edition of this book appeared (under a different title) in 1998.**
An advanced (graduate-level?) text on database technology. It includes a comprehensive definition of the language Tutorial D, which I use as a basis for examples throughout the body of the present book. Don't be fooled by that 2006 copyright date, by the way; the book is actually due to be published in 2005.

Fabian Pascal: ***Practical Issues in Database Management: A Reference for the Thinking Practitioner.*** **Boston, Mass.: Addison-Wesley (2000).**
This book focuses on a number of common database issues—normalization, redundancy, integrity, missing information, and others—and discusses theoretically correct approaches to those issues, with the emphasis on the practical application of that theory.

C. J. Date: ***The Database Relational Model: A Retrospective Review and Analysis.*** **Reading, Mass.: Addison-Wesley (2001).**
An unbiased retrospective review and analysis of Codd's relational contribution as documented in the papers he published in the period 1969–1979.

C. J. Date: "Double Trouble, Double Trouble" (in two parts), ***www.dbdebunk.com*** **(April 2002).**
A detailed treatment of the problems caused by duplicates. The discussion of duplicates in Chapter 3 is based on an example from this paper.

Jim Melton and Alan R. Simon: ***SQL:1999—Understanding Relational Components;*** **Jim Melton:** ***Advanced SQL:1999—Understanding Object-Relational and Other Advanced Features.*** **San Francisco, Calif.: Morgan Kaufmann (2002 and 2003, respectively).**
So far as I know, these two books are the only comprehensive tutorials available on all aspects of SQL:1999 (the immediate predecessor of the current standard SQL:2003). Melton was the editor of the SQL standard for many years.

C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: ***Temporal Data and the Relational Model.*** **San Francisco, Calif.: Morgan Kaufmann (2003).**
Some indication of what this book covers can be found in Chapter 8 of the present book.

Hugh Darwen: "How to Handle Missing Information Without Using Nulls" (presentation slides), ***www.thethirdmanifesto.com*** **(May 9th, 2003).**
In Chapter 7 of the present book, I briefly touched on a technique for representing missing information without using nulls. This presentation elaborates on another such technique, similar but not identical to the one mentioned in Chapter 7.

C. J. Date: "What First Normal Form Really Means," ***www.dbdebunk.com*** **(June 2003).**
First normal form has been the subject of much misunderstanding over the years. This paper is an attempt to set the record straight—even to be definitive, as far as possible.

The crux of the argument, as indicated in Chapter 2 of the present book, is that the concept of *atomicity* (in terms of which first normal form was originally defined) has no absolute meaning.

C. J. Date: "A Sweet Disorder," *www.dbdebunk.com* (August 2003).
Relations don't have a left-to-right ordering to their attributes, but SQL tables do have such an ordering to their columns. This paper explores some of the consequences of this state of affairs, which turn out to be rather less trivial than you might think.

C. J. Date: "On the Notion of Logical Difference," "On the Logical Difference Between Model and Implementation," and "On the Logical Differences Between Types, Values, and Variables" (in two parts), *www.dbdebunk.com* (July 2004).
The titles say it all.

C. J. Date: "Data Redundancy and Database Design" (in three parts), *www.dbdebunk.com* (March/April/May 2005).
A discussion of the theory (such as it is) of database design, with the emphasis on reducing data redundancy. In particular, the paper includes a detailed treatment of *The Principle of Orthogonal Design.*

C. J. Date: *Go Faster! The TransRelational™ Approach to DBMS Implementation*. To appear.
A detailed tutorial on *The TransRelational™ Model* (see Chapter 7 of the present book).

Numbers

0-tuple, 44
1NF (see first normal form)
2NF (see second normal form)
2VL (see two-valued logic)
3NF (see third normal form)
3VL (see three-valued logic)
4NF (see fourth normal form)
5NF (see fifth normal form)
6NF (see sixth normal form)

A

Abbey, Edward, 73
ACID properties, 122
aggregate operators, 98, 111
  versus "summaries", 98
Aristotle, 169
arity, 14
assignment, 32
  multiple (see multiple assignment)
  relational (see relational assignment)
  tuple, 35
Assignment Principle, The, 38, 106, 169
  SQL violations, 165
associativity, 103
atomicity
  scalar value, 29, 34
  transaction, 122
attribute
  attribute-name:type-name pair, 14, 42
  extracting value from tuple, 44
  multi-valued, 30
  pictured as column, 5
  relation, 45
  relation-valued (see relation-valued attribute)
  relvar, 21
  tuple, 42
attribute constraint, 119
attribute ordering (left to right), 22
  in SQL tables, 15
  not in relations, 15, 200
axiom (database), 76

B

bag, 32
base relation, 16
base relvar, 61
  (see also base relation)
BCNF (see Boyce/Codd normal form)
binary relation, 5
binary tuple, 42
body
  relation, 14, 45
  relvar, 21
BOOLEAN
  required in relational model, 164
bound variable, 74, 187
Boyce/Codd normal form, 140
built-in type, 23, 32
Bush, George W., 185
business rules, 115

C

candidate key, 5, 63
cardinality
  relation, 14, 45
  relvar, 21
Carroll, Lewis, 57
cartesian product (see product)
CASCADE (referential action), 66
Celko, Joe, 169
CHAR (SQL versus **Tutorial D**), 23
Chudnovsky, Gregory, 171
Closed World Assumption, The, 73, 168
closure, 8, 81–85, 110
CODASYL, 169
coercion, 27, 28, 106
column (see attribute)
commutativity, 102
connection trap, 149
connective, 181
consistency, 122
  versus correctness, 128
CONSTRAINT, 119
constraint, 115–132
  attribute, 119
  database (see database constraint)

constraint (*continued*)
relvar (see relvar constraint)
transition, 130
tuple, 119
type (see type constraint)
versus performance, 130
versus predicate, 127
versus proposition, 129
correctness (see consistency)
correlation name (SQL), 83
cost-based optimizing, 100

D

da Vinci, Leonardo, 4
Darwen, Hugh, xiv, xv, 19, 71, 100, 147, 160, 167, 171, 173, 174, 200
data independence, 13, 17, 135, 157
data model, 12
data type (see type)
database
collection of propositions, 72
logical system, 76
versus DBMS, 24
database administrator, 25
database constraint, 115, 119–121
checked immediately, 121–126
the (total) database constraint, 128
database management system, 24
versus database, 24
database statistics, 100
Date, C. J., 70, 71, 196, 200–202
DBA, 25
DBMS, 24
DCO, 25
declarative, 21
DEE (see TABLE_DEE)
degree
relation, 14, 45
relvar, 21
tuple, 42
DELETE
expansion, 107
via cursor (SQL), 62
denormalization, 152
dependency, 138
(see also FD; JD; MVD)
dependency preservation, 150
derived relation, 16
derived relvar (see snapshot; view)
difference, 10, 91
(see also semidifference)
direct image, 157
disjoint union (D_UNION), 90
distributivity, 102
divide, 10, 92
DIVIDEBY, 92
domain, 5, 24, 24–31
(see also type)
"domain check override", 25
double underlining, 6
DUM (see TABLE_DEE)
duplicates, 22, 59, 200
in SQL tables, 15
not in relations, 14, 48–52
(see also tuple equality)
durability, 122
dyadic predicate, 184

E

E/R modeling, 135, 162
empty heading, 56
empty key, 79
empty range, 189
empty relation, 47
empty tuple, 44
empty type, 38
entity integrity, 7, 69
entity/relationship modeling, 135, 162
equality, 32, 164
relation (see relation equality)
tuple (see tuple equality)
equijoin, 94
existential quantifier, 185
EXISTS, 184
iterated OR, 190
SQL, 188, 196
(see also existential quantifier)
expression transformation, 48, 100
expression versus statement, 25, 38
EXTEND, 95
Extensible Markup Language (see XML)
extension versus intension, 73

F

Fagin's theorem, 161
Fagin, Ron, 142, 161
Faroult, Stephane, xvi
FD, 64, 138
implied by superkey, 144
trivial, 139
(see also Boyce/Codd normal form)
fifth normal form, 144–150
first normal form, 16, 29, 201
relvar, 143
"flat relation", 47
FORALL, 185
iterated AND, 190
not in SQL, 188
(see also universal quantifier)
FOREIGN KEY, 61, 121
foreign key, 6, 65
not fundamental, 67

fourth normal form, 149
free variable, 113, 114, 183
(see also bound variable)
function (SQL), 3
functional dependency (see FD)

G

generated type, 35
Gennick, Jonathan, 53
Golden Rule, The, 128, 169
Gray, Jim, 122, 200
GROUP, 100

H

Hall, Patrick, 200
heading
relation, 14, 45
relvar, 21
tuple, 42
Heath's theorem, 142
Heath, Ian, 142
hierarchic model, 143, 169
Hitchcock, Peter, 200
Hodges, Wilfrid, 200

I

idempotence, 112
identity projection, 145
implementation, 12
versus model, 12
IMS, 143
Information Principle, The, 36, 70, 167, 168, 200
INSERT expansion, 107
instantiation, 72, 183
integrity constraint (see constraint)
intended interpretation, 72
intension, 72
interpretation (see intended interpretation)
INTERSECT, 88–90
intersect, 10, 88
irreducibility
key, 63
relvar, 147
IS_EMPTY, 105, 195
isolation, 122, 123

J

JD, 144
implied by superkeys, 144
trivial, 145
(see also fifth normal form; sixth normal form)
JOIN, 87
join, 10, 87
(see also equijoin; natural join; semijoin; theta-join)
n-adic, 89
join dependency (see JD)

K

KEY, 35, 120
key (see candidate key; see also foreign key; primary key; surrogate)
values are tuples, 64
Knuth, Donald E., 166

L

Lincoln, Abraham, 196
literal, 22, 58, 117
(see also selector)
logical system, 76
Lorentzos, Nikos A., xv, 147, 174
lossless decomposition (see nonloss decomposition)
lossy decomposition, 141, 160

M

MATCHING, 88
"materialized view", 72
Melton, Jim, 201
method (SQL), 3
MINUS, 91
model, 12
versus implementation, 12
monadic predicate, 184
multi-dimensional database, 47
multiple assignment, 126
multi-relvar constraint, 121
multiset, 32
multi-valued dependency (see MVD)
MVD, 149
implied by superkey, 161
trivial, 161
(see also fourth normal form)

N

n-adic predicate, 184
n-ary relation, 5
n-ary tuple, 42
natural join, 10, 87
network model, 169
nonloss decomposition, 141
redundancy removal, 146
nonscalar, 34
normal form, 137
normalization, 137–153
goals, 150
principles, 169

normalized (see first normal form)
NOT MATCHING, 91
n-place predicate, 184
n-tuple (see tuple)
null, 7, 200
  not a value, 7
  not in a relational model, 163
  not in relations, 52–56
  not in tuples, 43

O

object, 21
object ID, 69
object/relational, 32, 173
object-oriented model, 169
OPERATOR, 33
operator (SQL), 3
optimizer, 48
ORDER, 108
ORDER BY, 15, 108
  not in views, 108
ordering (see attribute ordering; ORDER; ORDER BY; tuple ordering)
orthogonal decomposition, 156
orthogonality, 153–156

P

Pascal, Fabian, 200
performance
  not a model issue, 12
  versus constraint, 130
physical design, 156
physical representation hidden, 26, 33
physical storage
  not in relational model, 17
  (see also physical design)
PJ/NF (see fifth normal form)
pointers not in relational model, 37, 69
possible representation, 115–118
POSSREP, 115
"possrep", 116
predicate, 72, 183
  compound, 184
  dyadic, 184
  monadic, 184
  n-adic, 184
  n-place, 184
  relvar (see relvar predicate)
  simple, 184
  versus constraint, 127
primary key, 6
primitive operators, 93
Principle of Identity of Indiscernibles, The, 169
Principle of Interchangeability, The, 70
Principle of Orthogonal Design, The, 202
principles of normalization (see normalization)
procedural, 21
procedure (SQL), 3
product, 10, 88, 91
project, 9, 86
projection-join normal form (see fifth normal form)
proper subset, 63, 103
proper superset, 63, 103
proposition, 72, 181
  compound, 182
  simple, 182
  versus constraint, 129
  versus predicate, 74

Q

quantifier, 74, 184–191
  don't need both, 188
  existential, 185
  other kinds, 191
  range-coupled, 192
  sequence, 186
  universal, 185
  (see also EXISTS; FORALL)
query rewrite, 48, 100
quota query, 114

R

referenced relvar, 65
referencing relvar, 65
referential action, 66
referential constraint (see foreign key)
referential integrity, 7
  metaconstraint, 129
refresh (see snapshot)
relation, 5, 45
  pictured as table, 4, 45, 47
  versus relvar, 18
  versus type, 75
  (see also relvar)
RELATION {*H*}, 46
relation constant, 71
relation equality, 103
relation selector, 46
relation type, 46
  inference, 84
  name, 46
RELATION type generator, 35
relation value (see relation)
relation variable (see relvar)
relational algebra, 8, 81
relational assignment, 8, 9, 106
  not in SQL, 19
relational calculus, 10, 181

relational comparisons, 103
relational completeness, 167
relational database (see database; *Information Principle, The*)
relational model
  formal definition, 164
  informal definition, 4–10
  objectives, 168
  versus other models, 169–172
relations, tuples, and attributes, 2
relation-valued attribute, 32, 99, 143
relcon, 71
relvar, 18
  base versus stored, 17
  versus file, 72
  versus relation, 18
  versus type, 75
  virtual (see view)
  (see also relation)
relvar constraint, 121
  the (total) relvar constraint, 128
relvar predicate, 72
relvar reference, 81
RENAME, 84
repeating group, 32
representation versus type
  physical (see physical representation hidden)
  possible (see possible representation)
restrict, 9, 86
Reuter, Andreas, 122, 200
routine (SQL), 3
row (see tuple)
row ID, 69
row type (SQL), 43
row type constructor (SQL), 43
row value constructor (SQL), 43
rules of inference, 76
  relation type (see relation type)
Russell, Bertrand, 163, 184
RVA (see relation-valued attribute)

S

scalar, 34
second normal form, 136, 138
SELECT - FROM - WHERE too rigid, 96
SELECT DISTINCT (SQL), 52
selector, 27, 116
  relation, 46
  scalar, 27
  tuple, 43
self-referencing relvar, 78
semantic optimization, 124
semidifference, 91
SEMIJOIN, 88
semijoin, 88
SEMIMINUS, 91
semistructured model, 169
set membership, 106
set-level operations, 61
Shakespeare, William, 73
Simon, Alan R., 201
single-relvar constraint, 121
sixth normal form, 147, 158
snapshot, 71
SQL, xvi
  expression evaluation, 93
  not same as relational model, 1
  pronouncing SQL, xvi
SQL:1999, 201
SQL:2003, 201
standard SQL, xvi
statement versus expression, 25, 41
Stoll, Robert R., 200
Stonebraker, Mike, 170
strong typing, 28
subset, 63, 103
summarize, 97–99
SUMMARIZE BY, 98
SUMMARIZE PER, 97–99
superkey, 120, 139
superset, 63, 103
surrogate, 160
system-defined type, 23, 32

T

table (see relation; relvar)
table value constructor (SQL), 46
TABLE_DEE, 89, 98
  and TABLE_DUM, 56
  identity with respect to join, 89
TABLE_DUM (see TABLE_DEE)
"tables and views", 17
tables, rows, and columns, 2
ternary relation, 5
ternary tuple, 42
THE_ operator, 27, 116
theorem (database), 76
theta-join, 93
Third Manifesto, The, xiv, 19, 70, 100, 171, 173, 175
third normal form, 136, 146
three-valued logic, 54
  truth tables, 54
TIMES, 88, 92
Todd, Stephen, 200
transaction, 121
transition constraint, 130
transitivity, 113
TransRelational™ Model, The, 157, 173, 201
"trigger", 78
triggered procedure, 78

trivial dependency (see FD; JD; MVD)
truth tables (see three-valued logic; two-valued logic)
tuple, 5, 41
  extracting attribute value from, 44
  pictured as row, 5
  relvar, 21
  versus proposition, 73
TUPLE {*H*}, 43
tuple constraint, 119
tuple equality, 45, 64, 65, 110
TUPLE FROM, 35
tuple ordering, 22
  not in relations, 15
  (see also ORDER; ORDER BY)
tuple selector, 43
tuple type, 43
  name, 43
TUPLE type generator, 35
tuple value (see tuple)
tuple variable, 35
  not in relational model, 35
Tutorial D
  used in *The Third Manifesto*, 19
  versus SQL, 82
two-valued logic, 54
  truth tables, 182
TYPE, 5, 32, 115
type, 5, 32
  versus physical representation, 28
  versus possible representation (see possible representation)
  versus variable, 38
type constraint, 115–118
  checked during selector invocation, 118
  not in SQL, 118
type error, 27
type generator, 35
  RELATION, 35
  TUPLE, 35

U

unary relation, 5
unary tuple, 42
UNGROUP, 100
UNION, 90
union, 10, 90
  disjoint, 90
  set-theoretic, 84, 87
UNIQUE (quantifier), 191
uniqueness (key), 63
universal quantifier, 185
UPDATE
  expansion, 107
  via cursor (SQL), 62
update anomalies, 151
update versus UPDATE, 8
"updating attributes", 62
"updating tuples", 62
updating views (see view)
user-defined type, 23, 32, 33

V

value, 19
  can't be updated, 19
  nonscalar, 34
  relation (see relation)
  scalar, 34
  tuple (see tuple)
  versus variable, 19
VAR
  base relvar, 35
  scalar, 98
  snapshot, 71
  tuple, 35
  view, 67
variable, 20
  can be updated, 21
  nonscalar, 34
  relation (see relvar)
  scalar, 34
  tuple, 36
  versus value, 19
view, 16, 67
  "materialized" (see snapshot)
  retrieval, 67
  two different purposes, 70
  updating, 68
  versus base relvar, 16
virtual relvar (see view)

W

WITH, 104
Wittgenstein, Ludwig, 157

X

XML, 31, 161, 165, 179
XPath, 32

ABOUT THE AUTHOR

C. J. Date is an independent author, lecturer, researcher, and consultant specializing in relational database systems. He was one of the first persons anywhere to recognize the fundamental importance of Codd's pioneering work on the relational model. He was also involved in technical planning for the IBM products SQL/DS and DB2 at the IBM Santa Teresa Laboratory in San Jose, California. He is best known for his books, in particular *An Introduction to Database Systems*, Eighth Edition (Addison-Wesley, 2004), the standard text in the field, which has sold nearly three quarters of a million copies worldwide, and (with Hugh Darwen and Nikos A. Lorentzos) *Temporal Data and the Relational Model* (Morgan Kaufmann, 2003). He enjoys a reputation that is second to none for his ability to explain complex technical material in a clear and understandable fashion.

COLOPHON

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

Genevieve d'Entremont was the production editor and copyeditor for *Database in Depth*. Matt Hutchinson proofread the book. Adam Witwer and Colleen Gorman provided quality control. Lydia Onofrei provided production assistance. Reg Aubry wrote the index.

Michele Wetherbee is the creative director for the Theory in Practice book series. MendeDesign designed the cover of this book. Karen Montgomery produced the cover layout with Adobe InDesign CS using the Akzidenz Grotesk and Orator fonts.

Marcia Friedman designed the interior layout. Melanie Wang designed the template. This book was converted by Keith Fahlgren and Andrew Savakis to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Adobe's Meridien; the heading font is ITC Bailey; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand MX and Adobe Photoshop CS.

Keep in touch with O'Reilly

1. Download examples from our books

To find example files for a book, go to:

www.oreilly.com/catalog

select the book, and follow the "Examples" link.

2. Register your O'Reilly books

Register your book at *register.oreilly.com*

Why register your books?
Once you've registered your O'Reilly books you can:

- Win O'Reilly books, T-shirts or discount coupons in our monthly drawing.
- Get special offers available only to registered O'Reilly customers.
- Get catalogs announcing new books (US and UK only).
- Get email notification of new editions of the O'Reilly books you own.

3. Join our email lists

Sign up to get topic-specific email announcements of new books and conferences, special offers, and O'Reilly Network technology newsletters at:

elists.oreilly.com

It's easy to customize your free elists subscription so you'll get exactly the O'Reilly news you want.

4. Get the latest news, tips, and tools

www.oreilly.com

- "Top 100 Sites on the Web"—PC Magazine
- CIO Magazine's Web Business 50 Awards

Our web site contains a library of comprehensive product information (including book excerpts and tables of contents), downloadable software, background articles, interviews with technology leaders, links to relevant sites, book cover art, and more.

5. Work for O'Reilly

Check out our web site for current employment opportunities:

jobs.oreilly.com

6. Contact us

O'Reilly Media
1005 Gravenstein Hwy North
Sebastopol, CA 95472 USA

TEL: 707-827-7000 or 800-998-9938
(6am to 5pm PST)

FAX: 707-829-0104

order@oreilly.com
For answers to problems regarding your order or our products. To place a book order online, visit:

www.oreilly.com/order_new

catalog@oreilly.com
To request a copy of our latest catalog.

booktech@oreilly.com
For book content technical questions or corrections.

corporate@oreilly.com
For educational, library, government, and corporate sales.

proposals@oreilly.com
To submit new book proposals to our editors and product managers.

international@oreilly.com
For information about our international distributors or translation queries. For a list of our distributors outside of North America check out:

international.oreilly.com/distributors.html

adoption@oreilly.com
For information about academic use of O'Reilly books, visit:

academic.oreilly.com

O'REILLY®

Our books are available at most retail and online bookstores.
To order direct: 1-800-998-9938 • *order@oreilly.com* • *www.oreilly.com*
Online editions of most O'Reilly titles are available by subscription at *safari.oreilly.com*

Buy *Database in Depth* and access the digital edition

FREE on Safari for 45 days.

Go to **www.oreilly.com/go/safarienabled**
and type in coupon code **N8PQ-1YPN-RWRI-ENOG-BN5Q**

Better than e-books

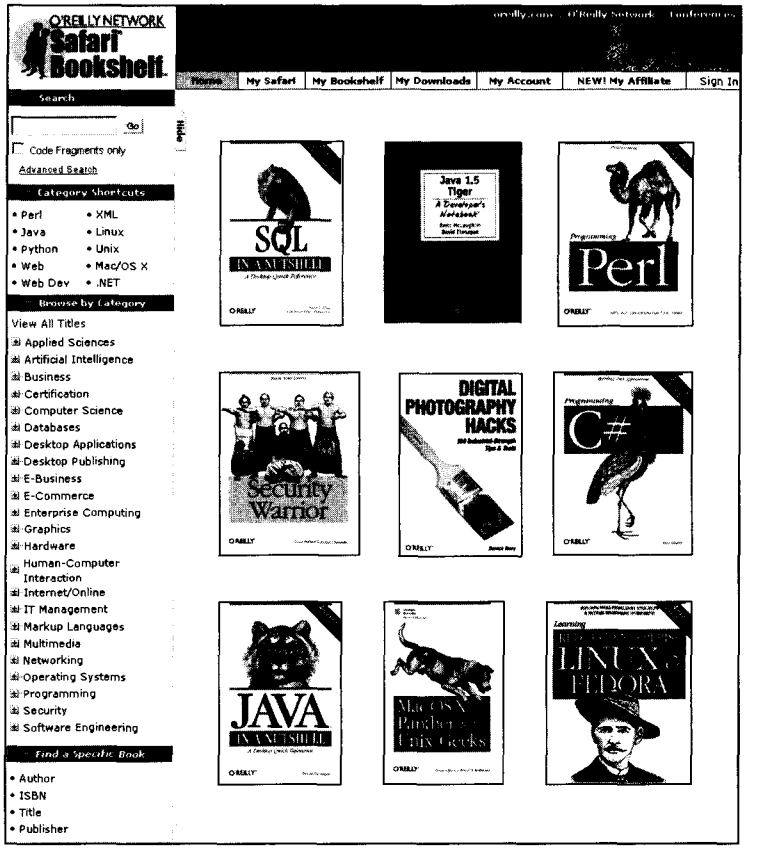
Search
over 2000 top tech books

Download
whole chapters

Cut and Paste
code examples

Find
answers fast

Read books from cover to cover. Or, simply click to the page you need.



Search Safari! The premier electronic reference library for programmers and IT professionals

O'REILLY NETWORK Safari Bookshelf

Addison Wesley · Sun microsystems · ALPHA · Microsoft Press · Peachpit Press · O'REILLY · Java · que · Adobe Press · SAMS · New Riders · Cisco Press · macromedia PRESS · PRENTICE HALL PTR

Part# 40421

RELATIONAL ALGEBRA AND UPDATE OPERATORS:

*A Simplified Syntax for the Dialect of **Tutorial D** Used in this Book*

RENAME:	<*relation*> RENAME (<*attribute*> AS <*name*>)
Restrict:	<*relation*> WHERE <*boolean*>
Project:	<*relation*> { <attributes> }
JOIN:	<*relation*> JOIN <*relation*> | JOIN { <*relations*> }
INTERSECT:	<*relation*> INTERSECT <*relation*> | INTERSECT { <*relations*> }
UNION:	<*relation*> UNION <*relation*> | UNION { <*relations*> }
Difference:	<*relation*> MINUS <*relation*>
Product:	<*relation*> TIMES <*relation*>
Divide:	<*relation*> DIVIDEBY <*relation*>
Semijoin:	<*relation*> MATCHING <*relation*>
Semidifference:	<*relation*> NOT MATCHING <*relation*>
EXTEND:	EXTEND <*relation*> ADD (<*exp*> AS <*name*>)
SUMMARIZE:	SUMMARIZE <*relation*> PER (<*relation*>) ADD (<*summary*> AS <*name*>)
GROUP:	<*relation*> GROUP ({ <*attributes*> } AS <*name*>)
UNGROUP:	<*relation*> UNGROUP (<*attributes*>)
Relation Assign:	<*relvar*> := <*relation*>
INSERT:	INSERT <*relvar*> <*relation*>
DELETE:	DELETE <*relvar*> [WHERE <*boolean*>]
UPDATE:	UPDATE <*relvar*> [WHERE <*boolean*>] (<*attribute assigns*>)
Attribute Assign:	<*attribute*> := <*exp*>

THE SUPPLIERS-AND-PARTS DATABASE:

Sample Values

S

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

P

PNO	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

SNO	PNO	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

Relational databases are ubiquitous in today's world. They're everywhere, from large e-commerce sites such as Amazon.com to small MP3 players that you can hold in the palm of your hand. Few technology professionals can afford to be without a good grounding in the fundamentals of relational database technology, yet many today who work with databases have had no formal training in relational theory.

In *Database in Depth*, author and well-known database authority Chris Date lays out the fundamentals of the relational model. This model was first introduced to the world in 1969 by E. F. Codd in his seminal paper "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," and it forms the basis for all database products in wide use today.

In this book, Chris dispels many commonly held misconceptions about the relational model, explaining that:

- **The term "relational" has nothing to do with relating two tables on a common set of columns.**
- **Relations are multidimensional. They are not flat. They are not two-dimensional. Don't let the term "table" mislead you!**
- **Nulls are most certainly not values, even though the SQL standard calls them so.**
- **Attributes of a relation can contain values of arbitrary complexity, including such things as arrays, XML documents, and even other relations.**
- **Base relations do not necessarily have to be physically stored.**
- **SQL is not a set-oriented language, but rather is bag-oriented.**

If you work with databases, you cannot afford to be without this book. Chris writes for the practitioner—for you—explaining clearly the fundamental principles that you need to know in order to be successful at what you do. Don't let a lack of formal education in database theory hold you back. Instead, let Chris's clear explanation of relational concepts, set theory, the difference between model and implementation, relational algebra, normalization, and much more set you apart and well above the competition when it comes to getting work done with a relational database.

C. J. Date was one of the first to recognize the genius in E. F. Codd's vision. He became a colleague of Codd's early on, worked closely with Codd during the formative years of the relational model, and has been a strong influence in the development of the database technology that you use day-in and day-out. This is your chance to learn from the master. Don't pass it by.

O'REILLY®
www.oreilly.com

ISBN 0-596-10012-4 US $29.95 CAN $41.95

9 780596 100124 6 36920 10012 6

Safari BOOKS ONLINE ENABLED
Includes FREE 45-Day Online Edition