/THEORY/IN/PRACTICE

SQL and Relational Theory

How to Write Accurate SQL Code

O'REILLY®

C. J. Date

Computers/Databases

SQL and Relational Theory

Understanding SQL's underlying theory is the best way to guarantee that your SQL code is correct and your database schema is robust and maintainable. On the other hand, if you're not well versed in the theory, you can fall into several traps. In *SQL and Relational Theory*, author C. J. Date demonstrates how you can apply relational theory directly to your use of SQL. With numerous examples and clear explanations of the reasoning behind them, you'll learn how to deal with common SQL dilemmas, such as:

- Should database access be granted through views instead of base tables?
- Nulls in your database are causing you to get wrong answers. Why? What can you do about it?
- Could you write an SQL query to find employees who haven't been in the same department for more than six months at a time?
- SQL supports "quantified comparisons," but they're better avoided. Why? How do you avoid them?
- Constraints are crucially important, but most SQL products don't support them properly. What can you do to resolve this situation?

Database theory and practice have evolved since E. F. Codd originally defined the relational model back in 1969. Independent of any SQL products, *SQL and Relational Theory* draws on decades of research to present the most up-to-date treatment of the material available anywhere. Anyone with a modest to advanced background in SQL will benefit from the many insights in this book.

C. J. Date began working on databases in early 1970 at IBM. A prolific writer, he is best known for his textbook *An Introduction to Database Systems* (Addison-Wesley), widely regarded as one of the fundamental texts on all aspects of database management.

US $39.99 CAN $39.99
ISBN: 978-0-596-52306-0
53999
9 780596 523060

Safari Books Online **Free online edition** for 45 days with purchase of this book. Details on last page.

O'REILLY® www.oreilly.com

SQL and Relational Theory

How to Write Accurate SQL Code

SQL and Relational Theory

How to Write Accurate SQL Code

C. J. Date

O'REILLY®

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

SQL and Relational Theory: How to Write Accurate SQL Code
by C. J. Date

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 c*orporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Adam Witwer
Indexer: C.J. Date
Cover Designer: Mark Paglietti
Interior Designer: Marcia Friedman
Illustrators: Jessamyn Read

Printing History:
January 2009: First Edition.

ISBN: 978-0-596-52306-0
[V]

To all those who think an exercise like this one is worthwhile, and in particular to the memory of Lex de Haan, who is very much missed.

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)

Languages die…
mathematical ideas do not.

G. H. Hardy

Unfortunately, the gap between theory and
practice is not as wide in theory
as it is in practice.

Anonymous

CONTENTS

PREFACE xiii

1 SETTING THE SCENE 1

The Relational Model Is Much Misunderstood 2
Some Remarks on Terminology 3
Principles, Not Products 4
A Review of the Original Model 5
Model vs. Implementation 13
Properties of Relations 17
Base vs. Derived Relations 19
Relations vs. Relvars 21
Values vs. Variables 23
Concluding Remarks 24
Exercises 25

2 TYPES AND DOMAINS 27

Types and Relations 27
Equality Comparisons 28
Data Value Atomicity 34
What's a Type? 37
Scalar vs. Nonscalar Types 40
Scalar Types in SQL 42
Type Checking and Coercion in SQL 43
Collations in SQL 45
Row and Table Types in SQL 47
Concluding Remarks 49
Exercises 50

3 TUPLES AND RELATIONS, ROWS AND TABLES 53

What's a Tuple? 53
Rows in SQL 57
What's a Relation? 60
Relations and Their Bodies 61
Relations Are n-Dimensional 62
Relational Comparisons 63
TABLE_DUM and TABLE_DEE 64
Tables in SQL 65

	Column Naming in SQL	*66*
	Concluding Remarks	*68*
	Exercises	*69*
4	NO DUPLICATES, NO NULLS	71
	What's Wrong with Duplicates?	*71*
	Duplicates: Further Issues	*76*
	Avoiding Duplicates in SQL	*77*
	What's Wrong with Nulls?	*79*
	Avoiding Nulls in SQL	*82*
	A Remark on Outer Join	*84*
	Concluding Remarks	*85*
	Exercises	*86*
5	BASE RELVARS, BASE TABLES	89
	Data Definitions	*90*
	Updating Is Set Level	*90*
	Relational Assignment	*92*
	More on Candidate Keys	*94*
	More on Foreign Keys	*97*
	Relvars and Predicates	*100*
	Relations vs. Types	*102*
	Exercises	*105*
6	SQL AND RELATIONAL ALGEBRA I: THE ORIGINAL OPERATORS	107
	Some Preliminaries	*107*
	More on Closure	*110*
	Restriction	*113*
	Projection	*114*
	Join	*115*
	Union, Intersection, and Difference	*118*
	Which Operators Are Primitive?	*120*
	Formulating Expressions a Step at a Time	*121*
	What Do Relational Expressions Mean?	*122*
	Evaluating SQL Table Expressions	*123*
	Expression Transformation	*124*
	The Reliance on Attribute Names	*127*
	Exercises	*129*

7 SQL AND RELATIONAL ALGEBRA II: ADDITIONAL OPERATORS 133

Semijoin and Semidifference *133*
Extend *134*
Image Relations *136*
Divide *139*
Aggregate Operators *141*
Image Relations bis *145*
Summarization *146*
Summarization bis *151*
Group and Ungroup *152*
"What If" Queries *154*
What About ORDER BY? *155*
Exercises *156*

8 SQL AND CONSTRAINTS 161

Type Constraints *162*
Type Constraints in SQL *165*
Database Constraints *166*
Database Constraints in SQL *170*
Transactions *171*
Why Database Constraint Checking Must Be Immediate *172*
But Doesn't Some Checking Have to Be Deferred? *175*
Constraints and Predicates *177*
Miscellaneous Issues *179*
Exercises *181*

9 SQL AND VIEWS 185

Views Are Relvars *186*
Views and Predicates *190*
Retrieval Operations *190*
Views and Constraints *192*
Update Operations *195*
What Are Views For? *199*
Views and Snapshots *200*
Exercises *201*

10 SQL AND LOGIC 205

Simple and Compound Propositions *206*
Simple and Compound Predicates *209*
Quantification *211*
Relational Calculus *215*

More on Quantification 221
Some Equivalences 227
Concluding Remarks 229
Exercises 230

11 USING LOGIC TO FORMULATE SQL EXPRESSIONS 233
Some Transformation Laws 234
Example 1: Logical Implication 236
Example 2: Universal Quantification 236
Example 3: Implication and Universal Quantification 238
Example 4: Correlated Subqueries 240
Example 5: Naming Subexpressions 241
Example 6: More on Naming Subexpressions 244
Example 7: Dealing with Ambiguity 245
Example 8: Using COUNT 247
Example 9: Join Queries 248
Example 10: UNIQUE Quantification 248
Example 11: ALL or ANY Comparisons 249
Example 12: GROUP BY and HAVING 253
Exercises 254

12 MISCELLANEOUS SQL TOPICS 255
*SELECT ** 256
Explicit Tables 256
Name Qualification 256
Range Variables 257
Subqueries 259
"Possibly Nondeterministic" Expressions 262
Empty Sets 263
A BNF Grammar for SQL Table Expressions 264
Exercises 266

A THE RELATIONAL MODEL 269
The Relational Model vs. Others 271
The Relational Model Defined 274
Objectives of the Relational Model 278
Some Database Principles 279
What Remains to Be Done? 280

B DATABASE DESIGN THEORY 285

The Place of Design Theory 286
Functional Dependencies and Boyce/Codd Normal Form 288
Join Dependencies and Fifth Normal Form 294
Two Cheers for Normalization 300
Orthogonality 304
Some Remarks on Physical Design 306
Concluding Remarks 308
Exercises 309

C ANSWERS TO EXERCISES 313

Chapter 1 313
Chapter 2 319
Chapter 3 326
Chapter 4 330
Chapter 5 335
Chapter 6 340
Chapter 7 348
Chapter 8 358
Chapter 9 366
Chapter 10 372
Chapter 11 378
Chapter 12 380
Appendix B 380

D SUGGESTIONS FOR FURTHER READING 389

INDEX 397

Preface

SQL is ubiquitous. But SQL is hard to use: It's complicated, confusing, and error prone—much more so, I venture to suggest, than its apologists would have you believe. In order to have any hope of writing SQL code that you can be sure is accurate, therefore (meaning it does exactly what it's supposed to do, no more and no less), you must follow some appropriate discipline—and it's the thesis of this book that *using SQL relationally* is the discipline you need. But what does this mean? Isn't SQL relational anyway?

Well, it's true that SQL is the standard language for use with relational databases—but that fact in itself doesn't make it relational. The sad truth is, SQL departs from relational theory in all too many ways; duplicate rows and nulls are two obvious examples, but they're not the only ones. As a consequence, it gives you rope to hang yourself with, as it were. So if you don't want to hang yourself, you need to understand relational theory (what it is and why); you need to know about SQL's departures from that theory; and you need to know how to avoid the problems they can cause. In a word, you need to use SQL relationally. Then you can behave as if SQL truly were relational, and you can enjoy the benefits of working with what is, in effect, a truly relational system.

Now, a book like this wouldn't be needed if everyone was using SQL relationally already—but they aren't. On the contrary, I observe much bad practice in current SQL usage. I even observe such practice being recommended, in textbooks and similar publications, by writers who really ought to know better (no names, no pack drill); in fact, a review of the literature in this regard is a pretty dispiriting exercise. The relational model first saw the light of day in 1969, and yet here we are, almost 40 years on, and it still doesn't seem to be very well understood by the database community at large. Partly for such reasons, this book uses the relational model itself as an organizing principle; it explains various features of the model in depth, and shows in every case how best to use SQL to implement the feature in question.

Prerequisites

I assume you're a database practitioner and therefore reasonably familiar with SQL already. To be specific, I assume you have a working knowledge of either the SQL standard or (perhaps more likely in practice) at least one SQL product. However, I don't assume you have a deep knowledge of relational theory as such (though I do hope you understand that the relational model is a good thing in general, and adhering to it wherever possible is a desirable goal). In order to avoid misunderstandings, therefore, I'll be describing various features of the relational model in detail, as well as showing how to use SQL to conform to those features. But what I won't do is attempt to justify all of those features; rather, I'll assume you're sufficiently experienced in database matters to understand why, e.g., the notion of a key makes sense, or why you sometimes need to do a join, or why many to many relationships need to be supported. (If I were to include such justifications, this would be a very different book—quite apart from anything else, it would be much bigger than it already is—and in any case, that book has already been written.)

I've said I expect you to be reasonably familiar with SQL. However, I should add that I'll be explaining certain aspects of SQL in detail anyway—especially aspects that might be encountered less frequently in practice. (The SQL notion of "possibly nondeterministic expressions" is a case in point here. See Chapter 12.)

Database in Depth

This book is based on, and intended to replace, an earlier one with the title *Database in Depth: Relational Theory for Practitioners* (O'Reilly, 2005). My aim in that earlier book was as follows (this is a quote from the 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 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?
>
> What exactly is first normal form?
>
> What's the connection between relations and predicates?
>
> What's semantic optimization?
>
> What's an image relation?
>
> Why is semidifference important?
>
> Why doesn't deferred integrity checking make sense?
>
> What's a relation variable?
>
> What's prenex normal form?
>
> Can a relation have an attribute whose values are relations?
>
> Is SQL relationally complete?
>
> Why is *The Information Principle* important?
>
> 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.

As the final sentence in this extract indicates, it was my hope that readers of that book would be able to apply its ideas for themselves, without further assistance from me as it were. But I've since come to realize that, contrary to popular opinion, SQL is such a difficult language that it can be far from obvious how to use it without violating relational principles. I therefore decided to expand the original book to include explicit, concrete advice on exactly that issue (how to use SQL relationally, I mean). So my aim in the present book is still the same as before (i.e., I want to help database practitioners understand relational theory in depth and make good use of that understanding in their professional activities), but I've tried to make the material a little easier to digest, perhaps, and certainly easier to apply. In other words, I've included a great deal of SQL-specific material (and it's this fact, more than anything else, that accounts for the increase in size over the previous version).

[*] For reasons that aren't important here, I've replaced a few of the questions in this list by new ones.

Further Remarks on the Text

I need to take care of several further preliminaries. First of all, 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 technical discrepancies—and there are a few—between this book and other books you might have seen by myself (including in particular the one this book is meant to replace), the present book should be taken as superseding. 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, wherever I felt it was necessary to do so.

Second, I will, as advertised, be talking 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 opposite: namely, that if something's theoretical, it can't be practical. But the truth is that theory (at least, relational theory, which is what I'm talking about here) 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. As one reviewer of the earlier book, Stéphane Faroult, wrote: "When you have a bit of practice, you realize there's no way to avoid having to know the theory." What's more, 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).

Of course, 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. In fact, it really shouldn't be necessary to have 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 be continually justifying ourselves to our critics—which is another reason why I think a book like this one is needed.

Third, as I've said, the book does go into a fair amount of detail regarding features of SQL or the relational model or both. (It deliberately has little to say on topics that aren't particularly relational; for example, there isn't much on transactions.) Throughout, I've tried to make it clear when the discussions apply to SQL specifically, when they apply to the relational model specifically, and when they apply to both. I should emphasize, however, that the SQL discussions in particular aren't meant to be exhaustive. SQL is such a complex language, and provides so many different ways of doing the same thing, and is subject to so many exceptions and special cases, that to be exhaustive—even if it were possible, which I tend to doubt—would be counterproductive; certainly it would make the book much too long. So I've tried to focus on what I think are the most important issues, and I've tried to be as brief as possible on the issues I've chosen to cover. And I'd like to claim that if you do everything I tell you, and don't do anything I don't tell you, then to a first approximation you'll be safe: You'll be using SQL relationally. But whether that claim is justified, or to what extent it is, must be for you to judge.

To the foregoing I have to add that, unfortunately, there are some situations in which SQL just can't be used relationally. For example, some SQL integrity checking simply has to be deferred (usually to commit time), even though the relational model rejects such checking as logically flawed. The book does offer advice on what to do in such cases, but I fear it often boils down to just *Do the best you can*. At least I hope you'll understand the risks involved in departing from the model.

I should say too that some of the recommendations offered aren't specifically relational anyway but are, rather, just matters of general good practice—though sometimes there are relational implications (implications that can be a little unobvious, too, perhaps I should add). *Avoid coercions* is a good example here.

Fourth, please note that I use the term *SQL* throughout the book to mean the standard version of that language exclusively, not some proprietary dialect (barring explicit statements to the contrary). In particular, I follow the standard in assuming the pronunciation "ess cue ell," not "sequel" (though this latter is common in the field), thereby saying things like *an* SQL table, not *a* SQL table.

Fifth, the book is meant to be read in sequence, pretty much, except as noted here and there in the text itself (most of the chapters do rely to some extent on material covered in earlier ones, so you shouldn't jump around too much). Also, each chapter includes a set of exercises. You don't have 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, are given in Appendix C.

Finally, I'd like to mention that I have some live seminars available based on the material in this book. See *http://www.clik.to/chris_date* 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. Indicates new terms. 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: "*SQL and Relational Theory* by C. J. Date. Copyright 2009 C. J. Date, 978-0-596-52306-0."

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*.

Comments and Questions

I've done my best to make this book as error-free as I can, but you might find mistakes. If so, please notify the publisher 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/9780596523060

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

http://www.oreilly.com

Safari® Books Online

Acknowledgments

I'd been thinking for some time about revising the earlier book to include more on SQL in particular, but the spur that finally got me down to it was sitting in on a class, late in 2007, for database practitioners. The class was taught by Toon Koppelaars and was based on the book he wrote with Lex de Haan (see Appendix D of the present book), and very good it was, too. But what struck me most about that class was seeing at first hand how apparently incapable the attendees were of applying relational and logical principles to their use of SQL. Now, I do assume those attendees had some knowledge of those topics—they were database practitioners, after all—but it seemed to me they really needed some guidance in the application of those ideas to their daily database activities. And so I put this book together. So I'm thankful, first of all, to Toon and Lex for providing me with the necessary impetus to get started on this project. I'm grateful also to my reviewers Herb Edelstein, Sheeri Ktitzer, Andy Oram, Peter Robson, and Baron Schwartz for their comments on earlier drafts, and Hugh Darwen and Jim Melton for other technical assistance. 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 Isabel Kunkle, Andy Oram, and Adam Witwer—for their encouragement, contributions, and support throughout the production of this book.

C. J. DATE
HEALDSBURG, CALIFORNIA
2008

CHAPTER ONE

Setting the Scene

A RELATIONAL APPROACH TO SQL: THAT'S THE THEME, OR ONE OF THE THEMES, OF THIS BOOK. Of course, to treat such a topic adequately, I need to cover relational issues as well as issues of SQL per se—and while this remark obviously applies to the book as a whole, it applies to this first chapter with special force. As a consequence, this chapter has comparatively little to say about SQL as such. What I want to do is review material that for the most part, at any rate, I hope you already know. My intent is to establish a point of departure: in other words, to lay some groundwork on which the rest of the book can build. But even though I hope you're familiar with most of what I have to say in this chapter, I'd like to suggest, respectfully, that you not skip it. You need to know what you need to know (if you see what I mean); in particular, you need to be sure you have the prerequisites needed to understand the material to come in later chapters. In fact I'd like to recommend, politely, that throughout the book you not skip the discussion of some topic just because you think you're familiar with that topic already. For example, are you absolutely sure you know what a key is, in relational terms? Or a join?*

The Relational Model Is Much Misunderstood

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 the relational model is the foundation (or a huge part of the foundation, at any rate) 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; certainly the model isn't well 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.
- Perhaps 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 practitioners in general, and SQL practitioners in particular, who have had some exposure to the relational model but don't know as much about it as they ought to or would like to. It's definitely *not* meant for beginners; however, it isn't just a refresher course, either. To be more specific, I'm sure you know something about SQL; but—and I apologize for the possibly offensive tone 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." I can't say it too strongly: *SQL and the relational model aren't the same thing*. 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 relation values and relation variables
- The relevance of predicates and propositions

* There's at least one pundit who doesn't. The following is a direct quote from a document purporting (like this book!) to offer advice to SQL users: "Don't use joins ... Oracle and SQL Server have fundamentally different approaches to the concept ... You can end up with unexpected result sets ... You should understand the basic types of join clauses ... Equi-joins are formed by retrieving all the data from two separate sources and combining it into one, large table ... Inner joins are joined on the inner columns of two tables. Outer joins are joined on the outer columns of two tables. Left joins are joined on the left columns of two tables. Right joins are joined on the right columns of two tables."

- The importance of attribute names
- The crucial role of integrity constraints

and so on (this isn't an exhaustive list). All of these issues, and 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 consequence of this state of affairs is that you might find, in reading this book, that you have to do some unlearning—and unlearning, unfortunately, is very hard to do.

Some Remarks on Terminology

You probably noticed right away, in that list of relational issues in the previous section, that I used the formal terms relation, tuple,* and attribute. SQL doesn't use these terms, however—it uses the more "user friendly" terms table, row, and column instead. And I'm generally sympathetic to the idea of using 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 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 often do exactly that myself—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 tend to use those more formal terms—at least when I'm talking about the relational model as opposed to SQL—and I'll give precise definitions for them at the relevant juncture. In SQL contexts, by contrast, I'll use SQL's own terms.

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

Talking of SQL, incidentally, let me remind you that (as stated in the preface) I use that term to mean the standard version of the language exclusively,† except in a few places where the context demands otherwise. However:

* Usually pronounced to rhyme with couple.

† International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:2003 (2003).

- Sometimes I use terminology that differs from that of the standard. For example, I use the term *table expression* in place of the standard term *query expression*, because (a) the value such expressions denote is a table, not a query, and (b) queries aren't the only context in which such expressions are used anyway. (As a matter of fact the standard does use the term *table expression*, but with a much more limited meaning; to be specific, it uses it to refer to what comes after the SELECT clause in a SELECT expression.)
- Following on from the previous point, I should add that not all table expressions are legal in SQL in all contexts where they might be expected to be. In particular, an explicit JOIN invocation, although it certainly does denote a table, can't appear as a "stand alone" table expression (i.e., at the outermost level of nesting), nor can it appear as the table expression in parentheses that constitutes a subquery (see Chapter 12). *Please note that these remarks apply to many of the individual discussions in the body of the book; however, it would be very tedious to keep on repeating them, and I won't.* (They're reflected in the BNF grammar in Chapter 12, however.)
- I ignore aspects of the standard that might be regarded as a trifle esoteric—especially if they aren't part of what the standard calls Core SQL or don't have much to do with relational processing as such. Examples here include the so called analytic or window (OLAP) functions; dynamic SQL; recursive queries; temporary tables; and details of user defined types.
- Partly for typographical reasons, I use a style for comments that differs from that of the standard. To be specific, I show comments as text strings in italics, bracketed by "/*" and "*/" delimiters.

Be aware, however, that all SQL products include features that aren't part of the standard per se. Row IDs provide a common example. My general advice regarding such features is: By all means use them if you want to—but not if they violate relational principles (after all, this book is supposed to be describing a *relational* approach to SQL). For example, row IDs are likely to violate what's called *The Principle of Interchangeability* (see Chapter 9); and if they do, then I certainly wouldn't use them. But, here and everywhere, the overriding rule is: You can do what you like, so long as you know what you're doing.

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; instead, it's concerned with principles. What do I mean by principles? Well, here's a dictionary 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: 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 make it harder to make 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 will never be obsolete.

In this book, therefore, we'll be concerned with principles, not products, and foundations, not fads. But I realize you do have to make compromises and tradeoffs sometimes, in the real world. For one example, sometimes you might have good pragmatic reasons for not designing the database in the theoretically optimal way. 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 be more or less forced into doing something not "truly relational" (like writing a query in some unnatural way to force the implementation to use an index). However, I believe very firmly that you should always make such compromises and tradeoffs from *a position of conceptual strength*. That is:

- You should understand what you're doing when you do decide to make such a compromise.
- You should know what the theoretically correct situation is, and you should have 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 due 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

The purpose of this section is to serve as a kickoff point for subsequent discussions; it reviews some of the most basic aspects of the relational model as originally defined. Note that 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:

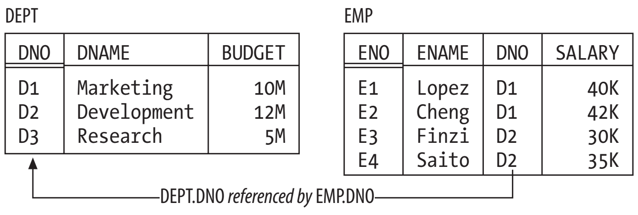
Mathematics too is not a static thing but changes over time. In fact, the relational model can itself be seen as a small branch of mathematics; as such, it evolves over time as new theorems are proved and new results discovered. What's more, those new contributions can be made by anyone who's competent 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 (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). It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into a field, database management, that was all too deficient in any 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 D.

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 on paper as tables (see Figure 1-1, below, 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, and then the attribute called DNO in the DEPT relation and the attribute called DNO in the EMP relation would both contain values that are taken from that conceptual pool. (By the way, it isn't necessary—though it's sometimes a good idea—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 that picture correspond to *attributes* of the relation and the rows to *tuples*. The value *n* can be any non-negative integer. A 1-ary relation is said to be *unary*; a 2-ary relation, *binary*; a 3-ary relation, *ternary*; and so on.

The relational model also supports various kinds of *keys*. To begin with—and this point is crucial!—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" consisting of 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; to repeat, candidate keys are always combinations, or sets, of attributes—even when the set in question contains just one attribute—and the conventional representation of a set on paper is as a commalist of elements enclosed in braces.

ASIDE

This is the first time I've used the term *commalist*, but I'll be using it a lot in the pages ahead. It can be defined as follows: Let *xyz* be some syntactic construct (for example, "attribute name"). Then the term *xyz commalist* denotes a sequence of zero or more *xyz*'s in which each pair of adjacent *xyz*'s is separated by a comma (as well as, optionally, one or more spaces before or after the comma or both). For example, if *A*, *B*, and *C* are attribute names, then the following are all attribute name commalists:

A , B , C

C , A , B

A , C

B

So too is the empty sequence of attribute names. Moreover, when some commalist is enclosed in braces, and therefore denotes a set, then (a) the order in which the elements appear within that commalist is immaterial (because sets have no ordering to their elements), and (b) if an element appears more than once, it's treated as if it appeared just once (because sets don't contain duplicate elements).

Next, a *primary* key is a candidate key that's been singled out for special treatment in some way. Now, if the relation in question has just one candidate key, then it won't make any real difference if we say it's the primary key. But if that relation has two or more candidate

* Strictly speaking, this sentence should read "Every *relvar* has at least one candidate key" (see the section "Relations vs. Relvars" later). A similar remark applies at various places elsewhere in this chapter, too. See Exercise 1-1 at the end of the chapter.

keys, then it's usual to choose one of them as primary, meaning it's somehow "more equal than the others." Suppose, for example, that every employee always has both a unique employee number and a unique employee name—not a very realistic example, perhaps, but good enough to make the point—so that {ENO} and {ENAME} are both candidate keys for EMP. Then we might choose {ENO}, say, to be the primary key.

Note that I said it's *usual* to choose a primary key. Indeed it is usual—but it's not 100 percent necessary. Now, if there's just one candidate key, then there's no choice and no problem; but if there are two or more, then having to choose one and make 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, therefore, 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 want to stress the fact 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 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* here, I mean that if, for example, EMP contains a tuple in which DNO has the value D2, then DEPT must also contain a tuple in which DNO has the value D2—for otherwise EMP would show some employee as being in a nonexistent department, and the database wouldn't be "a faithful model of reality."

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; however, all of those constraints will necessarily be specific to that database and will thus be expressed in terms of the relations in that database. By contrast, the relational model as originally formulated includes two *generic* integrity constraints—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. Here they are:

The entity integrity rule

Primary key attributes don't permit nulls.

The referential integrity rule

There mustn't be any unmatched foreign key values.

I'll explain the second rule 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 DEPT tuple with that same DNO value. So the referential integrity rule simply spells out the semantics of foreign keys; the name "referential integrity" derives from the fact that a 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 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 a few moments). Which I'll now proceed to do...but please understand that I'll be revisiting the whole issue of nulls in Chapters 3 and 4.

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	▨

Now, it's important to understand that this tuple contains *nothing at all* in the SALARY position. But it's very hard to draw pictures of nothing at all! I've tried to show that the SALARY position is empty in the picture above by shading it, but it would be more accurate not to show that position at all. Be that as it may, I'll use this same convention of representing empty positions by shading elsewhere in this book—but that shading does not, to repeat, represent any kind of value at all. You can think of it as constituting the null "marker," or flag, if you like.

In terms of relation EMP, then, the entity integrity rule says, loosely, that a given employee might have an unknown name, or an unknown department, or an unknown salary, but it can't have 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 further notice.

Manipulative Features

The manipulative part of the model in turn consists of two parts:

- The *relational algebra*, which is a collection of operators such as difference (or MINUS) that can be applied to relations
- A *relational assignment* operator, which allows the value of some relational expression (e.g., *r1* MINUS *r2*, where *r1* and *r2* 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 vs. Relvars." *Note:* I follow the usual convention throughout this book in using the generic term *update* to refer to the relational 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.

As for the relational algebra, it consists of a set of operators that—speaking very loosely—allow "new" relations to be derived from "old" ones. Each such operator takes one or more relations as input and produces another relation as output; for example, difference (i.e., 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 *r1* MINUS *r2*, feed the result as input to a union with some relation *r3*, feed *that* result as input to an intersection with some relation *r4*, and so on.

Now, any number of operators can be defined that fit the simple definition of "one or more relations in, exactly one relation out." Here I'll briefly describe what are usually thought of as the original operators (essentially the ones that Codd defined in his earliest papers);* I'll give more details in Chapters 6 and 7 I'll describe a number of additional operators as well. Figure 1-2 (opposite) is a pictorial representation of those original operators. *Note:* If you're unfamiliar with these operators and find the descriptions hard to follow, don't worry about it; as I've already said, I'll be going into much more detail, with lots of examples, in later chapters.

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 those tuples where the DNO value is D2.

* Except that Codd additionally defined an operator called *divide*. I'll explain in Chapter 7 why I omit that operator here.

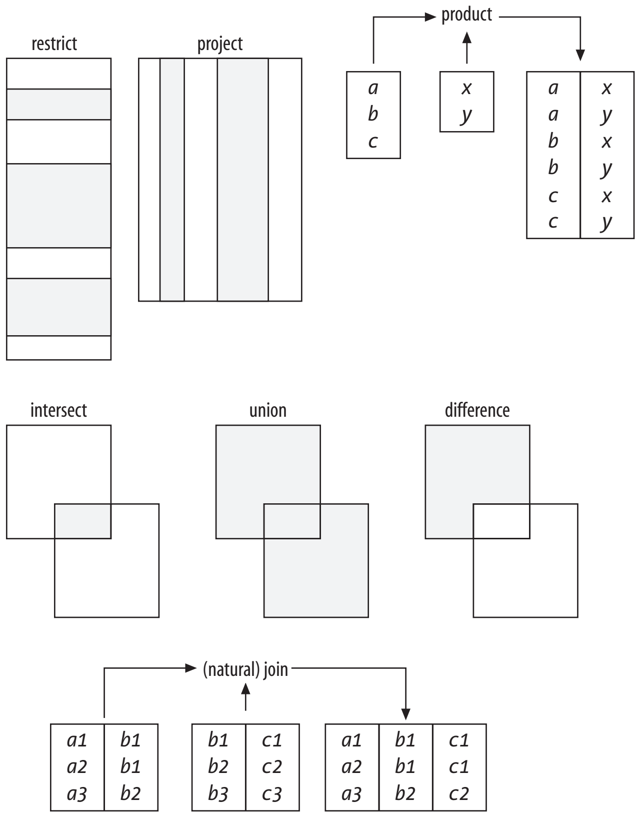


FIGURE 1-2. The original relational algebra

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 (thereby removing the ENAME and DNO attributes).

Product

Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations. The operator is also known variously as *cartesian product* (sometimes *extended* or *expanded* cartesian product), *cross product*, *cross join*, and *cartesian join*; in fact, it's really just a special case of join, as we'll see in Chapter 6.

Intersect

Returns a relation containing all tuples that appear in both of two specified relations. (Actually intersect is also a special case of join, as we'll see in Chapter 6.)

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.

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 more to say about the calculus later, mostly in Chapters 10 and 11.

The Running Example

I'll finish up this brief review by introducing the example I'll be using as a basis for most if not all of the discussions in the rest of the book: the familiar—not to say hackneyed—suppliers-and-parts database. (I apologize for dragging out this old warhorse yet one more time, but I believe that using the same example in a variety of different publications can help, not hinder, learning.) Sample values are shown in Figure 1-3.

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

To elaborate:

Suppliers
Relation S denotes suppliers (more accurately, suppliers under contract). Each supplier has one supplier number (SNO), unique to that supplier (as you can see, I've made {SNO} the primary key); one name (SNAME), not necessarily unique (though the SNAME values in Figure 1-3 do happen to be unique); one status value (STATUS), representing some kind of preference level among available suppliers; 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 ({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, or shipped, 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 ({SNO,PNO} is the primary key; also, {SNO} and {PNO} are both foreign keys, matching the primary keys of S and P, respectively). Notice that the database of Figure 1-3 includes one supplier, supplier S5, with no shipments at all.

Model vs. Implementation

Before going any further, there's one very important point I need to explain, because it underpins everything else to be discussed 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 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 of all, the concept *relation* itself is 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's 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's 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's 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 operations occur; all that's part of the implementation, not part of the model.

And one more example: *Candidate keys* (*keys* for short) are, again, part of the model, and users definitely have to know what keys are. In practice, key uniqueness is often enforced by means of what's called a "unique index"; but indexes in general, and unique indexes in particular, aren't part of the model, they're part of the implementation. Thus, a unique index mustn't be confused with a key in the relational sense, even though the former might be used to implement the latter (more precisely, to implement some *key constraint*—see Chapter 8).

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.

Please note that 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 implementation should be, at least potentially, *hidden from the user*.

Here are some important consequences of the foregoing definitions. First of all (and despite extremely common misconceptions to the contrary), everything to do with performance is fundamentally an implementation issue, not a model issue. For example, we often hear remarks to the effect 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* does—but that's all.

Now, I don't want to give the wrong impression here. It's true that performance is fundamentally an implementation issue; however, that doesn't mean a good implementation will perform well if you use the model badly. Indeed, that's precisely one of the reasons why you need to know the model: 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 (pseudocode)—

```
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. **Recommendation:** Don't do this. Relational systems shouldn't be used like simple access methods.*

By the way, these remarks about performance apply to SQL too. Like the relational operators (join and the rest), SQL as such can't be said to be fast or slow—only implementations can sensibly be described in such terms—but it's also possible to use SQL in such a way as to guarantee bad performance. Although I'll generally have little to say about performance in this book, therefore, I will occasionally point out certain performance implications of what I'm recommending.

ASIDE

I'd like to elaborate on this matter of performance for a moment. By and large, my recommendations in this book are never based on performance as a prime motivator; after all, it has always been an objective of the relational model to take performance concerns out of the hands of the user and put them into the hands of the system instead. However, it goes without saying that this objective hasn't yet been fully achieved, and so (as I've already said) the goal of using SQL relationally must sometimes be compromised in the interest of achieving satisfactory performance. That's another reason why, as I said earlier in this chapter, the overriding rule has to be: *You can do what you like, so long as you know what you're doing*.

Back to model vs. implementation. The second point is that, as you probably realize, it's precisely the separation of model and implementation that allows us to achieve *physical data independence*. Physical data independence—not a great term, by the way, but we're probably stuck with it—means we have the freedom to make changes in 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 specifics is, typically, 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 programs, queries, and the like can all still work after the change. Very importantly, therefore, physical data independence means *protecting investment in user training and applications*—investment in logical database design also, I might add.

* More than one reviewer observed that this sentence didn't make sense (how can a system be used as a method?). Well, if you're too young to be familiar with the term *access method*, then I envy you; but the fact is, that term, inappropriate though it certainly was (and is), was widely used in the past to mean a simple record level I/O facility of one kind or another.

It follows from all of the above that (as previously indicated) indexes, and indeed physical access paths of any kind, are properly part of the implementation, not the model; they belong under the covers and should be hidden from the user. (Note that access paths as such are nowhere mentioned in the relational model.) For the same reasons, they should be rigorously excluded from SQL also. **Recommendation:** Avoid the use of any SQL construct that violates this precept. (Actually there's nothing in the standard that does, so far as I'm aware, but I know the same isn't true of certain SQL products.)

Anyway, 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 considerations in general. Sadly, however, most of today's SQL systems don't make those distinctions as clearly as they should. As a direct consequence, they deliver far less physical data independence than they should, and far less than relational systems are or should be capable of. I'll come back to this issue in the next section.

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 data (especially the persistent data) of some particular enterprise.

In other words, a data model in the second sense is just a (logical, and 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 explained these two different meanings, I'd now 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, then 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* relational model (with the definite article). I'll have more to say on this latter point in Appendix A.

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

Properties of Relations

Now let's get back to our 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 by the term *attribute* I mean 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 in 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 simplicity.

By the way, although it's strictly correct to say 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 to pretend the heading is a set of attribute names only. For example, the STATUS attribute does have a type—INTEGER, 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. *Note:* The term *degree* is used in connection with tuples also. For example, the tuples in relation S are (like relation S itself) all of degree 4.

Next, relations *never* contain duplicate tuples. This property follows because a body is defined to be a set of tuples, and sets in mathematics don't contain duplicate elements. Now, SQL fails here, as I'm sure you know: SQL tables are allowed to contain duplicate rows and thus aren't 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 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 no primary key a fortiori.)

Next, the tuples of a relation are *unordered*, top to bottom. This property follows because, again, a body is defined to be 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 a similar remark naturally applies to 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 as depicted in 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 relations have no ordering to their tuples 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 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 as depicted in 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 two pictures below represent the same relation but 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 SELECT SNO, CITY FROM S and SELECT CITY, SNO FROM S, respectively. Now, you might be thinking that the difference between these two queries, and 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. See, for example, the discussion of the SQL JOIN operator in Chapter 6.)

Finally, relations are always *normalized* (equivalently, they're in *first normal form*, 1NF).* Informally, what this means is 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.

* "First" normal form because, as I'm sure you know, it's possible to define a series of "higher" normal forms—second normal form, third normal form, and so on—that are relevant to the business of database design. See Exercise 2-9 in Chapter 2, also Appendix B.

To close this section, I'd like to emphasize something I've touched on several times already: namely, the fact that there's a logical difference between a relation as such and a picture of a relation as shown in, for example, Figures 1-1 and 1-3. To say it one more time, the constructs in Figures 1-1 and 1-3 aren't relations at all but, rather, pictures of relations—which I generally refer to as *tables*, despite the fact that *table* is a loaded word in SQL contexts. Of course, relations and tables do have certain points of resemblance, and in informal contexts it's usual, and possibly acceptable, to say they're the same thing. But when we're trying to be precise—and right now I am trying to be precise—then we do have to recognize that the two concepts are not identical.

As an aside, I observe that, more generally, there's a logical difference between a thing of any kind and a picture of that thing. There's a famous painting by Magritte that illustrates the point I'm trying to make here. The painting is of an ordinary tobacco pipe, but underneath Magritte has written *Ceçi n'est pas une pipe*…the point being, of course, that *obviously* the painting isn't a pipe—rather, it's a picture of a pipe.

All of that being said, I should now say too that it's actually a major advantage of the relational model that its basic abstract object, the relation, does have such a simple representation on paper; it's that simple representation that makes relational systems easy to use and easy to understand, and makes it easy to reason about the way such systems behave. However, it's unfortunately also the case that that simple representation does suggest some things that aren't true (e.g., that there's a top to bottom tuple ordering).

And one further point: I've said there's a logical difference between a relation and a picture of a relation. The concept of *logical difference* derives from a dictum of Wittgenstein's:

> All logical differences are big differences.

This notion is an extraordinarily useful one; as a "mind tool," it's a great aid to clear and precise thinking, and it can be very helpful in pinpointing and analyzing some of the confusions that are, unfortunately, all too common in the database world. I'll be appealing to it many times in the pages ahead. Meanwhile, let me point out that we've encountered a few logical differences already. Here are some of them:

- SQL vs. the relational model
- Model vs. implementation
- Data model (first sense) vs. data model (second sense)

And more are described in the next three sections.

Base vs. Derived Relations

As I explained earlier, the operators of the relational algebra allow us to start with some given relations—perhaps the ones depicted in Figure 1-3—and obtain further relations from those given ones (for example, by doing queries). The given relations are referred to as *base* relations, the others are *derived* relations. In order to get us started, as it were, a relational system therefore 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, naturally, a base table). And base relations obviously need to be named—for example:

```
CREATE TABLE S ... ;
```

But certain derived relations, including in particular what are called views, are named too. A *view* (also known as a *virtual relation*) is a named relation whose value at any given time *t* is the result of evaluating a certain relational expression at that time *t*. Here's an SQL example:

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

In principle, you can operate on views as if they were base relations,* but they aren't base relations. Instead, you can think of a view as being "materialized"—in effect, you can think of a base relation being constructed whose value is obtained by evaluating the specified relational expression—at the time the view in question is referenced. Though I must emphasize that thinking of views being materialized in this way when they're referenced is purely conceptual; it's just a way of thinking; it's not what's really supposed to happen; and it wouldn't work for update operations in any case. How views are really supposed to work is explained in Chapter 9.

By the way, there's an important point I need to make here. You'll often hear 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 categorically 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 obtained when they're needed (conceptually, at any rate). If the base relations can be obtained 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; then base relations S and SP could be obtained, conceptually, by taking appropriate projections of that join. In other words: Base relations

* You might be thinking this claim can't be 100 percent true for update operations. If so, you might be right as far as today's products are concerned; however, I still claim it's true in principle. See the section "Update Operations" in Chapter 9 for further discussion.

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. The idea was to give implementers lots of freedom to implement the model in whatever way they chose—in particular, in whatever way seemed likely to yield good performance—without compromising on physical data independence. The sad fact is, however, SQL vendors seem mostly not to have understood this point; instead, they map base tables fairly directly to physical storage,* and (as noted previously) their products therefore provide far less physical data independence than relational systems are or should be capable of. Indeed, this state of affairs is reflected in the SQL standard itself (as well as in most other SQL documentation), which typically—quite ubiquitously, in fact—uses expressions like "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 want to mean (for example) a base relation specifically, then I'll say "base relation." **Recommendation:** I suggest strongly that you adopt the same discipline for yourself." Don't fall into the common trap of thinking the term *relation* means a base relation specifically—or, in SQL terms, thinking the term *table* means a base table specifically.

Relations vs. Relvars

Now, it's entirely possible that you already knew everything I've been telling you in this chapter so far; in fact, I rather hope you did, though I also hope that didn't mean you found the text boring. Anyway, now I come to something you might not know already. The fact is, historically there's been a lot of confusion over yet another logical difference: namely, the difference between relations as such, on the one hand, and relation *variables* on the other.

Forget about databases and relations for a moment; consider instead the following simple programming language example. Suppose I say in some 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. We all understand that. Well, in exactly the same way, if I say in SQL:

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

* 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. The fact is, I still consider the mapping to physical storage in those products to be fairly direct.

then T *is not a table*: It's a variable, 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).

Take another look at Figure 1-3, the suppliers-and-parts database. That figure shows three relation *values*—namely, the relation values that happen to exist 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, that is—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, in a sense, 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' ) ;
```

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

ASIDE

I can't show the foregoing assignment in SQL because SQL doesn't directly support relational assignment. Instead, I've shown it (as well as the original DELETE) in a more or less self-explanatory language called Tutorial D. 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* (see Appendix D)—and I'll use it in the present book too, when I'm explaining relational concepts. But since my intended audience is SQL practitioners, I'll show SQL equivalents as well, most of the time.

In like fashion, the familiar INSERT and UPDATE statements are also basically just 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; indeed it's the only update operator we really need, logically speaking.

So there's a logical difference between relation values and relation variables. The trouble is, the database 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 *relvar*; for example, I'll say the suppliers-and-parts database contains three relvars (more precisely, three *base* 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 been using the term *relvar* instead (or as well).

Values vs. Variables

The logical difference between relations and relvars is actually a special case of the logical 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 such a great help, in so many ways.) Here then are some definitions:

> **Definition:** A *value* is what the logicians call an "individual constant": for example, the integer 3. A value has no location in time or space. However, values can be represented in memory by means of some encoding, and those representations or encodings 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 following definition—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 wouldn't be that value any longer.

> **Definition:** A *variable* is a holder for a representation of a value. A variable does have location in time and space. Also, variables, unlike values, can be updated; that is, the current value of the variable can be replaced by another

* SQL makes the same mistake, of course, because it too has just one term, *table*, that has to be understood as sometimes meaning a table value and sometimes a table variable.

value. (After all, that's what "variable" means—to be a variable is to be updatable; equivalently, to be a variable is to be assignable to.)

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 about such matters in the next two chapters.

Now, you might think it's hard to imagine people getting confused over a distinction as obvious and fundamental as the one between values and variables. In fact, however, 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 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*].

Concluding Remarks

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

- I explained why we'd be concerned with principles, not products, and why I'd be using formal terminology such as *relations*, *tuples*, and *attributes* (at least in relational contexts) in place of their more "user friendly" SQL counterparts.
- 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 the *closure* property and very briefly described the operators *restrict*, *project*, *product*, *intersection*, *union*, *difference*, and *join*.
- 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 the difference between *base relations* (or base relvars, rather) and *views*.
- I discussed the logical differences between *model* and *implementation*, *values* and *variables* in general, and *relations* and *relvars* in particular. The model vs. implementation discussion in particular led to a discussion of *physical data independence*.
- 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 nulls and duplicate rows,

the fact that SQL tables have a left to right column ordering, and the fact that SQL doesn't clearly distinguish between table values and table variables (because it uses the same term, *table*, for both)—and we'll see many more in the pages to come. All of these issues will be dealt with in depth in later chapters.

One last point (I didn't mention this explicitly before, but I hope it's clear from everything I did say): Overall, the relational model is declarative, not procedural, in nature; that is, it always favors 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.

NOTE

After I first wrote the foregoing paragraph, 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*.

Exercises

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.

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 in your own words (a) physical data independence, (b) the difference between model and implementation.

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. *(Try this exercise without looking back at the body of the chapter.)* What relvars does the suppliers-and-parts database contain? What attributes do they involve? What candidate and foreign keys do they have? (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. It's not so important to remember the actual data values in detail—though it wouldn't hurt if you did.)

Exercise 1-10. "There's only one relational model." Explain this remark.

Exercise 1-11. 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 1-12. 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 1-13. Write the CREATE TABLE statements for an SQL version of the suppliers-and-parts database.

Exercise 1-14. The following is a typical SQL INSERT statement against the suppliers-and-parts database:

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

Show an equivalent relational assignment operation. *Note:* 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 1-15. *(Harder.)* The following is a typical SQL UPDATE statement against the suppliers-and-parts database:

```
UPDATE S SET STATUS = 25 WHERE 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 7 for further discussion.)

Exercise 1-16. In the body of the chapter, I said that SQL doesn't directly support relational assignment. Does it support it indirectly? If so, how? A related question: Can all relational assignments be expressed in terms of INSERT and/or DELETE and/or UPDATE? If not, why not? What are the implications?

Exercise 1-17. From a *practical* standpoint, 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 be taking a closer look at such matters later in the book.)

CHAPTER TWO

Types and Domains

THIS CHAPTER IS RELATED ONLY TANGENTIALLY TO THE MAIN THEME OF THE BOOK. Types are certainly fundamental, and the ideas discussed in this chapter are certainly important (they might help to dispel certain common misconceptions, too). However, type theory as such isn't a specially relational topic, and type-related matters don't seem—at least on the surface—to have much to do with SQL daily life, as it were. What's more, while there are certainly SQL problems in this area, there isn't much you can do about them, for the most part; I mean, there isn't much concrete advice I can offer to help with the goal of using SQL relationally (though there is some, as you'll see). So you might want to give this chapter just a "once over lightly" reading on a first pass, and come back to it after you've absorbed more of the material from later chapters.

Types and Relations

Data types (types for short) are fundamental to computer science. Relational theory in particular requires a supporting type theory, because relations are defined over types; that is, every attribute of every relation is defined to be of some type (and the same is true of relvars too, of course). For example, attribute STATUS of the suppliers relvar S might be defined to be of type INTEGER. If it is, then every relation that's a possible value for relvar

S must have a STATUS attribute of type INTEGER—which means in turn that every tuple in such a relation must also have a STATUS attribute that's of type INTEGER, which means in turn that the tuple in question 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—a relational attribute can be of any type whatsoever, implying among other things that such types can be arbitrarily complex. In particular, those types can be either system or user defined. In this book, however, I don't plan to say much about user defined types as such, because:

- The whole point about user defined types (from the point of view of the user who is merely using them, at any rate, as opposed to the user who actually has the job of defining them) is that they're supposed to behave just like system defined types anyway.
- Comparatively few users will ever be faced with the job of creating a user defined type—and creating such a type doesn't really involve any specifically relational considerations in any case.

From this point forward, therefore, you can take the term *type* to mean a system defined type specifically, unless the context demands otherwise. The relational model prescribes just one such type, BOOLEAN (the most fundamental type of all). Type BOOLEAN contains exactly two values: two truth values, to be specific, denoted by the literals TRUE and FALSE, respectively. Of course, practical systems support a variety of other system defined types as well, and I'll assume for definiteness that types INTEGER (integers), FIXED (fixed point numbers), and CHAR (character strings of arbitrary length) are among those supported. *Note:* I'll discuss the system defined types supported by SQL in particular in the section "Scalar Types in SQL" later.

In the interest of historical accuracy, I should now explain 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 a series of arguments 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 in each of the next two sections:

Equality comparisons and "domain check override"
  This part of the discussion will convince you (I hope) that domains really are types.

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

Equality Comparisons

Despite what I said a few moments ago about ignoring user defined types, I'm going to assume in the present section, for the sake of the argument, that the supplier number (SNO) attributes in relvars S and SP are declared to be of some user defined type—sorry, domain—which I'll assume for simplicity is called SNO as well. (*Note:* Throughout this book I treat *declared* and *defined* as synonymous.) Likewise, I'm going to assume that the part number (PNO) attributes in relvars P and SP are also declared to be of a user defined type (or domain) with the same name, PNO. Please note that these assumptions aren't crucial to my argument; it's just that I think they make that argument a little more convincing, and perhaps easier to follow.

I'll start with the fact that, as everyone knows (?), two values can be compared for equality in the relational model only if they come from the same domain. For example, the following comparison (which might be part of the WHERE clause in some SQL query) is obviously valid:

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

By contrast, this one obviously (?) isn't:

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

Why not? Because part numbers and supplier numbers are different kinds of things—they're defined on different domains. So the general idea is that the DBMS[*] should reject any attempt to perform any relational operation (join, union, whatever) that involves, either explicitly or implicitly, an equality 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 *
       FROM   SP
       WHERE  SP.PNO = S.SNO )      /* not OK */
```

(There's no terminating semicolon because this is an expression, not a statement. See Exercise 2-23 at the end of the chapter.)

As the comment says, this query is certainly not OK. The reason is that, in the last line, the user presumably meant to write WHERE SP.SNO = S.SNO, but by mistake—probably just a slip of the typing fingers—he or she wrote 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 the user at this point, highlight the error, and perhaps ask if the user would like to correct it before proceeding.

Now, I don't know any SQL 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

* DBMS = database management system. 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?)

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. However, Codd felt there should be a way in such a situation for the user to make the DBMS go ahead and do the comparison anyway, even though it's apparently not valid, on the grounds that sometimes the user will know more than the DBMS does. Now, it's hard for me to do justice to this idea, because quite frankly I don't think it makes sense—but let me give it a try. Suppose it's your job to design a database involving, let's say, customers and suppliers, and you therefore decide to have a domain of customer numbers and a domain of supplier numbers; and 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 comparison) to see if they're equal. And if it does, well, certainly the system mustn't prevent you from doing that comparison; certainly the system mustn't prevent you from posing a reasonable query.

On the basis of such arguments, Codd proposed what he called "domain check override" (DCO) versions of certain of his relational 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 (perhaps only the database administrator 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 on the suppliers-and-parts database:

```
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), page 47, he says that in such a situation "the DBMS [merely] checks that the basic data types are the same"; in the case at hand, those "basic data types" are all just numbers (loosely speaking), and so that check succeeds.

To me, this conclusion is unacceptable. Clearly, the expressions P.WEIGHT = SP.QTY and P.WEIGHT - SP.QTY = 0 mean essentially the same thing. Surely, therefore, they must both be valid or both be invalid; the idea that one might be valid and the other not surely makes no sense. 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 simple attribute references. Observe that the comparison P.WEIGHT = SP.QTY falls into this special category but the comparison P.WEIGHT - SP.QTY = 0 doesn't.)

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 plausible that the first two of these might be valid (and execute successfully, and even give TRUE) 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's going on?

I return now to the fact that attributes S.SNO and P.PNO are defined on domains SNO and PNO, respectively, and my claim that domains are actually types; in fact, I suggested earlier that domains SNO and PNO in particular were user defined types. Now, it's likely that those types are both physically represented in terms of the system defined type CHAR—but such representations are part of the implementation, not the model; they're irrelevant to the user, and in fact they're hidden from the user (at least, they 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 (see the section "What's a Type?" later in this chapter). 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 were an operator defined in connection with type SNO).

Now, when we define a type, we also have to define the operators that can be used in connection with values and variables of the type in question (again, see the section "What's a Type?"). And 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.† In the case of type SNO, for example, the selector (which in practice would 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')
```

* Throughout this book I use the term *representation* to mean a physical representation specifically, unless the context demands otherwise.

† This observation is valid regardless of whether we're in an SQL context (as in the present discussion) or otherwise—but I should make it clear that selectors in SQL aren't as straightforward as they might be, and *selector* as such isn't an SQL term. I should also make it clear that selectors have nothing to do with the SQL SELECT operator.

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

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

is an invocation of the PNO selector, and it returns a certain part number (namely, the one represented by the character string 'P1'). In other words, the SNO and PNO selectors effectively work by taking a certain CHAR value and converting it to a certain SNO value and a certain PNO value, respectively.

Now let's get back to the comparison S.SNO = 'X4'. As you can see, the comparands here are of different types (types SNO and CHAR, to be specific). Since they're of different types, they certainly can't be equal (recall that two values can be compared for equality "only if they come from the same domain"). But the system does at least know there's an operator—namely, the SNO selector—that effectively performs CHAR to SNO conversions. So it can invoke that operator, implicitly, to convert the CHAR comparand to a supplier number, thereby effectively replacing the original comparison by this one:

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

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

In the same kind of way, the system can effectively replace the comparison P.PNO = 'X4' by 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.

NOTE

Implicit type 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. I'll have a little more to say about coercion in SQL in particular in the section "Type Checking and Coercion in SQL," later.

To continue with the example: Another operator we must define when we 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 that the THE_ operators for types SNO and

* Again this observation is valid regardless of whether we're in an SQL context or some other context—though (as with selectors) "THE_ operators" in SQL aren't as straightforward as they might be, and "*THE_ operator*" as such isn't an SQL term. I note too that a given type might have more than one corresponding THE_ operator; see the discussion of type POINT in the section "What's a Type?" for an example of such a type.

PNO are called THE_SC and THE_PC, respectively. Then, if we really did want to compare S.SNO and P.PNO for equality, the only sense I can make of that requirement is that we want to test whether the character string representations are the same, which we might do like this:

```
THE_SC ( S.SNO ) = THE_PC ( 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 been sketching, involving selectors and THE_ operators, effectively provides both (a) the domain checking we want in the first place, and (b) a way of overriding that checking, when desired, 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 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 checks that the operands are of the right types for the operation in question (or, possibly, are coercible to those right types). Observe too that this mechanism works for all operations, not just for the equality comparisons I've been discussing; the emphasis on equality and other comparison operations in discussions of domain checking in the literature 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?).

I'd like to close this section by stressing the absolutely fundamental role played by the equality operator ("="). It wasn't just an accident that the discussions above happened to focus on the question of comparing two values for equality. The fact is, equality truly is central, and the relational model requires it to be supported for every type. Indeed, since a type is basically a set of values (see the section "What's a Type?"), without the "=" operator we couldn't even say what values constitute the type in question! That is, given some type *T* and some value *v*, we couldn't say, absent that operator, whether or not *v* was one of the values in the set of values constituting type *T*.

What's more, the relational model also specifies the semantics of the "=" operator, as follows: If *v1* and *v2* are values of the same type, then *v1* = *v2* evaluates to TRUE if *v1* and *v2* are the very same value and FALSE otherwise. (By contrast, if *v1* and *v2* are values of different types, then *v1* = *v2* has no meaning—it's not even a legal comparison—unless *v1* can be coerced to the type of *v2* or the other way around, in which case we aren't really talking about a comparison between *v1* and *v2* as such anyway.)

Data Value Atomicity

Well, I hope the previous section succeeded in convincing you that domains really are 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) 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 question: What does it mean for data to be atomic?

Well, on page 6 of the book mentioned earlier (*The Relational Model for Database Management Version 2*), Codd defines atomic data as data that "cannot be decomposed into smaller pieces by the DBMS (excluding certain special functions)." Even if we ignore that parenthetical exclusion, however, this definition is a trifle puzzling and/or not very precise. For example, what about character strings? Are character strings atomic? Well, every product I know provides a variety of operators—LIKE, SUBSTR (substring), "||" (concatenate), and so on—that rely by definition 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 is itself open to a variety of interpretations)
- 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

And so on.

Now I'd like to move on to what you might regard as a 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 writers 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 agree 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!* (I don't claim it's well designed—indeed, it probably isn't—but that's not the point. I'm concerned here with what's legal, not with questions of good design. The design of R3 is legal.) For consider:

- First, note that I've renamed the attribute PNO_SET, and I've shown the groups of part numbers that are PNO_SET values enclosed in braces, 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.

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, sometimes we want to deal with individual part numbers within that set—but then we're descending to a lower level of detail, or 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 atoms as indivisible things, sometimes we want to think about the subatomic particles (i.e., 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 (?).

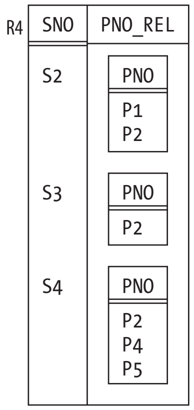
Let's return for a moment to relation R3. In Figure 2-1, I showed PNO_SET values as general sets. But it would be more useful in practice if they were, more specifically, relations (see Figure 2-2 opposite, where I've changed the attribute name to PNO_REL). Why would it be more useful? Because relations, 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 were to use general sets instead of relations, then we would 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!

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 Chapter 7; here let me just note that SQL doesn't support them. (More precisely, it doesn't support what would be its analog of RVAs, *table valued columns*. Oddly enough, however, it does support columns whose values are arrays, and columns whose values are rows, and even columns whose values are "multisets of rows"—where 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 little bit like "table valued columns"; 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 and thus aren't regular SQL table values, by definition.)

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," you might care to think of. Attributes, and therefore domains, can contain *anything* (any *values*, that is).

Incidentally, the foregoing paragraph goes a long way 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 about 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 this is: A proper object/relational system is just a relational system with proper type support (including proper user defined type support in particular)—which just means it's a proper relational system, no more and no less. And what some people are pleased to

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



R4

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

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

call "the object/relational model" is, likewise, just the relational model, no more and no less.

What's a Type?

From this point forward I'll favor the term *type* over the term *domain*. So 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:* It follows that types are disjoint (nonoverlapping), by definition. However, perhaps I need to elaborate on this point 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 the realm of what's called *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, e.g., 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

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

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 as yet: namely, the fact that *associated with every type there's a set of operators for operating on values and variables of the type in question*—where to say that operator *Op* is "associated with" type *T* means, precisely, that operator *Op* has a parameter of declared type *T*.* 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. *Note:* User defined operators can be defined in association with system defined types as well as user defined ones, as you would expect.

Observe now that, by definition, values and variables of a given type can be operated upon only by means of the operators associated with 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 aren't 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

* The logical difference between type and representation is important here. To spell the matter out, the operators associated with type *T* are the operators associated with type *T*—*not* the operators associated with the representation of type *T*. For example, just because the representation for type SNO happens to be CHAR, it doesn't follow that we can concatenate two supplier numbers; we can do that only if concatenation ("||") is an operator that's defined for type SNO. (In fact I did mention exactly this example in passing in the section "Equality Comparisons," as you might recall.)

supplier numbers wouldn't be supported (what could it possibly mean 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 the type (obviously enough).
2. Specifying the values that make up that type. I'll discuss this point in detail in Chapter 8.
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 a selector operator for selecting, or specifying, values of that type.
5. Specifying the operators—including THE_ operators in particular—that apply to values and variables of that type (see below).
6. For those operators that return a result, specifying the type of that result (again, see below).

Observe that points 4, 5, and 6 taken together imply that the system knows precisely which expressions are legal, and for those expressions that are legal it knows the type of the result as well.

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—I could have used SQL, but operator definitions in SQL involve a number of details that I don't want to get into here—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 a point, and it's obtained by invoking the POINT selector; 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, the discussions in this section to this point have been framed in terms of user defined types, for the most part. But similar considerations apply to system defined types too, except that in this case the various definitions are furnished by the system instead of by some user. For example, if INTEGER is a system defined type, then it's the system that defines the name, specifies legal integers, defines the hidden representation, and—as we've already seen—defines the corresponding operators ":=", "=", "+", and so on (though users can define additional operators too, of course).

There's one last point I want to make. I've mentioned selector operators several times; what I haven't mentioned, however, is that selectors—more precisely, selector invocations—are really just a generalization of the more familiar concept of a *literal*. What I mean by this remark is that all literals are selector invocations, although not all selector invocations are literals (in fact, a selector invocation is a literal if and only if its arguments are themselves all specified as literals in turn); for example, POINT(X,Y) and POINT(1.0,2.5) are both invocations of the POINT selector, but only the second is a POINT literal. It follows that every type must have, not just an associated selector, but also an associated format for writing literals. (And perhaps I should add for completeness that every value of every type must be denotable by means of some literal.)

Scalar vs. 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 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 the corresponding attributes—and hence tuple and relation values, variables, and so on are also all nonscalar.

That said, I must now emphasize that these notions are quite informal. Indeed, we've already seen that the concept of data value atomicity has no absolute meaning, and "scalarness" is just that same concept by another name. Thus, the relational model certainly doesn't rely on the scalar vs. nonscalar distinction in any formal sense. In this book,

* This paragraph touches on another important logical difference, incidentally: namely, that between arguments and parameters (see Exercise 2-5 at the end of the chapter). Note too that the POINT selector, unlike the selectors discussed earlier, takes two arguments (because points are represented by pairs of values, not just by a single value).

however, I do rely on it informally (I mean, I find it intuitively useful); 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")—and note that, for simplicity, I now define the attributes all to be of some system defined type:

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

Explanation:

- The keyword VAR in line 1 means this is a variable definition; S is the name of that variable; and 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 defined to be 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 CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

is an example of a *generated* type. A generated type is a type that's obtained by invoking some *type generator* (in the example, the type generator is, specifically, 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're interested in are TUPLE and RELATION. Here's an example involving the TUPLE type generator:

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

The value of variable S_TUPLE at any given time is a tuple with the same heading as that of relvar S (I've deliberately specified the attributes in a different sequence, just to show 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 S_TUPLE. In **Tutorial D**:

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

Important: I don't want you to misunderstand me here. While a variable like S_TUPLE might certainly be needed in some application program that accesses the suppliers-and-parts database, I'm not saying 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 Appendix A, in connection with what's called *The Information Principle*.

By the way, note carefully that (as the foregoing example suggests) there's a logical difference between a given tuple *t* and the relation *r* that contains just that tuple *t*. In particular, they're of different types—*t* is of some tuple type and *r* is of some relation type (though the types do at least have the same heading).

Finally, a few miscellaneous points to close this section:

- Even though tuple and relation types do 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. (Recall the discussion of physical data independence in Chapter 1.)
- Like scalar types, tuple and relation types certainly need associated selector operators (and literals as a special case). I'll defer the details to the next chapter. They don't need THE_ operators, however; instead, they have operators that provide access to the corresponding attributes (and those operators play a role somewhat analogous to that played by THE_ operators in connection with scalar types).
- Tuple and relation types also need assignment and equality comparison operators. I gave an example of tuple assignment earlier in the present section; I'll defer details of the other operators to the next chapter.

Scalar Types in SQL

I turn now to SQL. SQL supports the following more or less self-explanatory system defined scalar types:*

```
BOOLEAN                 NUMERIC(p,q)      DATE
CHARACTER(n)            DECIMAL(p,q)      TIME
CHARACTER VARYING(n)    INTEGER           TIMESTAMP
FLOAT(p)                SMALLINT          INTERVAL
```

A number of defaults, abbreviations, and alternative spellings—e.g., CHAR for CHARACTER, VARCHAR for CHARACTER VARYING, INT for INTEGER—are also supported; I won't bother to go into details, except to note that SQL, unlike **Tutorial D**, requires its character string types to have an associated length specification. Points arising:

* It also allows users to define their own types, but as I've already said I'm more or less ignoring user defined types in this chapter.

- Literals of more or less conventional format are supported for all of these types. *Note:* For reasons I don't need to get into here, more general selectors aren't needed for these types, and neither are THE_ operators.
- An explicit assignment operator is supported for all of these types. The syntax is:

  ```
  SET <scalar variable ref> = <scalar exp> ;
  ```

  Scalar assignments are also performed implicitly when various other operations (e.g., FETCH) are executed. *Note:* Throughout this book in formal syntax definitions like the one just shown, I use *ref* and *exp* as abbreviations for *reference* and *expression*, respectively.
- An explicit equality comparison operator is also supported for all of these types.* Equality comparisons are also performed implicitly when numerous other operations (for example, joins and unions, as well as grouping and duplicate elimination operations) are executed.
- Regarding type BOOLEAN in particular, I should add that although it's included in the SQL standard, it's supported by few if any of the mainstream SQL products. Of course, boolean expressions can always appear in WHERE, ON, and HAVING clauses, even if the system doesn't support type BOOLEAN as such. In such a system, however, no table can have a column of type BOOLEAN, and no variable can be declared to be of type BOOLEAN. As a consequence, workarounds might sometimes be needed.
- Several further system defined types are also supported, including "character large objects" (CLOB), "binary large objects" (BLOB), "national character strings" (NCHAR, etc.), and others (including an XML type in particular). Details of these types are beyond the scope of this book.
- Finally, in addition to the foregoing scalar types, SQL also supports something it calls domains—but SQL's domains aren't types at all; rather, they're just a kind of factored out "common column definition," with a number of rather strange properties that are, again, beyond the scope of this book. You can use them if you like, but don't make the mistake of thinking they're true relational domains (i.e., types).

Type Checking and Coercion in SQL

SQL supports only a weak form of strong typing (if you see what I mean). To be specific:

- BOOLEAN values can be assigned only to BOOLEAN variables and compared only with BOOLEAN values.

* Unfortunately that support is severely flawed, however. For example, in the case of CHAR and VARCHAR types, it's possible for "=" to give TRUE even when the comparands are clearly distinct (see the section "Collations in SQL"). And it's also possible—for all types, not just CHAR and VARCHAR types—for "=" not to give TRUE even when the comparands aren't distinguishable; in particular, this happens when (but not only when) the comparands are both null.

- Character string values can be assigned only to character string variables and compared only with character string values (where "character string" means either CHAR or VARCHAR).
- Numeric values can be assigned only to numeric variables and compared only with numeric values (where "numeric" means FLOAT, NUMERIC, DECIMAL, INTEGER, or SMALLINT).
- Dates, times, timestamps, and intervals are also subject to certain type checking rules, but the details are beyond the scope of this book.

Thus, for example, an attempt to compare a number and a character string is illegal. However, an attempt to compare two numbers is legal, even if those numbers are of different types—say INTEGER and FLOAT, respectively (in this example, the INTEGER value will be coerced to type FLOAT before the comparison is done). Which brings me to the question of type coercion ... It's a widely accepted principle in computing in general that coercions are best avoided, because they're error prone. In SQL in particular, one bizarre consequence of permitting coercions is that certain unions, intersections, and differences can yield a result with rows that don't appear in either operand! For example, consider the SQL tables T1 and T2 shown in Figure 2-3. Assume that column X is of type INTEGER in table T1 but NUMERIC(5,1) in table T2, and column Y is of type NUMERIC(5,1) in table T1 but INTEGER in table T2. Now consider the SQL query:

```
SELECT X , Y FROM T1
UNION
SELECT X , Y FROM T2
```

As Figure 2-3 suggests, columns X and Y in the result of this query (see the rightmost table in the figure) are both of type NUMERIC(5,1), and all values in those columns are obtained, in effect, by coercing some INTEGER value to type NUMERIC(5,1). Thus, the result consists exclusively of rows that appear in neither T1 nor T2!—a very strange kind of union, you might be forgiven for thinking.[*]

T1

X	Y
0	1.0
0	2.0

T2

X	Y
0.0	0
0.0	1
1.0	2

X	Y
0.0	1.0
0.0	2.0
0.0	0.0
1.0	2.0

FIGURE 2-3. A very strange "union"

[*] In connection with this example, one reviewer suggested that the "strangeness" of the union might not matter in practice, since at least no information has been lost in the result. Well, that observation might be valid, in this particular example. But if SQL wants to define an operator that manifestly doesn't behave like the union operator of the relational model, then it seems to me that, first, it doesn't help the cause of understanding to call that operator "union"; second (and more important), it isn't incumbent on me to show that that "union" can sometimes cause problems—rather, it's incumbent on the SQL designers to show that it can't.

Strong recommendation: Do your best to avoid coercions wherever possible. (My own clear preference would be to do away with them entirely, regardless of whether we're in the SQL context or some other context.) In the SQL context in particular, I recommend that you ensure that *columns with the same name are always of the same type*; this discipline, along with others recommended in this book, will go a long way toward ensuring that type conversions in general are avoided. And when they can't be avoided, I recommend doing them explicitly, using CAST or some CAST equivalent. For example (with reference to the foregoing UNION query):

```
SELECT CAST ( X AS NUMERIC(5,1) ) AS X , Y FROM T1
UNION
SELECT X , CAST ( Y AS NUMERIC(5,1) ) AS Y FROM T2
```

For completeness, however, I need to add that certain coercions are unfortunately built into SQL and can't be avoided. (I realize the following remarks might not make much sense at this point in the book, but I don't want to lose them.) To be specific:

- If a table expression *tx* is used as a row subquery, then the table *t* denoted by *tx* is supposed to have just one row *r*, and that table *t* is coerced to that row *r*. *Note:* The term *subquery* occurs ubiquitously in SQL contexts. I'll explain it in detail in Chapter 12; prior to that point, you can take it to mean just a table expression enclosed in parentheses.
- If a table expression *tx* is used as a scalar subquery, then the table *t* denoted by *tx* is supposed to have just one column and just one row and hence to contain just one value *v*, and that table *t* is doubly coerced to that value *v*. *Note:* This case occurs in connection with SQL-style aggregation in particular (see Chapter 7).
- In practice, the row expression *rx* in the ALL or ANY comparison *rx* θ *sq* (where θ is a comparison operator such as >ALL or <ANY and *sq* is a subquery) often consists of a simple *scalar* expression, in which case the scalar value denoted by that expression is effectively coerced to a row that contains just that scalar value. *Note:* Throughout this book, I use the term *row expression* to mean either a row subquery or a row selector invocation (where *row selector* in turn is my preferred term for what SQL calls a row value constructor—see Chapter 3); in other words, I use *row expression* to mean any expression that denotes a row, just as I use *table expression* to mean any expression that denotes a table. As for ALL or ANY comparisons, they're discussed in Chapter 11.

Finally, SQL also uses the term *coercion* in a very special sense in connection with character strings. The details are beyond the scope of this book.

Collations in SQL

SQL's rules regarding type checking and coercion, in the case of character strings in particular, are actually much more complex than I've been pretending so far, and I need to elaborate somewhat. Actually it's impossible in a book of this nature to do more than just scratch the surface of the matter, but the basic idea is this: Any given character string (a) consists of characters from one associated *character set* and (b) has one associated *collation*.

A collation—also known as a collating sequence—is a rule that's associated with a specific character set and governs the comparison of strings of characters from that character set. Let *C* be a collation for character set *S*, and let *a* and *b* be any two characters from *S*. Then *C* must be such that exactly one of the comparisons $a < b$, $a = b$, and $a > b$ evaluates to TRUE and the other two to FALSE (under *C*).

So much for the basic idea. Then those two strings are clearly distinct, and yet they're considered to "compare equal" if PAD Space applies. R

So much for the basic idea. However, there are complications. One arises from the fact that any given collation can have either PAD SPACE or NO PAD defined for it. Suppose the character strings 'AB' and 'AB ' (note the trailing space in the second of these) both have the same associated character set and the same collation. Then those two strings are clearly distinct, and yet they're considered to "compare equal" if PAD SPACE applies. **Recommendation:** Don't use PAD SPACE—always use NO PAD instead, if possible. (Note, however, that the choice between PAD SPACE and NO PAD affects comparisons only—it makes no difference to assignments.)

Another complication arises from the fact that the comparison $a = b$ might evaluate to TRUE, under a given collation, even if the characters *a* and *b* are distinct. For example, we might define a collation called CASE_INSENSITIVE in which each lowercase letter is considered to compare equal to its uppercase counterpart. As a consequence, again, strings that are clearly distinct will sometimes compare equal.

We see, therefore, that certain comparisons of the form $v1 = v2$ can give TRUE in SQL even if *v1* and *v2* are distinct (and possibly even if they're of different types). I'll use the term "equal but distinguishable" to refer to such pairs of values. Now, equality comparisons are performed, often implicitly, in numerous contexts (examples include MATCH, LIKE, UNIQUE, UNION, and JOIN), and the kind of equality involved in all such cases is indeed "equal even if distinguishable." For example, let collation CASE_INSENSITIVE be as defined above, and let PAD SPACE apply to that collation. Then, if the PNO columns of tables P and SP both use that collation, and if 'P2' and 'p2 ' are PNO values in, respectively, some row of P and some row of SP, those two rows will be regarded as satisfying the foreign key constraint from SP to P, despite the lowercase 'p' and trailing space in the foreign key value.

What's more, when evaluating expressions involving operators such as UNION, INTERSECT, EXCEPT, JOIN, GROUP BY, and DISTINCT, the system sometimes has to decide which of several equal but distinguishable values is to be chosen for some column in some result row. Unfortunately, SQL itself fails to give complete guidance in such situations. As a consequence, certain table expressions are indeterminate—the SQL term is *possibly nondeterministic*—in the sense that SQL doesn't fully specify how they should be evaluated; indeed, they might quite legitimately give different results on different occasions. For example, if collation CASE_INSENSITIVE applies to column Z in table T, then SELECT MAX(Z) FROM T might return 'ZZZ' on one occasion and 'zzz' on another, even if T hasn't changed in the interim.

I won't give SQL's rules here for when a given expression is "possibly nondeterministic" (I'll do that in Chapter 12). It's important to note, however, that such expressions aren't allowed in integrity constraints (see Chapter 8), because they could cause updates to succeed or fail unpredictably. Observe in particular, therefore, that this rule implies among other things that many table expressions—even simple SELECT expressions, sometimes—aren't allowed in constraints if they involve a column of some character string type! **Strong recommendation:** Avoid possibly nondeterministic expressions as much as you can.

Row and Table Types in SQL

Here repeated from the section "Scalar vs. Nonscalar Types" is an example of a tuple variable definition:

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

The expression TUPLE{...} here is, as you'll recall, an invocation of the TUPLE type generator. SQL has a corresponding ROW type generator (though it refers to it as a type *constructor*). Here's an SQL analog of the foregoing **Tutorial D** example:

```
DECLARE S_ROW /* SQL row variable */
    ROW ( SNO    VARCHAR(5) ,
          SNAME  VARCHAR(25) ,
          STATUS INTEGER ,
          CITY   VARCHAR(20) ) ;
```

Unlike tuples, however, rows in SQL have a left to right ordering to their components;* in the case at hand, there are actually 24 (= 4 * 3 * 2 * 1) different row types all consisting of the same four components (!).

SQL also supports row assignment. Here's an analog of the foregoing **Tutorial D** tuple assignment:

```
SET S_ROW = ( S WHERE SNO = 'S1' ) ;
```

> **NOTE**
> **The expression on the right side here is a row subquery—i.e., it's a table expression, syntactically speaking, but it's one that's acting as a row expression (see the earlier section "Type Checking and Coercion in SQL").**

Row assignments are also involved, in effect, in SQL UPDATE statements (see Chapter 3).

Turning to tables: Interestingly, SQL doesn't really have a TABLE type generator (or constructor) at all!—i.e., it has nothing directly analogous to the RELATION type generator described earlier in this chapter. However, it does have a mechanism, CREATE TABLE, for

* Oddly enough, SQL refers to the components of row types produced by explicit invocation of the ROW type constructor (and to the components of rows of such types) not as columns but as *fields*.

defining what by rights should be called table variables. For example, recall this definition from the section "Scalar vs. Nonscalar Types":

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

Here's an SQL analog:

```
CREATE TABLE S
  ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER      NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( SNO ) ) ;
```

Note carefully, however, that there's nothing—no sequence of linguistic tokens—in this example that can logically be labeled "an invocation of the TABLE type constructor." (This fact might become more apparent when you realize that the specification UNIQUE(SNO) doesn't have to come after the column definitions but can appear almost anywhere—e.g., between the definitions of columns SNO and SNAME.) In fact, to the extent that the variable S can be regarded (in SQL) as having any type at all, that type is nothing more than *bag of rows*, where the rows in question are of type ROW (SNO VARCHAR(5), SNAME VARCHAR(25), STATUS INTEGER, CITY VARCHAR(20)).

That being said, I should also say that SQL does support something it calls "typed tables." The term isn't very appropriate, however, because if TT is a "typed table" that has been defined to be "of type *T*," then TT is *not* of type *T*, and neither are its rows! More important, I think you should avoid such tables anyway, because they're inextricably intertwined with SQL's support for *pointers*, and pointers are prohibited in the relational model.* In fact, if some table has a column whose values are pointers to rows in some other table, then that table can't possibly represent a relation in the relational model sense. As I've just indicated, however, such tables are unfortunately permitted in SQL; the pointers are called *reference values*, and the columns that contain them are said to be of some *REF type*. Quite frankly, it's not clear why these features are included in SQL at all; certainly there seems to be no useful functionality that can be achieved with them that can't equally well—in fact, better—be achieved without them. **Strong recommendation:** Don't use them, nor any features related to them.

* Perhaps I should elaborate briefly on what I mean by the term *pointer*. A pointer is a value (an *address*, essentially) for which certain special operators—notably referencing and dereferencing operators—are, and in fact must be, defined. Here are rough definitions of those operators: (a) Given a variable *V*, the referencing operator applied to *V* returns the address of *V*; (b) given a value *v* of type pointer (i.e., an address), the dereferencing operator applied to *v* returns the variable that *v* points to (i.e., the variable with the given address).

ASIDE

To avoid a possible confusion, I should add that SQL actually uses the terminology of "referencing" in two quite different senses. One is as sketched above. The other, and older, sense has to do with foreign keys; a foreign key value in one row is said to "reference" the row that contains the corresponding key value. Note, however, that foreign keys aren't pointers!—there are several logical differences between the two concepts, including in particular the fact that foreign keys refer to rows, which are values, whereas pointers are addresses and therefore, by definition, refer to variables (recall from Chapter 1 that it's variables, not values, that "have location").

Concluding Remarks

It's a 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 among other things that this is indeed a misconception. Rather, relations can have attributes of *any type whatsoever*—the relational model nowhere prescribes just what those types must be, and in fact they can be as complex as you like (other than 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 we know that 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 were 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. Prerelational databases were full of pointers, and access to such databases involved a lot of pointer chasing, a state of affairs that made application programming error prone and direct end user access impossible. (These aren't the only problems with pointers, but they're among the most obvious ones.) 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*? And what exactly is a literal?

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. This chapter has touched on several more logical differences (refer back to Chapter 1 if you need to refresh your memory regarding this important notion), including:

argument	*vs.*	*parameter*
database	*vs.*	*DBMS*
foreign key	*vs.*	*pointer*
generated type	*vs.*	*nongenerated type*
relation	*vs.*	*type*
scalar	*vs.*	*nonscalar*
type	*vs.*	*representation*
user defined type	*vs.*	*system defined type*
user defined operator	*vs.*	*system defined operator*

What exactly is the logical difference in each of these cases?

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*. Why do you think it's so called?

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 (section "What's a Type?") as a template, 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. With reference to the departments-and-employees database from Chapter 1 (see Figure 1-1), suppose the attributes are of the following user defined types:

```
DNO    : DNO
DNAME  : NAME
BUDGET : MONEY
ENO    : ENO
ENAME  : NAME
SALARY : MONEY
```

Suppose departments also have a LOCATION attribute, of user defined type CITY (say). 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. `LOCATION = 'London'`

b. `ENAME = DNAME`

c. `SALARY * 5`

d. `BUDGET + 50000`

e. `ENO > 'E2'`

f. `ENAME || DNAME`

g. `LOCATION || '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? Does SQL support such a type?

Exercise 2-18. In the relational world, the equality operator "=" applies to every type. By contrast, SQL doesn't require "=" to apply to every type (I didn't mention this fact in the body of the chapter, but it's true of user defined types in particular), and it doesn't fully define 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 in the relational world 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 (see Chapter 5). 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 first example of an SQL SELECT expression in this chapter, I pointed out that there was no terminating semicolon because the expression *was* an expression and not a statement. But what's the difference?

Exercise 2-24. Explain as carefully as you can the logical difference between a relation with a relation valued attribute (RVA) and a "relation" with a repeating group.

Exercise 2-25. What's a subquery?

Exercise 2-26. In the body of the chapter, I said the equality operator "=" is supposed to apply to every type. But what about SQL row and table types?

CHAPTER THREE

Tuples and Relations, Rows and Tables

FROM THE FIRST TWO CHAPTERS YOU SHOULD HAVE GAINED A PRETTY GOOD UNDERSTANDING OF WHAT TUPLES AND RELATIONS ARE, AT LEAST INTUITIVELY. Now I want to define those concepts more precisely, and I want to explore some of the consequences of those more precise definitions; also, I want to describe the analogous SQL constructs (viz., rows and tables) and offer some specific recommendations to help with our goal of using SQL relationally. Perhaps I should warn you that the formal definitions might look a little daunting—but that's not unusual with formal definitions; the 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 quite familiar to you.

What's a Tuple?

Is this a tuple?—

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

Well, no, 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 in that picture as well as the attribute names). 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 (bad?) picture of the very same tuple:

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

Thus, while I'll certainly be making use of pictures like these in the pages 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 an *attribute 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: 4. The heading is also said to have degree 4.

Type names: CHAR, CHAR, INTEGER, and CHAR.

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

Corresponding attribute values: 'S1', 'Smith', 20, and 'London'. Note the quotes enclosing the character string values here, incidentally; I didn't show any such quotes in the pictures, but perhaps I should have done—it would have been more correct.

ASIDE

Suppose for a moment, as we did in the section “Equality Comparisons” in Chapter 2, that attribute SNO was of type SNO (a user defined type) instead of type CHAR. Then it would be even more incorrect to say the SNO value in the tuple we’re talking about was S1, or even ‘S1’; rather, it would be SNO(‘S1’). A value of type SNO is a value of type SNO, not a value of type CHAR!—a difference in type is certainly a major logical difference. (Recall from Chapter 2 that the expression SNO(‘S1’) is a selector invocation—in fact, a literal—of type SNO.)

Heading: The easiest thing to do here is show another picture:

SNO	CHAR	SNAME	CHAR	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	CHAR	CITY	CHAR	SNO	CHAR

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 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 CHAR , SNAME CHAR , 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, naturally, if the value is a tuple). Here’s a tuple selector invocation for our example (**Tutorial D** again):

```
TUPLE { SNO 'S1' , SNAME '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/expression pair, where the specified expression denotes the corresponding attribute value; the attribute type is omitted because it can always be inferred from the type of the specified expression.

Here’s another example (unlike the previous one, this one isn’t a literal, because not all of its arguments are specified as literals in turn):

```
TUPLE { SNO SX , SNAME 'James' , STATUS STX , CITY SCX }
```

I’m assuming here that SX, STX, and SCX are variables of types CHAR, INTEGER, and CHAR, respectively.

As these examples indicate, a tuple selector invocation in **Tutorial D** consists in general of the keyword TUPLE, followed by a commalist of attribute-name/expression pairs, the whole commalist enclosed in braces. Note, therefore, that the keyword TUPLE does double duty in **Tutorial D**—it's used in connection both with tuple selector invocations, as we've just seen, and with tuple type names as we saw earlier. An analogous remark applies to the keyword RELATION (see the section "What's a Relation?" later in this chapter).

Consequences of the Definitions

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—despite the fact that SQL often, though not always, refers to them explicitly as null *values*. **Recommendation:** Since the phrase "null value" is a contradiction in terms, don't use it; always say just "null" instead.

Now, 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 in the next chapter I'll give some much more pragmatic reasons as well.

The next consequence is: *Every subset of a tuple is a tuple and every subset of a heading is a heading.* (*Note:* In accordance with usual practice, throughout this book I take expressions 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. In the case at hand, for example, every tuple is a subset of itself, and so is every heading. When I want to exclude such possibilities, I'll talk explicitly in terms of *proper* subsets and supersets.) By way of example, given our usual tuple for supplier S1, what we might call "the {SNO,CITY} value" within that tuple is itself another tuple (of degree two):

SNO : CHAR	CITY : CHAR
S1	London

Its heading is as indicated, and its type is TUPLE {SNO CHAR, CITY CHAR}. Likewise, the following is a tuple too:

SNO : CHAR
S1

This tuple is of degree one, and its type is TUPLE {SNO CHAR}. Note, therefore, that if we're given such a tuple and we want to access the actual attribute *value*—'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.

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*, in order 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, however, 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.

The last thing I want to discuss in this section is 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, 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 in detail, 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 *tx* and *ty* 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 *vx* of *Ai* in *tx* is equal to the value *vy* of *Ai* in *ty*.

Also (this might seem obvious, but it needs to be said), two tuples are *duplicates* of each other 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. The reason is that tuples are fundamentally sets (sets of components, to be specific), and such operators make no sense for sets.

Rows in SQL

SQL supports rows, not tuples; in particular, it supports *row types*, a *row type constructor*, and *row value constructors*, which are analogous, somewhat, to tuple types, the TUPLE type generator, and tuple selectors, respectively, in **Tutorial D**. (Row types and row type constructors, though not row value constructors, were also discussed in Chapter 2.) But these analogies are loose at best, because, crucially, rows, unlike tuples, have a left to right ordering to their components. For example, the expressions ROW(1,2) and ROW(2,1)—both of which are legitimate row value constructor invocations—represent two different rows in SQL. *Note:* The keyword ROW in an SQL row value constructor invocation is optional, and in fact is almost always omitted in practice.

Thanks to that left to right ordering, row components in SQL can be, and are, identified by ordinal position instead of by name. For example, consider the following row value constructor invocation (actually it's a row literal, though SQL doesn't use that term):

```
( 'S1' , 'Smith' , 20 , 'London' )
```

This row clearly has (among other things) a component with the value 'Smith'; logically speaking, however, we can't say that component is "the SNAME component," we can say only that it's the *second* component.

I should add that rows in SQL always contain at least one component; SQL has no analog of the 0-tuple of the relational model.

As noted in Chapter 2, SQL also supports a row assignment operation.* In particular, such assignments are involved (in effect) in SQL UPDATE statements. For example, the following UPDATE statement—

```
UPDATE S
SET    STATUS = 20 , CITY = 'London'
WHERE  CITY = 'Paris' ;
```

—is defined to be logically equivalent to this one (note the row assignment in the second line):

```
UPDATE S
SET  ( STATUS , CITY ) = ( 20 , 'London' )
WHERE  CITY = 'Paris' ;
```

As for comparison operations, most boolean expressions in SQL, including (believe it or not) simple "scalar" comparisons in particular, are actually defined in terms of rows rather than scalars. Here's an example of a SELECT expression in which the WHERE clause contains an explicit row comparison:

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) = ( 20 , 'London' )
```

This SELECT expression is logically equivalent to the following one:

```
SELECT SNO
FROM   S
WHERE  STATUS = 20 AND CITY = 'London'
```

As another example, the expression:

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) <> ( 20 , 'London' )
```

* Strictly speaking, I shouldn't be talking about assignments of any kind in this chapter, since the chapter is concerned with values, not variables. But it's convenient to include at least this brief mention of SQL row assignment here.

is logically equivalent to:

```
SELECT SNO
FROM   S
WHERE  STATUS <> 20
OR     CITY <> 'London'
```

Note that OR in the last line here!

Moreover, since row components have a left to right ordering, SQL is also able to support "<" and ">" as row comparison operators. Here's an example:

```
SELECT SNO
FROM   S
WHERE  ( STATUS , CITY ) > ( 20 , 'London' )
```

This expression is logically equivalent to:

```
SELECT SNO
FROM   S
WHERE  STATUS > 20
OR   ( STATUS = 20 AND CITY > 'London' )
```

In practice, however, the vast majority of row comparisons involve rows of degree one, as here:

```
SELECT SNO
FROM   S
WHERE  ( STATUS ) = ( 20 )
```

Now, all of the row expression comparands in the examples so far have been, specifically, row value constructors. But if a row value constructor consists of a single scalar expression enclosed in parentheses, then the parentheses can be dropped, as here:

```
SELECT SNO
FROM   S
WHERE  STATUS = 20
```

The "row comparison" in the WHERE clause here is effectively a scalar comparison (STATUS and 20 are both scalar expressions). Strictly speaking, however, there's no such thing as a scalar comparison in SQL; the expression STATUS = 20 is still technically a row comparison (and the "scalar" comparands are effectively coerced to rows), so far as SQL is concerned.

Recommendation: Unless the rows being compared are of degree one (and thus effectively scalars), don't use the comparison operators "<", "<=", ">", and ">="; they rely on left to right column ordering, they have no straightforward counterpart in the relational model, and in any case they're seriously error prone. (It's not irrelevant to mention in this connection that when this functionality was first proposed for SQL, the standardizers had great difficulty in defining the semantics properly; in fact, it took them several attempts before they got it right.)

What's a Relation?

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

SNO : CHAR	SNAME : CHAR	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 all of 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 for you 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 of that term, 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 informally as "tables" but are known in mathematics, formally, as relations.

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—for example:

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

The order in which the tuples are specified is arbitrary. Here's another example (unlike the previous one, this one isn't a literal):

```
RELATION { tx1 , tx2 , tx3 }
```

I'm assuming here that *tx1*, *tx2*, and *tx3* are tuple expressions and are all of the same tuple type. As these examples suggest, a relation selector invocation in **Tutorial D** consists in general of the keyword RELATION, followed by a commalist of tuple expressions enclosed in braces.

Consequences of the Definitions

Most of the properties of relations I talked about in Chapter 1 are direct consequences of the definitions discussed above, but there are some points I didn't call out explicitly before, and I want to elaborate on some of the others. The first two I want to mention are as follows:

- Relations never contain duplicate tuples—because the body of a relation is a set (a set of tuples) and sets in mathematics don't contain duplicate elements.
- Relations never contain nulls—because the body of a relation is a set of tuples, and we've already seen that tuples in turn never contain nulls.

But these two points are so significant, and there's so much I need to say about them, that I'll defer detailed treatment to the next chapter. In the next few sections, I'll address a series of possibly (?) less weighty issues.

Relations and Their Bodies

The first point I want to discuss here is this: *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. Then relvar SP will have as its current value the empty shipments relation, which we might draw like this (and now I revert to the convention by which we omit the type names from a heading in informal contexts; throughout the rest of the book, in fact, I'll feel free to regard headings as either including or excluding type names—whichever best suits my purpose at the time):

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).

Consider now the relation depicted below:

SNO	PNO	QTY
S1	P1	300

This relation contains just one tuple (equivalently, it's of cardinality one). If we want to access the single tuple it contains, then we'll have to extract it somehow from its containing relation. **Tutorial D** uses syntax of the form TUPLE FROM *rx* for this purpose (where *rx* is any expression that denotes a relation of cardinality one—for example, the expression RELATION {TUPLE {SNO 'S1', PNO 'P1', QTY 300}}, which is in fact a relation selector invocation). SQL, by contrast, uses coercion: If (a) *tx* is a table expression that's being used as a row subquery (meaning it appears where a row expression is expected), then (b) the table *t* denoted by *tx* is supposed to contain just one row *r*, and (c) that table *t* is coerced to that row r. Here's an example:

```
SET S_ROW = ( S WHERE SNO = 'S1' ) ;
```

We also need to be able to test whether a given tuple *t* appears in a given relation *r*. In **Tutorial D**:

$t \in r$

This expression returns TRUE if *t* appears in *r* and FALSE otherwise. The symbol "∈" denotes the *set membership* operator; the expression *t* ∈ *r* can be pronounced "*t* [is] in *r*." In fact, as you've probably realized, "∈" is essentially SQL's IN—except that the left operand of SQL's IN is usually a scalar, not a row, which means there's some coercion going on once again (i.e., the scalar is coerced to the row that contains it).* Here's an example ("Get suppliers who supply at least one part"):

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO IN ( SELECT SNO
                FROM   SP )
```

Relations Are n-Dimensional

I've stressed the point several times that, while a relation can be pictured as a table, it *isn't* a table. (To say it one more time, a picture of a thing isn't the same as the thing.) Of course, it can be very convenient to think of a relation as a table; after all, tables are user friendly; indeed, as noted in Chapter 1, 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.

* Why exactly is the definite article correct here ("the" row)?

Unfortunately, however, 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 suppliers relation 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 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.*

Relational Comparisons

Like tuple types, relation types are no exception to the rule that the "=" comparison operator must be defined for every type; that is, given two relations *rx* and *ry* of the same relation type *T*, we must at least be able to test whether they're equal. Other comparisons might be useful, too; for example, we might want to test whether *rx* is *included* in *ry* (meaning every tuple in *rx* is also in *ry*), or whether *rx* is *properly* included in *ry* (meaning every tuple in *rx* is also in *ry* but *ry* contains at least one tuple that isn't in *rx*). Here's an example, expressed in **Tutorial D** as usual, of an equality comparison on relations:

```
S { CITY } = P { CITY }
```

The left comparand here 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 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 }
```

The symbol "⊃" here means "properly includes." The meaning of this comparison (considerably paraphrased) is: Are there any suppliers who supply no parts at all?

Other useful relational comparison operators include "⊇" ("includes"), "⊆" ("is included in"), and "⊂" ("is properly included in"). In particular, one extremely common requirement is to be able to perform an "=" comparison between some given relation *rx* and an empty relation of the same type—in other words, a test to see whether *rx* is empty. So it's convenient to define a shorthand:

```
IS_EMPTY ( rx )
```

This expression is defined to return TRUE if the relation denoted by the relational expression *rx* is empty and FALSE otherwise; I'll be relying on it heavily in chapters to come (especially in Chapter 8). The inverse operator is useful too:

* 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 fail to realize that relations are multidimensional already.

```
IS_NOT_EMPTY ( rx )
```

This expression is logically equivalent to NOT (IS_EMPTY(*rx*)).

TABLE_DUM and TABLE_DEE

Recall from the discussion of tuples earlier in this chapter 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. In the same kind of way, 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.

Let *r* be a relation of degree zero, then. How many such relations are there? The answer is: Just two. First, *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 the one place where the idea of thinking of relations as tables breaks down completely).

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—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 be discussing the whole notion of what relations mean in general in Chapters 5 and 6.

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

Now, 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: These two relations (especially TABLE_DEE) play a role in the relational algebra

* Perhaps I should 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 perhaps 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 in the relational world that we're probably not going to change them.

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. Which brings us to SQL…SQL, since it has no counterpart to the 0-tuple, clearly (but unfortunately) has no counterpart to TABLE_DUM or TABLE_DEE either.

Tables in SQL

NOTE

Throughout this section, by the term *table* I mean a table value specifically—an SQL table value, that is—and not a table variable (which is what CREATE TABLE creates). I'll discuss table variables in Chapter 5.

Now, I explained in Chapter 2 that SQL doesn't really have anything analogous to the concept of a relation type at all; instead, an SQL table is considered to consist of rows (a bag of rows, in general, not necessarily a set) that are of a certain row type. It follows that SQL doesn't really have anything analogous to the RELATION type generator, either—though it does support other type generators, including ARRAY, MULTISET, and the one we already know from Chapter 2, ROW. It does, however, have something called a *table value constructor* that's analogous, somewhat, to a relation selector. Here's an example:

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

This expression (actually it's a table literal, though SQL doesn't use this term) evaluates to a table with four—not three!—rows and two columns. What's more, those columns have no names. As I've already explained, the columns of an SQL table are ordered, left to right; as a consequence, those columns can be, and often are, identified by ordinal position instead of name.

By way of another example, consider the following table value constructor invocation:

```
VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
       ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
       ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
       ( 'S4' , 'Clark' , 20 , 'London' ) ,
       ( 'S5' , 'Adams' , 30 , 'Athens' )
```

Note that, in order for this expression to be regarded as a fair approximation to its relational counterpart (i.e., a relation literal denoting the relation that's the current value of relvar S as shown in Figure 1-3), we must:

1. Ensure, for each column of the table specified by the VALUES expression, that all of the values are of the pertinent type. (In particular, if a given relational attribute corresponds to the *i*th ordinal position within any of the specified rows, we must make sure it corresponds to the *i*th ordinal position in all of them.)
2. Ensure that we don't specify the same row twice.

NOTE

As you know, in the relational model a heading is a set of attributes. In SQL, by contrast, because columns have a left to right ordering, it would be more correct to regard a heading as a sequence, not a set, of attributes (or columns, rather). If the recommendations of this book are followed, however, this logical difference can mostly (?) be ignored.

What about table assignment and comparison operators? Well, table assignment is a big topic, and I'll defer the details to Chapters 5 and 7. As for table comparisons, SQL has no direct support, but workarounds are available. For example, here's an SQL counterpart to the relational comparison S{CITY} = P{CITY}:

```
NOT EXISTS ( SELECT CITY FROM S
             EXCEPT
             SELECT CITY FROM P )
AND
NOT EXISTS ( SELECT CITY FROM P
             EXCEPT
             SELECT CITY FROM S )
```

And here's a counterpart to the comparison S{SNO} ⊃ SP{SNO}:

```
EXISTS ( SELECT SNO FROM S
         EXCEPT
         SELECT SNO FROM SP )
AND
NOT EXISTS ( SELECT SNO FROM SP
             EXCEPT
             SELECT SNO FROM S )
```

Column Naming in SQL

In the relational model, (a) every attribute of every relation has a name (i.e., anonymous attributes are prohibited), and (b) such names are unique within the relevant relation (i.e., duplicate attributes names are prohibited). In SQL, such rules are enforced sometimes, but not always. To be specific, they're enforced for the tables that happen to be the current values of table variables—defined via CREATE TABLE or CREATE VIEW—but not for the tables that result from evaluation of some table expression.* **Strong recommendation:** Use AS specifications as appropriate to give proper column names to any column that otherwise (a) wouldn't have a name at all or (b) would have a name that wasn't unique. Here are some examples:

* It's certainly true in this case that SQL fails to enforce the rule against duplicate column names. However, it doesn't quite fail to enforce the rule against anonymous columns; if some column would otherwise have no name, the implementation is supposed to give that column a name that's unique within its containing table but is otherwise implementation dependent. In practical terms, however, there's no real difference between saying something is implementation dependent and saying it's undefined (see Chapter 12). Calling such columns anonymous is thus not too far from the truth.

```
SELECT DISTINCT SNAME , 'Supplier' AS TAG
FROM   S

SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
FROM   S

SELECT MAX ( WEIGHT ) AS MBW
FROM   P
WHERE  COLOR = 'Blue'

CREATE VIEW SDS AS
       SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
       FROM   S ;

SELECT DISTINCT S.CITY AS SCITY , P.CITY AS PCITY
FROM   S , SP , P
WHERE  S.SNO = SP.SNO
AND    SP.PNO = P.PNO

SELECT TEMP.*
FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
     ( SNO , SNAME , STATUS , SCITY ,
       PNO , PNAME , COLOR , WEIGHT , PCITY )
```

Of course, the foregoing recommendation can be ignored if there's no subsequent need to reference the otherwise anonymous or nonuniquely named columns. For example, the third of the foregoing examples could safely be abbreviated in some circumstances (in a WHERE or HAVING clause, perhaps) to just:

```
SELECT MAX ( WEIGHT )
FROM   P
WHERE  COLOR = 'Blue'
```

Note, moreover, that the recommendation can't be followed at all in the case of tables specified by means of VALUES expressions.

Important note: The operators of the relational algebra rely on proper attribute naming in a variety of ways. For example, as we'll see in Chapter 6, the relational UNION operator requires its operands to have the same heading (and hence the same attribute names), and the result then has the same heading as well. One advantage of this scheme is precisely that it avoids the complexities caused by reliance on ordinal position! In order to use SQL relationally, therefore, you should apply the same discipline to the SQL analogs of those relational operators. **Strong recommendation:** As a prerequisite to enforcing such a discipline, if two columns in SQL represent "the same kind of information," give them the same name wherever possible. (That's why, for example, the two supplier number columns in the suppliers-and-parts database are both called SNO and not, say, SNO in one table and SNUM in the other.) Conversely, if two columns represent different kinds of information, it's usually a good idea to give them different names.

The only case where it's impossible to follow the foregoing recommendation is when two columns in the same table both represent the same kind of information. For example, consider a table EMP with columns representing employee number and manager number, respectively, where manager number is itself another employee number. These two columns will have to have different names, say ENO and MNO, respectively. As a consequence, some column renaming will sometimes have to be done, as in this example:

```
( SELECT ENO , MNO FROM EMP ) AS TEMP1
  NATURAL JOIN
( SELECT ENO AS MNO , ... FROM EMP ) AS TEMP2
/* where "..." is EMP columns other than ENO and MNO */
```

Such renaming will also have to be done, if you want to use SQL relationally, if columns simply haven't been named appropriately in the first place (e.g., if you're confronted with a database that's been defined by somebody else). A strategy you might want to consider in such circumstances is the following:

- For every base table, define a view identical to that base table except possibly for some column renaming.
- Make sure the set of views so defined abides by the column naming discipline described above.
- Operate in terms of those views instead of the underlying base tables. (Observe in particular that those views will certainly be updatable.)

Unfortunately, it's not possible to ignore the fact 100 percent that columns have an ordinal position in SQL. (Of course, it's precisely because of this fact that SQL is able to get away with its anonymous columns and duplicate column names.) Note in particular that columns still have an ordinal position in SQL even when they don't need to (i.e., when they're all properly named anyway); this observation applies to columns in base tables and views in particular. **Strong recommendation:** Never write SQL code that relies on such ordinal positioning. Examples of where SQL attaches significance to such positioning include:

- SELECT *
- JOIN, UNION, INTERSECT, and EXCEPT—especially if CORRESPONDING isn't specified, in the last three cases (see Chapter 6)
- In the column name commalist, if specified, following the definition of a range variable (see Chapter 12)
- In the column name commalist, if specified, in CREATE VIEW (see Chapter 9)
- INSERT, if no column name commalist is specified
- ALL and ANY comparisons, if the operands are of degree greater than one (see Chapter 11)
- VALUES expressions

Concluding Remarks

In this chapter I've given precise definitions for the fundamental concepts *tuple* and *relation*. As I said earlier, 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, selectors, and comparisons, as well as a number of important consequences of the definitions (in particular, I briefly described the important relations TABLE_DUM and TABLE_DEE). And I discussed the SQL counterparts of all of these notions, where such counterparts exist. In closing, I'd like to stress the importance of the recommendations (in the section immediately preceding this one) regarding *column naming* in SQL. Later chapters will rely heavily on those recommendations.

Exercises

Exercise 3-1. Define as precisely as you can the terms *attribute*, *body*, *cardinality*, *degree*, *heading*, *relation*, *relation type*, and *tuple*.

Exercise 3-2. State as precisely as you can what it means for (a) two tuples to be equal; (b) two relations to be equal.

Exercise 3-3. 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 3-4. Write a typical **Tutorial D** relation selector invocation. Also show SQL's counterpart, if any, to that selector invocation.

Exercise 3-5. (*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 3-6. 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 3-7. 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 don't involve RVAs. Also give an example of a relation with an RVA such that there's no relation that represents precisely the same information but has no RVA.

Exercise 3-8. Explain the relations TABLE_DUM and TABLE_DEE in your own words. Why exactly doesn't SQL support them?

Exercise 3-9. 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 3-10. What's the logical difference if any—as opposed to the obvious syntactic difference—between the following two SQL expressions?

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

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

Exercise 3-11. What exactly does the following SQL expression mean?

```
SELECT SNO
FROM   S
WHERE  ( NOT ( ( STATUS , SNO ) <= ( 20 , 'S4' ) ) ) IS NOT FALSE
```

Exercise 3-12. Explain in your own words what it means to say that relations are *n*-dimensional.

Exercise 3-13. List as many situations as you can think of in which SQL regards left to right column ordering as significant.

Exercise 3-14. Give an SQL analog for the **Tutorial D** expression IS_NOT_EMPTY(*rx*).

Exercise 3-15. State in your own words, as carefully as you can, the discipline described in the body of the chapter regarding SQL column names.

Exercise 3-16. The column naming discipline referred to in the previous exercise relies on the use of AS specifications. But such specifications can appear in several different contexts; moreover, the syntax sometimes takes the form "*X* AS ..." and sometimes "... AS *X*" (if you see what I mean); and the keyword is sometimes optional and sometimes mandatory.* List all of the contexts in which AS can appear, showing which are of the form "*X* AS ..." and which "... AS *X*", and in which cases the keyword is optional.

* For this reason, in fact, I always show the keyword explicitly, even when it's not required. It can be hard to remember when keywords are optional in SQL and when they're mandatory. And in any case it would surely seem strange, in the case of AS in particular, to talk about something being an "AS clause" or "AS specification" if there isn't any AS.

CHAPTER FOUR

No Duplicates, No Nulls

IN THE PREVIOUS CHAPTER, I SAID THE FOLLOWING (APPROXIMATELY):

- Relations never contain duplicate tuples, because the body of a relation is a set (a set of tuples) and sets in mathematics don't contain duplicate elements.
- Relations never contain nulls, because the body of a relation is a set of tuples, and tuples in turn never contain nulls.

I also suggested that since there was so much to be said about these topics, it was better to devote a separate chapter to them. This is that chapter. *Note:* By definition, the topics in question are SQL topics, not relational ones; in what follows, therefore, I'll use the terminology of SQL rather than that of the relational model (for the most part, at any rate). As you'll soon see, I'll also adopt a simplified, though I hope self-explanatory, shorthand notation for rows.

What's Wrong with Duplicates?

There are numerous practical arguments in support of the position that duplicate rows ("duplicates" for short) should be prohibited. Here I want to emphasize 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:

1. 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*).
2. 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 *exp1*—representing some user query, say—into another such expression *exp2*, such that *exp1* and *exp2* are guaranteed to produce the same result when evaluated but *exp2* has better performance characteristics than *exp1* (at least, we hope so). *Note:* Be aware, however, that the term *query rewrite* is also used in certain products with a more restricted meaning.

Now I can present my argument. The fundamental point I want to make is that certain expression transformations, and hence certain optimizations, that would be valid if SQL were truly relational aren't valid in the presence of duplicates. By way of example, consider the (nonrelational) database shown in Figure 4-1. Note in passing that the tables in that database have no keys (there's no double underlining in the figure). And by the way: If you're thinking the database is unrealistic—and especially if you're thinking you're not going to be convinced by the arguments that follow, therefore—please see the further remarks on this example at the beginning of the next section.

P

PNO	PNAME
P1	Screw
P1	Screw
P1	Screw
P2	Screw

SP

SNO	PNO
S1	P1
S1	P1
S1	P2

FIGURE 4-1. 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.[†]

[*] One reviewer felt strongly that an even more powerful practical argument (in fact, the most practical one of all) is simply that duplicates don't match reality—a database that permits duplicates just hasn't been designed properly and can't be, as I put it in Chapter 1, "a faithful model of reality." I'm very sympathetic to this position. But this book isn't about database design, and duplicates aren't just a database design issue in any case. Thus, what I'm trying to do here is show the problems duplicates can cause, whether or not they're due to bad design. A detailed analysis of this whole issue, design aspects included, can be found in the paper "Double Trouble, Double Trouble" (see Appendix D).

[†] I once quoted this line in a seminar, and an attendee said "You can say that again!" To which I replied "Yes—there's a logical difference between logic and rhetoric."

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 violate one of the original objectives of the relational model, which was *explicitness*—the meaning of the data should be as explicit as possible, since we're supposed to be talking about a shared database. The presence of duplicates strongly suggests that part of that 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. For if not, then (to repeat) why are the duplicates there in the first place?

Now consider the 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 = 'Screw'
OR     P.PNO IN
     ( SELECT SP.PNO
       FROM   SP
       WHERE  SP.SNO = 'S1' )
```

Result: P1 * 3, P2 * 1.

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

Result: P1 * 2, P2 * 1.

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

Result: P1 * 9, P2 * 4.

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

Result: P1 * 8, P2 * 4.

5.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION  ALL
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Result: P1 * 5, P2 * 2.

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

Result: P1 * 3, P2 * 2.

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

Result: P1 * 4, P2 * 2.

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

Result: P1 * 3, P2 * 1.

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

Result: P1 * 1, P2 * 1.

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

Result: P1 * 2, P2 * 2.

12.
```
SELECT P.PNO
FROM   P
WHERE  P.PNAME = 'Screw'
UNION
SELECT SP.PNO
FROM   SP
WHERE  SP.SNO = 'S1'
```

Result: P1 * 1, P2 * 1.

(Actually, certain of the foregoing formulations—which?—are a little suspect, because they effectively assume that every screw is supplied by at least one supplier. But this fact makes no material difference to the argument that follows.)

The first point is that the twelve different formulations produce nine different results—different, that is, with respect to their *degree of duplication*. (By the way, I make no claim 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, then he or she needs to be extremely careful in formulating the query in such a way as to obtain exactly the desired result.

Furthermore, 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 more expensive and less reliable, as well as later in delivery to the marketplace, than it might be.
- 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.

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 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, then there can be no good reason for having duplicates in the first place!). However, I'll have more to say about this suggestion in the next section.

Duplicates: Further Issues

There's much, much more that could be said regarding duplicates and what's wrong with them, but I'll limit myself here to just three more points. First of all, you might reasonably object that in practice base tables, at least, never do include duplicates, and the foregoing 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, even if the input tables themselves have no duplicates at all. For example, here are two possible formulations of the query "Get supplier numbers for suppliers who supply at least one part" (and note here that the input tables certainly don't contain any duplicates):

```
SELECT SNO                    |    SELECT SNO
FROM   S                      |    FROM   S NATURAL JOIN SP
WHERE  SNO IN                 |
     ( SELECT SNO             |
       FROM   SP )            |
```

At least one of these expressions will produce a result with duplicates, in general. (*Exercise*: Given our usual sample data values, what results do the two expressions produce?) So if you don't want to think of the tables in Figure 4-1 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.

Second, 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 Chapter 3, 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, at least in SQL as such.) 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.

Avoiding Duplicates in SQL

The relational model prohibits duplicates; to use SQL relationally, therefore, steps must be taken to prevent them from occurring. Now, if every base table has at least one key (see Chapter 5), then duplicates will never occur in base tables as such. As noted in the previous section, however, certain SQL expressions can still yield result tables with duplicates. Here are some of the cases in which such tables can be produced:

- SELECT ALL
- UNION ALL
- VALUES (i.e., table value constructor invocations)

Regarding VALUES, see Chapter 3. Regarding ALL, note first that this keyword (and its alternative, DISTINCT) can be specified:

- In a SELECT clause, immediately following the SELECT keyword
- In a union, intersection, or difference, immediately following the applicable keyword (UNION, INTERSECT, and EXCEPT, respectively)
- Inside the parentheses in an invocation of a "set function" such as SUM, immediately preceding the argument expression

NOTE

DISTINCT is the default for UNION, INTERSECT, and EXCEPT; ALL is the default in the other cases.

Now, the "set function" case is special; you must specify ALL, at least implicitly, if you want the function to take duplicate values into account, which sometimes you do (see Chapter 7). But the other cases have to do with elimination of duplicate rows, which must always be done, at least in principle, if you want to use SQL relationally. Thus, the obvious recommendations in those cases are: Always specify DISTINCT; preferably do so explicitly; and never specify ALL. Then you can just forget about duplicate rows entirely.

In practice, however, matters aren't quite that simple. Why not? Well, I don't think I can do better here than repeat the essence of what I wrote in this book's predecessor:

> At this point in the original draft, I added 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 SNO FROM S ..., that DISTINCT can safely be ignored. If I write either EXISTS (SELECT DISTINCT ...) or IN (SELECT DISTINCT ...), those DISTINCTs can safely be ignored. If I write SELECT DISTINCT SNO FROM SP ... GROUP BY 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 interest of maintaining good relations (with my reviewers, I mean)—not to follow my own advice elsewhere in this book but only to request duplicate elimination explicitly when it seemed to be 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.

So the **recommendation** (sadly) boils down to this: First, make sure you know when SQL eliminates duplicates without you asking it to. Second, in those cases where you do

* The implication is that SELECT DISTINCT might take longer to execute than SELECT ALL, even if the DISTINCT is effectively a "no op." Well, that might be so; I don't want to labor the point; I'll just observe that the reason those implementations typically can't optimize away unnecessary DISTINCTs is that they don't understand how *key inference* works (i.e., they can't figure out the keys that apply to the result of an arbitrary table expression).

have to ask, make sure you know whether it matters if you don't. Third, in those cases where it matters, specify DISTINCT (but, as Hugh Darwen once said, be annoyed about it). And never specify ALL!

What's Wrong with Nulls?

The opening paragraph from the section "What's Wrong with Duplicates?" 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 want to emphasize 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:

1. I assume you know that any comparison in which at least one 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 it's also 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, as understood in SQL, 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.)
2. I assume you're also familiar with the 3VL truth tables for the familiar logical operators—or *connectives*—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; and 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 in particular—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 4-2, 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 at all*—

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 4-2. A nonrelational database, with a null

Consider now the following (admittedly rather contrived) query on the database of Figure 4-2: "Get (SNO,PNO) pairs where either the supplier and part cities are different or the part city isn't Paris (or both)." Here's an SQL formulation of this query:

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

Now I want to focus on the boolean expression in the WHERE clause:

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

(I've added parentheses for clarity.) 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 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*. Now, 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' )
```

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

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

which 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 (meaning, specifically, 3VL) and the result that's correct in the real world are different!

* It must, because if it didn't, then the part wouldn't satisfy the corresponding *relvar predicate*. This is an important point, but I'm not yet in a position to explain it or even state it properly. See Chapter 5 for further discussion.

By way of another example, consider the following query on the same table P as shown in Figure 4-2 (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 PNO
FROM   P
WHERE  CITY = CITY
```

The real world answer here is surely 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 4-2). SQL, however, will return no part numbers at all.

To sum up: If you have any nulls in your database, then you're getting wrong answers to certain of your queries. What's more, you have no way of knowing, in general, just which queries you're getting wrong answers to and which not; 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.

ASIDE

To all of the above, I can't resist adding that even though SQL does support 3VL, and even though it does support the keyword UNKNOWN, that keyword does *not*—unlike the keywords TRUE and FALSE—denote a value of type BOOLEAN! (This is just one of the numerous flaws in SQL's 3VL support; there are many, many others, but most are beyond the scope of this book.) In other words, type BOOLEAN still contains just two values; "the third truth value" is represented, quite incorrectly, by null! Here are some consequences of this fact:

- Assigning UNKNOWN to a variable X of type BOOLEAN actually sets X to null.
- After such an assignment, the comparison X = UNKNOWN doesn't give TRUE—instead, it gives null (meaning, to spell the point out, that SQL apparently believes, or claims, that it's unknown whether X is UNKNOWN).
- In fact, the comparison X = UNKNOWN always gives null (meaning UNKNOWN), regardless of the value of X, because it's logically equivalent to the comparison "X = NULL" (not meant to be valid syntax).

To understand the seriousness of such flaws, you might care to meditate on the analogy of a numeric type that uses null instead of zero to represent zero.

As with the business of duplicates earlier, there's a lot more that could be said on the whole issue of nulls, but I just want to close with a review of the *formal* argument against them. 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).
- In fact, nulls violate the most fundamental relational principle of all—viz., *The Information Principle* (see Appendix A).

The net of all this 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*.

Avoiding Nulls in SQL

The relational model prohibits nulls; to use SQL relationally, therefore, steps must be taken to prevent them from occurring. First of all, a NOT NULL constraint should be specified, explicitly or implicitly, for every column in every base table (see Chapter 5); then nulls will never occur in base tables as such. Unfortunately, however, certain SQL expressions can still yield result tables containing nulls. Here are some of the situations in which nulls can be produced:

- The SQL "set functions" such as SUM all return null if their argument is empty (except for COUNT and COUNT(*), which correctly return zero).
- If a scalar subquery evaluates to an empty table, that empty table is coerced to a null.
- If a row subquery evaluates to an empty table, that empty table is coerced to a row of all nulls. *Note:* A row of all nulls and a null row aren't the same thing at all, logically speaking (another logical difference here, in fact)—yet SQL does think they're the same thing, at least some of the time. It would take us much too far afield to get into such matters in detail here, however.
- Outer joins and union joins are expressly designed to produce nulls in their result.
- If the ELSE clause is omitted from a CASE expression, an ELSE clause of the form ELSE NULL is assumed.
- If $x = y$, the expression NULLIF(x,y) returns a null.
- The "referential triggered actions" ON DELETE SET NULL and ON UPDATE SET NULL both generate nulls (obviously enough).

Strong recommendations:

- Do specify NOT NULL, explicitly or implicitly, for every column in every base table.
- Don't use the keyword NULL in any other context whatsoever (i.e., anywhere other than a NOT NULL constraint).
- Don't use the keyword UNKNOWN in any context whatsoever.
- Don't omit the ELSE clause from a CASE expression unless you're certain it would never be reached anyway.
- Don't use NULLIF.
- Don't use outer join, and don't use the keywords OUTER, FULL, LEFT, and RIGHT (except possibly as suggested in the section "A Remark on Outer Join" below).
- Don't use union join.
- Don't specify either PARTIAL or FULL on MATCH (they have meaning only when nulls are present). For similar reasons, don't use the MATCH option on foreign key constraints, and don't use IS DISTINCT FROM. (In the absence of nulls, the expression *x* IS DISTINCT FROM *y* is equivalent to the expression *x* <> *y*.)
- Don't use IS TRUE, IS NOT TRUE, IS FALSE, or IS NOT FALSE. The reason is that, if *bx* is a boolean expression, then the following equivalences are invalid only if nulls are present:

  ```
  bx IS TRUE       ≡  bx
  bx IS NOT TRUE   ≡  NOT bx
  bx IS FALSE      ≡  NOT bx
  bx IS NOT FALSE  ≡  bx
  ```

  In other words, IS TRUE and the rest are distractions at best, in the absence of nulls.
- Finally, do use COALESCE on every scalar expression that might otherwise "evaluate to null." (Apologies for the quotation marks, but the fact is that the phrase "evaluates to null" is a solecism.)

This last point requires a little more explanation, however. Basically, COALESCE is an operator that lets us replace a null by some nonnull value "as soon as it appears" (i.e., before it has a chance to do any significant damage). Here's the definition: Let *x*, *y*, ..., *z* be scalar expressions. Then the expression COALESCE (*x*,*y*,...,*z*) returns null if its arguments are all null, or the value of its first nonnull argument otherwise. Of course, you're recommended to make sure at least one of *x*, *y*, ..., *z* is nonnull. Here's an example:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

In this example, if the SUM invocation “evaluates to null”—which it will do in particular for any supplier that doesn’t have any matching shipments—then the COALESCE invocation will replace that null by a zero. Given our usual sample data, therefore, the query produces the following result:

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

A Remark on Outer Join

Outer join is expressly designed to produce nulls in its result and should therefore be avoided, in general. Relationally speaking, it’s a kind of shotgun marriage: It forces tables into a kind of union—yes, I do mean union, not join—even when the tables in question fail to conform to the usual requirements for union (see Chapter 6). It does this, in effect, by padding one or both of the tables with nulls before doing the union, thereby making them conform to those usual requirements after all. But there’s no reason why that padding shouldn’t be done with proper values instead of nulls, as in this example:

```
SELECT SNO , PNO
FROM   SP
UNION
SELECT SNO , 'nil' AS PNO
FROM   S
WHERE  SNO NOT IN ( SELECT SNO FROM SP )
```

Result (note the last line in particular):

SNO	PNO
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5
S5	*nil*

Alternatively, the same result could be obtained by using the SQL outer join operator in conjunction with COALESCE, as here:

```
SELECT SNO , COALESCE ( PNO , 'nil' ) AS PNO
FROM ( S NATURAL LEFT OUTER JOIN SP ) AS TEMP
```

Concluding Remarks

There are a few final remarks I want to make regarding nulls and 3VL specifically. Nulls and 3VL are supposed to be a solution to the "missing information" problem—but I believe I've shown that, to the extent they can be considered a "solution" at all, they're a disastrously bad one. Before I leave the topic, however, I'd like to raise, and respond to, an argument that's often heard in this connection. That argument goes something like this:

> All of those examples you give where nulls lead to the wrong answer are very artificial. Real world queries aren't like that! More generally, most of your criticisms seem very academic and theoretical—I bet you can't show any real practical situations where nulls have given rise to the kinds of problems you worry about, and I bet you can't prove that such practical situations do occur.

Needless to say, I have several responses to this argument. The first is: How do we know nulls *haven't* caused real practical problems, anyway? It seems to me that if some serious real world situation—an oil spill, a collapsed bridge, a wrong medical diagnosis—were found to be due to nulls, there might be valid reasons (nontechnical ones, I mean) why the information would never get out. We've all heard stories of embarrassing failures caused by computing glitches of other kinds, even in the absence of nulls; in my opinion, nulls can only serve to make such failures more likely.

Second, suppose someone (me, for example) were to go around claiming that some software product or application contained a serious logical error due to nulls. Can you imagine the lawsuits?

Third, and most important, I think those of us who criticize nulls don't need to be defensive, anyway; I think we should stand the counterarguments on their head, as it were. After all, it's undeniable that nulls can lead to errors in certain cases. So it's not up to us to prove those "certain cases" might include practical, real world situations; rather, it's up to those who want to defend nulls to prove they don't. I venture to add that it seems to me it would be quite difficult, and very likely impossible, to prove any such thing.

Of course, if nulls are prohibited, then missing information will have to be handled by some other means. Unfortunately, those other means are much too complex to be discussed in detail here. The SQL mechanism of (nonnull) default values can be used in simple cases; but for a more comprehensive examination of the issues involved, including in particular an explanation of how you can still get "don't know" answers when you want them, even from a database without nulls, I'm afraid I'll have to refer you to some of the publications listed in Appendix D.

Exercises

Exercise 4-1. "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 4-2. Let *r* be a relation and let *bx* and *by* be boolean expressions. Then there's a law (used in relational systems to help in optimization) that states that (*r* WHERE *bx*) UNION (*r* WHERE *by*) ≡ *r* WHERE *bx* OR *by* (where the symbol "≡" means *is equivalent to*). If *r* isn't a relation but an SQL table with duplicates, does this law still apply?

Exercise 4-3. Let *a*, *b*, and *c* be sets. Then *the distributive law of intersection over union* (also used in relational systems to help in optimization) states that *a* INTERSECT (*b* UNION *c*) ≡ (*a* INTERSECT *b*) UNION (*a* INTERSECT *c*). If *a*, *b*, and *c* are bags instead of sets, does this law still apply?

Exercise 4-4. Part of the SQL standard's explanation of the FROM clause (as in a SELECT - FROM - WHERE expression) reads as follows:

> [The] result of the <from clause> is the ... cartesian product of the tables identified by [the] <table reference>s [in that <from clause>]. The … cartesian product, *CP*, is the multiset of all rows *r* such that *r* is the concatenation of a row from each of the identified tables …

Note, therefore, that *CP* isn't well defined!—the fact that the standard goes on to say that "The cardinality of *CP* is the product of the cardinalities of the identified tables" notwithstanding. For example, consider the tables T1 and T2 shown below:

T1	C1
	0
	0

T2	C2
	1
	2

Observe now that either of the following fits the above definition for "the" cartesian product *CP* of T1 and T2 (that is, either one could be "the" multiset referred to):

CP1	C1	C2
	0	1
	0	1
	0	2
	0	2

CP2	C1	C2
	0	1
	0	2
	0	2
	0	2

Can you fix up the wording of the standard appropriately?

Exercise 4-5. Consider the following cursor definition:

```
DECLARE X CURSOR FOR SELECT SNO , QTY FROM SP ;
```

Note that (a) cursor X permits updates, (b) the table visible through cursor X permits duplicates, but (c) the underlying table SP doesn't. Now suppose the operation DELETE ... WHERE CURRENT OF X is executed. Then there's no way, in general, of saying which specific row of table SP is deleted by that operation. How would you fix *this* problem?

Exercise 4-6. Please write out one googol times: There's no such thing as a duplicate.

Exercise 4-7. Do you think nulls occur naturally in the real world?

Exercise 4-8. There's a logical difference between null and the third truth value: True or false? (Perhaps I should ask: True, false, or unknown?)

Exercise 4-9. In the body of the chapter, I gave truth tables for one monadic 3VL connective (NOT) and two dyadic 3VL connectives (AND and OR), but there are many other connectives as well (see the next exercise). Another useful monadic connective is MAYBE, with truth table as follows:

p	MAYBE *p*
T	F
U	T
F	F

Does SQL support this connective?

Exercise 4-10. Following on from the previous exercise, how many connectives are there altogether in 2VL? And how many in 3VL? What do you conclude from your answers to these two questions?

Exercise 4-11. A logic is *truth functionally complete* if it supports, directly or indirectly, all possible connectives. Truth functional completeness is an extremely important property; a logic that didn't satisfy it would be like an arithmetic that had no support for certain operations, say "+". Is 2VL truth functionally complete? Is SQL's 3VL truth functionally complete?

Exercise 4-12. Let *bx* be a boolean expression. Then *bx* OR NOT *bx* is also a boolean expression, and in 2VL it's guaranteed to evaluate to TRUE (it's an example of what logicians call a *tautology*). Is it a tautology in 3VL? If not, is there an analogous tautology in 3VL?

Exercise 4-13. With *bx* as in the previous exercise, *bx* AND NOT *bx* is also a boolean expression, and in 2VL it's guaranteed to evaluate to FALSE (it's an example of what logicians call a *contradiction*). Is it a contradiction in 3VL? If not, is there an analogous contradiction in 3VL?

Exercise 4-14. In 2VL, *r* JOIN *r* is equal to *r* and INTERSECT and TIMES are both special cases of JOIN (see Chapter 6). Are these observations still valid in 3VL?

Exercise 4-15. The following is a legitimate SQL row value constructor invocation: ROW (1,NULL). Is the row it represents null or nonnull?

Exercise 4-16. Let *bx* be an SQL boolean expression. Then NOT (*bx*) and (*bx*) IS NOT TRUE are both SQL boolean expressions. Are they equivalent?

Exercise 4-17. Let *x* be an SQL expression. Then *x* IS NOT NULL and NOT (*x* IS NULL) are both SQL boolean expressions. Are they equivalent?

Exercise 4-18. Let DEPT and EMP be SQL tables; let DNO be a column in both; let ENO be a column in EMP; and consider the expression DEPT.DNO = EMP.DNO AND EMP.DNO = 'D1' (this expression might be part of the WHERE clause in some query, for example). Now, a "good" optimizer might very well transform that expression into DEPT.DNO = EMP.DNO AND EMP.DNO = 'D1' AND DEPT.DNO = 'D1', on the grounds that $a = b$ and $b = c$ together imply that $a = c$ (see Exercise 6-13 in Chapter 6). But is this transformation valid? If not, why not? And what are the implications?

Exercise 4-19. Suppose the suppliers-and-parts database permits nulls. Here then is a query on that database, expressed for reasons beyond the scope of this chapter not in SQL but in a kind of pidgin form of relational calculus (see Chapter 10):

```
S WHERE NOT EXISTS SP ( SP.SNO = S.SNO AND SP.PNO = 'P2' )
```

What does this query mean? And is the following formulation equivalent?

```
S WHERE NOT ( S.SNO IN ( SP.SNO WHERE SP.PNO = 'P2' ) )
```

Exercise 4-20. Let *k1* and *k2* be values of the same type. In SQL, then, what exactly do a., b., and c. below mean?

a. *k1* and *k2* are "the same" for the purposes of a comparison in, e.g., a WHERE clause.

b. *k1* and *k2* are "the same" for the purposes of key uniqueness.

c. *k1* and *k2* are "the same" for the purposes of duplicate elimination.

Exercise 4-21. In the body of the chapter, I said that UNION ALL can generate duplicates. But what about INTERSECT ALL and EXCEPT ALL?

Exercise 4-22. Are the recommendations "Always specify DISTINCT" and "Never specify ALL" duplicates of each other?

Exercise 4-23. If TABLE_DEE corresponds to TRUE (or *yes*) and TABLE_DUM to FALSE (or *no*), then what corresponds to UNKNOWN (or *maybe*)?

CHAPTER FIVE

Base Relvars, Base Tables

BY NOW YOU SHOULD BE FAMILIAR WITH THE IDEA THAT RELATION VALUES (RELATIONS FOR SHORT) VS. RELATION VARIABLES (RELVARS FOR SHORT) IS ONE OF THE GREAT LOGICAL DIFFERENCES. Now it's time to take a closer look at that difference; more specifically, it's time to take a closer look at issues that are relevant to relvars in particular, as opposed to relations. *Caveat:* Unfortunately, you might find the SQL portions of the discussion that follows a little confusing, because SQL doesn't clearly distinguish between the two concepts—instead, it uses the same term *table* to mean sometimes a table value, sometimes a table variable. For example, the keyword TABLE in CREATE TABLE clearly refers to a table variable; but when we say that, e.g., table S has five rows, the phrase "table S" clearly refers to a table value (namely, the current value of the table variable called S). Be on your guard for potential confusion in this area.

Let me also remind you of a few further points:

- First of all, a relvar is a variable whose permitted values are relations, and it's specifically relvars, not relations, that are the target for INSERT, DELETE, and UPDATE operations (more generally, for relational assignment operations—recall that INSERT, DELETE, and UPDATE are all just shorthand for certain relational assignments).

- Next, 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.
- Last, 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.

The present chapter deals with base relvars (base tables, in SQL); in fact, it won't hurt too much if you assume throughout the book until further notice that all relvars are base relvars, barring explicit statements to the contrary. Chapter 9 discusses the special considerations, such as they are, that apply to virtual relvars or views.

Data Definitions

As a basis for examples, I'll use the following definitions for the suppliers-and-parts database (**Tutorial D** on the left and SQL on the right, a pattern I'll follow in most of my examples in this chapter):

```
VAR S BASE RELATION         |    CREATE TABLE S
  { SNO    CHAR ,           |      ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  CHAR ,           |        SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER ,        |        STATUS INTEGER      NOT NULL ,
    CITY   CHAR }           |        CITY   VARCHAR(20)  NOT NULL ,
  KEY { SNO } ;             |        UNIQUE ( SNO ) ) ;

VAR P BASE RELATION         |    CREATE TABLE P
  { PNO    CHAR ,           |      ( PNO    VARCHAR(6)   NOT NULL ,
    PNAME  CHAR             |        PNAME  VARCHAR(25)  NOT NULL ,
    COLOR  CHAR ,           |        COLOR  CHAR(10)     NOT NULL ,
    WEIGHT FIXED ,          |        WEIGHT NUMERIC(5,1) NOT NULL ,
    CITY   CHAR }           |        CITY   VARCHAR(20)  NOT NULL ,
  KEY { PNO } ;             |        UNIQUE ( PNO ) ) ;

VAR SP BASE RELATION        |    CREATE TABLE SP
  { SNO    CHAR ,           |      ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    CHAR ,           |        PNO    VARCHAR(6)   NOT NULL ,
    QTY    INTEGER }        |        QTY    INTEGER      NOT NULL ,
  KEY { SNO , PNO }         |        UNIQUE ( SNO , PNO ) ,
  FOREIGN KEY { SNO }       |        FOREIGN KEY ( SNO )
        REFERENCES S        |           REFERENCES S ( SNO ) ,
  FOREIGN KEY { PNO }       |        FOREIGN KEY ( PNO )
        REFERENCES P ;      |           REFERENCES P ( PNO ) ) ;
```

Updating Is Set Level

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, meaning they take entire relations or entire relvars as operands, not just individual tuples.) 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 8—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 change them both at the same time, perhaps like this:

```
UPDATE S                        |    UPDATE S
WHERE SNO = 'S1'                |    SET    CITY = 'New York'
   OR SNO = 'S4' :              |    WHERE  SNO = 'S1'
    { CITY := 'New York' } ;    |    OR     SNO = 'S4' ;
```

What's being updated in this example is a set of two tuples.

One consequence of the foregoing is that there's nothing in the relational model corresponding to SQL's "positioned updates" (i.e., UPDATE or DELETE "WHERE CURRENT OF cursor"), because those operations are tuple level (or row level, rather), not set level, by definition. They do happen to work, most of the time, in today's SQL products, but that's because those products aren't very good at supporting integrity constraints. If they 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. **Recommendation:** Don't do SQL updates through a cursor, unless you can be absolutely certain that problems like the one in the example will never arise. (I say this in full knowledge of the fact that many SQL updates are done through a cursor at the time of writing.)

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. Recall the definitions of *value* and *variable* from Chapter 1. If *V* is subject to update, then *V* must be a variable, by definition—but tuples (like relations) are values and can't be updated, again by definition. What we really mean when we talk of updating tuple *t1* to *t2* (say), 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) you'll let me 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.

Constraint Checking

The fact that update operators are set level implies among other things that integrity constraint checking mustn't be done until all of the updating has been done (see Chapter 8 for further discussion). In other words, a set level update mustn't be treated as a sequence of individual tuple level updates (or row level updates, in SQL). Now, I believe the SQL standard does conform to this requirement—or maybe not; its "row level triggers" might be a little suspect in this connection (see the subsection immediately following). In any case, even if the standard does conform, that's not to say all products do; thus, you should still be on your lookout for violations in this regard.

Triggered Actions

The fact that update operators are set level has another implication too: namely, that "referential triggered actions" such as ON DELETE CASCADE (see the section "More on Foreign Keys" later in this chapter)—more generally, triggered actions of all kinds—also mustn't be done until all of the updating has been done. Here, however, SQL unfortunately does treat set level updates as a sequence of row level ones, at least (as already suggested) in its support for row level triggers if nowhere else. **Recommendation:** Try to avoid operations that are inherently row level. Of course, this recommendation doesn't prohibit set level operations in which the set just happens to be of cardinality one, as in the following example:

```
UPDATE S WHERE SNO = 'S5' :    |    UPDATE S
     { CITY := 'New York' } ;  |    SET    CITY = 'New York'
                               |    WHERE  SNO = 'S5' ;
```

A Final Remark

The net of the discussions in this section overall is that update operations—in fact, all operations—in the relational model are always *semantically atomic*; that is, either they execute in their entirety or they have no effect (except possibly for returning a status code or equivalent). Thus, although we do sometimes describe some set level operation, informally, as if it were shorthand for a sequence of tuple level operations, it's important to understand that all such descriptions are (as I said before) only approximations to the truth.

Relational Assignment

Relational assignment in general works by assigning a relation value, denoted by some relational expression, to a relation variable, denoted by the pertinent relvar name. Here's an example:

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

As we saw in Chapter 1, this assignment is logically equivalent to the following **Tutorial D** DELETE statement:

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

More generally, the DELETE statement

```
DELETE R WHERE bx ;
```

(where *R* is a relvar name and *bx* is a boolean expression) is shorthand for the following relational assignment:

```
R := R WHERE NOT ( bx ) ;
```

Likewise, the **Tutorial D** INSERT statement

```
INSERT R rx ;
```

(where *R* is again a relvar name and *rx* is a relational expression) is shorthand for:

```
R := R D_UNION rx ;
```

NOTE

D_UNION here denotes disjoint union. Disjoint union is like regular union, except that its operand relations are required to have no tuples in common (see Chapter 6).

Finally, the **Tutorial D** UPDATE statement also corresponds to a certain relational assignment—but the details are a little more complicated in this case than they are for INSERT and DELETE, and I'll defer them to Chapter 7.

Table Assignment in SQL

SQL's INSERT, DELETE, and UPDATE are directly analogous to their **Tutorial D** counterparts, and there's not much more to be said, except for a couple of small points regarding INSERT specifically. In SQL, the source for an INSERT operation is specified by means of a table expression (frequently but not necessarily a VALUES expression—see Chapter 3). Thus, INSERT in SQL really does insert a table, not a row, though that table might contain just one row, or even no rows at all. Also, INSERT in SQL supports an option according to which the target table specification can be followed by a parenthesized column name commalist. **Recommendation:** Always exercise this option. For example, the INSERT statement

```
INSERT INTO SP ( PNO , SNO , QTY ) VALUES ( 'P6' , 'S5' , 700 ) ;
```

is preferable to this one—

```
INSERT INTO SP VALUES ( 'S5' , 'P6' , 700 ) ;
```

—because this second formulation relies on the left to right ordering of columns in table SP and the first one doesn't. Here's another example:

```
INSERT INTO SP ( SNO , PNO , QTY ) VALUES ( 'S3' , 'P1' , 500 ) ,
                                          ( 'S2' , 'P5' , 400 ) ;
```

Unfortunately SQL doesn't have a direct analog of relational assignment as such. Indeed, the best it can do for the generic assignment

```
R := rx ;
```

is the following sequence of statements:

```
DELETE FROM T ;
INSERT INTO T ( ... ) tx ;
```

(*T* and *tx* here are the SQL analogs of *R* and *rx*, respectively.) Note in particular that this sequence of statements could fail where its relational counterpart (i.e., the relational assignment) would succeed—for example, if table *T* is subject to the constraint that it mustn't be empty.

The Assignment Principle

I'll close this section by drawing your attention to a principle that, though it's really quite simple, has far reaching consequences: *The Assignment Principle*, which states that after assignment of value *v* to variable *V*, the comparison *v* = *V* must evaluate to TRUE. *Note: The Assignment Principle* is a fundamental principle, not just for the relational model, but for computing in general. It applies to relational assignment in particular, of course, but (to repeat) it's actually relevant to assignments of all kinds. In fact, it's more or less the definition of assignment, as I'm sure you realize. I'll have more to say about it in Chapter 8, when I discuss what's called *multiple* assignment.

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 first is 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:
>
> 1. *Uniqueness:* No possible value for *R* contains two distinct tuples with the same value for *K*.
>
> 2. *Irreducibility:* No proper subset of *K* has the uniqueness property.
>
> If *K* consists of *n* attributes, then *n* is the *degree* of *K*.

Now, the uniqueness property is self-explanatory, but I need to say a little 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 value for relvar S 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 important reason is that if we were to specify a "key" that wasn't irreducible, the DBMS wouldn't be able to enforce the proper uniqueness constraint. For example, suppose we told the DBMS (lying!) that {SNO,CITY} was a key. Then the DBMS 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 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. **Recommendation:** In SQL, never lie to the system by defining as a key some column combination that you know isn't irreducible.

Now, all of the relvars we've seen so far have had just one key. Here by contrast are several self-explanatory examples (in **Tutorial D** only, for simplicity) of relvars with two or more. 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 } ;
```

By the way, you might have noticed that there's a tiny sleight of hand going on here. A key is supposed to be a set of attributes, and an attribute is supposed to be an attribute-name/type-name pair; yet the **Tutorial D** KEY syntax specifies just attribute names, not attribute-name/type-name pairs. The syntax works, however, because attribute names are unique within the pertinent heading, and the corresponding type names are thus specified implicitly. In fact, analogous remarks apply at many places in the **Tutorial D** language, and I won't bother to repeat them every time, letting this one paragraph do duty for all.

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. (By definition, integrity constraints constrain updates, and updates apply to variables, not values. See Chapter 8 for further discussion.)

Second, in the case of base relvars in particular, it's usual, as noted in Chapter 1, to single out one key as the *primary* key (and any other keys for the relvar in question are then said to be *alternate* keys). But whether some key is to be chosen as primary, and if so which one, are essentially psychological issues, beyond the purview of the relational model as such. As a matter of good practice, most base relvars probably should have a primary key—but, to repeat, this rule, if it is a rule, really isn't a relational issue as such.

Third, if *R* is a relvar, then *R* certainly does have, and in fact must 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*—i.e., the entire heading—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 certainly something that's both unique and irreducible. *Note:* These remarks don't necessarily apply to SQL tables—SQL tables allow duplicate rows and so might have no key at all. **Strong recommendation:** For base tables, at any rate, use PRIMARY KEY and/or UNIQUE specifications to ensure that every such table does have at least one key.

Fourth, note that key values are *tuples* (rows, in SQL), not scalars. 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 'S1' }
```

(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 'S1', rather—but it really isn't.

By the way, it should now be clear that keys, like so much else in the relational model, rely crucially on the concept of *tuple equality*. That is, in order to enforce the uniqueness constraint, we need to be able to tell whether 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.

* On the other hand, it does make sense to say of some relation that it either does or does not *satisfy* some key constraint. We might even go further and say, a trifle sloppily, that a relation that satisfies a given key constraint actually "has" that key—though such a manner of speaking is likely to cause confusion, and I wouldn't recommend it.

Fifth, let *SK* be a subset of the heading of relvar *R* that possesses the uniqueness property but not necessarily the irreducibility property. Then *SK* is a *superkey* for *R* (and a superkey that isn't a key is called a *proper* superkey). For example, {SNO} and {SNO,CITY} are both superkeys—and the latter is a proper superkey—for relvar S. Note that the heading of any relvar *R* is always a superkey for *R*, by definition.

My final point has to do with the notion of *functional dependence*. I don't want to get into a lot of detail regarding that concept here—I'll come back to it in Chapter 8, also in Appendix B—but you're probably familiar with it anyway; all I want to do here is call your attention to the following. Let *SK* be a superkey (possibly a key) for relvar *R*, and let *A* be any subset of the heading of *R*. Then *R* necessarily satisfies the functional dependence

$$SK \rightarrow A$$

To elaborate briefly: In general, the functional dependence $SK \rightarrow A$ means that whenever two tuples of *R* have the same value for *SK*, they also have the same value for *A*. But if two tuples have the same value for *SK*, where *SK* is a superkey, then by definition they must be the very same tuple!—and so they *must* have the same value for *A*. In other words, loosely: We always have "functional dependence arrows" out of superkeys (and therefore out of keys in particular) to everything else in the relvar. Again, I'll have a little more to say on these matters in Chapter 8 and Appendix B.

More on Foreign Keys

I remind you from Chapter 1 that, loosely speaking, a foreign key is a set of attributes in one relvar whose values are required to match the values of some candidate key in some other relvar (or possibly the same relvar). In the suppliers-and-parts database, for example, {SNO} and {PNO} are foreign keys in SP whose values are required to match, respectively, values of the candidate key {SNO} in S and values of the candidate key {PNO} in P. (By *required to match* here, I mean that if, e.g., relvar SP contains a tuple with SNO value S1, then relvar S must also contain a tuple with SNO value S1—for otherwise SP would show some shipment as being supplied by a nonexistent supplier, and the database wouldn't be "a faithful model of reality.") Here now is a more 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* such that there exists a possibly empty sequence of attribute renamings on *R1* that maps *K* into *K'* (say), where *K'* and *FK* contain exactly the same attributes. Further, let *R2* and *R1* be subject to the constraint that, at all times, every tuple *t2* in *R2* has an *FK* value that's the *K'* value for some (necessarily unique) tuple *t1* in *R1* at the time in question. Then *FK* is a *foreign key* (with the same *degree* as *K*); the associated constraint is a *referential constraint*; and *R2* and *R1* are the *referencing relvar* and the corresponding *referenced relvar*, respectively, for that constraint.

As an aside, I note that the relational model as originally formulated required foreign keys to match not just some key, but specifically the primary key, of the referenced relvar. Since we don't insist on primary keys, however, we certainly can't insist that foreign keys match primary keys specifically, and we don't (and SQL agrees with this position).

In the suppliers-and-parts database, to repeat, {SNO} and {PNO} are foreign keys in SP, referencing the sole candidate key—in fact, the primary key—in S and P, respectively. Here now is a more complicated example:

```
VAR EMP BASE RELATION           |  CREATE TABLE EMP
  { ENO CHAR ,                  |    ( ENO VARCHAR(6) NOT NULL ,
    MNO CHAR ,                  |      MNO VARCHAR(6) NOT NULL ,
    ... }                       |       ..... ,
  KEY { ENO }                   |      UNIQUE ( ENO ) ,
  FOREIGN KEY { MNO }           |      FOREIGN KEY ( MNO )
     REFERENCES EMP { ENO }     |      REFERENCES EMP ( ENO ) ) ;
        RENAME ( ENO AS MNO ) ; |
```

As you can see, there's a significant difference between the two FOREIGN KEY specifications here. I'll explain the **Tutorial D** one first.* Attribute MNO denotes the employee number of the manager of the employee identified by ENO (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), and so the "referencing relvar" (*R2* in the definition) and the "referenced relvar" (*R1* in the definition) are one and the same in this example. More to the point, foreign key values, like candidate key values, are *tuples*; so we have to do some renaming in the foreign key specification, in order for the tuple equality comparison to be at least syntactically valid. (What tuple equality comparison? *Answer:* 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 attributes and thus certainly the same attribute names.) That's why, in the **Tutorial D** specification, the target isn't just EMP but, rather, EMP{ENO} RENAME (ENO AS MNO). *Note:* The RENAME operator is described in detail in the next chapter; for now, I hope it's self-explanatory.

Turning now to SQL: With reference to the definition of the foreign key concept given earlier, in SQL the key *K* and matching foreign key *FK* are sequences, not sets, of columns. (In other words, left to right column ordering is significant once again.) Let those columns, in sequence as defined within the applicable FOREIGN KEY specification, be *B1*, *B2*, ..., *Bn* (for

* In the interest of accuracy, I should explain that **Tutorial D** doesn't actually include any explicit foreign key support at the time of writing. However, proposals to add such support are under active consideration (see the paper "Inclusion Dependencies and Foreign Keys," mentioned in Appendix D), and it's convenient to pretend for the purposes of this book that those proposals have in fact been adopted.

FK) and *A1*, *A2*, ..., *An* (for *K*).* Then columns *Bi* and *Ai* ($1 \le i \le n$) must be of the same type, but they don't have to have the same name. That's why the SQL specification

```
FOREIGN KEY ( MNO ) REFERENCES EMP ( ENO )
```

is correct as it stands, without any need for renaming.

Recommendation: Despite this last point, ensure that foreign key columns do have the same name in SQL as the corresponding key columns wherever possible (see the discussion of column naming in Chapter 3). However, there are certain situations—two of them, to be precise—in which this recommendation can't be followed 100 percent:

- When some table *T* has a foreign key matching some key of *T* itself (as in the EMP example)
- When some table *T2* has two distinct foreign keys both matching the same key *K* in table *T1*

Even here, however, you should at least try to follow the recommendation in spirit, as it were. For example, you might want to ensure in the second case that one of the foreign keys has the same column names as *K*, even though the other one doesn't (and can't). See Exercise 5-15 at the end of the chapter.

Referential Actions

As you probably know, SQL supports not just foreign keys as such but also certain *referential actions*, such as CASCADE. Such actions 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.

Now, referential actions might well 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. Thus, 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 any of the prescriptions of the model (and are in the spirit of the model and can be shown to be useful, I suppose I should add). To elaborate:

* Columns *A1*, *A2*, ..., *An* must be the columns named in some UNIQUE or PRIMARY KEY specification for the target table, but they don't necessarily have to appear in that specification in the same sequence as they do in the FOREIGN KEY specification. Moreover, they and the parentheses surrounding them can be omitted entirely from this latter specification—but then they must appear in a PRIMARY KEY specification, not a UNIQUE specification, for the target table, and of course they must also appear in that specification in the appropriate sequence.

- 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—perhaps even including support for type inheritance—is highly desirable, to say the least.
- *Triggered procedures:* Strictly speaking, a triggered procedure is an action (the *triggered action*) to be performed if a specified event (the *triggering event*) occurs—but the term is often used loosely to include the triggering event as well. *Referential triggered actions* such as ON DELETE CASCADE are just a pragmatically important example of this more general construct, in which the action is DELETE (actually the "procedure" in this particular case is specified declaratively), and the triggering event is ON DELETE.* No triggered procedures are prescribed by the relational model, but they aren't necessarily proscribed either—though they would be if they led to a violation of either the model's set level nature or *The Assignment Principle*, both of which they're likely to do in practice. *Note:* The combination of a triggering event and the corresponding triggered action is often known just as a *trigger*. **Recommendation:** As discussed earlier, avoid use of SQL's row level triggers, and don't use triggers of any kind in such a way as to violate *The Assignment Principle*.
- By way of a third example, the relational model has almost nothing to say about recovery and concurrency controls, but this fact 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 does rely 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, also because they're part of the model as originally defined. But I'd like to stress the fact 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 8. (In fact, much the same could be said for candidate keys as well, but there the practical benefits of providing a shorthand are overwhelming.)

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 (*disciplined* might 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.

* In case you're wondering about the SQL terminology here, ON DELETE CASCADE is a "referential triggered action" and CASCADE by itself is a "referential action."

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.

In general, 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), and they stand for values of the applicable types (CHAR, CHAR, 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 statement:

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

This statement is in fact a *proposition*, which in logic is a statement that's either true or false, unconditionally. Here are a couple of examples:

1. Edward Abbey wrote *The Monkey Wrench Gang*.
2. 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 that propositions must always be true! However, the ones I'm talking about at the moment *are* supposed to be true ones specifically, as I now explain:

- First of all, every relvar has an associated predicate, called the *relvar predicate* for the relvar in question. (The predicate shown above is thus the relvar predicate for relvar S.)
- 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.

Given our usual sample value for relvar S, for example, we assume the following propositions all evaluate to TRUE at this time:

> *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. For example, the tuple

```
TUPLE { SNO 'S6' , SNAME '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're entitled to 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.*

To sum up: A given relvar *R* contains, at any given time, *all* and *only* the tuples that represent true propositions (true instantiations of the relvar predicate for *R*) at the time in question—or, at least, that's what we always assume in practice. In other words, in practice we adopt what's called *The Closed World Assumption* (see Appendix A for more on this topic).

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 for a given relvar varies over time, but the intension does not.

Two final points regarding terminology:

- You might possibly be familiar with the term *predicate* already, since SQL uses it extensively to refer to what this book calls a boolean expression (i.e., SQL talks about "IN predicates," "EXISTS predicates," and so on). Now, this usage on SQL's part isn't exactly incorrect, but it does usurp a very general term—one that's extremely important in relational contexts—and give it a rather specialized meaning, which is why I prefer not to follow that usage myself.
- Talking of usurping general terms and giving them specialized meanings, there's another potential confusion in this area. It has to do with the term *statement*. As you might have realized, logic uses this term in a sense that's very close to its natural language meaning. By contrast, programming languages give it a more limited and specialized meaning: They use it to mean a construct that causes some action to occur, such as defining or updating a variable or changing the flow of control. And I'm afraid this book uses the term in both senses, relying on context to make it clear which meaning is intended. *Caveat lector.*

Relations vs. Types

Chapter 2 discussed types and relations, among other things. However, I wasn't in a position in that chapter to explain the most important logical 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, where 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 we make 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 character strings and integers—and nothing else. (In a real database, of course, our vocabulary will usually be much more extensive than this, especially if any user defined types are involved.)
- The things we can say are things of the form "The supplier with the supplier number denoted by the specified character string is under contract; has the name denoted by another specified character string; has the status denoted by the specified integer; and is located in the city denoted by yet another specified character string"—and nothing else. (Nothing else, that is, except for things *logically implied* by things we can say explicitly. For example, given the things we already know we can say explicitly 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 said is very reminiscent of, and probably has some deep connection to, relational projection… Well, you'd be absolutely right. See the section "What Do Relational Expressions Mean?" in Chapter 6 for further discussion.)

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 would have nothing to talk about; without relations, we couldn't say anything.
2. Types and relations 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.)

ASIDE

When I say that types and relations are necessary and sufficient, I am of course talking only about the logical level. Obviously other constructs (pointers, for example) are needed at the physical level, as we all know—but that's because the objectives are different at that level. The physical level is beyond the purview of the relational model, deliberately.

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 products try to do (though it goes without saying that 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. (In fact, at least one of the products I have in mind has already failed.) The products I have in mind aren't relational products, though; typically, they're products that support "objects" in the object oriented sense, or products that try somehow to marry such objects and SQL tables. Further details are beyond the scope of this book, however.

Here's a slightly more formal perspective on what I've been saying. As we've seen, 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 sets of such propositions, rather), is *a logical system*. And by "logical system" here, 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 invented the relational model back in 1969, that a database (despite the name) 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 to say, which are the ones represented by the tuples in the base relvars—are the axioms of the logical system under discussion. And 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.

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 "data 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 Appendix A.

Exercises

Exercise 5-1. It's sometimes suggested that a relvar is really just a traditional computer file, with tuples instead of records and attributes instead of fields. Discuss.

Exercise 5-2. 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 5-3. Why are SQL's "positioned update" operations a bad idea?

Exercise 5-4. Let SS be a base table with the same columns as table S, and consider the following SQL INSERT statements:

```
INSERT INTO SS ( SNO , SNAME , STATUS , CITY )
     ( SELECT SNO , SNAME , STATUS , CITY
       FROM   S
       WHERE  SNO = 'S6' ) ;

INSERT INTO SS ( SNO , SNAME , STATUS , CITY ) VALUES
     ( SELECT SNO , SNAME , STATUS , CITY
       FROM   S
       WHERE  SNO = 'S6' ) ;
```

Are these statements logically equivalent? If not, what's the difference between them?

Exercise 5-5. *(This is essentially a repeat of Exercise 2-21 from Chapter 2, but you should be able to give a more comprehensive answer now.)* State *The Assignment Principle*. Can you think of any situations in which SQL violates that principle? Can you identify any negative consequences of such violations?

Exercise 5-6. Give definitions for SQL base tables corresponding to the TAX_BRACKET, ROSTER, and MARRIAGE relvars in the section "More on Candidate Keys."

Exercise 5-7. Why doesn't it make sense to say a relation has a key?

Exercise 5-8. In the body of the chapter, I gave one reason why key irreducibility is a good idea. Can you think of any others?

Exercise 5-9. "Key values are not scalars but tuples." Explain this remark.

Exercise 5-10. Let relvar *R* be of degree *n*. What's the maximum number of keys *R* can have?

Exercise 5-11. What's the difference between a key and a superkey? And given that there's such a thing as a superkey, do you think it would make sense to define any such thing as a subkey?

Exercise 5-12. 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. Do such examples lead inevitably to a requirement for null support? (*Answer:* No, but they do serve to show how seductive the nulls idea can be.) What can be done in the example if nulls are prohibited?

Exercise 5-13. Why doesn't SQL have anything analogous to **Tutorial D**'s renaming option in its foreign key specifications?

Exercise 5-14. Can you think of a situation in which two relvars *R1* and *R2* might each have a foreign key referencing the other? If so, what are the implications?

Exercise 5-15. The well known *bill of materials* application involves a relvar (PP, say) showing which parts ("major" parts) contain which parts ("minor" parts) as components, and showing also the corresponding quantities (e.g., "part P1 contains 4 of part P2"). Give appropriate **Tutorial D** and SQL definitions. What referential actions do you think might make sense in this example?

Exercise 5-16. 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 5-17. Define the terms *proposition* and *predicate*. Give examples.

Exercise 5-18. State the predicates for relvars P and SP from the suppliers-and-parts database.

Exercise 5-19. What do you understand by the terms *intension* and *extension*?

Exercise 5-20. 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 some of the 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 5-21. Explain *The Closed World Assumption* in your own terms. Could there be such a thing as *The Open World Assumption*?

Exercise 5-22. 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. What are the implications? Can you think of any uses for such a key?

Exercise 5-23. A predicate has a set of parameters and the empty set is a legitimate set; thus, a predicate could have an empty set of parameters. What are the implications?

Exercise 5-24. What's the predicate for a relvar of degree zero? (Does this question even make sense? Justify your answer.)

Exercise 5-25. Every relvar has some relation as its value. Is the converse true?—that is, is every relation a value of some relvar?

CHAPTER SIX

SQL and Relational Algebra I: The Original Operators

THIS IS THE FIRST OF TWO CHAPTERS ON THE OPERATORS OF THE RELATIONAL ALGEBRA; it discusses the original operators (i.e., the ones mentioned in Chapter 1) in depth, and it also examines certain ancillary but important issues—e.g., the significance of proper attribute (or column) naming once again. It also explains the implications of such matters for our overall goal of using SQL relationally.

Some Preliminaries

Let me begin by reviewing a few points from Chapter 1. First, recall that each algebraic operator takes at least one relation as input and produces another relation as output. Second, recall too 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 operators" (restrict, project, product, intersect, union, difference, and join); now I'm in a position to define those operators, and others, much more carefully. Before I can do that, however, I need to make a few more general points:

1. The operators of the algebra 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 remark applies to object systems?)
2. The operators are also *read-only:* They "read" their operands and they return a result, but they don't update anything. In other words, they operate on relations, not relvars.
3. Please note that the previous point doesn't mean that relational expressions can't refer to relvars. For example, if *R1* and *R2* are relvar names, then *R1* UNION *R2* is certainly a valid relational expression in **Tutorial D** (so long as those relvars are of the same type, that is). In that expression, however, *R1* and *R2* don't denote those relvars as such; rather, they denote the relations that happen to be the current values of those relvars at that time. In other words, we can certainly use a relvar name to denote a relation operand—and such a *relvar reference* in itself 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 latter 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 thing as 3 + 2, no more and no less.
4. Finally, given that the operators of the algebra are indeed all read-only, it follows that INSERT, DELETE, and UPDATE (and relational assignment), though they're certainly relational operators, aren't part of the algebra as such—though, regrettably, you'll often come across remarks to the contrary in the literature.

I also need to say something about the design of **Tutorial D**, because its support for the algebra is significantly different from SQL's. The overriding point is that, in operations like UNION or JOIN that need some correspondence to be established between operand attributes, **Tutorial D** does so by requiring the attributes in question to be, formally, the very same attribute (i.e., to have the same name and same type). For example, here's a **Tutorial D** expression for the join of parts and suppliers on cities:

```
P JOIN S
```

The join operation here is performed, by definition, on the basis of part and supplier cities, CITY being the sole attribute that P and S have in common. Here by contrast is the same operation in SQL (note the last line in particular, where the required correspondence of attributes—or columns, rather—is spelled out explicitly):

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

* Not necessarily in SQL, though! For example, if *T1* and *T2* are SQL table names, we typically can't write things like *T1* UNION *T2*—we have to write something like SELECT * FROM *T1* UNION SELECT * FROM *T2* instead.

Actually this example can be formulated in many different ways in SQL. Here are three more. As you can see, 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 column called either P.CITY or S.CITY):

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

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
```

However, 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 as realized in **Tutorial D**:

- SQL permits, and sometimes requires, dot qualified names. **Tutorial D** doesn't. *Note:* I'll have more to say about SQL's dot qualified names in Chapter 12.
- **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 name" concept; in effect, it replaces that concept by the idea that attributes sometimes need to be renamed, as mentioned under the previous point. *Note:* I'll be discussing SQL's correlation names in detail in Chapter 12.
- The foregoing example didn't illustrate the point, but SQL sometimes uses left to right column ordering to define column correspondences, too. **Tutorial D** doesn't.
- As well as either explicitly or implicitly supporting certain features of the relational algebra, SQL also explicitly supports certain features of the relational calculus (correlation names are a case in point, and EXISTS is another). **Tutorial D** doesn't.

One result 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 consequences for the optimizer. (I once wrote a paper on this topic called "Fifty Ways to Quote Your Query"—see Appendix D—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.)

- SQL requires most queries to conform to its SELECT - FROM - WHERE template. **Tutorial D** has no analogous requirement. *Note:* I'll have more to say on this particular issue in the next chapter.

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

More on Closure

To say it again, the result of every relational operation is a relation. Conversely, any operator that produces a result that isn't a relation is, by definition, not a relational operator. For example, any operator that produces an ordered result isn't a relational operator (see the discussion of ORDER BY in the next chapter). And in SQL in particular, the same is true of any operator that produces a result with duplicate rows, or left to right column ordering, or nulls, or anonymous columns, or duplicate column names. Closure is crucial! As I've already said, closure is what makes it possible to write nested expressions in the relational model, and (as we'll see later) it's also important in expression transformation, and hence in optimization. **Strong recommendation:** Don't use any operation that violates closure if you want the result to be amenable to further relational processing.

ASIDE

The remarks in the foregoing paragraph notwithstanding, some writers do regard *relational inclusion* ("⊆"), not without some justification, to be a relational operation—more specifically, to be part of the relational algebra—even though it produces a result that's a truth value, not a relation. The point isn't worth fighting over here, however.

Now, when I say the output from each algebraic operation is another relation, I hope it's clear that I'm talking conceptually; I don't mean the system actually has to materialize that output in its entirety. For example, consider the following expression (a restriction of a join—**Tutorial D** on the left and SQL on the right as usual, and I've deliberately shown all name qualifications explicitly in the SQL version):

```
( P JOIN S )          |    SELECT P.* , S.SNO , S.SNAME , S.STATUS
WHERE PNAME > SNAME   |    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, discarding it if not. Thus, the intermediate result that's the output from the join might never have to exist as a fully materialized relation in its own right at all. (In practice, in fact, 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 of that join 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 rules—to be more specific, *relation type inference rules*—such that if we know the headings (and hence types) of the input relations for an operation, we can infer the heading (and hence type) of the output relation from that operation. And the relational model does include such a set of rules. In the case of join, for example, those rules say the output from P JOIN S is of this type:

```
RELATION { PNO CHAR , PNAME CHAR , COLOR CHAR , WEIGHT FIXED ,
           CITY CHAR , SNO CHAR , SNAME CHAR , 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 theory 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 the type of the result of P JOIN S is as follows (for example):

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

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 a heading as well as a body; thus, every result must have a proper relational heading as well as a proper relational body.

Now, the RENAME operator mentioned in the introduction to this chapter is needed in large part because of the foregoing type inference rules; it allows us to perform, e.g., a join, even when the relations involved don't meet the attribute naming requirements for that operation (speaking a trifle loosely). Here's the definition:

> **Definition:** Let *r* be a relation and let *A* be an attribute of *r*. Then the *renaming r* RENAME (*A* AS *B*) is a relation with (a) heading identical to that of *r* except that attribute *A* in that heading is renamed *B*, and (b) body identical to that of *r* (except that references to *A* in that body are replaced by references to *B*, a nicety that can be ignored for present purposes).

For example:

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

NOTE

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.

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, since it *is* an expression, not a statement or "command," it can be nested inside other expressions. We'll see plenty of examples of such nesting later.

So how does SQL handle this business of result type inference? The answer is: Not very well. First of all, we saw in Chapter 3 that it 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 2 * WEIGHT FROM P). Third, it can also produce results with duplicate column names (for example, consider SELECT DISTINCT P.CITY, S.CITY FROM P, S).

Strong recommendation: Follow the column naming discipline from Chapter 3 wherever necessary to ensure that SQL conforms as far as possible to the relational rules described in this chapter. Just to remind you, that discipline involved using AS specifications to give proper column names to any column that otherwise (a) wouldn't have a name at all or (b) would have a name that wasn't unique—unless there's no subsequent need to reference the otherwise anonymous or nonuniquely named columns, perhaps. My SQL examples in this chapter and the next (indeed, throughout the rest of this book) will all abide by this discipline.

I haven't finished with the example from the beginning of this section. Let's take another look at it:

```
( P JOIN S )            |    SELECT P.* , S.SNO , S.SNAME , S.STATUS
WHERE PNAME > SNAME     |    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 if you think about it, because that expression is supposed to apply to the result of the FROM clause (see the section "Evaluating SQL Expressions," later), 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 type inference 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 this omission by appealing to the fact that you're supposed to be familiar with SQL already. It's tempting to ask, though, whether you had ever thought about this issue before…but I won't.

Now I can go on to describe some other algebraic operators. Please note that I'm not trying to be exhaustive in this chapter (or the next); I won't be covering "all known operators," and I won't even describe 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.

Restriction

> **Definition**: Let *r* be a relation and let *bx* be a boolean expression in which every attribute reference identifies some attribute of *r* and there aren't any relvar references. Then *bx* is a *restriction condition*, and the *restriction* of *r* according to *bx*, *r* WHERE *bx*, is a relation with (a) heading the same as that of *r* and (b) body consisting of all tuples of *r* for which *bx* evaluates to TRUE.

For example:

```
P WHERE WEIGHT < 17.5         |    SELECT *
                              |    FROM   P
                              |    WHERE  WEIGHT < 17.5
```

Let *r* be a relation. Then the restriction *r* WHERE TRUE (or, more generally, any restriction of the form *r* WHERE *bx* where *bx* is any expression, such as 1 = 1, that's identically TRUE) just returns *r*. Such a restriction is known as an *identity restriction*.

NOTE

Tutorial D does support expressions of the form *r* WHERE *bx*, but those expressions aren't limited to being simple restrictions as defined above, because the boolean expression *bx* isn't limited to being a restriction condition but can be more general. Similar remarks apply to SQL also. Examples are given in later chapters.

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. SELECT in SQL—meaning, more precisely, the SELECT clause portion of a SELECT expression—isn't restriction at all but is, rather, a kind of loose combination of UNGROUP, EXTEND, RENAME, and "project" ("project" in quotes because it doesn't eliminate duplicates unless asked to do so). *Note:* UNGROUP and EXTEND are described in the next chapter.

Projection

> **Definition**: Let *r* be a relation and let *A*, *B*, ..., *C* be attributes of *r*. Then the *projection* of *r* on those attributes, *r*{*A*,*B*,...,*C*}, is a relation with (a) heading {*A*,*B*,...,*C*} and (b) body the set of all tuples *x* such that there exists some tuple *t* in *r* with *A* value equal to the *A* value in *x*, *B* value equal to the *B* value in *x*, ..., and *C* value equal to the *C* value in *x*.

For example:

```
P { COLOR , CITY }          |    SELECT DISTINCT COLOR , CITY
                            |    FROM   P
```

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.* The result heading has attributes—or columns—COLOR and CITY (in that order, in SQL).

Let *r* be a relation. Then the projection *r*{*H*}, where {*H*} is all of the attributes—in other words, the heading—of *r*, just returns *r*. Such a projection is known as an *identity projection*.

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

```
P { COLOR , CITY }  and  P { ALL BUT PNO , PNAME , WEIGHT }
```

are equivalent. This feature can save a lot of writing (think of projecting a relation of degree 100 on 99 of its attributes).† Analogous remarks apply to all operators in **Tutorial D** where they make sense.

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

```
S JOIN P { PNO }
```

to mean

* I can't help pointing out that the term "duplicate elimination," which is used almost universally (not just in SQL contexts), would more accurately be *duplication* elimination.

† A relvar, as opposed to a relation, of such a high degree is unlikely, since it would probably be in violation of the principles of normalization (see Appendix B). But such relvars aren't exactly unknown.

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

and not

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

Exercise: Show the difference between these two interpretations, given our usual sample data.

Join

Before I define join as such, it's useful to introduce the concept of "joinability." Relations *r1* and *r2* are *joinable* if and only if attributes with the same name are of the same type—equivalently, if and only if the set theory union of their headings is itself a legal heading. Note that this concept applies not only to join as such but to various other operations as well, as we'll see in the next chapter. Anyway, armed with this notion, I can now define the join operation:

> **Definition:** Let relations *r1* and *r2* be joinable. Then their *natural join* (or just *join* for short), *r1* JOIN *r2*, is a relation with (a) heading the set theory union of the headings of *r1* and *r2* and (b) body the set of all tuples *t* such that *t* is the set theory union of a tuple from *r1* and a tuple from *r2*.

The following example is repeated from the section "Some Preliminaries," except that now I've dropped the explicit name qualifiers in the SQL version where they aren't needed:

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

I remind you, however, that SQL also allows this join 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 commalist of column references in the SELECT clause by a simple "*"):

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

The result heading, given this latter formulation, has attributes or columns CITY, PNO, PNAME, COLOR, WEIGHT, SNO, SNAME, and STATUS (in that order in SQL, though not of course in **Tutorial D**).

There are several more points to be made in connection with the natural join operation. First of all, observe that intersection is a special case (i.e., *r1* INTERSECT *r2* is a special case of *r1* JOIN *r2*, in **Tutorial D** terms). To be specific, it's the special case in which relations *r1* and *r2* aren't merely joinable but are actually of the same type. However, I'll have more to say about INTERSECT later in this chapter.

Next, product is a special case, too (i.e., *r1* TIMES *r2* is a special case of *r1* JOIN *r2*, in **Tutorial D** terms). To be specific, it's the special case in which relations *r1* and *r2* have no attribute names in common. Why? Because, in this case, (a) the set of common attributes is empty; (b) every tuple has the same value for the empty set of attributes (namely, the 0-tuple); thus, (c) every tuple in *r1* joins to every tuple in *r2*, and so we get the product as stated. For completeness, however, I'll give the definition anyway:

> **Definition:** The *cartesian product* (or just *product* for short) of relations *r1* and *r2*, *r1* TIMES *r2*, where *r1* and *r2* have no common attribute names, is a relation with (a) heading the set theory union of the headings of *r1* and *r2* and (b) body the set of all tuples *t* such that *t* is the set theory union of a tuple from *r1* and a tuple from *r2*.

Here's an example:

```
( P RENAME ( CITY AS PCITY ) )    |   SELECT PNO , PNAME , COLOR ,
  TIMES  /* or JOIN */            |      WEIGHT , P.CITY AS PCITY ,
( S RENAME ( CITY AS SCITY ) )    |      SNO , SNAME , STATUS ,
                                  |      S.CITY AS SCITY
                                  |   FROM   P , S
```

Note the need to rename at least one of the two CITY attributes in this example. The result heading has attributes or columns PNO, PNAME, COLOR, WEIGHT, PCITY, SNO, SNAME, STATUS, and SCITY (in that order, in SQL).

Last, join is fundamentally a dyadic operator; however, it's possible, and useful, to define an *n*-adic version of the operator (and **Tutorial D** does), according to which we can write expressions of the form

```
JOIN { r1 , r2 , ... , rn }
```

to join any number of relations *r1, r2, ..., rn*.* 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 just one tuple.) Why is the result TABLE_DEE? Well, consider the following:

- In ordinary arithmetic, 0 is what's called the *identity* (or *identity value*) 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,

* For completeness, **Tutorial D** also supports *n*-adic versions of INTERSECT and TIMES.

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 identically equal to *r* itself (see below). 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 do suggest you try to convince yourself that *r* JOIN TABLE_DEE and TABLE_DEE JOIN *r* are indeed both identically equal to *r*. It might help to point out that the joins in question are actually cartesian products (right?).

Explicit JOINs in SQL

In SQL, the keyword JOIN can be used to express various kinds of join operations (although those operations can always be expressed without it, too). Simplifying slightly, the possibilities—I've numbered them for purposes of subsequent reference—are as follows (*t1* and *t2* are tables, *bx* is a boolean expression, and *C1*, *C2*, ..., *Cn* are columns appearing in both *t1* and *t2*):

1. `t1 NATURAL JOIN t2`
2. `t1 JOIN t2 ON bx`
3. `t1 JOIN t2 USING ( C1 , C2 , ... , Cn )`
4. `t1 CROSS JOIN t2`

I'll elaborate on the four cases briefly, since the differences between them are a little subtle and can be hard to remember:

1. Case 1 has effectively already been explained.
2. Case 2 is logically equivalent to the following:
   ```
   ( SELECT * FROM t1 , t2 WHERE bx )
   ```
3. Case 3 is logically equivalent to a Case 2 expression in which *bx* takes the form
   ```
   t1.C1 = t2.C1 AND t1.C2 = t2.C2 AND ... AND t1.Cn = t2.Cn
   ```
   —except that columns *C1*, *C2*, ..., *Cn* appear once, not twice, in the result, and the column ordering in the heading of the result is (in general) different: Columns *C1*, *C2*, ..., *Cn* appear first (in that order), then the other columns of *t1* in the order in which they appear in *t1*, then the other columns of *t2* in the order in which they appear in *t2*. (Do you begin to see what a pain this left to right ordering business is?)
4. Finally, Case 4 is logically equivalent to the following:
   ```
   ( SELECT * FROM t1 , t2 )
   ```

Recommendations:

1. Use Case 1 (NATURAL JOIN) in preference to other methods of formulating a join (but make sure columns with the same name are of the same type). Note that the NATURAL JOIN formulation will often be the most succinct if other recommendations in this book are followed.
2. Avoid Case 2 (JOIN ON), because it's guaranteed to produce a result with duplicate column names (unless tables *t1* and *t2* have no common column names in the first place). But if you really do want to use Case 2—which you just might, if you want to formulate a greater-than join, say*—then make sure you do some appropriate renaming as well. For example:

   ```
   SELECT TEMP.*
   FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
        ( SNO , SNAME , STATUS , SCITY ,
          PNO , PNAME , COLOR , WEIGHT , PCITY )
   ```
3. In Case 3, make sure columns with the same name are of the same type.
4. In Case 4, make sure there aren't any common column names.

In each of the four cases, the operands *t1* and *t2* are specified by means of what SQL calls *table references*. Let *tr* be such a reference. If the table expression in *tr* is a table subquery, then *tr* must also include an AS clause—even if the "correlation name" defined by that AS clause is never explicitly mentioned anywhere else in the overall expression (see Chapter 12 for further explanation). For example:

```
( SELECT SNO , CITY FROM S ) AS TEMP1
  NATURAL JOIN
( SELECT PNO , CITY FROM P ) AS TEMP2
```

One last point: Be aware that an explicit JOIN invocation isn't allowed in SQL as either (a) a "stand alone" table expression (i.e., one at the outermost level of nesting) or (b) the table expression in parentheses that constitutes a subquery.

Union, Intersection, and Difference

Union, intersection, and difference (UNION, INTERSECT, and MINUS in **Tutorial D**; UNION, INTERSECT, and EXCEPT in SQL) all follow the same general pattern. I'll start with union.

Union

Definition: Let relations *r1* and *r2* be of the same type; then their *union*, *r1* UNION *r2*, is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in *r1* or *r2* or both.

* Greater-than join is a special case of what's called θ-join, which is discussed later in this chapter.

For example (I'll assume for the sake of all of the examples in this section that parts have an extra attribute called STATUS, of type INTEGER):

```
P { STATUS , CITY } UNION     |    SELECT STATUS , CITY
S { CITY , STATUS }           |    FROM   P
                              |    UNION  CORRESPONDING
                              |    SELECT CITY , STATUS
                              |    FROM   S
```

As with projection, it's worth noting explicitly in connection with union that "duplicates are eliminated." Note that we don't need to specify DISTINCT in the SQL version to achieve this effect; although UNION provides the same options as SELECT does (DISTINCT vs. ALL), the default for UNION is DISTINCT, not ALL (for SELECT it's the other way around, as you'll recall from Chapter 4). The result heading has attributes or columns STATUS and CITY—in that order, in SQL. Note that the CORRESPONDING specification in the SQL formulation allows us to ignore the possibility that those columns might appear at different ordinal positions within the operand tables. **Recommendations:**

- Make sure corresponding columns have the same name and type.
- Always specify CORRESPONDING if possible.* If it isn't—in particular, if the SQL product you're using doesn't support it—then at least make sure columns line up properly, as in this revised version of the example:

  ```
  SELECT STATUS , CITY FROM P
  UNION
  SELECT STATUS , CITY FROM S  /* note the reordering */
  ```
- Don't include the "BY (column name commalist)" option in the CORRESPONDING specification, unless it makes no difference anyway (e.g., specifying BY (STATUS,CITY) would make no difference in the example).
- Never specify ALL. *Note:* The usual reason for specifying ALL on UNION isn't that users want to see duplicate rows in the output; rather, it's that they know there aren't any duplicate rows in the input—i.e., the union is disjoint (see below)—and so they're trying to prevent the system from having to do the extra work of trying to eliminate duplicates that they know aren't there in the first place. In other words, it's a performance reason. See the discussion of such matters in Chapter 4, in the section "Avoiding Duplicates in SQL."

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

```
S { CITY } D_UNION P { CITY }
```

Given our usual sample data, this expression will produce a run time error, because supplier cities and part cities aren't disjoint. SQL has no direct counterpart to D_UNION.

* I omitted CORRESPONDING from examples in earlier chapters because at the time it would only have been distracting.

Tutorial D also supports *n*-adic forms of both UNION and D_UNION. I'll skip the details here.

Intersection

> **Definition:** Let relations *r1* and *r2* be of the same type; then their *intersection*, *r1* INTERSECT *r2*, is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in both *r1* and *r2*.

For example:

```
P { STATUS , CITY } INTERSECT    |    SELECT    STATUS , CITY
S { CITY , STATUS }              |    FROM      P
                                 |    INTERSECT CORRESPONDING
                                 |    SELECT    CITY , STATUS
                                 |    FROM      S
```

All comments and recommendations noted under "Union" apply here also, mutatis mutandis. *Note:* 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. As mentioned in a footnote earlier, **Tutorial D** also supports an *n*-adic form, but I'll skip the details here.

Difference

> **Definition:** Let relations *r1* and *r2* be of the same type; then their *difference*, *r1* MINUS *r2* (in that order), is a relation of the same type, with body consisting of all tuples *t* such that *t* appears in *r1* and not *r2*.

For example:

```
P { STATUS , CITY } MINUS      |    SELECT STATUS , CITY
S { CITY , STATUS }            |    FROM   P
                               |    EXCEPT CORRESPONDING
                               |    SELECT CITY , STATUS
                               |    FROM   S
```

All comments and recommendations noted under "Union" apply here also, mutatis mutandis.

Which Operators Are Primitive?

I've now covered all of the operators I want to cover in this chapter. As I've more or less said already, however, not all of those operators are primitive—some of them can be defined in terms of others. One possible primitive set is the one consisting of restrict, project, join (or product), union, and difference. *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 (see Exercise 7-3 in Chapter 7). What this example does show, 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.

Formulating Expressions a Step at a Time

Consider the following **Tutorial D** expression (the query is "Get pairs of supplier numbers such that the suppliers concerned are colocated (i.e., are in the same city)":

```
( ( ( S RENAME ( SNO AS SA ) ) { SA , CITY } JOIN
    ( S RENAME ( SNO AS SB ) ) { SB , CITY } )
        WHERE SA < SB ) { SA , SB }
```

The result has two attributes, called SA and SB (note that it would have been sufficient to do just one renaming; I did two for symmetry). The purpose of the condition SA < SB is twofold:

- It eliminates pairs of supplier numbers of the form (*a,a*).
- It guarantees that the pairs (*a,b*) and (*b,a*) won't both appear.

Be that as it may, I now show another formulation of the query in order to show how **Tutorial D**'s WITH construct can be used to simplify the business of formulating what otherwise might be rather complicated expressions:

```
WITH ( S RENAME ( SNO AS SA ) ) { SA , CITY } AS R1 ,
     ( S RENAME ( SNO AS SB ) ) { SB , CITY } AS R2 ,
     R1 JOIN R2 AS R3 ,
     R3 WHERE SA < SB AS R4 :
     R4 { SA, SB }
```

As the example suggests, a WITH clause in **Tutorial D** consists of the keyword WITH followed by a commalist of specifications of the form *expression* AS *name*, the whole commalist followed by a colon. For each of those "*expression* AS *name*" specifications, the expression is evaluated and the result effectively assigned to a temporary variable with the specified name. Note that any given specification in the commalist is allowed to refer to names introduced in specifications earlier in the same commalist. Note too that WITH isn't really an operator of the relational algebra as such; it's just a device to help with the formulation of complicated expressions (especially ones involving common subexpressions). I'll be making extensive use of it in subsequent chapters.

SQL too supports a WITH construct, with these differences:

- It writes the operands the other way around, thus: WITH *name* AS *expression*, …, *name* AS *expression*.
- It doesn't use the colon separator.
- WITH in **Tutorial D** can be used in connection with expressions of any kind. By contrast, WITH in SQL can be used only in connection with table expressions specifically.

Also, in SQL, the *name* portion of a "*name* AS *expression*" specification can optionally be followed by a parenthesized column name commalist (much as in a range variable definition—see Chapter 12), but it shouldn't be necessary to exercise this option if other recommendations in this book are followed.

Here's an SQL version of the example:

```
WITH T1 AS ( SELECT SNO AS SA , CITY
             FROM   S ) ,
     T2 AS ( SELECT SNO AS SB , CITY
             FROM   S ) ,
     T3 AS ( SELECT *
             FROM   T1 NATURAL JOIN T2 ) ,
     T4 AS ( SELECT *
             FROM   T3
             WHERE  SA < SB )
     SELECT SA , SB
     FROM   T4
```

What Do Relational Expressions Mean?

Recall now from the previous chapter that every relvar has a certain *relvar predicate*, which is, loosely, what the relvar means. For example, the predicate for the suppliers relvar S is:

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

What I didn't mention in that previous chapter, however, is that the foregoing notion extends in a natural way to apply to arbitrary relational expressions. For example, consider the projection of suppliers on all attributes but CITY:

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

This expression denotes a relation containing all tuples of the form

```
TUPLE { SNO sno , SNAME sn , STATUS st }
```

such that a tuple of the form

```
TUPLE { SNO sno , SNAME sn , STATUS st , CITY sc }
```

currently exists in relvar S for some CITY value *sc*. In other words, the result represents the current extension of a predicate that looks like this (see Chapter 5 if you need to refresh your memory regarding the notion of a predicate's extension):

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

This predicate thus represents the meaning of the relational expression (i.e., the projection) S{SNO,SNAME,STATUS}. Observe that it has just three parameters and the corresponding relation has just three attributes—CITY isn't a parameter to that predicate but what logicians call a "bound variable" instead, owing to the fact that it's "quantified" by the phrase *There exists some city* (see Chapter 10 for further explanation of bound variables and quantifiers).* *Note:* A possibly clearer way of making the same point—viz., 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).

Remarks analogous to the foregoing apply to every possible relational expression. To be specific: Every relational expression *rx* always has an associated meaning, or predicate; moreover, the predicate for *rx* can be determined from the predicates for any relvars involved in that expression, together with the semantics of any relational operations involved. As an exercise, you might like to revisit some of the relational (or SQL) expressions shown earlier in this chapter, with a view to determining what the corresponding predicate might look like in each case.

Evaluating SQL Table Expressions

In addition to natural join, Codd originally defined an operator he called θ-join, where θ denoted any of the usual scalar comparison operators ("=", "≠", "<", and so on). Now, θ-join isn't primitive; in fact, it's defined to be a restriction of a product. Here by way of example is the "not equals" join of suppliers and parts on cities (so θ here is "≠"):

```
( ( S RENAME ( CITY AS SCITY ) )   |   SELECT SNO , SNAME , STATUS ,
  TIMES                            |          S.CITY AS SCITY , PNO ,
  ( P RENAME ( CITY AS PCITY ) ) ) |          PNAME , COLOR , WEIGHT ,
WHERE SCITY ≠ PCITY                |          P.CITY AS PCITY
                                   |   FROM   S , P
                                   |   WHERE  S.CITY <> P.CITY
```

Now I want to focus on the SQL formulation specifically. We can think of that SQL expression as being implemented in three steps, as follows:

1. The FROM clause is executed and yields the product of tables S and P. *Note:* If we were doing this relationally, we would have to rename the CITY attributes before that product could be 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 product by eliminating rows in which the two city values are equal. *Note:* If θ had been "=" instead of "≠" (or "<>", rather, in SQL), this step would have been: Restrict the 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

* One reviewer asked why CITY is mentioned in the predicate at all, since it isn't part of the result of the projection. This is an important question! A short answer is: Because that result is obtained by projecting away the CITY attribute specifically, nothing more and nothing less. A much longer answer can be found in my book *Logic and Databases: The Roots of Relational Theory* (Trafford, 2007), pages 387–391.

cities. In other words, an equijoin is a θ-join for which θ is "=". *Exercise:* What's the difference between an equijoin and a natural join?

3. Finally, the SELECT clause is executed and yields a projection of that restriction on the columns specified in the SELECT clause. (Actually it's doing some renaming as well, in this particular example, and I mentioned earlier in this chapter that SELECT provides other functionality too, in general—but I want to ignore these details as well, for simplicity.)

At least to a first approximation, then, the FROM clause corresponds to a product, the WHERE clause to a restriction, and the SELECT clause to a projection; thus, the overall SELECT - FROM - WHERE expression represents a projection of a restriction of a 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. Now, there's no implication that the implementation has to use exactly that algorithm in order to evaluate such expressions; on the contrary, 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) evaluating 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 optimizer is to find an algorithm that's guaranteed to be equivalent to the conceptual one but performs better…which brings us to the next section.

Expression Transformation

In this section, I want to take a slightly closer look at what the optimizer does. More specifically, I want to consider what's involved in transforming some relational expression into another, logically equivalent, expression. *Note:* I mentioned this notion under the discussion of duplicates in Chapter 4, 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).

I'll start with a trivial example. Consider the following **Tutorial D** expression (the query is "Get suppliers who supply part P2, together with the corresponding quantities," and I'll ignore the SQL analog for simplicity):

```
( ( S JOIN SP ) WHERE 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 reads and writes" as a reasonable measure of performance. Neither of these assumptions is very 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. (By contrast, I was assuming for the sake of the example in Step 1, realistically or not, that the 1,000,000 intermediate result tuples couldn't be kept in main memory.)
3. *Project the result of Step 2:* This step involves no tuple reads or writes 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 reads and writes, 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 rather than the first! If it does, then what it's doing (in effect) is transforming the original expression

```
( S JOIN SP ) WHERE 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 = '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, 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 (that was the law I was using in the example). And a good optimizer will know those laws, and will apply them—because the performance of a query ideally shouldn't depend on the specific syntax used to express that query in the first place. *Note:* Actually, 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 I hope is obvious.

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 and only if $f(g(a,b)) = g(f(a),f(b))$ for all *a* and *b*. In ordinary arithmetic, for example, SQRT (nonnegative 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 by 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 union, though not over intersection and 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 above for restrictions.

Two more important general laws are the laws of *commutativity* and *associativity*:

- The dyadic operator *g* is *commutative* if and only if $g(a,b) = g(b,a)$ for all *a* and *b*. In ordinary arithmetic, for example, addition and multiplication are commutative, but subtraction and division aren't. In relational algebra, intersect, union, and join are all commutative,* but difference isn't. So, for example, if a query involves a join of two

* Strictly speaking, the SQL analogs of these operators are *not* commutative, because—among other things—the left to right column order of the result depends on which operand is specified first; indeed, the disciplines recommended in this book in connection with these operators are designed, in part, precisely to avoid such problems. More generally, the possibility of such problems occurring is one reason out of many why you're recommended never to write SQL code that relies on column positioning.

relations *r1* and *r2*, the commutative law tells us it doesn't matter which of *r1* and *r2* 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.

- The dyadic operator *g* is *associative* if and only 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 isn't. So, for example, if a query involves a join of three relations *r1*, *r2*, and *r3*, the associative and commutative laws taken together tell us we can join the relations pairwise in any order we like. The system is thus free to decide which of the various possible sequences is most efficient.

Note, incidentally, that all of these transformations can be performed without any regard for either actual data values or access paths (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.

The Reliance on Attribute Names

There's one question that might have been bothering you but hasn't been addressed in this chapter so far. The operators of the relational algebra, at least as described in this book, all rely heavily on attribute naming. For example, the **Tutorial D** expression *R1* JOIN *R2*—where I'll suppose, just to be definite, that *R1* and *R2* are base relvars—is defined to do the join on the basis of those attributes of *R1* and *R2* that have the same names. But the question often arises: Isn't this approach rather fragile? For example, what happens if we later add a new attribute to relvar *R2*, say, that has the same name as one already existing in relvar *R1*?

Well, let me begin by clarifying one point. It's true that the operators do rely, considerably, on proper attribute naming. However, they also require attributes of the same name to be of the same type (and hence in fact to be the very same attribute, formally speaking). Thus, for example, an error would occur—at compile time, too, I would hope—if, in the expression *R1* JOIN *R2*, *R1* and *R2* both had an attribute called *A* but the two *A*'s were of different types.* Note that this requirement (that attributes of the same name be of the same type) imposes no serious limitation on functionality, thanks to the availability of the RENAME operator.

Now to the substance of the question. In fact, there's a popular misconception here, and I'm very glad to have this opportunity to dispel it. In today's SQL systems, application program access to the database is provided either through a call level interface or through an embedded, but conceptually distinct, data sublanguage ("embedded SQL"). But embedded SQL is really just a call level interface with a superficial dusting of syntactic sugar, so the two approaches come to the same thing from the DBMS's point of view, and

* Actually such an error might not occur in SQL, because SQL permits coercions; but **Tutorial D** doesn't, and the observation is certainly true of **Tutorial D**.

indeed from the host language's point of view as well. In other words, the DBMS and the host language are typically only loosely coupled in most systems today. As a result, much of the advantage of using a well designed, well structured programming language is lost in today's database environment. Here's a quote:* "Most programming errors in database applications would show up as *type errors* [if the database definition were] part of the type structure of the program."

Now, the fact that the database definition is not "part of the type structure of the program" in today's systems can be traced back to a fundamental misunderstanding that existed in the database community in the early 1960s or so. The perception at that time was that, in order to achieve data independence (more specifically, *logical* data independence—see Chapter 9), it was necessary to move the database definition out of the program so that, in principle, that definition could be changed later without changing the program. But that perception was at least partly incorrect. What was, and is, really needed is *two separate definitions*, one inside the program and one outside; the one inside would represent the programmer's perception of the database (and would provide the necessary compile time checking on queries, etc.), the one outside would represent the database "as it really is." Then, if it subsequently becomes necessary to change the definition of the database "as it really is," data independence is preserved by changing the mapping between the two definitions.

Here's how the mechanism I've just briefly described might look in SQL. First we introduce the notion of a *public table*, which represents the application's perception of some portion of the database. For example:

```
CREATE PUBLIC TABLE X           /* hypothetical syntax! */
  ( SNO   VARCHAR(5)  NOT NULL ,
    SNAME VARCHAR(25) NOT NULL ,
    CITY  VARCHAR(20) NOT NULL ,
    UNIQUE ( SNO ) ) ;

CREATE PUBLIC TABLE Y           /* hypothetical syntax! */
  ( SNO   VARCHAR(5)  NOT NULL ,
    PNO   VARCHAR(6)  NOT NULL ,
    UNIQUE ( SNO , PNO ) ) ,
    FOREIGN KEY ( SNO ) REFERENCES X ( SNO ) ) ;
```

These definitions effectively assert that "the application believes" there are tables in the suppliers-and-parts database called X and Y, with columns and keys as specified. Such is not the case—but there are database tables called S and SP (with columns and keys as specified for X and Y, respectively, but with one additional column in each case), and we can define mappings as follows:

* From Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen: "Database Programming in Machiavelli—A Polymorphic Language with Static Type Inference," Proc. ACM SIGMOD International Conference on Management of Data, Portland, Ore. (June 1989).

```
X ::= SELECT SNO , SNAME , CITY FROM S ; /* hypothetical syntax! */

Y ::= SELECT SNO , PNO FROM SP ;         /* hypothetical syntax! */
```

These mappings are defined outside the application (the symbol "::=" here means "is defined as").

Now consider the SQL expression X NATURAL JOIN Y. Clearly the join here is being done on the basis of the common column, SNO. And if, say, a column SNAME is added to the database table SP, all we have to do is change the mapping—actually no change is required at all, in this particular example!—and everything will continue to work as before; in other words, data independence will be preserved.

Unfortunately, today's SQL products don't work this way. Thus, for example, the SQL expression S NATURAL JOIN SP is, sadly, subject to exactly the "fragility" problem mentioned in the original question (so too is the simpler expression SELECT * FROM S, for that matter). However, you can reduce that problem to more manageable proportions by adopting the strategy suggested under the discussion of column naming in Chapter 3. For convenience, I repeat that strategy here:

- For every base table, define a view identical to that base table except possibly for some column renaming.
- Make sure the set of views so defined abides by the naming discipline described in that same discussion (i.e., of column naming) in Chapter 3.
- Operate in terms of those views instead of the underlying base tables.

Now, if the base tables change subsequently, all you'll have to do is change the view definitions accordingly.

Exercises

Exercise 6-1. What if anything is wrong with the following SQL expressions (from a relational perspective or otherwise)?

a. `SELECT * FROM S , SP`

b. `SELECT SNO , CITY FROM S`

c. `SELECT SNO , PNO , 2 * QTY FROM SP`

d. `SELECT S.SNO FROM S , SP`

e. `SELECT S.SNO , S.CITY FROM S NATURAL JOIN P`

f. `SELECT CITY FROM S UNION SELECT CITY FROM P`

g. `SELECT S.* FROM S NATURAL JOIN SP`

Exercise 6-2. 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 6-3. Given the usual suppliers-and-parts database, what's the value of the **Tutorial D** expression JOIN{S,SP,P}? What's the corresponding predicate? And how would you express this join in SQL?

Exercise 6-4. Why do you think the project operator is so called?

Exercise 6-5. Given our usual sample values for the suppliers-and-parts database, what values do the following **Tutorial D** expressions denote? In each case, give both (a) an SQL analog and (b) an informal interpretation of the expression (i.e., a corresponding predicate) in natural language.

a. `( S JOIN ( SP WHERE PNO = 'P2' ) ) { CITY }`

b. `( P { PNO } MINUS ( SP WHERE SNO = 'S2' ) { PNO } ) JOIN P`

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 6-6. Union, intersection, product, and join are all both commutative and associative. Verify these claims. Are they valid in SQL?

Exercise 6-7. Which of the operators described in this chapter have a definition that doesn't rely on tuple equality?

Exercise 6-8. The SQL FROM clause FROM *t1*, *t2*, ..., *tn* (where each *ti* denotes a table) returns the product of its arguments. But what if *n* = 1?—what's the 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 6-9. 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 all pairs of part numbers such that some supplier supplies both of the indicated parts.

g. Get supplier numbers for suppliers with a status lower than that of supplier S1.

h. Get part numbers for parts supplied by all suppliers in London.

i. Get (SNO,PNO) pairs such that the indicated supplier does not supply the indicated part.

j. Get suppliers who supply at least all parts supplied by supplier S2.

Exercise 6-10. Prove the following statements (making them more precise where necessary):

a . A sequence of restrictions of a given relation can be transformed into a single restriction.

b . A sequence of projections of 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 6-11. 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 6-12. Let *r* be a relation. What does the **Tutorial D** expression *r*{} mean (i.e., what's the corresponding predicate)? What does it return? Also, what does the **Tutorial D** expression *r*{ALL BUT} mean, and what does it return?

Exercise 6-13. The boolean expression

```
x > y AND y > 3
```

(which might be part of a query) is 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*. Note that the transformation is certainly 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 SQL products that actually perform this kind of optimization?

Exercise 6-14. Consider the following **Tutorial D** expression:

```
WITH ( P WHERE COLOR = 'Purple' ) AS PP ,
     ( SP RENAME ( SNO AS X ) ) AS T :
S WHERE ( T WHERE X = SNO ) { PNO } ⊇ PP { PNO }
```

What does this expression mean? Given our usual sample data values, show the result returned. Does that result accord with your understanding of what the expression means? Justify your answer.

Exercise 6-15. SQL has no direct counterpart to D_UNION. How best might the D_UNION example from the body of the chapter be simulated in SQL?

Exercise 6-16. What do you understand by the term *joinable?* How could the definition of the term be extended to cover the case of *n* relations for arbitrary *n* (instead of just *n* = 2, which was the case discussed in the body of the chapter)?

Exercise 6-17. What exactly is it that makes it possible to define *n*-adic versions of JOIN and UNION (and D_UNION)? Does SQL have anything analogous? Why doesn't an *n*-adic version of MINUS make sense?

Exercise 6-18. I claimed earlier in the book that TABLE_DEE meant TRUE and TABLE_DUM meant FALSE. Substantiate and/or elaborate on these claims.

SQL and Relational Algebra II: Additional Operators

AS I'VE SAID SEVERAL TIMES ALREADY, AN OPERATOR OF THE RELATIONAL ALGEBRA IS AN OPERATOR THAT TAKES ONE OR MORE RELATIONS AS INPUT AND PRODUCES ANOTHER RELATION AS OUTPUT. As I observed in Chapter 1, however, any number of operators can be defined that conform to this simple characterization. Chapter 6 described the original operators (join, project, etc.); the present chapter describes some of the many additional operators that have been defined since the relational model was first invented. It also considers how those operators can best be realized in SQL.

Semijoin and Semidifference

Join is one of the most familiar of all of the relational operators. In practice, however, it often turns out that many queries that require join at all 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).

> **Definition:** The *semijoin* of relations *r1* and *r2* (in that order), *r1* MATCHING *r2*, is equivalent to (*r1* JOIN *r2*){*A*,*B*,...,*C*}, where *A*, *B*, ..., *C* are all of the attributes of *r1*.

In other words, *r1* MATCHING *r2* is the join of *r1* and *r2*, projected back on the attributes of *r1*. Here's an example ("Get suppliers who currently supply at least one part"):

```
S MATCHING SP              |    SELECT S.* FROM S
                           |    WHERE  SNO IN
                           |         ( SELECT SNO FROM SP )
```

The result heading is the same as that of S. Note that the expressions *r1* MATCHING *r2* and *r2* MATCHING *r1* aren't equivalent, in general. Note too that we could replace IN by MATCH in the SQL version; interestingly, however, we can't replace NOT IN by NOT MATCH in the semidifference analog (see below), because there's no "NOT MATCH" operator in SQL.

Turning to semidifference: 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 at all really require semidifference.

> **Definition:** The *semidifference* between relations *r1* and *r2* (in that order), *r1* NOT MATCHING *r2*, is equivalent to *r1* MINUS (*r1* MATCHING *r2*).

Here's an example ("Get suppliers who currently supply no parts at all"):

```
S NOT MATCHING SP          |    SELECT S.* FROM S
                           |    WHERE  SNO NOT IN
                           |         ( SELECT SNO FROM SP )
```

Again the result heading is the same as that of S. *Note:* If *r1* and *r2* are of the same type, *r1* NOT MATCHING *r2* degenerates to *r1* MINUS *r2*; in other words, difference (MINUS) is a special case of semidifference, relationally speaking. By contrast, join isn't a special case of semijoin—they're really different operators, though it's true that (loosely speaking) some joins are semijoins and some semijoins are joins. See Exercise 7-19 at the end of the chapter.

Extend

You might have noticed that the algebra as I've described it so far has no conventional computational capabilities. Now, SQL does; for example, we can certainly write queries along the lines of SELECT A + B AS C ... (for example). However, as soon as we write that "+" sign, we've gone beyond the bounds of the algebra as originally defined. So we need to add something to the algebra in order to provide this kind of functionality, and that's what EXTEND is for. By way of example, suppose part weights (in P) 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                       |    SELECT P.* ,
   ADD ( WEIGHT * 454 AS GMWT )  |           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

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":

```
( ( EXTEND P ADD              |    SELECT PNO ,
    ( WEIGHT * 454 AS GMWT ) ) |           WEIGHT * 454 AS GMWT
        WHERE GMWT > 7000.0 )  |    FROM   P
                { PNO , GMWT } |    WHERE  WEIGHT * 454 > 7000.0
```

As you can see, the expression WEIGHT * 454 appears twice in the SQL version, and we have to hope the implementation will be smart enough to recognize that it need evaluate that expression just once per tuple (or row) instead of twice. In the **Tutorial D** version, by contrast, the expression appears only once.

The problem this example illustrates is that SQL's SELECT - FROM - WHERE template is just 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 and nested in arbitrary ways; but SQL's SELECT - FROM - WHERE template effectively means that queries *must* be expressed as a product, followed by a restrict, followed by some combination of project and/or extend and/or ungroup and/or rename—and many queries just don't fit this pattern.

Incidentally, you might be wondering why I didn't formulate the SQL version like this:

```
SELECT PNO , 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 at compile time.

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** (and now I'll show all name qualifiers explicitly, for clarity):

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT P.PNO , ( P.WEIGHT * 454 ) AS GMWT
       FROM P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

But not all SQL products allow nested subqueries to appear in the FROM clause in this manner. Note too that this kind of formulation inevitably leads to a need to reference certain variables (TEMP, in the example) before they're defined—quite possibly a long way before they're defined, in fact, in real SQL queries.

I'll close this section with a formal definition:

> **Definition:** Let *r* be a relation. Then the *extension* EXTEND *r* ADD (*exp* AS *X*) is a relation with (a) heading the heading of *r* extended with attribute *X*, and (b) body the set of all tuples *t* such that *t* is a tuple of *r* extended with a value for attribute *X* that's 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 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*.

Image Relations

An image relation is, loosely, the "image" within some relation of some tuple (usually a tuple within some other relation). For example, given the suppliers-and-parts database and our usual sample values, the following is the image within the shipments relation of the supplier tuple for supplier S4:

PNO	QTY
P2	200
P4	300
P5	400

Clearly, this particular image relation can be obtained by means of the following **Tutorial D** expression:

```
( SP WHERE SNO = 'S4' ) { ALL BUT SNO }
```

Here's a formal definition of image relations in general:

> **Definition:** Let relations *r1* and *r2* be joinable (i.e., such that attributes with the same name are of the same type); let *t1* be a tuple of *r1*; let *t2* be a tuple of *r2* that has the same values for those common attributes as tuple *t1* does; let relation *r3* be that restriction of *r2* that contains all and only such tuples *t2*; and let relation *r4* be the projection of *r3* on all but those common attributes. Then *r4* is the *image relation* (with respect to *r2*) corresponding to *t1*.

Here's an example that illustrates the usefulness of image relations:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

Note, incidentally, that the boolean expression in the WHERE clause here is a relational comparison. *Explanation:*

- First of all, the roles of *r1* and *r2* from the definition are being played by the suppliers relation and the shipments relation, respectively (where by "the suppliers relation" I mean the current value of relvar S, and similarly for "the shipments relation").
- Next, we can imagine the boolean expression in the WHERE clause being evaluated for each tuple *t1* in *r1* (i.e., each tuple in the suppliers relation) in turn.
- Consider one such tuple, say that for supplier S*x*. For that tuple, then, the expression !!SP—pronounced "bang bang SP" or "double bang SP"—denotes the corresponding image relation *r4* within *r2*; in other words, it denotes the set of (PNO,QTY) pairs within SP for parts supplied by that supplier S*x*. The expression !!SP is an *image relation reference*.
- The expression (!!SP){PNO}—i.e., the projection of the image relation on {PNO}—thus denotes the set of part numbers for parts supplied by supplier S*x*.
- The expression overall (i.e., S WHERE ...) thus denotes suppliers from S for whom that set of part numbers is equal to the set of all part numbers in the projection of P on {PNO}. In other words, it represents the query "Get suppliers who supply all parts" (speaking a little loosely).

NOTE

Since the concept of an image relation is defined in terms of some given tuple (*t1*, in the formal definition), it's clear that an image relation reference can appear, not in all possible contexts in which relational expressions in general can appear, but only in certain specific contexts: namely, those in which the given tuple *t1* is understood. WHERE clauses are one such context, as the foregoing example indicates, and we'll see another in the section "Image Relations bis."

ASIDE

SQL has no direct support for image relations as such. Here for interest is an SQL analog of the foregoing Tutorial D expression (I show it for your consideration, but I'm not going to discuss it in detail, except to note that it can obviously (?) be improved in a variety of ways):

```
SELECT *
FROM   S
WHERE  NOT EXISTS
     ( SELECT PNO
       FROM   SP
       WHERE  SP.SNO = S.SNO
       EXCEPT
       SELECT PNO
       FROM   P )
AND    NOT EXISTS
     ( SELECT PNO
       FROM   P
       EXCEPT
       SELECT PNO
       FROM   P
       WHERE  SP.SNO = S.SNO )
```

Back to image relations as such: It's worth noting that the "!!" operator can be defined in terms of MATCHING. For example, the example discussed above—

```
S WHERE ( !! SP ) { PNO } = P { PNO }
```

—is logically equivalent to the following:

```
S WHERE ( SP MATCHING RELATION { TUPLE { SNO SNO } } ) { PNO } = P { PNO }
```

Explanation: Again consider some tuple of S, say that for supplier S*x*. For that tuple, then, the expression TUPLE{SNO SNO}—which is a tuple selector invocation—denotes a tuple containing just the SNO value S*x* (the first SNO is an attribute name, the second denotes the value of the attribute of that name in the tuple for S*x* within relvar S). So the expression

```
RELATION { TUPLE { SNO SNO } }
```

—which is a relation selector invocation—denotes the relation that contains just that tuple. Hence, the expression

```
SP MATCHING RELATION { TUPLE { SNO SNO } }
```

denotes a certain restriction of SP: namely, that restriction that contains just those shipment tuples that have the same SNO value as the supplier tuple for supplier S*x* does. The overall result follows.

Suppose now that we're given a revised version of the suppliers-and-parts database—one that's simultaneously both extended and simplified, compared to the usual version—that looks like this (in outline):

```
S   { SNO }       /* suppliers               */
SP  { SNO, PNO }  /* supplier supplies part  */
PJ  { PNO, JNO }  /* part is used in project */
J   { JNO }       /* projects                */
```

Relvar J here represents *projects* (JNO stands for project number), and relvar PJ indicates which parts are used in which projects. Now consider the query "Get all (*sno,jno*) pairs such that *sno* is an SNO value currently appearing in relvar S, *jno* is a JNO value currently appearing in relvar J, and supplier *sno* supplies all parts used in project *jno*." This is a complicated query!—but a formulation using image relations is almost trivial:

```
( S JOIN J ) WHERE !! PJ ⊆ !! SP
```

Reverting now to the usual suppliers-and-parts database, here's another example ("Delete shipments from suppliers in London"—and this time I'll show an SQL analog as well):

```
DELETE SP WHERE IS_NOT_EMPTY   |   DELETE FROM SP
  ( !! (S WHERE                |   WHERE  SNO IN
        CITY = 'London') ) ;   |        ( SELECT SNO FROM S
                               |          WHERE CITY = 'London' ) ;
```

For a given shipment, the specified image relation !! (S WHERE ...) is either empty, if the corresponding supplier isn't in London, or contains exactly one tuple otherwise.

Divide

I include the following discussion of divide in this chapter only to show why (contrary to conventional wisdom, perhaps) I don't think it's very important; in fact, I think it should be dropped. You can skip this section if you like.

I have several reasons (three at least) for wanting to drop divide. One is that any query that can be formulated in terms of divide can alternatively, and much more simply, be formulated in terms of image relations instead, as I'll demonstrate in just a moment. Another is that there are at least seven different divide operators anyway!—that is, there are, unfortunately, at least seven different operators all having some claim to be 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.

Definition: Let relations *r1* and *r2* be such that the heading {*Y*} of *r2* is some subset of the heading of *r1* and the set {*X*} is the other attributes of *r1*. Then the *division* of *r1* by *r2*, *r1* DIVIDEBY *r2*,[*] is shorthand for the following:

```
r1 { X } NOT MATCHING ( ( r1 { X } JOIN r2 ) NOT MATCHING r1 )
```

For example, the expression

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

(given our usual sample data values) yields:

SNO
S1

The expression can thus be loosely characterized as a representation of the query "Get supplier numbers for suppliers who supply all parts" (I'll explain the reason for that qualifier "loosely" in a few moments). In practice, however, we're more likely to want full supplier details (not just supplier numbers) for the suppliers in question, in which case the division will need to be followed by a join:

```
( SP { SNO , PNO } DIVIDEBY P { PNO } ) JOIN S
```

But we already know how to formulate this query more simply using image relations:

```
S WHERE ( !! SP ) { PNO } = P { PNO }
```

This latter formulation is (a) more succinct, (b) easier to understand (at least, it seems so to me), and (c) *correct*. This last point is the crucial one, of course, and I'll explain it below. First, however, I want to explain why divide is called divide, anyway. The reason is that if *r1* and *r2* are relations with no attribute names in common and we form the product *r1* TIMES *r2*, and then divide the result by *r2*, we get back to *r1*;[†] in other words, product and divide are inverses of each other, in a sense.

As I've said, the expression

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

can be loosely characterized as a formulation of the query "Get supplier numbers for suppliers who supply all parts"; in fact, this very example is often used as a basis for explaining, and justifying, the divide operator in the first place. Unfortunately, however, that characterization isn't quite correct. Rather, the expression is a formulation of the

[*] **Tutorial D** doesn't directly support this operator, and *r1* DIVIDEBY *r2* is thus not valid **Tutorial D** syntax.

[†] So long as *r2* isn't empty. What happens if it is?

query “Get supplier numbers for suppliers who *supply at least one part and in fact* supply all parts.”* In other words, the divide operator not only suffers from problems of complexity and lack of succinctness—it doesn’t even solve the problem it was originally, and specifically, designed to address.

Aggregate Operators

In a sense this section is a bit of a digression, because the operators to be discussed aren’t relational but scalar—they return a scalar result.† But I do need to say something about them before I can get back to the main theme of the chapter.

An aggregate operator in the relational model is an operator that derives a single value from the “aggregate” (i.e., the bag or set) of values appearing within some attribute within some relation—or, in the case of COUNT(*), which is slightly special, from the “aggregate” that’s the entire relation as such. Here are two examples:

```
X := COUNT ( S ) ;             |   SELECT COUNT ( * ) AS X
                               |   FROM   S

Y := COUNT ( S { STATUS } ) ;  |   SELECT COUNT ( DISTINCT STATUS )
                               |          AS Y
                               |   FROM   S
```

I’ll focus on the **Tutorial D** statements on the left first. Given our usual sample values, the first assigns the value 5 (the number of tuples in the current value of relvar S) to the variable X; the second assigns the value 3 (the number of tuples in the projection of the current value of relvar S on {STATUS}, which is to say the number of distinct STATUS values in that current value) to the variable Y.

In general, a **Tutorial D** aggregate operator invocation looks like this:

```
<agg op name> ( <relation exp> [, <exp> ] )
```

Legal <*agg op name*>s include COUNT, SUM, AVG, MAX, MIN, AND, OR, and XOR (the last three of which apply to aggregates of boolean values). Within the <*exp*>, an <*attribute ref*> can appear wherever a literal would be allowed. That <*exp*> must be omitted if the <*agg op name*> is COUNT; otherwise, it can be omitted only if the <*relation exp*> denotes a relation of degree one, in which case an <*exp*> consisting of a reference to the sole attribute of that relation is assumed. Here are some examples:

* If you’re wondering what the logical difference is here, consider the slightly different query “Get suppliers who supply all purple parts” (the point being, of course, that there are no purple parts). If there aren’t any purple parts, then every supplier supplies all of them!—even supplier S5, who supplies no parts at all, and is thus not represented in relvar SP, and so can’t be returned by an analogous DIVIDEBY expression. And if you’re still wondering, then see the further discussion of this example in Chapter 11.

† Nonscalar aggregate operators can be defined too, but they’re beyond the scope of this book.

1. `SUM ( SP , QTY )`

   This expression denotes the sum of all quantities in relvar SP (given our usual sample values, the result is 3100).

2. `SUM ( SP { QTY } )`

   This expression is shorthand for SUM(SP{QTY},QTY), and it denotes the sum of all *distinct* quantities in SP (i.e., 1000).

3. `AVG ( SP , 3 * QTY )`

   This expression effectively asks what the average shipment quantity would be if quantities were all triple their current value (the answer is 775). More generally, the expression

   ```
   agg ( rx , x )
   ```

   (where *x* is some expression more complicated than a simple <*attribute ref*>) is essentially shorthand for the following:

   ```
   agg ( EXTEND rx ADD ( x AS y ) , y )
   ```

Now I turn to SQL. For convenience, let me first repeat the examples:

```
X := COUNT ( S ) ;            |   SELECT COUNT ( * ) AS X
                              |   FROM   S

Y := COUNT ( S { STATUS } ) ; |   SELECT COUNT ( DISTINCT STATUS )
                              |          AS Y
                              |   FROM   S
```

Now, you might be surprised to hear me claim that SQL doesn't really support aggregate operators at all! I say this knowing full well that most people would consider expressions like those on the right above to be, precisely, SQL aggregate operator invocations.* But they aren't. Let me explain. As we know, the counts are 5 and 3, respectively. But those SQL expressions don't evaluate to those counts as such, as true aggregate operator invocations would; rather, they evaluate to tables that contain those counts. More precisely, each yields a table with one row and one column, and the sole value in that row is the actual count:

X
5

Y
3

As you can see, therefore, the SELECT expressions really don't represent aggregate operator invocations as such; at best, they represent only approximations to such invocations. In fact, aggregation is treated in SQL as if it were a special case of *summarization*. Now, I haven't discussed summarization in this chapter yet; for present purposes,

* It might be claimed, somewhat more reasonably, that *the COUNT invocations within* those expressions are SQL aggregate operator invocations. But the whole point is that such invocations can't appear as "stand alone" expressions in SQL; rather, they always appear as part of some table expression.

however, you can regard it as what's represented in SQL by a SELECT expression with a GROUP BY clause. Now, the foregoing SQL expressions don't have a GROUP BY clause—but they're defined to be shorthand for the following, which do (and do therefore represent summarizations as claimed):

```
SELECT COUNT ( * ) AS X
FROM   S
GROUP  BY ( )

SELECT COUNT ( DISTINCT STATUS ) AS Y
FROM   S
GROUP  BY ( )
```

> **NOTE**
> **In case these expressions look strange to you, I should explain that SQL does allow both (a) GROUP BY operand commalists enclosed in parentheses and (b) GROUP BY clauses with no operands at all. Specifying a GROUP BY clause with no operands at all is equivalent to omitting the GROUP BY clause entirely.**

So SQL does support summarization—but it doesn't support aggregation as such. Sadly, the two concepts are often confused, and perhaps you can begin to see why. What's more, the picture is confused still further by the fact that, in SQL, it's common in practice for the table that results from an "aggregation" to be coerced to the single row it contains, or even doubly coerced to the single value that row contains: two separate errors (of judgment, if nothing else) thus compounding to make the "aggregation" look more like a true aggregation after all! Such double coercion occurs in particular when the SELECT expression is enclosed in parentheses to form a scalar subquery, as in the following SQL assignments:

```
SET X = ( SELECT COUNT ( * ) FROM S ) ;

SET Y = ( SELECT COUNT ( DISTINCT STATUS ) FROM S ) ;
```

But assignment as such is far from being the only context in which such coercions occur (see Chapters 2 and 12).

> **ASIDE**
> **Actually there's another oddity arising in connection with SQL-style aggregation (I include this observation here because this is where it logically belongs, but it does rely on an understanding of SQL-style summarization, and you can skip it if you like):**
>
> - **In general, an expression of the form SELECT - FROM *T* - WHERE - GROUP BY - HAVING delivers a result containing exactly one row for each "group" in *G*, where *G* is the "grouped table" resulting from applying**

the WHERE, GROUP BY, and HAVING clauses to table *T*.

- Omitting the WHERE and HAVING clauses, as in a typical SQL-style aggregation, is equivalent to specifying WHERE TRUE and HAVING TRUE, respectively. For present purposes, therefore, we need consider the effect of the GROUP BY clause, only, in determining the grouped table *G*.
- Suppose table *T* has *nT* rows. Then arranging those rows into groups can produce at most *nT* groups; in other words, the grouped table *G* has *nG* groups for some *nG* (*nG* ≤ *nT*), and the overall result, obtained by applying the SELECT clause to *G*, thus has *nG* rows.
- Now suppose *nT* is zero (i.e., table *T* is empty); then *nG* must clearly be zero as well (i.e., table *G*, and hence the result of the SELECT expression overall, must both be empty as well).
- In particular, therefore, the expression

  ```
  SELECT COUNT ( * ) AS X
  FROM   S
  GROUP  BY ( )
  ```

  —which is, recall, the expanded form of SELECT COUNT(*) AS X FROM S—ought logically to produce the result shown on the left, not the one shown on the right, if table S is empty:

X

X
0

  In fact, however, it produces the result on the right. How? *Answer:* By special casing. Here's a direct quote from the standard: "If there are no grouping columns, then the result of the <group by clause> is the grouped table consisting of *T* as its only group." In other words, while grouping an empty table in SQL does indeed (as argued above) produce an empty set of groups in general, *the case where the set of grouping columns is empty is special*; in that case, it produces a set containing exactly one group, that group being identical to the empty table *T*. In the example, therefore, the COUNT operator is applied to an empty group, and thus "correctly" returns the value zero.

Now, you might be thinking the discrepancy here is hardly earth shattering; you might even be thinking the result on the right above is somehow "better" than the one on the left. But (to state the obvious) there's a logical difference between the two, and—to quote Wittgenstein again—*all logical differences are big differences*. Logical mistakes like the one under discussion are simply unacceptable in a system that's meant to be solidly based on logic, as relational systems are.

Image Relations bis

In this section, I just want to present a series of examples that show the usefulness of aggregate operators as discussed in the previous section in connection with image relations. *Note:* The examples in question all involve some summarization again, at least implicitly, but I still don't want to discuss summarization as such (I'll do that in the next two sections).

Example 1: Get suppliers for whom the total shipment quantity, taken over all shipments for the supplier in question, is less than 1000.

```
S WHERE SUM ( !! SP , QTY ) < 1000
```

For any given supplier, the expression SUM(!!SP,QTY) denotes, precisely, the total shipment quantity for the supplier in question. An equivalent formulation without the image relation is:

```
S WHERE SUM ( SP MATCHING RELATION { TUPLE { SNO SNO } } , QTY ) < 1000
```

Here for interest is an SQL "analog"—"analog" in quotes because actually there's a trap in this example; the SQL expression shown is not quite equivalent to the **Tutorial D** expressions shown previously (why not?):

```
SELECT S.*
FROM   S , SP
WHERE  S.SNO = SP.SNO
GROUP  BY S.SNO , S.SNAME , S.STATUS , S.CITY
HAVING SUM ( SP.QTY ) < 1000
```

NOTE

I can't resist pointing out in passing that (as this example suggests) SQL lets us say "S.*" in the SELECT clause but not in the GROUP BY clause, where it would make just as much sense.

Example 2: Get suppliers with fewer than three shipments.

```
S WHERE COUNT ( !! SP ) < 3
```

Example 3: Get suppliers for whom the maximum shipment quantity is less than twice the minimum shipment quantity (taken over all shipments for the supplier in question in both cases).

```
S WHERE MAX ( !! SP , QTY ) < 2 * MIN ( !! SP , QTY )
```

Example 4: Get shipments such that at least two other shipments involve the same quantity.

```
SP WHERE COUNT ( !! (SP RENAME ( SNO AS SN , PNO AS PN )) ) > 2
```

This example is admittedly very contrived, but the point is to illustrate the occasional need for some attribute renaming in connection with image relation references. In the example, the renaming is needed in order to ensure that the image relation we want, in connection with a given shipment tuple, is defined in terms of attribute QTY only. The introduced names SN and PN are arbitrary.

I remark in passing that this example also illustrates the "multiple" form of RENAME:

```
SP RENAME ( SNO AS SN , PNO AS PN )
```

This expression is shorthand for the following:

```
( SP RENAME ( SNO AS SN ) ) RENAME ( PNO AS PN )
```

Similar shorthands are defined for various other operators, too, including EXTEND in particular (I'll give an example later).

Example 5: Update suppliers for whom the total shipment quantity, taken over all shipments for the supplier in question, is less than 1000, reducing their status to half its previous value.

```
UPDATE S WHERE SUM ( !!SP , QTY ) < 1000 :
                 { STATUS := 0.5 * STATUS } ;
```

Summarization

> **Definition:** Let relations *r1* and *r2* be such that *r2* has the same heading as some projection of *r1*, and let the attributes of *r2* be *A*, *B*, ..., *C*. Then the *summarization* SUMMARIZE *r1* PER (*r2*) ADD (*summary* AS *X*) is a relation with (a) heading the heading of *r2* extended with attribute *X*, and (b) body the set of all tuples *t* such that *t* is a tuple of *r2* extended with a value *x* for attribute *X*. That value *x* is computed by evaluating *summary* over all tuples of *r1* that have the same value for attributes *A*, *B*, ..., *C* as tuple *t* does. Relation *r2* must not have an attribute called *X*, and *summary* must not refer to *X*. Observe that the result has cardinality equal to that of *r2* and degree equal to that of *r2* plus one. The type of *X* in that result is the type of *exp*.

Here's an example (which I'll label SX1—"SUMMARIZE Example 1"—for purposes of subsequent reference):

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

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

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

In other words, the result contains one tuple for each tuple in the PER relation—i.e., one tuple for each of the five supplier numbers, in the example—extended with the corresponding count.

ASIDE

Note carefully that the syntactic construct COUNT(PNO)—I deliberately don't call it an expression, because it isn't one (at least, not in the Tutorial D sense)—in the foregoing SUMMARIZE is *not* an invocation of the aggregate operator called COUNT. That aggregate operator takes a relation as its argument. By contrast, the argument to the COUNT in the foregoing SUMMARIZE is an attribute: an attribute of some relation, of course, but just which relation is specified only very indirectly. In fact, the syntactic construct COUNT(PNO) is really very special—it has no meaning outside the context of an appropriate SUMMARIZE, and it can't be used outside that context. All of which begins to make it look as if SUMMARIZE might be not quite respectable, in a way, and it might be nice if we could replace it by something better ... See the section "Summarization bis," later.

As a shorthand, if relation *r2* doesn't merely have the same heading as some projection of relation *r1* but actually is such a projection, the PER specification can be replaced by a BY specification, as in this example ("Example SX2"):

```
SUMMARIZE SP BY { SNO } ADD ( COUNT ( PNO ) AS PCT )
```

Here's the result:

SNO	PCT
S1	6
S2	2
S3	1
S4	3

As you can see, this result differs from the previous one in that it contains no tuple for supplier S5. That's because BY {SNO} in the example is defined to be shorthand for PER (SP{SNO})—SP, because SP is what we want to summarize—and the projection SP{SNO} doesn't contain a tuple for supplier S5.

Now, Example SX2 can be expressed in SQL as follows:

```
SELECT SNO , COUNT ( ALL PNO ) AS PCT
FROM   SP
GROUP  BY SNO
```

As this example suggests, summarizations—as opposed to "aggregations"—are typically formulated in SQL by means of a SELECT expression with an explicit GROUP BY clause (but see later!). Points arising:

- You can think of such expressions as being evaluated as follows. First, the table specified by the FROM clause is divided up into a set of disjoint "groups"—actually tables—as specified by the grouping column(s) in the GROUP BY clause; result rows are then obtained, one for each group, by computing the specified summary (or summaries, plural) for that group and appending other items as specified by the SELECT item commalist. *Note:* The SQL analog of the term *summary* is "set function"; the term is doubly inappropriate, however, because (a) the argument to such a function isn't a set but a bag, in general, and (b) the result isn't a set either.
- It's safe to specify just SELECT, not SELECT DISTINCT, in the example because (a) the result table is guaranteed to contain just one row for each group, by definition, and (b) each group contains just one value for the grouping column(s), again by definition.
- The ALL specification could be omitted from the COUNT invocation in this example, because for set functions ALL is the default. (In the example, in fact, it makes no difference whether ALL or DISTINCT is specified, because the combination of supplier number and part number is a key for table SP.)
- The set function COUNT(*) is a special case—it applies, not to values in some column (as, e.g., SUM does), but to rows in some table. (In the example, the specification COUNT(PNO) could be replaced by COUNT(*) without changing the result.)

Now let's get back to Example SX1. That example can be formulated in SQL as follows:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

The important point here is that the result does now contain a row for supplier S5, because by definition that result contains one row for each supplier number in table S. And as you can see, this formulation differs from the one given for Example SX2—the one that missed supplier S5—in that it doesn't include a GROUP BY clause, and it doesn't do any grouping (at least, not overtly).

ASIDE

By the way, there's a trap for the unwary in SQL here. As you can see, the second item in the SELECT item commalist in the foregoing SQL expression—i.e., the subexpression (SELECT ... S.SNO) AS PCT—is of the form *subquery* AS *name* (and the subquery in question is in fact a scalar one). Now, if that very same text were to appear in a FROM clause, the specification AS *name* would be understood as defining a range variable (see Chapter 10). In the SELECT clause, however, that same specification is understood as defining a name for the pertinent column of the overall result. It follows that the following formulation of the example is *not* logically equivalent to the one shown above:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO ) AS PCT
                 FROM   SP
                 WHERE  SP.SNO = S.SNO )
FROM   S
```

With this formulation, the table *t* that's returned by evaluation of the subquery has a column called PCT. That table *t* is then doubly coerced to the sole scalar value it contains, producing a column value in the overall result—but (believe it or not) that column in the overall result is *not* called PCT; instead, it has no name.

To revert to the main thread of the discussion: As a matter of fact, Example SX2 could also be expressed without using GROUP BY, as follows:

```
SELECT DISTINCT SPX.SNO , ( SELECT COUNT ( ALL SPY.PNO )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS PCT
FROM   SP AS SPX
```

As these example suggest, SQL's GROUP BY clause is logically redundant—any relational expression that can be represented with it can also be represented without it. Be that as it may, there's another point that needs to be made here. Suppose Example SX1 had requested, not the count of part numbers, but the sum of quantities, for each supplier:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ )
```

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

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

By contrast, this SQL expression—

```
SELECT S.SNO , ( SELECT SUM ( ALL QTY )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

—gives a result in which the TOTQ value for supplier S5 is shown as null, not zero. That's because, in SQL, if any set function other than COUNT(*) or COUNT is invoked on an empty argument, the result is (incorrectly) defined to be null. To get the correct result, therefore, we need to use COALESCE, as follows:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

Suppose now that Example SX1 had asked for the sum of quantities for each supplier, but only where that sum is greater than 250:

```
( SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ ) )
                                                  WHERE TOTQ > 250
```

Result:

SNO	TOTQ
S1	1300
S2	700
S4	900

The "natural" SQL formulation of this query would be:

```
SELECT SNO , SUM ( ALL QTY ) AS TOTQ
FROM   SP
GROUP  BY SNO
HAVING SUM ( ALL QTY ) > 250  /* not TOTQ > 250 !!! */
```

But it could also be formulated like this:

```
SELECT DISTINCT SPX.SNO , ( SELECT SUM ( ALL SPY.QTY )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS TOTQ
FROM   SP AS SPX
WHERE  ( SELECT SUM ( ALL SPY.QTY )
         FROM   SP AS SPY
         WHERE  SPY.SNO = SPX.SNO ) > 250
```

As this example suggests, HAVING, like GROUP BY, is also logically redundant—any relational expression that can be represented with it can also be represented without it. It's true that the GROUP BY and HAVING versions are often more succinct; on the other hand, it's also true that they sometimes deliver the wrong answer (consider, for example, what would happen if the foregoing example had specified that the sum should be less than, instead of greater than, 250).

Recommendations: If you use GROUP BY or HAVING, make sure the table you're summarizing is the one you really want to summarize (typically suppliers rather than shipments, in terms of the examples in this section). Also, be on the lookout for the possibility that some summarization is being done on an empty set, and use COALESCE wherever necessary.

There's one more thing I need to say about GROUP BY and HAVING. Consider the following SQL expression:

```
SELECT SNO , CITY , SUM ( ALL QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO
```

Observe that CITY appears in the SELECT item commalist here but isn't one of the grouping columns. That appearance is legitimate, however, because table S satisfies a certain *functional dependence*—see Chapter 8 or Appendix B—according to which each SNO value in that table has just one corresponding CITY value (again, in that table); what's more, the SQL standard includes rules according to which the system will in fact be aware of that functional dependence. As a consequence, even though it isn't a grouping column, CITY is still known to be single valued per group, and it can therefore indeed appear in the SELECT clause as shown (also in the HAVING clause, if there is one).

Of course, it's not logically wrong—though there might be negative performance implications—to specify the column as a grouping column anyway, as here:

```
SELECT SNO , CITY , SUM ( QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO , CITY
```

Summarization bis

The SUMMARIZE operator has been part of **Tutorial D** since its inception. With the introduction of image relations, that operator became logically redundant, however—and while there might be reasons (perhaps pedagogic ones) to retain it, the fact is that most summarizations can be more succinctly expressed by means of EXTEND.* Recall Example SX1 from the previous section ("For each supplier, get the supplier number and a count of the number of parts supplied"). The SUMMARIZE formulation looked like this:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

Here by contrast is an equivalent EXTEND formulation:

```
EXTEND S { SNO } ADD ( COUNT ( !! SP ) AS PCT )
```

(Since the combination {SNO,PNO} is a key for relvar SP, there's no need to project the image relation on {PNO} before computing the count.) As the example suggests, EXTEND

* Not to mention the fact that as pointed out in the earlier section on summarization, SUMMARIZE has to make use of a syntactic construct that unfortunately looks a bit like an aggregate operator invocation but isn't one. As noted earlier, therefore, it might be a good idea to dispense with SUMMARIZE altogether.

is certainly another context in which image relations make sense; in fact, they're arguably even more useful in this context than they are in WHERE clauses.

The rest of this section consists of more examples. I've continued the numbering from the examples in the section "Image Relations bis."

Example 6: For each supplier, get supplier details and total shipment quantity, taken over all shipments for the supplier in question.

```
EXTEND S ADD ( SUM ( !! SP , QTY ) AS TOTQ )
```

Example 7: For each supplier, get supplier details and total, maximum, and minimum shipment quantity, taken over all shipments for the supplier in question.

```
EXTEND S ADD ( SUM ( !! SP , QTY ) AS TOTQ ,
               MAX ( !! SP , QTY ) AS MAXQ ,
               MIN ( !! SP , QTY ) AS MINQ )
```

Note the use of the multiple form of EXTEND in this example.

Example 8: For each supplier, get supplier details, total shipment quantity taken over all shipments for the supplier in question, and total shipment quantity taken over all shipments for all suppliers.

```
EXTEND S ADD ( SUM ( !! SP , QTY ) AS  TOTQ ,
               SUM (    SP , QTY ) AS GTOTQ )
```

Result:

SNO	TOTQ	GTOTQ
S1	1300	3100
S2	700	3100
S3	200	3100
S4	900	3100
S5	0	3100

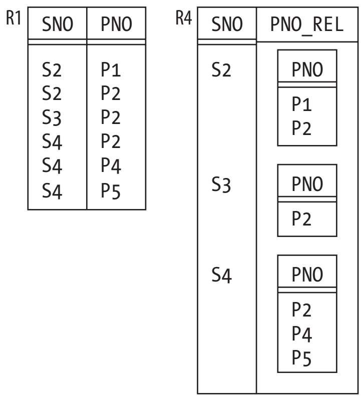
Example 9: For each city *c*, get *c* and the total and average shipment quantities for all shipments for which the supplier city and part city are both *c*.

```
WITH ( S JOIN SP JOIN P ) AS TEMP :
EXTEND TEMP { CITY } ADD ( SUM ( !! TEMP , QTY ) AS TOTQ ,
                           AVG ( !! TEMP , QTY ) AS AVGQ )
```

The point of this rather contrived example is to illustrate the usefulness of WITH, in connection with summarizations in particular, in avoiding the need to write out some possibly lengthy subexpression several times.

Group and Ungroup

Recall from Chapter 2 that relations with relation valued attributes (RVAs for short) are legal. Figure 7-1 opposite 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 7-1. Relations R1 and R4 from Figures 2-1 and 2-2 in Chapter 2

Obviously 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 the relations shown in Figure 7-1, 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.

Incidentally, it's worth noting that the following expression—

```
EXTEND R1 { SNO } ADD ( !! R1 AS PNO_REL )
```

—will produce exactly the same result as the GROUP example shown above. In other words, GROUP can be defined in terms of EXTEND and image relations. Now, I'm not suggesting that we get rid of our useful GROUP operator; quite apart from anything else, a language that had an explicit UNGROUP operator (as **Tutorial D** does) but no explicit GROUP operator could certainly be criticized on ergonomic grounds, if nothing else. But it's at least interesting, and perhaps pedagogically helpful, to note that the semantics of GROUP can so easily be explained in terms of EXTEND and image relations.

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* (see Appendix D) or the book *Databases, Types, and the Relational Model: The Third Manifesto* (again, see Appendix D), 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 this chapter and its predecessor. See Exercises 7-11–7-13 at the end of the chapter.

"What If" Queries

"What if" queries are a frequent requirement; they're used to explore the effect of making certain changes without actually having to make (and possibly unmake) the changes in question. Here's an example ("What if parts in Paris were in Nice and had double the weight?"):

```
UPDATE P WHERE CITY = 'Paris' : | WITH T1 AS
     { CITY := 'Nice' ,         |  ( SELECT P.*
       WEIGHT := 2 * WEIGHT }   |    FROM   P
                                |    WHERE  CITY = 'Paris' ) ,
                                |      T2 AS
                                |  ( SELECT P.* , 'Nice' AS NC ,
                                |           2 * WEIGHT AS NW
                                |    FROM   T1 )
                                |    SELECT PNO , PNAME , COLOR ,
                                |           NW AS WEIGHT ,
                                |           NC AS CITY
                                |    FROM   T2
```

Observe that, even though it uses the keyword UPDATE, the **Tutorial D** expression on the left here is indeed an expression and not a statement (it has no terminating semicolon); in particular, it has no effect on relvar P. (The keyword UPDATE here thus denotes a read-only operator! Lewis Carroll, where are you?) What the expression does do is evaluate to a relation containing exactly one tuple *t2* for each tuple *t1* in the current value of relvar P for which the city is Paris—except that, in that tuple *t2*, the status is double that in tuple *t1* and the city is Nice, not Paris. In other words, the expression overall is shorthand for the following (and this expansion should help you understand the SQL version of the query):

```
WITH ( P WHERE CITY = 'Paris' ) AS R1 ,
     ( EXTEND R1 ADD ( 'Nice' AS NC , 2 * WEIGHT AS NW ) ) AS R2 ,
     R2 { ALL BUT CITY , WEIGHT } AS R3 :
     R3 RENAME ( NC AS CITY , NW AS WEIGHT )
```

And now I can take care of some unfinished business from Chapter 5. In that chapter, I said that UPDATE—by which I meant the UPDATE *statement*, not the read-only operator discussed above—corresponded to a certain relational assignment, but the details were a little more complicated than they were for INSERT and DELETE. Now I can explain those details. By way of example, consider the following UPDATE statement:

```
UPDATE P WHERE CITY = 'Paris' :
            { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } ;
```

This statement is logically equivalent to the following relational assignment (note the appeal to the read-only UPDATE operator as discussed above):

```
P := ( P WHERE CITY ≠ 'Paris' )
       UNION
     ( UPDATE P WHERE CITY = 'Paris' :
                { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } ) ;
```

What About ORDER BY?

The last topic I want to address in this chapter is ORDER BY. Now, despite the title of this chapter, ORDER BY isn't actually 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). Please don't misunderstand me here; I'm not saying ORDER BY isn't useful; however, I *am* saying it can't sensibly appear in a relational expression (unless it's treated simply as a "no op," I suppose).* By definition, therefore, the following expressions, though legal, aren't relational expressions as such:

```
S MATCHING SP                |    SELECT DISTINCT S.*
          ORDER ( ASC SNO )  |    FROM   S , SP
                             |    WHERE  S.SNO = SP.SNO
                             |    ORDER  BY 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, as we know from 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

Incidentally, it would be remiss of me not to mention in passing that most of the relational operators also have counterparts in SQL that aren't functions. This state of affairs is due to the fact that, as explained in Chapter 2, SQL sometimes defines the result of the comparison *v1* = *v2* to be TRUE even when *v1* and *v2* are distinct. For example, consider the character strings 'Paris' and 'Paris ', respectively (note the trailing space in the latter); these

* In particular, therefore, it can't appear in a view definition—despite the fact that at least one well known product allows it to! *Note:* It's sometimes suggested that ORDER BY is needed in connection with what are called "quota queries," but this is a popular misconception (see Exercise 7-14).

values are clearly distinct, and yet SQL sometimes regards them as equal. As explained in Chapter 2, therefore, certain SQL expressions are "possibly nondeterministic." Here's a simple example:

```
SELECT DISTINCT CITY FROM S
```

If one supplier has CITY 'Paris' and another 'Paris ', then the result will include either 'Paris' or 'Paris ' (or possibly both), but which result we get might not be defined. 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.

Exercises

Exercise 7-1. Given our usual sample values for the suppliers-and-parts database, what relations do the following **Tutorial D** expressions denote? In each case, give both (a) an SQL analog and (b) an informal interpretation of the expression (i.e., the corresponding predicate) in natural language.

a. `S MATCHING ( SP WHERE PNO = 'P2' )`

b. `P NOT MATCHING ( SP WHERE SNO = 'S2' )`

c. `P WHERE ( !! SP ) { SNO } = S { SNO }`

d. `P WHERE SUM ( !! SP , QTY ) < 500`

e. `P WHERE TUPLE { CITY CITY } ∈ S { CITY }`

f. `EXTEND S ADD ( 'Supplier' AS TAG )`

g. `EXTEND S { SNO } ADD ( 3 * STATUS AS TRIPLE_STATUS )`

h. `EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY AS SHIPWT )`

i. `EXTEND P ADD ( WEIGHT * 454 AS GMWT , WEIGHT * 16 AS OZWT )`

j. `EXTEND P ADD ( COUNT ( !! SP ) AS SCT )`

k.
```
EXTEND S
   ADD ( COUNT ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO )
       AS NP )
```

l. `SUMMARIZE S BY { CITY } ADD ( AVG ( STATUS ) AS AVG_STATUS )`

m.
```
SUMMARIZE ( S WHERE CITY = 'London' )
             PER ( TABLE_DEE ) ADD ( COUNT ( SNO ) AS N )
```

n. `UPDATE SP WHERE SNO = 'S1' : { SNO := 'S7' , QTY = 0.5 * QTY }`

Exercise 7-2. In what circumstances (if any) are *r1* MATCHING *r2* and *r2* MATCHING *r1* equivalent?

Exercise 7-3. Show that rename isn't primitive.

Exercise 7-4. Give an expression involving EXTEND instead of SUMMARIZE that's logically equivalent to the following:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS NP )
```

Exercise 7-5. Which if any of the following **Tutorial D** expressions are equivalent? Show an SQL analog in each case.

a. `SUMMARIZE r PER ( r { } ) ADD ( SUM ( 1 ) AS CT )`

b. `SUMMARIZE r PER ( TABLE_DEE ) ADD ( SUM ( 1 ) AS CT )`

c. `SUMMARIZE r BY { } ADD ( SUM ( 1 ) AS CT )`

d. `EXTEND TABLE_DEE ADD ( COUNT ( r ) AS CT )`

Exercise 7-6. 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 XOR aggregate operator here deliberately). What does SQL do in these circumstances? And why?

Exercise 7-7. Let relation R4 in Figure 7-1 denote the current value of some relvar. If R4 is as described in Chapter 2, what's the predicate for that relvar?

Exercise 7-8. Let *r* be the relation denoted by the following **Tutorial D** 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 7-9. Write **Tutorial D** and/or SQL expressions for the following queries on the suppliers-and-parts database:

a. Get the total number of parts supplied by supplier S1.

b. Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.

c. Get city names for cities in which at least two suppliers are located.

d. Since SNAME and CITY are both of type CHAR, it makes sense (sort of) to compare a supplier name and a supplier city ... If every supplier has a name alphabetically preceding that supplier's city, get the result 'Y'; otherwise get the result 'N'.

Exercise 7-10. Here are two **Tutorial D** expressions involving relation R4 from Figure 7-1. What queries do they represent?

```
( R4 WHERE TUPLE { PNO 'P2' } ∈ PNO_REL ) { SNO }

( ( R4 WHERE SNO = 'S2' ) UNGROUP ( PNO_REL ) ) { PNO }
```

Exercise 7-11. Given our usual sample values for the suppliers-and-parts database, what does the following **Tutorial D** expression denote?

```
EXTEND S
   ADD  ( ( ( SP RENAME ( SNO AS X ) ) WHERE X = SNO ) { PNO }
        AS PNO_REL )
```

Exercise 7-12. 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 'S6' , SNAME 'Lopez' , STATUS 30 , CITY 'Madrid' ,
           PNO_REL RELATION { TUPLE { PNO 'P5' } } } } ;

UPDATE SSP WHERE SNO = 'S2' :
 { INSERT PNO_REL RELATION { TUPLE { PNO 'P5' } } } ;
```

Exercise 7-13. Using relvar SSP from the previous exercise, write expressions for the following queries:

a. Get pairs of supplier numbers for suppliers who supply exactly the same set of parts.

b. Get pairs of part numbers for parts supplied by exactly the same set of suppliers.

Exercise 7-14. 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 two heaviest parts," for which the specified quota is two. Give **Tutorial D** and SQL formulations of this query. Given our usual data values, what exactly do these formulations return?

Exercise 7-15. Using the explicit SUMMARIZE operator, how would you deal with the query "For each supplier, get the supplier number and the average of *distinct* shipment quantities for shipments by that supplier"?

Exercise 7-16. Given a revised version of the suppliers-and-parts database that looks like this—

```
S   { SNO }      /* suppliers                */
SP  { SNO, PNO } /* supplier supplies part    */
SJ  { SNO, JNO } /* supplier supplies project */
J   { JNO }      /* projects                 */
```

—give **Tutorial D** and SQL formulations of the query "For each supplier, get supplier details, the number of parts supplied by that supplier, and the number of projects supplied by that supplier." For **Tutorial D**, give both EXTEND and SUMMARIZE formulations.

Exercise 7-17. What does the following **Tutorial D** expression mean?

```
S WHERE ( !! (!! SP) ) { PNO } = P { PNO }
```

Exercise 7-18. Is there a logical difference between the following two **Tutorial D** expressions? If so, what is it?

```
EXTEND TABLE_DEE ADD ( COUNT (   SP ) AS NSP )

EXTEND TABLE_DEE ADD ( COUNT ( !! SP ) AS NSP )
```

Exercise 7-19. Give an example of a join that's not a semijoin and a semijoin that's not a join. When exactly are the expressions *r1* JOIN *r2* and *r1* MATCHING *r2* equivalent?

Exercise 7-20. Let relations *r1* and *r2* be of the same type, and let *t1* be a tuple in *r1*. For that tuple *t1*, then, what exactly does the expression !!*r2* denote? And what happens if *r1* and *r2* aren't just of the same type but are in fact the very same relation?

Exercise 7-21. What's the logical difference, if any, between the following SQL expressions?

```
SELECT COUNT ( * ) FROM S

SELECT SUM ( 1 ) FROM S
```

CHAPTER EIGHT

SQL and Constraints

I'VE TOUCHED ON THE TOPIC OF INTEGRITY CONSTRAINTS OCCASIONALLY IN PREVIOUS CHAPTERS, BUT IT'S TIME TO GET MORE SPECIFIC. Here's 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 in general are so called because they constrain the values that can legally appear as values of some variable; but the ones we're interested in here are the ones that apply to database variables specifically. Such constraints 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 further defines the values that can appear in a given database (where "further" means over and above the constraints imposed by the pertinent type constraints). As usual, in what follows I'll discuss these concepts in both relational and SQL terms.

By the way, it's worth noting that constraints in general can be regarded as a formal version of what some people call *business rules*. Now, this latter term doesn't really have a precise definition (at least, not one that's universally accepted); in general, however, a business rule is a declarative statement—emphasis on *declarative*—of some aspect of the enterprise the database is meant to serve, and statements that constrain the values of database variables certainly fit that loose definition. In fact, I'll go further. In my opinion, constraints are really what database management is all about. The database is supposed to represent some aspect of the enterprise in question; that representation is supposed to be as faithful as possible, in order to guarantee that decisions made on the basis of what the database says are right ones; and constraints are the best mechanism we have for ensuring that the representation is indeed as faithful as possible. Constraints are crucial, and proper DBMS support for them is crucial as well.

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—and that's effectively what a type constraint does. In the case of system defined types, it's the system that carries out this task, and there's not much more to be said. In the case of user defined types, by contrast, there certainly is more to say, much more. So let's suppose for the sake of the example that shipment quantities, instead of being of the system defined type INTEGER, are of some user defined type—QTY, say. Here then is a possible **Tutorial D** definition for that type:

```
1  TYPE QTY
2       POSSREP QPR
3             { Q INTEGER
4                 CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

Explanation:

- Line 1 just says we're defining a type called QTY.
- 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*—definitely are visible to the user (in the example, users do definitely know that quantities have a possrep called QPR). Note very carefully, however, that there's 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 says those integers must lie in the range 0 to 5000 inclusive. Thus, lines 2–4 together define valid quantities to be, precisely, values that can possibly be represented by integers in the specified range, and it's that definition that constitutes the *type constraint* for type QTY. (Observe, therefore, that such constraints are specified not in terms of the type as such but, rather, in terms of a possrep for the type. Indeed,

* There are some minor exceptions to this rule that need not concern us here.

one of the reasons we require the possrep concept in the first place is precisely to serve as a vehicle for formulating type constraints, as I hope the example suggests.)

Here's a slightly more complicated example:

```
TYPE POINT
     POSSREP CARTESIAN { X FIXED , Y FIXED
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 } ;
```

Type POINT denotes points in two-dimensional space; it has a possrep called CARTESIAN with two components called X and Y (corresponding, presumably, to cartesian coordinates); those components are of type FIXED; and there's a CONSTRAINT specification that (in effect) says the only points we're interested in are those that lie on or inside a circle with center the origin and radius 100 (SQRT = nonnegative square root). *Note:* I used a type called POINT in an example in Chapter 2, as you might recall, but I deliberately didn't show the POSSREP and CONSTRAINT specifications for that type in that example; tacitly, however, I was assuming that type had a possrep called POINT, not CARTESIAN. See the subsection immediately following.

Selectors and THE_ Operators

Before I continue with my discussion of type constraints as such, I need to digress for a few moments in order to clarify a few points raised by the QTY and POINT examples.

Recall from Chapter 2 that types in general, including user defined types like QTY and POINT in particular, 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:

1. `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, **Tutorial D** has 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 above becomes just QTY(250)—which is the style I used for selectors in Chapter 2, if you care to go back and look. I'll assume this revised definition for type QTY for the rest of this chapter, barring explicit statements to the contrary.

2. `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). If that expression is a literal, as it was in the previous example, then the selector invocation is a literal in turn; thus, a literal is a special case of a selector invocation, as we already know from Chapter 2.

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 specific argument.

As for type POINT, let's first redefine that type so that the possrep has the same name as the type, as in the QTY example above:

```
TYPE POINT POSSREP { X FIXED , Y FIXED CONSTRAINT ... } ;
```

Now continuing with the examples:

1. `POINT ( 5.7 , -3.9 )`

   This expression is a POINT selector invocation (actually a POINT literal).

2. `THE_X ( P )`

   This expression returns the FIXED value that's the X coordinate of the cartesian possible representation of the point that's the current value of the variable P (which must be of type POINT).

Just as an aside, let me draw your attention to the fact that (as I said earlier) **Tutorial D** requires a TYPE statement to include *at least* one POSSREP specification. The fact is, **Tutorial D** does allow a type to have several distinct possreps. POINT is a good example; we might well want to define two distinct possreps for points, to reflect the fact that points in two-dimensional space can possibly be represented by either cartesian or polar coordinates. Further details don't belong in a book of this nature; I'll just note for the record that SQL has nothing analogous.

More on Type Constraints

Now let's get back to type constraints as such. Suppose I had defined type QTY as follows, with no explicit CONSTRAINT specification:

```
TYPE QTY POSSREP { Q INTEGER } ;
```

This definition is defined to be shorthand for the following:

```
TYPE QTY POSSREP { Q INTEGER CONSTRAINT TRUE } ;
```

Given this definition, anything that could possibly be represented by an integer would be a legitimate QTY value, and so type QTY would necessarily still have an associated type constraint, albeit a fairly weak one. In other words, the specified possrep defines an a priori constraint for the type, and the CONSTRAINT specification effectively imposes an additional constraint, over and above that a priori one. (Informally, however, we often take the term "type constraint" to refer to what's stated in the CONSTRAINT specification as such.)

Now, 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 such that they must be possibly representable as integers in the range 0 to 5000 inclusive. 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 fact, it should be obvious that we can never tolerate an expression that's supposed to denote a value of some type *T* but in fact doesn't; after all, "a value of type *T* that's not a value of type *T*" is a contradiction in terms. It follows that no variable—in particular, no relvar—can ever be assigned a value that's not of the right type.

One last point to close this section: Declaring anything to be of some particular type imposes a constraint on that thing, by definition.* In particular, declaring 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. *Note:* This constraint on attribute QTY is an example of what's sometimes called an *attribute constraint*.

Type Constraints in SQL

As I'm sure you noticed, I didn't give SQL versions of the examples in the previous section. That's because, believe it or not, SQL doesn't support type constraints at all!—apart from the rather trivial a priori ones, of course. For example, although SQL would certainly let you create a user defined type called QTY and specify that quantities must be representable as integers, it wouldn't let you say those integers must lie in a certain range. In other words, an SQL definition for that type might look like this:

```
CREATE TYPE QTY AS INTEGER FINAL ;
```

(The keyword FINAL here just means QTY doesn't have any proper subtypes. Subtypes in general are beyond the scope of this book.)

With the foregoing SQL definition, all available integers will be regarded as denoting valid quantities. If you want to constrain quantities to some particular range, therefore, you'll have to specify an appropriate *database* constraint—in practice, probably a base table constraint (see the section "Database Constraints in SQL")—on each and every use of the type. For example, you might need to extend the definition of base table SP as follows

* 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.

(note the revised data type specification for column QTY and the CONSTRAINT specification at the end):

```
CREATE TABLE SP
  ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    VARCHAR(6)   NOT NULL ,
    QTY    QTY          NOT NULL ,
    UNIQUE ( SNO , PNO ) ,
    FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ,
    FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ,
    CONSTRAINT SPQC CHECK ( QTY >= QTY(0) AND
                            QTY <= QTY(5000) ) ) ;
```

The expressions QTY(0) and QTY(5000) in the CONSTRAINT specification here can be regarded as QTY selector invocations. I remind you, however, that *selector* isn't an SQL term (and nor is *THE_ operator*); as I hinted in Chapter 2, in fact, the situation regarding selectors and THE_ operators in SQL is too complicated to describe in detail in this book. Suffice it to say that analogs of those operators are usually available, though they aren't always provided "automatically" as they are in **Tutorial D**.

For interest, I also show an SQL definition for type POINT (and here I've specified NOT FINAL instead of FINAL, just to illustrate that possibility):

```
CREATE TYPE POINT AS
     ( X NUMERIC(5,1) , Y NUMERIC(5,1) ) NOT FINAL ;
```

To say it again, then, SQL doesn't really support type constraints. The reasons for the omission are complex—they have to do with type inheritance and are therefore beyond the scope of this book—but the implications are serious.* **Recommendation:** Wherever possible, use database constraints to make up for the omission, as in the QTY example above. Of course, this recommendation might lead to a lot of duplicated effort, but such duplication is better than the alternative: namely, bad data in the database. See Exercise 8-8 at the end of the chapter.

Database Constraints

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 some logically equivalent shorthand); in SQL, they're specified by means of a CREATE ASSERTION statement (or, again, some equivalent shorthand). I don't want to get into details of those shorthands—at least, not yet—because they're essentially just a matter of syntax; for now, let me stay with the "longhand" forms. Here are some examples (**Tutorial D** on the left and SQL on the right as usual):

* Although I've said type inheritance in general is beyond the scope of this book, I can't resist pointing out one implication of SQL's lack of support for type constraints in particular: namely, that SQL has to permit such absurdities as nonsquare squares (by which I mean, more precisely, values of type SQUARE whose sides are of different lengths, and are thus merely rectangles).

Example 1:

```
CONSTRAINT CX1 IS_EMPTY        |    CREATE ASSERTION CX1 CHECK
  ( S WHERE STATUS < 1         |     ( NOT EXISTS
      OR    STATUS > 100 ) ;   |     ( SELECT * FROM S
                               |       WHERE  STATUS < 1
                               |       OR     STATUS > 100 ) ) ;
```

Constraint CX1 says: Status values must be in the range 1 to 100 inclusive. This constraint involves just a single attribute of a single relvar. Note in particular that 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 that reason, such constraints are sometimes referred to, informally, as tuple constraints, or row constraints in SQL—though this latter term is also used in SQL to mean, more specifically, a row constraint that can't be formulated as a column constraint (see the section "Database Constraints in SQL"). All such usages are deprecated, however, because constraints constrain updates and (as we saw in Chapter 5) there's no such thing as a tuple or row level update in the relational world.

Example 2:

```
CONSTRAINT CX2 IS_EMPTY        |   CREATE ASSERTION CX2 CHECK
  ( S WHERE CITY = 'London'    |    ( NOT EXISTS
     AND   STATUS ≠ 20 ) ;     |    ( SELECT * FROM S
                               |      WHERE  CITY = 'London'
                               |      AND    STATUS <> 20 ) ) ;
```

Constraint CX2 says: Suppliers in London must have status 20. This constraint involves two distinct attributes, but because they're attributes of the same relvar it's still the case, as it was with constraint CX1, that the constraint can be checked for a given supplier tuple by examining just that tuple in isolation (hence it too is a "tuple constraint" or "row constraint").

Example 3:

```
CONSTRAINT CX3                 |    CREATE ASSERTION CX3 CHECK
   COUNT ( S ) =               |     ( UNIQUE ( SELECT ALL SNO
   COUNT ( S { SNO } ) ;       |                FROM   S ) ) ;
```

Constraint CX3 says: No two tuples in relvar S have the same supplier number; in other words, {SNO} is a superkey—in fact, a key—for that relvar (recall from Chapter 5 that a superkey is a superset of a key, loosely speaking). Like constraints CX1 and CX2, this constraint still involves just one relvar; however, it can't be checked for a given supplier tuple by examining just that tuple in isolation (and so it isn't a "tuple" or "row" constraint). In practice, of course, it's very unlikely that constraint CX3 would be specified in longhand as shown—some kind of explicit KEY shorthand is almost certainly preferable. 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 CX3 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. I'll have more to say about this operator in Chapter 10.

Here for interest is an SQL formulation of constraint CX3 that more closely resembles the **Tutorial D** formulation:

```
CREATE ASSERTION CX3 CHECK
 ( ( SELECT COUNT ( ALL SNO ) FROM S ) =
   ( SELECT COUNT ( DISTINCT SNO ) FROM S ) ) ;
```

Example 4:

```
CONSTRAINT CX4                 |  CREATE ASSERTION CX4 CHECK
COUNT ( S { SNO } ) =          |  ( NOT EXISTS ( SELECT *
COUNT ( S { SNO , CITY } ) ;   |                 FROM   S AS SX
                               |  WHERE EXISTS ( SELECT *
                               |                 FROM   S AS SY
                               |  WHERE SX.SNO = SY.SNO
                               |  AND   SX.CITY <> SY.CITY ) ) ) ;
```

Constraint CX4 says: Whenever two tuples in relvar S have the same supplier number, they also have the same city. In other words, there's a *functional dependence* from {SNO} to {CITY}, which would more usually be expressed as follows:

```
{ SNO } → { CITY }
```

Here's the definition:

> **Definition:** Let *A* and *B* be subsets of the heading of relvar *R*. Then relvar *R* satisfies the *functional dependence* (FD) *A* → *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* → *B* is read as "*B* is functionally dependent on *A*," or "*A* functionally determines *B*," or, more simply, just "*A* arrow *B*." As the example shows, however, a functional dependence is basically just another integrity constraint (though, like CX3, it isn't a "tuple" or "row" constraint).

Now, as noted in Chapter 5, the fact that relvar S satisfies this particular FD is a logical consequence of the fact that {SNO} is a key for that relvar. For that reason, there's no need to state it explicitly, just so long as the fact that {SNO} is a key *is* stated explicitly. But not all FDs are consequences of keys. For example, suppose it's the case that if two suppliers are in the same city, then they must have the same status (note that this constraint isn't satisfied by our usual sample values). This hypothetical new constraint is clearly an FD:

```
{ CITY } → { STATUS }
```

It can thus be stated in the style of constraint CX4 (exercise for the reader).

Now, you might be thinking that some shorthand syntax would be desirable for stating FDs, similar to the shorthand we already have for stating keys. Myself, I don't think so, because although not all FDs are consequences of keys in general, all FDs *will* be consequences of keys if the database is well designed. In other words, the very fact that FDs are hard to state if the database is badly designed might be seen as a small argument in favor of not designing the database badly in the first place!

NOTE

By "well designed" in the previous paragraph, I really mean *properly normalized*. Normalization as such is discussed further in Appendix B; note, however, that relational (or SQL) statements and expressions will work regardless of whether or not the relvars (or tables) are fully normalized. But I will at least add that those statements and expressions will often be easier to formulate (and, contrary to popular opinion, will often perform better too) if the tables are fully normalized—but normalization as such is primarily a database design issue, not a relational model or SQL issue.

Example 5:

```
CONSTRAINT CX5 IS_EMPTY        |   CREATE ASSERTION CX5 CHECK
   ( ( S JOIN SP )             |   ( NOT EXISTS
     WHERE STATUS < 20         |   ( SELECT *
     AND   PNO = 'P6' ) ;      |     FROM   S NATURAL JOIN SP
                               |     WHERE  STATUS < 20
                               |     AND    PNO = 'P6' ) ) ;
```

Constraint CX5 says: No supplier with status less than 20 can supply part P6. Observe that this constraint involves—or, better, interrelates—two distinct relvars, S and SP. In general, a database constraint might involve, or interrelate, any number of distinct relvars. *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. (Thus, constraints CX3 and CX4 were single-relvar constraints, while constraint CX5 is a multi-relvar constraint. Constraints CX1 and CX2 were also single-relvar constraints.) All of these terms are somewhat deprecated, however, for reasons to be discussed in the next chapter.

Example 6:

```
CONSTRAINT CX6                 |  CREATE ASSERTION CX6 CHECK
   SP { SNO } ⊆ S { SNO } ;    |  ( NOT EXISTS
                               |      ( SELECT SNO
                               |        FROM   SP
                               |        EXCEPT
                               |        SELECT SNO
                               |        FROM   S ) ) ;
```

Constraint CX6 says: Every supplier number in relvar SP must appear in relvar S. As you can see, the **Tutorial D** formulation involves a relational comparison; SQL doesn't support relational comparisons, however, and so we have to indulge in some circumlocution in the SQL formulation. Given that {SNO} is a key—in fact, the sole key—for relvar S, it's clear that constraint CX6 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 CX6.

Database Constraints in SQL

Any constraint that can be formulated by means of a CONSTRAINT statement in **Tutorial D** can be formulated by means of a CREATE ASSERTION statement in SQL, as examples CX1-CX6 in the previous section should have been sufficient to suggest.* Unlike **Tutorial D**, however, SQL has a feature according to which any such constraint can alternatively be specified as a *base table constraint*—i.e., as part of the definition of some base table. For example, here again is the SQL version of constraint CX5 from the previous section:

```
CREATE ASSERTION CX5 CHECK
     ( NOT EXISTS ( SELECT *
                    FROM   S NATURAL JOIN SP
                    WHERE  STATUS < 20
                    AND    PNO = 'P6' ) ) ;
```

This example could have been stated (in a slightly different form) using a CONSTRAINT specification on the CREATE TABLE for base table SP, like this:

```
CREATE TABLE SP
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( PNO <> 'P6' OR ( SELECT STATUS FROM S
                         WHERE  SNO = SP.SNO ) >= 20 ) ) ;
```

Note, however, that a logically equivalent formulation could have been specified as part of the definition of base table S instead—or base table P, or absolutely any base table in the database, come to that (see Exercise 8-17 at the end of the chapter).

Now, this alternative style can certainly be useful for "row constraints" (i.e., constraints that can be checked for an individual row in isolation), because it's simpler than its CREATE ASSERTION counterpart. Here, for example, are constraints CX1 and CX2 from the previous section, reformulated as base table constraints on base table S:

```
CREATE TABLE S
  ( ... ,
    CONSTRAINT CX1 CHECK ( STATUS >= 1 AND STATUS <= 100 ) ) ;

CREATE TABLE S
  ( ... ,
    CONSTRAINT CX2 CHECK ( STATUS = 20 OR CITY <> 'London' ) ) ;
```

For a constraint involving more than one base table, however, CREATE ASSERTION is usually better, because it avoids having to make an arbitrary choice as to which table to attach the constraint to.

* Except that, as you'll recall from Chapter 2, SQL constraints are supposed not to contain "possibly nondeterministic expressions," a rule that could cause serious problems in practice if true. See Chapter 12 for further discussion.

NOTE

Certain constraints—for example, NOT NULL constraints and key constraints for keys that involve just one column—can optionally be formulated as "column constraints" in SQL. A column constraint is one that's specified, not just as part of the definition of the base table in question, but as part of the definition of some specific column of that base table. For simplicity, I'll ignore this possibility in this book, except for NOT NULL constraints in particular.

Two last points to close this section:

- Be aware that any constraint stated as part of the CREATE TABLE for base table *T* is automatically satisfied if *T* is empty—even if the constraint is of the form "*T* mustn't be empty"! (Or even if it's of the form "*T* must contain -5 rows," or the form "1 = 0," come to that.) See Exercises 8-15 and 8-16 at the end of the chapter.
- *(Important!)* While most current products do support key and foreign key constraints, they don't support CREATE ASSERTION at all, and they don't support base table constraints any more complicated than simple row constraints. (Formally, they don't permit base table constraints to contain a subquery.) In practice, therefore, most constraints will have to be enforced by means of procedural code (possibly triggered procedures)—and that code can be quite difficult to write, too.* This state of affairs represents a serious defect in today's products, and it needs to be remedied, urgently.

Transactions

Despite the defect identified at the end of the previous section, I do need to assume for the rest of the chapter (just as the relational model does, in fact) that database constraints of arbitrary complexity can be stated declaratively. The question now arises: When are such 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 such checking should 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.

Transaction theory is a large topic in its own right. But 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

* In this connection, I'd like to recommend the book *Applied Mathematics for Database Professionals*, by Lex de Haan and Toon Koppelaars (see Appendix D).

(see Appendix D). 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:

- *Atomicity* means that transactions are "all or nothing."
- *Consistency* means that any given transaction transforms a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points. (A database state is consistent if and only if it satisfies all defined constraints; consistency is just another word for integrity in this context.)
- *Isolation* means that any given transaction's updates are concealed from all other transactions until such time as the given transaction commits.
- *Durability* means that 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 statements have to be that unit; in other words, I believe database constraints must 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 this: As we know from Chapter 5, a database can be regarded as a collection of propositions, propositions that we believe to be true ones. 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, 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.

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 completeness 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 really believe in the isolation property. Part of the problem here is that the word *isolation* doesn't quite 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 *TX1* produces some result, in the database or elsewhere, that's subsequently read by transaction *TX2*, then *TX1* and *XT2* aren't truly isolated from each other (and this remark applies regardless of whether *TX1* and *TX2* run concurrently or otherwise). In particular, therefore, if (a) *TX1* sees an inconsistent state of the database and therefore produces an incorrect result, and (b) that result is then seen by *TX2*, then (c) the inconsistency seen by *TX1* has effectively been propagated to *TX2*. 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 deal with the possibility that the database might be inconsistent when it's invoked. There's a severe loss of orthogonality if some piece of code 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 the next chapter) 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, suppose we have two views with **Tutorial D** definitions as follows (LS = London suppliers, NLS = non London suppliers):

```
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 CX7 IS_EMPTY       |   CREATE ASSERTION CX7 CHECK
   ( LS { SNO } JOIN          |     ( NOT EXISTS
           NLS { SNO } ) ;    |     ( 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 (since, logically, there's no real difference between the two).

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 6). For example, consider the expression (SP JOIN S){PNO}. Now, the join here is based on the match between a foreign key in a referencing relvar, SP, and the corresponding key in the referenced relvar, S. As a consequence, every SP tuple does join to some S tuple, and every SP tuple thus does contribute a part number to the projection that's 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; with join in general, each operand will include some tuples that have no counterpart in the other 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 necessarily does 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.

Now, in principle any constraint whatsoever can be used in semantic optimization; we're not limited to foreign key constraints as in the example.* 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—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, then that constraint must be satisfied at all times (or rather, and 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:* Alternatively, 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.

To sum up: 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 must be undone and an exception raised.

* The constraint must be stated declaratively, however; obviously there's no way semantic optimization can "understand" and exploit constraints that have been specified procedurally (and so we have here another strong reason for requiring declarative constraint support).

But Doesn't Some Checking Have to Be Deferred?

The arguments of the previous section notwithstanding, the conventional wisdom is that multi-relvar constraint checking, at least, has to be deferred to commit time. By way of example, suppose the suppliers-and-parts database is subject to the following constraint:

```
CONSTRAINT CX8
    COUNT ( ( S WHERE SNO = 'S1' ) { CITY }
              UNION
            ( P WHERE 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 this constraint and our usual sample values, then, both of the following SQL UPDATEs will fail under immediate checking:

```
UPDATE S SET CITY = 'Paris' WHERE SNO = 'S1' ;

UPDATE P SET CITY = 'Paris' WHERE PNO = 'P1' ;
```

Note that I show these UPDATEs in SQL precisely because checking *is* immediate in **Tutorial D** and the conventional solution to the problem therefore doesn't work in **Tutorial D**. What is that conventional solution? *Answer:* We defer the checking of the constraint to commit time,* and we make sure the two UPDATEs are part of the same transaction, like this:

```
START TRANSACTION ;
   UPDATE S SET CITY = 'Paris' WHERE SNO = 'S1' ;
   UPDATE P SET CITY = 'Paris' WHERE PNO = 'P1' ;
COMMIT ;
```

In this conventional solution, the constraint is checked at end of transaction, 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 assuming that tuples or rows for S1 and P1 do exist), it would get the answer *yes*.

* In case you're wondering how the deferring is done, I should explain that in general—there are some exceptions that don't need to concern us here—every SQL constraint is defined to be (a) either DEFERRABLE or NOT DEFERRABLE and (b) either INITIALLY DEFERRED or INITIALLY IMMEDIATE. Then, at run time, the statement SET CONSTRAINTS <*constraint name commalist*> <*option*>, where <*option*> is either DEFERRED or IMMEDIATE, sets the "mode" of the specified constraint(s) accordingly. COMMIT forces all constraints into immediate mode; if some implied integrity check then fails, the COMMIT fails, and the transaction is rolled back.

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," as it were. For example (switching back now to **Tutorial D**):

```
UPDATE S WHERE SNO = 'S1' : { CITY := 'Paris' } ,
UPDATE P WHERE 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, as we know, 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 source expressions on the right sides are evaluated.
- Second, all of the individual assignments (to the variables on the left sides) are executed in sequence as written.

(Actually this definition requires a slight refinement in the case where two or more of the individual assignments specify the same target variable, but the details are beyond the scope of this book.) Observe that, precisely because all of the source expressions are evaluated before any of the individual assignments are executed, none of those individual assignments can depend on the result of any other (and so "executing them in sequence as written" is only a manner of speaking; in fact, the sequence is irrelevant). 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 the operation in the first place. In the example, therefore, the double assignment succeeds where the two separate single assignments failed. Note in particular 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.

ASIDE

Perhaps I should remind you here that all statements are *semantically atomic* in the relational model. In fact, most such statements are syntactically atomic too; multiple assignment is an exception, because it's semantically atomic but not syntactically so.

So what about multiple assignment in SQL? Well, SQL does have some support for this operation; in fact, it's had some such support for many years. First of all, referential actions such as CASCADE imply, in effect, that a single DELETE or UPDATE statement can cause several base tables to be updated "at the same time," as part of a single operation. Second, the ability to update (for example) certain join views, if it's supported, implies the same thing. Third, FETCH INTO and SELECT INTO are both multiple assignment operations, of a kind. Fourth, SQL explicitly supports a multiple assignment form of the SET statement (that's exactly what row assignment is—see Chapters 2 and 3). 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 "simultaneous" assignment to several different *tables*—which is precisely the case illustrated by the foregoing example, and precisely what we need in order to avoid having to do deferred checking.*

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.

Recommendation: Given the state of today's SQL products, some constraint checking will almost certainly have to be deferred, even though logically speaking it should be immediate. In such a case, you should do whatever it takes—which in practice probably means terminating the transaction—to force the check to be done before performing any operation that might rely on the constraint being satisfied.

Constraints and Predicates

Recall from Chapter 5 that the predicate for any given 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 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 operation) that there's a supplier S6 named Lopez with status 30 and city Madrid,

* I'm told, however, that this functionality will be provided in some future version of the standard.

† 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).

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 any more).

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.

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 it's inconsistent, then it contains at least one false proposition, but the converse isn't necessarily so. Or to put it another way, *correct* implies *consistent* (but not the other way around), and *inconsistent* implies *incorrect* (but not the other way around)—where to say 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, and let *C1*, *C2*, ..., *Cm* (*m* ≥ 0) be all of the database constraints, single- or multi-relvar, that mention *R*. Assume for simplicity that each *Ci* is just a boolean expression (i.e., ignore the constraint names, for simplicity). Then the boolean expression

```
( C1 ) AND ( C2 ) AND ... AND ( Cm ) 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 in the unlikely event[*] that no constraints at all are defined for a given relvar, then 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 *DB* be a database, and let *R1*, *R2*, ..., *Rn* (*n* ≥ 0) be all of the relvars in *DB*. Let the constraints for those relvars be *RC1*, *RC2*, ..., *RCn*, respectively. Then the boolean expression

```
( RC1 ) AND ( RC2 ) AND ... AND ( RCn ) AND TRUE
```

is *the total database constraint* for *DB* (but I'll refer to it for the purposes of this book as just *the* constraint for *DB*). And here's a correspondingly extended—in fact, the final—version of **The Golden Rule**:

* "Unlikely" is right; *every* relvar is supposed to have a key constraint at the very least.

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. *Note:* In case you're not convinced, I refer you to the further discussion of this issue in Chapter 10.

Miscellaneous Issues

There are a few more points to do with integrity that I need to cover but don't fit very well into any of the preceding sections.

First of all, a constraint, since it's basically a boolean expression that must evaluate to TRUE, is in fact a proposition (I more or less suggested as much in the previous section, but I never came out and stated it explicitly). For example, here's constraint CX1 once again from the section "Database Constraints":

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

The relvar name "S" here constitutes what logicians call a *designator*; when the constraint is checked, it designates a specific value—namely, the value of that relvar at the time in question. By definition, that value is a relation (*s*, say), and so the constraint effectively becomes:

```
CONSTRAINT CX1 IS_EMPTY ( s WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Clearly, the boolean expression here—which is really the constraint as such, "CONSTRAINT CX1" being little more than window dressing—is certainly either true or false, unconditionally, and that's the definition of what it means to be a proposition (see Chapter 5).

Second, suppose relvar S already contains a tuple that violates constraint CX1 when the CONSTRAINT statement just shown is executed; then that execution must fail. 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, relational databases are supposed to satisfy the referential integrity rule, which says there mustn't be any unmatched foreign key values. Now, in Chapter 1, I referred to this rule as a "generic integrity constraint." But it should be clear that it's different in kind from the constraints we've been examining in this chapter. It's really a *meta*constraint, in a sense; what it says is that every specific database must satisfy the specific referential constraints that apply to that particular database. In the case of the suppliers-and-parts database, for example, it says the referential constraints from SP to S and P must be satisfied—because if they aren't, then that database will violate the referential integrity metaconstraint. Likewise, in the case of the departments-and-employees database from Chapter 1, the referential constraint from EMP to DEPT must be satisfied, because if it isn't, then again that database will violate the referential integrity metaconstraint.

Another issue I didn't mention previously is the possibility of supporting what are called 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 (by contrast, a constraint that isn't a transition constraint is sometimes said to be a *state* constraint). For example, a person's marital status can change from "never married" to "married" but not the other way around. Here's a database example ("No supplier's status must ever decrease"):

```
CONSTRAINT CX9 IS_EMPTY
   ( ( ( S' { SNO , STATUS } RENAME ( STATUS AS OLD ) )
         JOIN
       ( S  { SNO , STATUS } RENAME ( STATUS AS NEW ) ) )
     WHERE OLD > NEW ) ;
```

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 CX9 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.)

Transition constraints aren't currently supported in either **Tutorial D** or SQL (other than procedurally).

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 current products; indeed, they seem to be underappreciated at best, if not completely misunderstood. The emphasis in practice 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.*

* I don't mean to suggest here that proper support for integrity checking implies bad performance; in fact, I think it ought to 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.

Now, I don't want you to misunderstand me here. Of course performance is important too. Functionally speaking, a system that doesn't deliver at least adequate performance isn't a system (not a usable one, at any rate). 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.

Exercises

Exercise 8-1. Define the terms *type constraint* and *database constraint*. When are such constraints checked? What happens if the check fails?

Exercise 8-2. State **The Golden Rule**. Is it true that this rule can be violated if and only if some individual single-relvar constraint is violated?

Exercise 8-3. What do you understand by the terms *assertion*; *attribute constraint*; *base table constraint*; *column constraint*; *multi-relvar constraint*; *referential constraint*; *relvar constraint*; *row constraint*; *single-relvar constraint*; *state constraint*; *"the" (total) database constraint*; *"the" (total) relvar constraint*; *transition constraint*; *tuple constraint*? Which of these categories if any do (a) key constraints, (b) foreign key constraints, fall into?

Exercise 8-4. Distinguish between possible and physical representations.

Exercise 8-5. With the **Tutorial D** definition of type QTY as given in the body of the chapter, what do the following expressions return?

a. `THE_Q ( QTY ( 345 ) )`

b. `QTY ( THE_Q ( QTY ) )`

Exercise 8-6. Explain as carefully as you can (a) what a selector is; (b) what a THE_ operator is. *Note:* This exercise essentially repeats ones in previous chapters, but now you should be able to be more specific in your answers.

Exercise 8-7. Suppose the only legal CITY values are London, Paris, Rome, Athens, Oslo, Stockholm, Madrid, and Amsterdam. Define a **Tutorial D** type called CITY that satisfies this constraint.

Exercise 8-8. Following on from the previous exercise, show how you could impose the corresponding constraint in SQL on the CITY columns in base tables S and P. Give at least two solutions. Compare and contrast those solutions with each other and with your answer to the previous exercise.

Exercise 8-9. Define supplier numbers as a **Tutorial D** user defined 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. State any assumptions you make regarding the availability of operators to help with your definition.

Exercise 8-10. A line segment is a straight line connecting two points in the euclidean plane. Give a corresponding **Tutorial D** type definition.

Exercise 8-11. Can you think of a type for which we might want to specify two different possreps? Does it make sense for two or more possreps for the same type each to include a type constraint?

Exercise 8-12. Can you think of a type for which different possreps might have different numbers of components?

Exercise 8-13. Which operations might cause constraints CX1-CX9 from the body of the chapter to be violated?

Exercise 8-14. Does **Tutorial D** have anything directly analogous to SQL's base table constraints?

Exercise 8-15. In SQL, what is it exactly (i.e., formally) that makes base table constraints a little easier to state than their CREATE ASSERTION counterparts? *Note:* I haven't covered enough in this book yet to enable you to answer this question. Nevertheless, you might want to think about it now, or possibly use it as a basis for group discussion.

Exercise 8-16. Following on from the previous question, a base table constraint is automatically regarded as satisfied in SQL if the pertinent base table is empty. Why exactly do you think this is so (I mean, what's the formal reason)? Does **Tutorial D** display any analogous behavior?

Exercise 8-17. In the body of the chapter, I gave a version of constraint CX5 as a base table constraint on table SP. However, I pointed out that it could alternatively have been formulated as such a constraint on base table S, or base table P, or in fact any base table in the database. Give such alternative formulations.

Exercise 8-18. Constraint CX1 (for example) had the property that it could be checked for a given tuple by examining just that tuple in isolation; constraint CX5 (for example) did not. What is it, formally, that accounts for this difference? What's the pragmatic significance, if any, of this difference?

Exercise 8-19. Can you give a **Tutorial D** database constraint that's exactly equivalent to the specification KEY{SNO} for relvar S?

Exercise 8-20. Give an SQL formulation of constraint CX8 from the body of the chapter.

Exercise 8-21. 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 be 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.

n. No shipment can have a total weight (part weight times shipment quantity) greater than 20,000 pounds.

In each case, state which operations might cause the constraint to be violated. *Note:* Don't forget image relations, which might be helpful in formulating some of the foregoing constraints.

Exercise 8-22. In the body of the chapter, I defined the total database constraint to be a boolean expression of this form:

```
( RC1 ) AND ( RC2 ) AND ... AND ( RCn ) AND TRUE
```

What's the significance of that "AND TRUE"?

Exercise 8-23. 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 8-24. Suppose no "cascade delete" rule is stated 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 (i.e., without raising the possibility of a referential integrity violation).

Exercise 8-25. 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 8-26. Distinguish between correctness and consistency.

Exercise 8-27. Investigate any SQL product that might be available to you. What semantic optimization does it support, if any?

Exercise 8-28. Why do you think SQL fails to support type constraints? What are the consequences of this state of affairs?

Exercise 8-29. 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. To what extent do the concepts discussed apply to nonscalar types and/or system defined types?

Exercise 8-30. Do any of your SQL solutions to previous exercises involve any “possibly nondeterministic expressions"? If so, is that a problem? If so, what can be done about it?

CHAPTER NINE

SQL and Views

INTUITIVELY, THERE ARE SEVERAL DIFFERENT WAYS OF LOOKING AT WHAT A VIEW IS, ALL OF WHICH ARE VALID AND ALL OF WHICH CAN BE HELPFUL IN THE RIGHT CIRCUMSTANCES:

- A view is a virtual relvar; i.e., it's a relvar that "looks and feels" just like a base relvar but (unlike a base relvar) doesn't exist independently of other relvars—in fact, it's defined in terms of such other relvars.
- A view is a derived relvar; in other words, it's a relvar that's explicitly derived (and known to be derived, at least by some people) from certain other relvars. *Note:* If you're wondering what the difference is between a derived relvar and a virtual one, I should explain that all virtual relvars are derived but some derived ones aren't virtual. See the section "Views and Snapshots" later in the chapter.
- A view is a window into the relvars from which it's derived; thus, operations on the view are to be understood as "really" being operations on those underlying relvars.
- A view is what some writers call a "canned query" (or, more precisely, a named relational expression).

As usual, in what follows I'll discuss these concepts in both relational and SQL terms. Regarding SQL specifically, however, let me remind you of something I said in Chapter 1: A view is a table!—or, as I would prefer to say, a relvar. SQL documentation often uses expressions like "tables and views," thereby suggesting that tables and views are different things—but they're not; in many ways, in fact, it's the whole point about a view that it *is* a table. So don't fall into the common trap of thinking the term *table* means a base table specifically. People who fall into that trap aren't thinking relationally, and they're likely to make mistakes as a consequence; in fact, several such mistakes can be found in the design of SQL itself. Indeed, it could be argued that the very names of the CREATE TABLE and CREATE VIEW operators are at least a psychological mistake, in that they tend to reinforce both (a) the idea that the term *table* means a base table specifically and (b) the idea that views and tables are different things. Be on the lookout for confusion in this area.

One last preliminary point: On the question of whether access to the database should "always" be through views, see the section "Column Naming in SQL" in Chapter 3 or the section "The Reliance on Attribute Names" in Chapter 6.

Views Are Relvars

Of those informal characterizations listed above of what a view is, the following definition might appear to favor one over the rest—but those informal characterizations are all equivalent anyway, loosely speaking:

> **Definition:** A *view 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 (the *view defining expression*) is specified when *V* is defined and must mention at least one relvar.

The following examples ("London suppliers" and "non London suppliers") are repeated from Chapter 8, except that I now give SQL definitions as well:

```
VAR LS VIRTUAL                   |  CREATE VIEW LS
  ( S WHERE CITY = 'London' ) ;  |    AS ( SELECT *
                                 |         FROM   S
                                 |         WHERE  CITY = 'London' )
                                 |         WITH CHECK OPTION ;

VAR NLS VIRTUAL                  |   CREATE VIEW NLS
 ( S WHERE CITY   'London' ) ;   |   AS ( SELECT *
                                 |        FROM   S
                                 |        WHERE  CITY <> 'London' )
                                 |        WITH CHECK OPTION ;
```

Note that these are both *restriction* views: Their value at any given time is a certain restriction of the value at that time of base relvar S. Some syntax issues:

- The parentheses in the examples are unnecessary but not wrong. I include them for clarity.
- CREATE VIEW in SQL allows a parenthesized commalist of column names to appear following the view name, as in this example:

  ```
  CREATE VIEW SDS ( SNAME , DOUBLE_STATUS )
    AS ( SELECT DISTINCT SNAME , 2 * STATUS
         FROM   S ) ;
  ```

  Recommendation: Don't do this—follow the recommendations given in Chapter 3 under "Column Naming in SQL" instead. For example, view SDS can equally well (in fact better) be defined like this:

  ```
  CREATE VIEW SDS
    AS ( SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
         FROM   S ) ;
  ```

  Note in particular that this latter style means you have to tell the system just once instead of twice that one of the view columns is called SNAME.
- CREATE VIEW in SQL also allows WITH CHECK OPTION to be specified if it regards the view as updatable. **Recommendation:** Always specify this option if possible. See the section "Update Operations" for further discussion.

The Principle of Interchangeability

Since views are relvars, essentially everything I said in previous chapters regarding relvars in general applies to views in particular. Subsequent sections discuss specific aspects of this observation in detail. First, however, there's a more fundamental point I need to explain.

Consider the example of London vs. non London suppliers again. In that example, S is a base relvar and LS and NLS are views. *But it could have been the other way around*—that is, we could have made LS and NLS base relvars and S a view, like this (**Tutorial D** only, for simplicity):

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

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

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

(Recall from Chapter 6 that D_UNION—"disjoint union"—is a version of union that requires its operands to have no tuples in common.) *Note:* In order to guarantee that this design is logically equivalent to the original one, we would also have to state and enforce certain constraints—in particular, constraints to the effect that every CITY value in LS is London and no CITY value in NLS is—but I omit such details here. See the section "Views and Constraints" later for further consideration of such matters.

Anyway, the message of the example is that, in general, which relvars are base ones and which virtual is arbitrary (at least from a formal point of view). In the example, we could design the database in at least two different ways: ways, that is, that are logically distinct but are information equivalent. (By *information equivalent* here, I mean the two designs represent the same information; i.e., for any query on one, there's a logically equivalent query on the other.) And *The Principle of Interchangeability* follows logically from such considerations:

> **Definition:** *The Principle of Interchangeability* (of base and virtual relvars) states that there must be no arbitrary and unnecessary distinctions between base and virtual relvars; i.e., virtual relvars should "look and feel" just like base relvars so far as users are concerned.

Here are some implications of this principle:

- As I've already suggested in passing, views are subject to integrity constraints, just like base relvars. (We usually think of integrity constraints as applying to base relvars specifically, but *The Principle of Interchangeability* shows that this position isn't really tenable.) See the section "Views and Constraints," later.
- 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. Again, see the section "Views and Constraints," later.
- I didn't mention this point in Chapter 1, but the "entity integrity" rule is supposed to apply specifically to base relvars, not views, and it thereby violates *The Principle of Interchangeability*. Of course, I reject that rule anyway, because it has to do with nulls. (I also reject it because it has to do with primary keys specifically instead of candidate keys in general, but let that pass.)
- 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—then it violates *The Principle of Interchangeability*. (It might violate *The Information Principle*, too. See Appendix A.) Now, 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 they are, most unfortunately, in the SQL standard, as well as in most of the major SQL products—as some kind of *object* ID in the object oriented sense, then they're very definitely prohibited! Object IDs are effectively pointers, and (to repeat from Chapter 2) the relational model explicitly prohibits pointers.
- The distinction discussed in the previous chapter between single- and multi-relvar constraints is more apparent than real (and the terminology is deprecated, somewhat, for that very reason). Indeed, an example in that chapter showed that the very same constraint could be a "single-relvar" constraint with one design for the database and a "multi-relvar" constraint with another.

- Perhaps most important of all, we must be able to update views—because if not, then that fact in itself would constitute the clearest possible violation of *The Principle of Interchangeability*. See the section "Update Operations," later.

Relation Constants

You might have noticed, in the formal definition of what a view was at the beginning of the present section, that the view defining expression is required to mention at least one relvar. Why? Because if it didn't, the "virtual relvar" wouldn't be a relvar at all!—I mean, it wouldn't be a variable, and it wouldn't be updatable. For example, the following is a valid CREATE VIEW statement in SQL:

```
CREATE VIEW S_CONST ( SNO , SNAME , STATUS , CITY ) AS
           VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
                  ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
                  ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
                  ( 'S4' , 'Clark' , 20 , 'London' ) ,
                  ( 'S5' , 'Adams' , 30 , 'Athens' ) ;
```

But this view certainly can't be updated. In other words, it's not a variable at all, let alone a virtual one; rather, it's what might be called a *named relation constant*. Let me elaborate.

- First of all, I regard the terms *constant* and *value* as synonymous. Note, therefore, that there's a logical difference between a constant and a literal; a literal isn't a constant but is, rather, a symbol—sometimes known as a "self defining" symbol—that denotes a constant.
- Strictly speaking, there's also a logical difference between a constant and a *named* constant; a constant is a value, but a named constant is like a variable, except that its value can't be changed. That said, however, for the remainder of this discussion I'll take the term *constant* to mean a named constant specifically.
- Constants can be of any type we like, naturally, but relation constants are my major focus here. Now, **Tutorial D** doesn't currently support relation constants, but if it did, a relation constant (or "relcon") definition would probably look something like this example:

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

Now, I do believe it would be desirable to provide some kind of "relcon" functionality along the lines sketched above. In fact, **Tutorial D** already provides two system defined relcons: namely, TABLE_DUM and TABLE_DEE, both of which are extremely important, as we know. Apart from these two, however, neither **Tutorial D** nor SQL currently provides any direct support for relcons. It's true that (as we've seen) such support can be simulated by means of the conventional view mechanism; however, there's a huge logical difference between constants and variables, and I don't think it helps the cause of understanding to pretend that relcons are relvars.

Views and Predicates

The Principle of Interchangeability implies that a view (just like a base relvar) has a relvar predicate, in which the parameters correspond one to one to the attributes of the relvar—i.e., the view—in question. However, the predicate that applies to a view *V* is a *derived* predicate: It's derived from the predicates for the relvars in terms of which *V* is defined, in accordance with the semantics of the relational operations involved in the view defining expression. In fact, you already know this: In Chapter 6, I explained that every relational expression has a corresponding predicate, and of course a view has exactly the predicate that corresponds to its view defining expression. For example, consider view LS ("London suppliers") once again, as defined near the beginning of the section "Views Are Relvars"). That view is a restriction of relvar S, and its predicate is therefore the logical AND of the predicate for S and the restriction condition:

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

Or more colloquially:

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

Note, however, that this more colloquial form obscures the fact that CITY is a parameter. Indeed it *is* a parameter, but the corresponding argument is always the constant 'London'. (Precisely for this reason, in fact, a more realistic version of view LS in practice would probably project away the CITY attribute.)

Similarly, the predicate for view NLS is:

> *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY, which isn't London.*

Retrieval Operations

The Principle of Interchangeability implies that users 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 operations on the base relvars in terms of which the views are ultimately defined. *Note:* I say "ultimately" here because if views really do behave just like base relvars, then one thing we can do is define further views on top of them, as in this SQL example:

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

In this section, I limit my attention to the mapping of read-only or "retrieval" operations, for simplicity (I remind you that the operations of the relational agebra are indeed all read-only). In fact, the process of mapping a read-only operation on a view to operations on the underlying relvars is essentially straightforward. For example, suppose we issue this SQL query on view LS:

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

First, then, 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 directly evaluated. However—and for performance reasons perhaps more significantly—it can also be simplified to:

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

In all likelihood, this latter expression is the one that will actually be evaluated.

Now, it's important to understand that the reason the foregoing procedure works is precisely because of the relational closure property. 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.

For obvious reasons, the foregoing procedure for implementing read-only operations on views is known as the *substitution* procedure. Incidentally, it's worth noting that the procedure didn't always work in early versions of SQL: to be specific, in versions prior to 1992. The reason was that those early versions didn't fully support the closure property. As a result, certain apparently innocuous queries against certain apparently innocuous tables (actually views) failed—and failed, moreover, in ways that were hard to explain. Here's a simple example. First the view definition:

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

Now a query:

* As elsewhere in this book, I would have preferred to use the more formal term *object* in this sentence in place of the vague term *thing*, but (as noted in the previous chapter) *object* is a rather loaded term in computing contexts.

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

This query failed in the SQL standard, prior to 1992, because simple substitution yielded something like the following syntactically invalid expression:

```
SELECT CITY
FROM   S
WHERE  SUM ( STATUS ) > 25
GROUP  BY CITY
```

Now, the standard has been fixed in this regard, as you probably know; however, it doesn't follow that the products have!—and indeed, the last time I looked, there was at least one major product that still hadn't. Indeed, precisely because of problems like the foregoing (among others), the product in question actually implements certain view retrievals by *materialization* instead of substitution; that is, it actually evaluates the view defining expression, builds a table to hold the result of that evaluation, and then executes the requested retrieval against that materialized table. And while such an implementation might be argued to conform to the letter of the relational model, as it were, I don't think it can be said to conform to the spirit.

Views and Constraints

The Principle of Interchangeability implies that views not only have relvar predicates like base relvars, they also have relvar constraints like base relvars—by which I mean they have both individual relvar constraints and what in Chapter 8 I called a *total* relvar constraint (for the relvar in question). As with predicates, however, the constraints that apply to a view *V* are *derived*: They're derived from the constraints for the relvars in terms of which *V* is defined, in accordance with the semantics of the relational operations involved in the view defining expression. By way of example, consider view LS once again. That view is a restriction of relvar S—i.e., its view defining expression specifies a restriction operation on relvar S—and so its (total) relvar constraint is the logical AND of the (total) relvar constraint for S and the specified restriction condition. Let's suppose for the sake of the example that the only constraint that applies to base relvar S is the key constraint implied by the fact that {SNO} is a key. Then the total relvar constraint for view LS is the AND of that key constraint and the constraint that the city is London, and view LS is required to satisfy that constraint at all times (in other words, **The Golden Rule** applies to views just as it does to base relvars).

For simplicity, from this point forward I'll use the term *view constraint* to refer to any constraint that applies to some view. Now, just because view constraints are always derived in the sense explained above, it doesn't follow that there's no need to declare them explicitly in practice. For one thing, the system might not be "intelligent" enough to carry out the inferences needed to determine the constraints that apply to some view; for another, such explicit declarations can at least serve documentation purposes (i.e., they can help to explain the semantics of the view in question to users, if not to the system); and there's another reason too, which I'll get to in a little while.

I claim, then, that it should be possible to declare explicit constraints for views. In particular, it should be possible (a) to include explicit KEY and FOREIGN KEY specifications in view definitions and (b) to allow the target relvar in a FOREIGN KEY specification to be a view. For example, here's an example to illustrate possibility (a):

```
VAR LS VIRTUAL ( S WHERE CITY = 'London' )
    KEY { SNO } ;
```

Tutorial D does permit such specifications; SQL doesn't. **Recommendation:** In SQL, include such specifications in the form of comments. For example:

```
CREATE VIEW LS
  AS ( SELECT *
       FROM   S
       WHERE  CITY = 'London' )
       /* UNIQUE ( SNO ) */
  WITH CHECK OPTION ;
```

NOTE

As I've said, SQL doesn't permit view constraints to be formulated explicitly as part of the view definition; however, logically equivalent constraints can be formulated by means of CREATE ASSERTION (if it's supported, that is). More generally, in fact, CREATE ASSERTION allows us to formulate constraints of any kind we like for any table that could be a view if we chose to define it as such—where by "any table" I mean, more precisely, more precisely, the value denoted by any arbitrary table expression. I'll have more to say about this possibility in a few moments.

Now, having said that it should be possible to declare explicit constraints on views, I should now add that sometimes it might be a good idea not to do so, because it could lead to redundant checking. For example, as I've said, the specification KEY{SNO} clearly applies to view LS—but that's because it applies to base relvar S as well, and declaring it explicitly for view LS might simply lead to the same constraint being checked twice. (But it should still be stated as part of the view documentation, somehow, because it's part of the semantics of the view.)

Perhaps more to the point, there certainly are situations where declaring view constraints explicitly could be a good idea. Here's an example, expressed in terms of SQL for definiteness. We're given two base tables that look like this (in outline):

```
CREATE TABLE FDH
     ( FLIGHT ... , DESTINATION ... , HOUR ... ,
       UNIQUE ( FLIGHT ) ) ;
```

```
CREATE TABLE DFGP
     ( DAY ... , FLIGHT ... , GATE ... , PILOT ... ,
       UNIQUE ( DAY , FLIGHT ) ) ;
```

These two tables have predicates *Flight FLIGHT leaves at hour HOUR for destination DESTINATION* (for FDH) and *On day DAY, flight FLIGHT with pilot PILOT leaves from gate GATE* (for DFGP), respectively. They're subject to the following constraints (expressed here in a kind of pseudo logical style):

```
IF ( f1,n1,h ), ( f2,n2,h ) IN FDH AND
   ( d,f1,g,p1 ), ( d,f2,g,p2 ) IN DFGP
THEN f1 = f2 AND p1 = p2

IF ( f1,n1,h ), ( f2,n2,h ) IN FDH AND
   ( d,f1,g1,p ), ( d,f2,g2,p ) IN DFGP
THEN f1 = f2 AND g1 = g2
```

Explanation: The first of these constraints says that if (a) two rows of FDH have the same HOUR *h*, and (b) two rows of DFGP, one each for the FLIGHTs *f1* and *f2* in the two FDH rows, have the same DAY *d* and GATE *g*, then the two FDH rows must be the same and the two DFGP rows must be the same—in other words, if we know the HOUR, DAY, and GATE, then the FLIGHT and PILOT are determined. The second constraint is analogous: If (a) two rows of FDH have the same HOUR *h*, and (b) two rows of DFGP, one each for the FLIGHTs *f1* and *f2* in the two FDH rows, have the same DAY *d* and PILOT *p*, then the two FDH rows must be the same and the two DFGP rows must be the same—in other words, if we know the HOUR, DAY, and PILOT, then the FLIGHT and GATE are determined.

Now, stating these constraints directly in terms of the two base tables is fairly nontrivial:

```
CREATE ASSERTION BTCX1 CHECK
 ( NOT ( EXISTS ( SELECT * FROM FDH AS FX WHERE
         EXISTS ( SELECT * FROM FDH AS FY WHERE
         EXISTS ( SELECT * FROM DFGP AS DX WHERE
         EXISTS ( SELECT * FROM DFGP AS DY WHERE
                  FY.HOUR = FX.HOUR AND
                  DX.FLIGHT = FX.FLIGHT AND
                  DY.FLIGHT = FY.FLIGHT AND
                  DY.DAY = DX.DAY AND
                  DY.GATE = DX.GATE AND
                ( FX.FLIGHT <> FY.FLIGHT OR
                  DX.PILOT <> DY.PILOT ) ) ) ) ) ) ) ;

CREATE ASSERTION BTCX2 CHECK
 ( NOT ( EXISTS ( SELECT * FROM FDH AS FX WHERE
         EXISTS ( SELECT * FROM FDH AS FY WHERE
         EXISTS ( SELECT * FROM DFGP AS DX WHERE
         EXISTS ( SELECT * FROM DFGP AS DY WHERE
                  FY.HOUR = FX.HOUR AND
                  DX.FLIGHT = FX.FLIGHT AND
                  DY.FLIGHT = FY.FLIGHT AND
                  DY.DAY = DX.DAY AND
                  DY.PILOT = DX.PILOT AND
                ( FX.FLIGHT <> FY.FLIGHT AND
                  DX.GATE <> DY.GATE ) ) ) ) ) ) ) ;
```

But stating them in the form of key constraints on a view definition, if that were permitted, would take care of matters nicely:

```
CREATE VIEW V AS
     ( FDH NATURAL JOIN DFGP ,
       UNIQUE ( DAY , HOUR , GATE ) ,     /* hypothetical */
       UNIQUE ( DAY , HOUR , PILOT ) ) ;  /* syntax !!!!! */
```

Since this solution isn't available, we should at least specify those hypothetical view constraints in terms of suitable assertions:

```
CREATE VIEW V AS FDH NATURAL JOIN DFGP ;

CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE FROM V ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT FROM V ) ) ;
```

In fact we don't actually have to define the view V in order to define these constraints—we could simply replace the references to V in the UNIQUE expression by the view defining expression for V, like this:

```
CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE
                        FROM   FDH NATURAL JOIN DFGP ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT
                        FROM   FDH NATURAL JOIN DFGP ) ) ;
```

Update Operations

The Principle of Interchangeability implies that views must be updatable (i.e., assignable to). Now, I can hear some readers objecting right away: Surely some views just can't be updated, can they? For example, consider a view defined as the join—a many to many join, observe—of relvars S and P on {CITY}; surely we can't insert a tuple into, or delete a tuple from, that view, can we? *Note:* I apologize for the sloppy manner of speaking here; as we know from Chapter 5, there's no such thing as "inserting or deleting a tuple" in the relational model. But to be too pedantic about such matters in the present discussion would get in the way of understanding, probably.

Well, even if it's true—which it might or might not be—that we can't insert a tuple into or delete a tuple from S JOIN P, let me point out that certain updates on certain *base* relvars can't be done, either. For example, inserting a tuple into relvar SP will fail if the SNO value in that tuple doesn't currently exist in relvar S. Thus, updates on base relvars can always fail on integrity constraint violations—*and the same is true for updates on views*. In other words, it isn't that some views are inherently nonupdatable, but rather that some updates on some views will fail on integrity constraint violations (i.e., violations of **The Golden Rule**). *Note:* Actually, updates, on both base relvars and views, can fail on violations of *The Assignment Principle*, too. For simplicity, I'll ignore this point, for the most part.

So let *V* be a view; in order to support updates on *V* properly, then, the system needs to know the total constraint, *VC* say, for *V*. In other words, it needs to be able to perform *constraint inference*, so that, given the constraints that apply to the relvars in terms of which *V* is defined, it can determine *VC*. As I'm sure you know, however, SQL products today do, or are capable of doing, very little in the way of such constraint inference. As a result, SQL's support for view updating is quite weak (what's more, this is true of the standard as well as the major products). Typically, in fact, SQL products don't allow updating on views any more complex than simple restrictions and/or projections of a single underlying base table (and even then there are problems). For example, consider view LS once again. That view is just a restriction of base table S, and so we can perform the following DELETE on it:

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

This DELETE maps to:

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

As I've indicated, however, few products provide support for updating views much more sophisticated than this one. *Note:* I'll have more to say regarding view updatability in SQL in the next subsection but one ("More on SQL").

The CHECK Option

Consider the following SQL INSERT on view LS:

```
INSERT INTO LS ( SNO , SNAME , STATUS , CITY )
       VALUES ( 'S6' , 'Lopez' , 30 , 'Madrid' ) ;
```

This INSERT maps to:

```
INSERT INTO S ( SNO , SNAME , STATUS , CITY )
       VALUES ( 'S6' , 'Lopez' , 30 , 'Madrid' ) ;
```

(The change is just in the target table name.) Observe now that "the new row" violates the constraint for view LS, because the city isn't London. So what happens? By default, SQL *will* insert that row into base table S; precisely because it doesn't satisfy the view defining expression for the view (i.e., view LS), however, it won't be visible through that view. From the perspective of view LS, in other words, the new row just drops out of sight (alternatively, we might say from that perspective that the INSERT is a no op). Actually, however, what's happened from the perspective of view LS is that *The Assignment Principle* has been violated.

Now, I hope it goes without saying that the foregoing behavior is logically incorrect. It isn't tolerated in **Tutorial D**. As for SQL, the CHECK option is provided to address the problem: If (but only if) WITH CASCADED CHECK OPTION is specified for a given view, then updates to that view are required to conform to the view defining expression for that view. **Recommendation:** Specify WITH CASCADED CHECK OPTION on view definitions whenever possible. Be aware, that SQL permits such a specification only if it regards the view as updatable, and (as previously noted) not all logically updatable views are regarded as such in SQL. *Note:* The alternative to CASCADED is LOCAL, but don't use it.* It's all right to specify neither CASCADED nor LOCAL, however, because CASCADED is the default.

More on SQL

As we've seen, SQL's support for view updating is limited. It's also extremely hard to understand!—in fact, the standard is even more impenetrable in this area than it usually is. The following excerpt (edited just slightly here) gives some idea of the complexities involved:

> [The] <query expression> *QE1* is *updatable* if and only if for every <query expression> or <query specification> *QE2* that is simply contained in *QE1:*
>
> a) *QE1* contains *QE2* without an intervening <non join query expression> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.
>
> b) If *QE1* simply contains a <non join query expression> *NJQE* that specifies UNION ALL, the:
>
> i) *NJQE* immediately contains a <query expression> *LO* and a <query term> *RO* such that no leaf generally underlying table of *LO* is also a leaf generally underlying table of *RO*.
>
> ii) For every column of *NJQE*, the underlying columns in the tables identified by *LO* and *RO*, respectively, are either both updatable or not updatable.
>
> c) *QE1* contains *QE2* without an intervening <non join query term> that specifies INTERSECT.
>
> d) *QE2* is updatable.

Here's my own gloss on the foregoing extract…I note that:

- The foregoing is just one of the many rules that have to be taken in combination in order to determine whether a given view is updatable in SQL.
- The rules in question aren't given all in one place but are scattered over many different parts of the document.
- All of those rules rely on a variety of additional concepts and constructs—e.g., updatable columns, leaf generally underlying tables, <non join query term>s—that are in turn defined in still further parts of the document.

* The semantics of WITH LOCAL CHECK OPTION are far too baroque to be spelled out in detail here. In any case, it's hard to see why anyone would ever want such semantics; indeed, it's hard to resist the suspicion that this alternative was included in the standard for no other reason than to allow certain flawed implementations, extant at the time, to be able to claim conformance.

Because of such considerations, I won't even attempt a precise characterization here of just which views SQL regards as updatable. Loosely speaking, however, they do at least include the following:

1. Views defined as a restriction and/or projection of a single base table
2. Views defined as a one to one or one to many join of two base tables (in the one to many case, only the many side is updatable)
3. Views defined as a UNION ALL or INTERSECT of two distinct base tables
4. Certain combinations of Cases 1–3 above

But even these limited cases are treated incorrectly, thanks to SQL's lack of understanding of constraint inference, **The Golden Rule**, and *The Assignment Principle*, and thanks also to the fact that SQL permits nulls and duplicate rows. And the picture is complicated still further by the fact that SQL identifies four distinct cases; to spell the matter out, a given view in SQL can be *updatable*, *potentially updatable*, *simply updatable*, or *insertable into*. Now, the standard defines these terms formally but gives no insight into their intuitive meaning or why they were given those names. However, I can at least say that "updatable" refers to UPDATE and DELETE and "insertable into" refers to INSERT, and a view can't be insertable into unless it's updatable. But note the suggestion that some views might permit some updates but not others (e.g., DELETEs but not INSERTs), and the further suggestion that it might therefore be possible that DELETE and INSERT might not be inverses of each other. Both of these facts, if facts they are, represent further violations of *The Principle of Interchangeability*.

Regarding Case 1 above, however, I can be a little more precise. To be specific, an SQL view is certainly updatable if all of the following conditions are satisfied:

- The view defining expression is either (a) a simple SELECT expression (no UNIONs, INTERSECTs, or EXCEPTs) or (b) an "explicit table" (see Chapter 12) that's logically equivalent to such an expression. *Note:* I'll assume for simplicity in what follows that Case (b) is automatically converted to Case (a).
- The SELECT clause in that SELECT expression specifies ALL (implicitly or explicitly).
- After expansion of any "asterisk style" items, every item in the SELECT item commalist is a simple column name (possibly qualified, and possibly with an AS specification), and no such item appears more than once.
- The FROM clause in that SELECT expression contains exactly one table reference (again, see Chapter 12), and the table expression in that table reference is just the name, *T* say, of an updatable table (either a base table or an updatable view).
- The WHERE clause, if any, in that SELECT expression contains no subquery in which the FROM clause references *T*.
- The SELECT expression has no GROUP BY or HAVING clause.

Recommendation: Lobby the SQL vendors to improve their support for view updating as soon as possible.

What Are Views For?

So far in this chapter, I've been tacitly assuming that you already know what views are for—but now I'd like to say something about that topic nonetheless. In fact, views serve two rather different purposes:

- The user who actually defines view *V* is, obviously, aware of the corresponding view defining expression *X*. So that user can use the name *V* wherever the expression *X* is intended, but such uses are basically just shorthand, and are explicitly understood to be just shorthand by the user in question.
- 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 I've already explained at length. 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 chapter prior to this point.

Logical Data Independence

The second of the foregoing purposes is intimately related to the question of *logical data independence*. Recall from Chapter 1 that physical data independence means we can 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. Reasonably enough, therefore, logical data independence means we can change the way the data is logically stored and accessed without having to make corresponding changes in the way the data is perceived by the user. And it's views that are supposed to provide that logical data independence.

By way of example, suppose that for some reason—the precise reason is not important for present purposes—we wish to replace base relvar S by base relvars LS and NLS, as follows:

```
VAR LS BASE RELATION       /* London suppliers */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR NLS BASE RELATION      /* non London suppliers */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;
```

As we know from discussions earlier in this chapter, the old relvar S is the disjoint union of the two new relvars LS and NLS (and LS and NLS are both restrictions of that old relvar S). So we can define a view that's exactly that union, and name it S:

```
VAR S VIRTUAL ( LS D_UNION NLS ) KEY { SNO } ;
```

Any expression that previously referred to base relvar S will now refer to view S instead. Assuming the system supports operations on views correctly—unfortunately a rather large assumption, given the state of today's products—users will be immune to this particular change in the logical structure of the database.

I note in passing that replacing the original suppliers relvar S by its two restrictions LS and NLS isn't a totally trivial matter. In particular, something might have to be done about the shipments relvar SP, since that relvar has a foreign key that references the original suppliers relvar. See Exercise 9-11 at the end of the chapter.

Views and Snapshots

Throughout this chapter, I've been using the term *view* in its original sense—the sense, that is, in which (in the relational context, at least) it was originally defined. Unfortunately, however, some terminological confusion has arisen in recent years: certainly in the academic world, and to some extent in the commercial world also. Recall 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 on the fly):

```
VAR LSS SNAPSHOT ( S WHERE CITY = 'London' )
    KEY { SNO }
    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 relation constant or "read-only relvar" (read-only, that is, apart from the periodic refresh—see the bullet item immediately following).
- Periodically (EVERY DAY in the example) the snapshot is refreshed, meaning 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 applications can often tolerate—in some cases 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, snapshots have come to be known (at least in some circles) not as snapshots at all but as *materialized views*. But they aren't views! Views aren't supposed to be materialized at all;* as we've seen, operations on views are supposed to be implemented by mapping them into suitable operations on the underlying relvars. Thus,

* Despite the fact that, as we saw earlier, there's at least one product on the market that does materialize them at least some of the time.

"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 I do use the term *view* in its original sense; but be warned that it doesn't always have that meaning elsewhere. **Recommendations:** Never use the term *view*, unqualified, to mean a snapshot; never use the term *materialized view*; and watch out for violations of these recommendations on the part of others!

Exercises

Exercise 9-1. Define a view consisting of supplier numbers and part numbers for suppliers and parts that aren't colocated. Give both **Tutorial D** and SQL definitions.

Exercise 9-2. 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 NATURAL JOIN SP
       WHERE  S.CITY = 'London' ) ;
```

Here's a query on this view:

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

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

Exercise 9-3. What key(s) does view LSSP from the previous exercise have? What's the predicate for that view?

Exercise 9-4. Given the following **Tutorial D** view definition—

```
VAR HEAVYWEIGHT VIRTUAL
  ( ( P RENAME ( WEIGHT AS WT , COLOR AS COL ) )
      WHERE WT > 14.0 ) { PNO , WT , COL } ;
```

—show the converted form after the substitution procedure has been applied for each of the following expressions and statements:

a. `HEAVYWEIGHT WHERE COL = 'Green'`

b. `( EXTEND HEAVYWEIGHT ADD ( WT + 5.3 AS WTP ) ) { PNO , WTP }`

c. `INSERT HEAVYWEIGHT RELATION { TUPLE { PNO 'P99' , WT 12.0 , COL 'Purple' } } ;`

d. `DELETE HEAVYWEIGHT WHERE WT < 9.0 ;`

e. `UPDATE HEAVYWEIGHT WHERE WT = 18.0 : { COL := 'White' } ;`

Exercise 9-5. Suppose the HEAVYWEIGHT view definition from Exercise 9-4 is revised as follows:

```
VAR HEAVYWEIGHT VIRTUAL
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) )
    RENAME ( COLOR AS COL ) ) WHERE WT > 6356.0 )
                                { PNO , WT , COL } ;
```

(i.e., attribute WT now denotes weights in grams rather than pounds). Now repeat Exercise 9-4.

Exercise 9-6. Give SQL solutions to Exercises 9-4 and 9-5.

Exercise 9-7. Which of the SQL views from previous exercises should have WITH CHECK OPTION specified?

Exercise 9-8. What keys do the views in Exercises 9-4 and 9-5 have? What are the relvar predicates? What are the total relvar constraints?

Exercise 9-9. Give as many reasons as you can think of for wanting to be able to declare keys for a view.

Exercise 9-10. Using either the suppliers-and-parts database or any database you happen to be familiar with, give some further examples (over and above the London vs. non London suppliers example, that is) to illustrate the point that which relvars are base and which virtual is largely arbitrary.

Exercise 9-11. In the body of the chapter, in the discussion of logical data independence, I discussed the possibility of restructuring—i.e., changing the logical structure of—the suppliers-and-parts database by replacing base relvar S by two of its restrictions (LS and NLS). However, I also observed that such a replacement wasn't a completely trivial matter. Why not?

Exercise 9-12. Investigate any SQL product available to you:

a. 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?

b. What updates on what views does that product support? Be as precise as you can in your answer. Are the view updating rules in that product identical to those in the SQL standard?

c. More generally, in what ways—there will be some!—does that product violate *The Principle of Interchangeability*?

Exercise 9-13. Distinguish between views and snapshots. Does SQL support snapshots? Does any product that you're aware of?

Exercise 9-14. What's a "materialized view"? Why is the term deprecated?

Exercise 9-15. Consider the suppliers-and-parts database, but ignore the parts relvar for simplicity. Here in outline are two possible designs for suppliers and shipments:

a. `S { SNO , SNAME , STATUS , CITY }`

   `SP { SNO , PNO , QTY }`

b. `SSP { SNO , SNAME , STATUS , CITY , PNO , QTY }`

   `XSS { SNO , SNAME , STATUS , CITY }`

Design a. is as usual. In Design b., by contrast, relvar SSP contains a tuple for every shipment, giving the applicable part number and quantity and full supplier details, and relvar XSS contains supplier details for suppliers who supply no parts at all. (Are these designs information equivalent?) Write view definitions to express Design b. as views over Design a. and vice versa. Also, show the applicable constraints for each design. Does either design have any obvious advantages over the other? If so, what are they?

Exercise 9-16. Views are supposed to provide logical data independence. But didn't I say in Chapter 6 that a hypothetical mechanism called "public tables" was supposed to perform that task? How do you account for the discrepancy?

CHAPTER TEN

SQL and Logic

As I mentioned in Chapter 1, there's an alternative to the relational algebra called the relational calculus. What this means is that queries, constraints, view definitions, and so forth can all be formulated in calculus terms as well as algebraic ones; sometimes, in fact, it's easier to come up with a calculus formulation, though the opposite can also be true.

What is the relational calculus? Essentially, it's an applied form of predicate calculus (also known as predicate logic), tailored to the needs of relational databases. So the aims of this chapter are to introduce the relevant features of predicate logic (hereinafter abbreviated to just *logic*); to show how those features are realized in concrete form in the relational calculus; and, finally, to consider the corresponding features of SQL as we go.

Incidentally, it follows from the above that a relational language can be based on either the algebra or the calculus. For example, **Tutorial D** is explicitly based on the algebra (which is why there aren't many references to **Tutorial D** in this chapter), and Query-By-Example (see Appendix D) is explicitly based on 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 on, 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—with the further implication that most queries, constraints, and so on that can be expressed in SQL at all can in fact be expressed in many different ways, as I mentioned in passing in Chapter 6.

Simple and Compound Propositions

Recall from Chapter 5 that, in logic, a proposition is a statement that evaluates unconditionally to either TRUE or FALSE. Here are some examples (of which Nos. 1, 4, and 5 are true and Nos. 2 and 3 false):

1. `2 + 3 = 5`
2. `2 + 3 > 7`
3. `Jupiter is a star`
4. `Mars has two moons`
5. `Venus is between Earth and Mercury`

Connectives

Given some set of propositions, we can combine propositions from that set to form further propositions, using various *connectives*. The connectives most commonly encountered in practice are NOT, AND, OR, IF ... THEN ... (also known as IMPLIES or "⇒"), and IF AND ONLY IF (also known as IFF, or BI-IMPLIES, or IS EQUIVALENT TO, or "⇔", or "≡"). Here are a few examples of propositions that can be formed from Nos. 3, 4, and 5 from the foregoing list:

6. `( Jupiter is a star ) OR ( Mars has two moons )`
7. `( Jupiter is a star ) AND ( Jupiter is a star )`
8. `( Venus is between Earth and Mercury ) AND NOT ( Jupiter is a star )`
9. `IF ( Mars has two moons ) THEN ( Venus is between Earth and Mercury )`
10. `IF ( Jupiter is a star ) THEN ( Mars has two moons )`

* That goal was based on what I regard as a fundamental misconception: namely, that the relational algebra and calculus could both be characterized as being somewhat "user hostile." But that characterization was, I believe, founded on a confusion over syntax vs. semantics. Certainly the syntax in Codd's early papers was a little daunting, based as it was on formal mathematical notation. But semantics is another matter; the algebra and the calculus both have (I would argue) very simple semantics, and it's fairly easy, as many writers have demonstrated, to wrap that semantics in syntax that's very user friendly indeed.

NOTE

I've used parentheses to make the scope of the connectives clear in these examples; 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, and sometimes they can improve clarity.

In general, the connectives can be regarded as *logical operators*—they take propositions as their input and return another proposition as their output. NOT is a monadic operator, the other four are dyadic. A proposition that involves no connectives is called a *simple* proposition; a proposition that isn't simple is called *compound*. And the truth value of a compound proposition can be determined from the truth values of the constituent simple propositions in accordance with the following truth tables (in which, for space reasons, I've abbreviated TRUE and FALSE to just T and F, respectively):

p	NOT *p*
T	F
F	T

p	*q*	*p* AND *q*	*p* OR *q*	IF *p* THEN *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

By the way, truth tables can also be drawn in the following slightly different style (and here I've abbreviated IF ... THEN ... to just IF, again for space reasons):

NOT	
T	F
F	T

AND	T	F
T	T	F
F	F	F

OR	T	F
T	T	T
F	T	F

IF	T	F
T	T	F
F	T	T

IFF	T	F
T	T	F
F	F	T

Sometimes one style is more convenient, sometimes the other is. Anyway, let's take a closer look at one of the sample compound propositions:

9. `IF ( Mars has two moons ) THEN ( Venus is between Earth and Mercury )`

This proposition is of the form IF *p* THEN *q* (equivalently, *p* IMPLIES *q*), where *p* is the *antecedent* and *q* is the *consequent*. Since the antecedent and the consequent both evaluate to TRUE, the overall proposition evaluates to TRUE also, as you can see from the truth table. But whether Venus is between Earth and Mercury obviously has nothing to do with whether Mars has two moons! So what exactly is going on here?

The foregoing example highlights a problem that people with no training in formal logic often experience—namely, that implication is notoriously difficult to come to grips with—

so I'd like to offer the following argument (or rationale) in an attempt to clarify the matter:

- First of all, note that there are exactly 16 dyadic connectives altogether, corresponding to the 16 possible dyadic truth tables (just four of which are shown above).
- Of those 16 dyadic connectives, some but not all are given common names such as AND and OR. But those names are really nothing more than a mnemonic device; they don't have any intrinsic meaning, they're chosen simply because the connectives so named have behavior that's similar (not necessarily identical) to that of their natural language counterparts. Indeed, it's easy to see that even AND doesn't mean quite the same thing as "and" in natural language. In logic, *p* AND *q* and *q* AND *p* are equivalent—but their natural language counterparts might not be. Here's an illustration: The natural language statements

  "I was seriously disappointed and I voted for a change in leadership"

  and

  "I voted for a change in leadership and I was seriously disappointed"

  are most certainly not equivalent! In other words, AND is a kind of logical distillate of "and" in natural language; very importantly—and unlike "and" in natural language—its meaning is *context independent*. And similar remarks apply to all of the other connectives, too.
- Of the 16 available dyadic connectives, the one called IMPLIES has behavior that most closely resembles that of implication as usually understood in natural language. For example, "if Mars has two moons, then it certainly has at least one moon" is a valid implication in both logic and natural language. But nobody would or should claim that logical implication and natural language implication are the same thing. In fact, logical implication, like all of the connectives, is (of necessity) *formally defined*—i.e., it's defined purely in terms of the truth values, not the meanings, of its operands—whereas the same obviously can't be said of its natural language counterpart.
- Let's look at another example (number 10 from the foregoing list):

  ```
  IF ( Jupiter is a star ) THEN ( Mars has two moons )
  ```

  Perhaps even more counterintuitively, this one evaluates to TRUE also, because the antecedent is false (check the truth table); yet whether Mars has two moons, again obviously, has nothing to do with whether Jupiter is a star. Again, part of the justification—for the fact that the implication evaluates to TRUE, that is—is just that IMPLIES is formally defined. In this case, however, there's another argument (a database example, in fact) that you might find a little more satisfying. Suppose the suppliers-and-parts database is subject to the constraint that red parts must be stored in London (I deliberately state that constraint here in somewhat simplified form):

  ```
  IF ( COLOR = 'Red' ) THEN ( CITY = 'London' )
  ```

Clearly we don't want this constraint to be violated by a part that isn't red. It follows, therefore, that we want the proposition overall (a logical implication) to evaluate to TRUE if the antecedent evaluates to FALSE.

It follows from all of the above that the proposition *p* IMPLIES *q* (equivalently, 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. And this equivalence illustrates the point that the connectives NOT, AND, OR, IMPLIES, and IF AND ONLY IF aren't all primitive; some of them can be expressed in terms of others. As a matter of fact, all possible monadic and dyadic connectives 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?

A Remark on Commutativity

The connectives AND and OR are commutative; that is, the compound propositions *p* AND *q* and *q* AND *p* are logically equivalent, and so are the compound propositions *p* OR *q* and *q* OR *p*. As a consequence, you should never write code involving such propositions that assumes that *p* will be evaluated before *q* or the other way around. For example, let the function SQRT ("nonnegative square root") be defined in such a way that an exception is raised if its argument is less than zero, and consider the following SQL expression:

```
SELECT ...
FROM   ...
WHERE  X >= 0 AND SQRT ( X ) ...
```

This expression isn't guaranteed to avoid raising the exception, because the SQRT function might be invoked before the test is done to ensure that X is nonnegative.

Simple and Compound Predicates

Consider the following statements:

11. `x is a star`

12. `x has two moons`

13. `x has m moons`

14. `x is between Earth and y`

15. `x is between y and z`

Here *x*, *y*, *z*, and *m* are *parameters*, also known as *placeholders*. As a consequence, the statements aren't propositions (i.e., they aren't unconditionally either true or false), precisely because they involve such parameters. For example, the statement "*x* is a star" involves the parameter *x*, and we can't say whether it's true or false unless and until 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, as the next paragraph makes clear.

Now, we can substitute *arguments* for the parameters and thereby obtain propositions from those parameterized 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 unconditionally 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 5—is a truth valued function; that is to say, 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," while the argument *the moon* does not.

As an aside, I remind you from Chapter 5 that logicians speak not of invoking a predicate but rather of *instantiating* it (in fact, for reasons that needn't concern us here, their concept of instantiation is slightly more general than that of the familiar notion of function invocation). However, I'll favor the terminology of invocation in this chapter. Also, Exercise 5-23 in Chapter 5 showed that a proposition can be regarded as a degenerate predicate; to be precise, it's a predicate for which the set of parameters is empty (and the truth valued function that is that predicate 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 predicate "*x* has *m* moons," which involves two parameters, *x* and *m*. Substituting the arguments *Mars* for *x* and *2* for *m* yields a true proposition; substituting the arguments *Earth* for *x* and *2* for *m* yields a false one. In fact, predicates can conveniently be classified according to the cardinality of their set of parameters. Thus we 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 *m* moons" is a 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 monadic; if $n = 2$, it's dyadic.

Given a set of predicates, we can combine predicates from that set to form further predicates using the logical connectives already discussed (NOT, AND, OR, and so forth); in other words, the 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 an example of a compound predicate:

17. `( x is a star ) OR ( x is between Earth and y )`

This predicate is 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 in the section "Relational Calculus" later.

The term used in logic for expressions like EXISTS *x* and FORALL *x* 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 (thus: ∃) and FORALL by an upside down A (thus: ∀). I use the keywords EXISTS and FORALL for readability.

ASIDE

At this point, one reviewer asked whether a quantifier is just another connective. No, it isn't. Let *p* and *q* be predicates, each with a single parameter *x*. Then *p* and *q* can be combined in various ways by means of connectives, but the result is always just another predicate with that same single parameter *x*. By contrast, quantifying over *x*—that is, forming an expression such as EXISTS *x* (*p*) or FORALL *x* (*q*)—has the effect of converting the predicate concerned into a proposition. Thus, there's a clear logical difference between the two concepts. (Though I should add that in the database context, at least, the quantifiers can be *defined in terms of* certain connectives. I'll explain this point later, in the section "More on Quantification.")

By way of another example, consider the dyadic predicate "*x* is taller than *y*." If we quantify existentially over *x*, we obtain:

```
EXISTS x ( x is taller than y )
```

This statement isn't a proposition, because it isn't unconditionally 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). But another way to obtain a proposition from the original predicate is to quantify over *both* parameters. 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 alone, it's necessary to quantify over *every* parameter. More generally, if we quantify over *m* parameters ($m \leq n$), we obtain a *k*-adic predicate, where $k = n - m$.
- 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 say 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. By contrast, with unlike quantifiers, the sequence matters (see the point immediately following).

- When we "quantify over everything," each individual quantifier can be either existential or universal. In the 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 them can be ignored 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 most of those interpretations that there do exist at least two people "in the universe," as it were. I'll come back to this assumption 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. *Note:* Actually I'm cheating a little bit here, because I haven't said what I mean by "shorter"! But I could have done—i.e., I could have stated explicitly, somehow, that the predicates "*x* is taller than *y*" and "*y* is shorter than *x*" are logically equivalent—and I'll assume for the rest of this section that I've done so.

  ```
  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.

Last (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

What I've been calling parameters are more usually known in logic as *free variables*—and quantifying over a free variable converts it into what's called a *bound* variable. For example, consider again the 2-place predicate from the previous section:

```
x is taller than y
```

Here *x* and *y* are free variables. If we now quantify existentially over *x*, we obtain:

```
EXISTS x ( x is taller than y )
```

Now *y* is free (still) but *x* is 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're just 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 later.) *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 mean 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 can't 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
```

As this example demonstrates, the terminology of free vs. bound "variables" doesn't really refer to variables per se, but rather to variable *occurrences*—occurrences of references to variables within some predicate, to be precise. In the second of the foregoing predicates, for example, it's the second *occurrence* of the *reference* to *y* that's bound, and the third such occurrence that's free. Despite this state of affairs, it's usual (perhaps regrettably) to talk about free and bound variables as such,* even though such talk is slightly sloppy. Be on your guard for confusion in this area!

* Even, sometimes, in logic textbooks, where the practice really ought to be deprecated.

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: equivalently, one that involves no free variables.

Relational Calculus

Everything I've described in this chapter so far maps very directly into the relational calculus. Let's look at a simple example—a relational calculus representation of the query "Get supplier number and status for suppliers in Paris who supply part P2." Here first for comparison purposes is an algebraic formulation:

```
( S WHERE CITY = 'Paris' ) { SNO , STATUS }
                           MATCHING ( SP WHERE PNO = 'P2' )
```

And here's a relational calculus equivalent:

```
RANGEVAR SX  RANGES OVER S  ;
RANGEVAR SPX RANGES OVER SP ;

{ SX.SNO , SX.STATUS }
           WHERE SX.CITY = 'Paris' AND
                 EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

Explanation:

- The first two lines are definitions, defining SX and SPX to be *range variables* that range over S and SP, respectively. What those definitions mean is that, at any given time, permitted values of SX are tuples in the relation that's the value of relvar S at that time; likewise, permitted values of SPX are tuples in the relation that's the value of relvar SP at that time.
- The remaining lines are the actual query. Observe that they take the following generic form:

  ```
  proto tuple WHERE predicate
  ```

  This expression overall is the relational calculus version of a relational expression (i.e., an expression that denotes a relation), and it evaluates to a relation containing every possible value of the proto tuple for which the predicate evaluates to TRUE, and no other tuples. (The term *proto tuple*, standing for "prototype tuple," is apt but nonstandard; in fact, a standard term for the concept doesn't seem to exist.) In the example, therefore, the result is a relation of degree two, containing every (SNO,STATUS) pair from relvar S such that (a) the corresponding city is Paris and (b) there exists a shipment in relvar SP with the same supplier number as in that pair and with part number P2.

Note the use of dot qualified names in this example (in both the proto tuple and the predicate); I won't go into details, because dot qualified names will be familiar to you from SQL. Indeed, SQL has a formulation of the query under discussion that's very similar in general terms to the foregoing relational calculus formulation:

```
SELECT SX.SNO , SX.STATUS
FROM   S AS SX
WHERE  SX.CITY = 'Paris'
AND    EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  SPX.SNO = SX.SNO
       AND    SPX.PNO = 'P2' )
```

Note that SQL does support range variables, though it doesn't usually use that term (I'll have more to say about range variables in SQL in Chapter 12). More important, note that it also supports EXISTS; however, that support is somewhat indirect.* To be specific, let *sq* be a subquery; then EXISTS *sq* is a boolean expression (and so represents a predicate), and it evaluates to FALSE if the table denoted by *sq* is empty and TRUE otherwise. (The table expression *tx* in parentheses that constitutes *sq* will usually, though not invariably, be of the form SELECT * FROM ... WHERE ..., and the WHERE clause will usually, though not invariably, include some reference to some "outer" table, meaning *sq* will be a *correlated* subquery specifically. In the foregoing example, S is that outer table, and it's referenced by means of the range variable SX. See Chapter 12 for further explanation.)

ASIDE

There's a certain irony here, though. As we saw in Chapter 4, SQL, because it supports nulls, is based on what's called *three-valued logic*, 3VL (instead of the conventional two-valued logic I'm discussing in this chapter, which is what the relational model is based on). In 3VL, the existential quantifier can return three different results: TRUE, FALSE, and UNKNOWN (again, see Chapter 4). But SQL's EXISTS operator always returns TRUE or FALSE, never UNKNOWN. For example, EXISTS(*tx*) will return TRUE, not UNKNOWN, if *tx* evaluates to a table containing nothing but nulls (I'm speaking a trifle loosely here); yet UNKNOWN is the logically correct result. As a consequence, (a) SQL's EXISTS isn't a faithful implementation of the existential quantifier of 3VL, and (b) once again, therefore, SQL queries sometimes return the wrong answer. I'll give a concrete example of such a situation in the next chapter.

Let's look at another example—the query "Get supplier names for suppliers who supply all parts." I'll assume the same range variables as before, plus another one for a range variable PX:

```
RANGEVAR PX RANGES OVER P ;

{ SX.SNAME } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

* It might help to point out that SQL's EXISTS is rather similar to **Tutorial D**'s IS_NOT_EMPTY (see Chapter 3). See the section "Some Equivalences," later.

In somewhat stilted natural language: "Get names of suppliers such that, for all parts, there exists a shipment with the same supplier number as the supplier and the same part number as the part." *Note:* As you probably know, SQL has no direct support for FORALL at all. For that reason, I won't show an SQL analog of this example here—I'll come back to it later, in the section "More on Quantification" I will point out, however, that there's a logical difference between the foregoing calculus expression and this one, where the quantifiers have been switched:

```
{ SX.SNAME } WHERE EXISTS SPX ( FORALL PX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Exercise: What does this latter expression mean? And do you think the query is a "sensible" one?

One more example ("Get supplier names for suppliers who supply all red parts"):

```
{ SX.SNAME } WHERE FORALL PX ( IF PX.COLOR = 'Red' THEN
                               EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Note the use of logical implication in this example. Note too that I'm assuming the same range variables as before; in fact, I'll continue to make that same assumption for the rest of this chapter.

By the way, here's another possible formulation of the foregoing query:

```
{ SX.SNAME } WHERE FORALL PX
                 ( EXISTS SPX ( IF PX.COLOR = 'Red' THEN
                                   SPX.SNO = SX.SNO AND
                                   SPX.PNO = PX.PNO ) )
```

In this latter formulation, the predicate in the WHERE clause is in what's called "prenex normal form," meaning, loosely, that the quantifiers all appear at the beginning. Here's a precise definition:

> **Definition:** A predicate is in *prenex normal form* (PNF) if and only if (a) it's quantifier free (i.e., it contains no quantifiers at all) or (b) it's of the form EXISTS *x* (*p*) or FORALL *x* (*p*), where *p* is in PNF in turn. In other words, a PNF predicate takes the form
>
> *Q1 x1 (Q2 x2 (... (Qn xn (q)) ...))*
>
> where $n \geq 0$, each *Qi* ($i = 1, 2, ..., n$) is either EXISTS or FORALL, and the predicate *q*—which is sometimes called the *matrix*—is quantifier free.

Prenex normal form isn't more or less correct than any other form, but with a little practice it does tend to become the most natural formulation in many cases.

More on Range Variables

From what's been said in this section so far, it should be clear that range variables in the relational calculus serve as the free and bound variables that are required by formal logic.

As I mentioned earlier, those variables always have to range over some set of permissible values; in the relational calculus context specifically, that set is always the body of some relation (usually but not necessarily the relation that's the current value of some relvar). *Note:* It follows that a given range variable always denotes some tuple. For that reason, the relational calculus is sometimes known more specifically as the *tuple* calculus, and the variables themselves as tuple variables. This latter usage can be confusing, however, since the term *tuple variable* already has a somewhat different meaning, and I won't adopt it in this book.

Now I can say a little more about the syntax of relational calculus expressions:

- First of all, a proto tuple is a commalist of items enclosed in braces, in which each item is either a *range attribute reference*—possibly with an associated AS clause to introduce a new attribute name—or a *range variable reference*. (There are other possibilities too, but I'll limit my attention to just these cases until further notice. See Example 5 below.) *Note:* It's usual to omit the braces if the commalist contains just a single item, but I'll generally show them even when they're not actually required, for clarity.
- A range attribute reference is an expression of the form *R.A*, where *A* is an attribute of the relation that range variable *R* ranges over; SX.SNO is an example. And a range variable reference is just a range variable name, like SX, and it's shorthand for a commalist of range attribute references, one for each attribute of the relation the range variable ranges over.
- Let some range attribute reference involving range variable *R* appear, explicitly or implicitly, within some proto tuple. Then the predicate in the corresponding WHERE clause can, and usually will, contain at least one free range attribute reference involving *R*—where by "free range attribute reference involving *R*" I mean a range attribute reference of the form *R.A* that's not within the scope of any quantifier in which *R* is the bound variable.
- The WHERE clause is optional; omitting it is equivalent to specifying WHERE TRUE.

More Sample Queries

I'll give a few more examples of relational calculus queries, in order to illustrate a few more points; however, I'm not trying to be exhaustive in my treatment. For simplicity, I'll omit the RANGEVAR definitions that would be needed in practice and will just assume that SX, SY, SZ, etc., have been defined as range variables over S; PX, PY, PZ, etc., have been defined as range variables over P; and SPX, SPY, SPZ, etc., have been defined as range variables over SP. Please note that the formulations shown aren't the only ones possible, in general. I'll leave it as an exercise for you to show equivalent SQL formulations in each case.

Example 1: Get all pairs of supplier numbers such that the suppliers concerned are colocated.

```
{ SX.SNO AS SA , SY.SNO AS SB } WHERE SX.CITY = SY.CITY
                                  AND SX.SNO < SY.SNO
```

Note the use of AS clauses in this example.

Example 2: Get supplier names for suppliers who supply at least one red part.

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS PX ( SX.SNO = SPX.SNO AND
                                            SPX.PNO = PX.PNO AND
                                            PX.COLOR = 'Red' ) )
```

Example 3: Get supplier names for suppliers who supply at least one part supplied by supplier S2.

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS SPY ( SX.SNO = SPX.SNO AND
                                             SPX.PNO = SPY.PNO AND
                                             SPY.SNO = 'S2' ) )
```

Example 4: Get supplier names for suppliers who don't supply part P2.

```
{ SX.SNAME } WHERE NOT EXISTS SPX ( SPX.SNO = SX.SNO AND
                                    SPX.PNO = 'P2' )
```

In practice the argument expression following NOT might need to be enclosed in parentheses.

Example 5: For each shipment, get full shipment details, including total shipment weight.

```
{ SPX , PX.WEIGHT * SPX.QTY AS SHIPWT } WHERE PX.PNO = SPX.PNO
```

Note the use of a computational expression in the proto tuple here. An algebraic version of this example would involve EXTEND.

Example 6: For each part, get the part number and the total shipment quantity.

```
{ PX.PNO , SUM ( SPX WHERE SPX.PNO = PX.PNO , QTY ) AS TOTQ }
```

This example illustrates the use of an aggregate operator invocation within the proto tuple (it's also the first example to omit the WHERE clause). Incidentally, note that the following expression, though syntactically legal, would not be a correct formulation of the query (why not?):

```
{ PX.PNO , SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO ) AS TOTQ }
```

Answer: Because duplicate quantities would be eliminated before the sum is computed.

Example 7: Get part cities that store more than five red parts.

```
{ PX.CITY } WHERE
     COUNT ( PY WHERE PY.CITY = PX.CITY AND PY.COLOR = 'Red' ) > 5
```

Sample Constraints

Now I'd like to give some examples of the use of relational calculus to formulate constraints. The examples are based on, and use the same numbering as, the ones in Chapter 8. I'll assume the availability of range variables as in the previous subsection. Please note again 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 CX1 FORALL SX ( SX.STATUS > 0 AND SX.STATUS < 101 ) ;
```

NOTE

SQL allows a constraint like this one to be simplified by (in effect) eliding both the explicit use of a range variable and, more important, the explicit universal quantification. To be specific, we can specify a *base table constraint*—see Chapter 8—as part of the definition of base table S that looks like this:

```
CONSTRAINT CX1 CHECK ( STATUS > 0 AND STATUS < 101 )
```

Similar remarks apply to subsequent examples also.

Example 2: Suppliers in London must have status 20.

```
CONSTRAINT CX2 FORALL SX ( IF SX.CITY = 'London'
                           THEN SX.STATUS = 20 ) ;
```

Example 3: No two tuples in relvar S have the same supplier number.

```
CONSTRAINT CX3 FORALL SX ( FORALL SY ( IF SX.SNO = SY.SNO THEN
                                          SX.SNAME = SY.SNAME AND
                                          SX.STATUS = SY.STATUS AND
                                          SX.CITY = SY.CITY ) ) ;
```

Example 4: Whenever two tuples in relvar S have the same supplier number, they also have the same city.

```
CONSTRAINT CX4 FORALL SX ( FORALL SY ( IF SX.SNO = SY.SNO
                                       THEN SX.CITY = SY.CITY ) ) ;
```

Example 5: No supplier with status less than 20 can supply part P6.

```
CONSTRAINT CX5 FORALL SX ( IF SX.STATUS < 20 THEN
                           NOT EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = 'P6' ) ) ;
```

Example 6: Every supplier number in relvar SP must appear in relvar S.

```
CONSTRAINT CX6 FORALL SPX ( EXISTS SX ( SX.SNO = SPX.SNO ) ) ;
```

I'll have more to say on this particular example in the next section.

Example 7: No supplier number appears in both relvar LS and relvar NLS.

```
CONSTRAINT CX7 FORALL LX ( FORALL NX ( LX.SNO ≠ NX.SNO ) ) ;
```

LX and NX range over LS and NLS, respectively.

Example 8: Supplier S1 and part P1 must never be in different cities.

```
CONSTRAINT CX8 NOT EXISTS SX ( EXISTS PX ( SX.SNO = 'S1' AND
                                           PX.PNO = 'P1' AND
                                           SX.CITY ≠ PX.CITY ) ) ;
```

Example 9: There must always be at least one supplier. (There's no counterpart to this example in Chapter 8.)

```
   CONSTRAINT CX9 EXISTS SX ( TRUE ) ;
```

The expression EXISTS SX (TRUE) evaluates to FALSE if and only if SX ranges over an empty relation.

More on Quantification

There are a number of further issues I need to discuss regarding quantification in particular.

We Don't Need Both Quantifiers

In practice, we don't actually need both EXISTS and FORALL, because any predicate 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 the following predicate once again:

```
   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 predicate

```
   EXISTS x ( p ( x ) )
```

is logically equivalent to the predicate

```
   NOT ( FORALL x ( NOT ( p ( x ) ) )
```

(where the predicate *p* might legitimately involve other parameters in addition to *x*). Likewise, the predicate

```
   FORALL x ( p ( x ) )
```

is logically equivalent to the predicate

```
   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 doesn't need to support both EXISTS and FORALL explicitly. But it's very desirable to support them both in practice. The reason is that some problems are "more naturally" formulated in terms of EXISTS, while others are "more naturally" formulated in terms of FORALL instead. For example, SQL supports EXISTS but not FORALL, as we know; as a consequence, certain queries are quite awkward to formulate in SQL. Consider again the query "Get suppliers who supply all parts," which can be expressed in relational calculus quite simply as follows:

```
   { SX } WHERE FORALL PX ( EXISTS SPX
                          ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

In SQL, by contrast, the query has to look something like this:

```
SELECT SX.*
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

("Get suppliers SX such that there does not exist a part PX such that there does not exist a shipment SPX linking that supplier SX to that part PX"). 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 predicates

```
EXISTS x ( p ( x ) )
```

and

```
NOT ( FORALL x ( NOT ( p ( x ) ) ) )
```

are logically equivalent. As we know, the bound variable *x* in each of these predicates 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 fifty feet tall (or in the database context, more realistically, it might be the set of tuples in a relvar that's currently empty). Then:

- The expression EXISTS *x* (*p*(*x*)) evaluates to FALSE, because "there is no *x*"—i.e., there's no value available to be substituted for *x* in order to make the expression true. Note carefully that these remarks are valid *regardless of what p(x) happens to be*. For example, "There exists a person over fifty feet tall who works for IBM" evaluates to FALSE (unsurprisingly).
- It follows that the negation NOT EXISTS *x* (*p*(*x*)) evaluates to TRUE—again, regardless of what *p*(*x*) happens to be. For example, "There doesn't exist a person over fifty feet tall who works for IBM"—more idiomatically, "No person who works for IBM is over fifty feet tall"—evaluates to TRUE (again unsurprisingly).
- But NOT EXISTS *x* (*p*(*x*)) is equivalent to FORALL *x* (NOT (*p*(*x*)), and so this latter expression also evaluates to TRUE—once again, regardless of what *p*(*x*) happens to be.
- But if the predicate *p*(*x*) is arbitrary, then so is the predicate NOT (*p*(*x*)). And so 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 fifty feet tall *do* work for IBM" also evaluates to TRUE—because, to say it again, there aren't any persons over fifty 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 query discussed earlier—

```
{ SX } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                      SPX.PNO = PX.PNO ) )
```

("Get suppliers who supply all parts")—will return all suppliers if there aren't any parts.

Defining EXISTS and FORALL

As you might have realized, 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*, ..., *xn*}. Then

```
EXISTS x ( p ( x ) )
```

is a predicate, and it's defined to be equivalent to (and hence shorthand for) the predicate

```
p ( x1 ) OR p ( x2 ) OR ... OR p ( xn ) OR FALSE
```

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

```
( Mercury has a moon ) OR ( Venus has a moon ) OR
( Earth has a moon )   OR ( Mars has a moon )  OR FALSE
```

which evaluates to TRUE because, e.g., "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

```
p ( x1 ) AND p ( x2 ) AND ... AND p ( xn ) AND TRUE
```

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

```
( Mercury has a moon ) AND ( Venus has a moon ) AND
( Earth has a moon )   AND ( Mars has a moon )  AND TRUE
```

which evaluates to FALSE because, e.g., "Venus has a moon" is false.

As an aside, let me remark that, as the examples 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 are finite in turn). In pure 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. Much more to the point, the quantifiers allow us to formulate queries without having to know the precise content of the database at any given time (which wouldn't be the case if we always had to use the explicit iterated OR and AND equivalents).

Other Kinds of Quantifiers

While it's certainly 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). We can't assign a truth value to this example because it's a (monadic) predicate and 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. *Exercise:* Does this predicate—which is in fact a proposition—evaluate to TRUE?)

* To elaborate: Consider by way of example the proposition EXISTS *x* (*p*), where *p* is a predicate with just one parameter, *x*. If *x* ranges over an infinite set, then any attempt to formulate an "iterated OR" algorithm for evaluating the proposition must fail, since the evaluation might never terminate (it might never find the one value of *x* that satisfies *p*). Likewise, any attempt to formulate an "iterated AND" algorithm for FORALL *x* (*p*) must also fail, since again the evaluation might never terminate (it might never find the one value of *x* that fails to satisfy *p*).

As another exercise, what does the following predicate mean?

```
FORALL x ( UNIQUE y ( x has social security number y ) )
```

Now recall the following constraint: "Every supplier number in relvar SP must appear in relvar S." Here's the formulation I gave previously:

```
CONSTRAINT CX6 FORALL SPX ( EXISTS SX ( SX.SNO = SPX.SNO ) ) ;
```

However, I hope you can see that a more accurate formulation is:

```
CONSTRAINT CX6 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

In other words, for a given tuple in relvar SP, we want there to be not at least one (EXISTS), but exactly one (UNIQUE), corresponding tuple in relvar S. The previous formulation "works" because there's an additional constraint in effect: viz., that {SNO} is a key for relvar S. But the revised formulation is closer to what we really want to say.

Now, SQL does support UNIQUE (sort of), though its support is more indirect than its support for EXISTS. To be specific, let *sq* be a subquery; then UNIQUE *sq* is a boolean expression, and it evaluates to FALSE if the table denoted by *sq* contains any duplicate rows and TRUE otherwise. Thus, whereas the logic expression

```
UNIQUE x ( p ( x ) )
```

means "There exists *exactly* one argument value *a* corresponding to the parameter *x* such that *p*(*a*) evaluates to TRUE," the SQL (very approximate!) analog—

```
UNIQUE ( SELECT * FROM T WHERE p ( x ) )
```

—means "Given an argument value *a* corresponding to the parameter *x*, there exists *at most* one row in the pertinent table *T* such that *p*(*a*) evaluates to TRUE." In particular, if the UNIQUE argument expression is not just a subquery but in fact a row subquery (see Chapter 12), the UNIQUE invocation returns TRUE if the table denoted by that query has either just one row *or no rows at all*.

Here then is a somewhat contrived example of an SQL query that uses UNIQUE ("Get names of suppliers who supply at least two distinct parts in the same quantity"):

```
SELECT DISTINCT SNAME
FROM   S
WHERE  NOT UNIQUE ( SELECT QTY
                    FROM   SP
                    WHERE  SP.SNO = S.SNO )
```

To repeat, this example is rather contrived; to be specific, it's designed to exploit the fact that SQL retains duplicates in the result of a SELECT expression if DISTINCT isn't specified.* As you'll recall, however, I've suggested elsewhere in this book—in Chapter 4,

* It's also the first example in this chapter to make use of implicit range variables (though plenty of other examples did the same thing in earlier chapters). See Chapter 12 if you need a formal explanation of what's going on here.

to be specific—that DISTINCT should "always" be specified. So although the example was meant to be reasonably realistic, don't infer from it that the system has to permit duplicate rows in order to support such queries. Here's a formulation of the same query that doesn't use UNIQUE:

```
SELECT DISTINCT SNAME
FROM   S
WHERE  ( SELECT COUNT ( DISTINCT QTY )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) <
       ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO )
```

SQL also uses the keyword UNIQUE in key constraints. For example, the CREATE TABLE for table S includes the following specification:

```
UNIQUE ( SNO )
```

In fact, however, this specification is defined to be shorthand for the following (which could be part of a more general base table constraint or a CREATE ASSERTION statement):

```
CHECK ( UNIQUE ( SELECT SNO FROM S ) )
```

Observe that for once the SELECT expression in the foregoing CHECK constraint must definitely not specify DISTINCT! (Why not?)

SQL also uses the keyword UNIQUE in MATCH expressions. Here's an example ("Get suppliers who supply exactly one part"):

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO MATCH UNIQUE ( SELECT SNO FROM SP )
```

But this usage too is basically just shorthand. For example, the example just shown is equivalent to the following—

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  EXISTS ( SELECT *    /* there's at least one ... */
                FROM   SP
                WHERE  SP.SNO = S.SNO )
AND    UNIQUE ( SELECT *    /* ... and there aren't two */
                FROM   SP
                WHERE  SP.SNO = S.SNO )
```

—which in turn is equivalent to:

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) = 1
```

Some Equivalences

I'll finish up this chapter with a few remarks regarding certain equivalences that might have already occurred to you. First of all, recall the IS_EMPTY operator, which I introduced in Chapter 3 and made heavy use of in Chapter 8. If the system supports that operator, then 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 ) )
```

(I'm assuming here that the variable *x* ranges over the set *X*.)

In fact, SQL's support for EXISTS (and for FORALL, such as it is) is based on exactly the foregoing equivalences. The fact is, SQL's EXISTS isn't really a quantifier, as such, at all, because it doesn't involve any bound variables. Instead, it's an *operator*, in the conventional sense of that term: a monadic operator of type BOOLEAN, to be precise. Like any monadic operator invocation, an invocation of the SQL EXISTS operator is evaluated by, first, evaluating the expression that denotes its sole argument; second, applying the operator per se—in this case EXISTS—to the result of that evaluation. Thus, given the expression EXISTS (*tx*), where *tx* is a table expression, the system first evaluates *tx* to obtain a table *t*; then it applies EXISTS to *t*, returning TRUE if *t* is nonempty and FALSE otherwise. (At least, that's the conceptual algorithm; numerous optimizations are possible, but they're irrelevant to the present discussion.)

And now I can explain why SQL doesn't support FORALL. The reason is that representing the universal quantifier by means of an operator with syntax of the form FORALL(*tx*)—where *tx* is again a table expression—couldn't possibly make any sense. For example, consider the hypothetical expression FORALL (SELECT * FROM S WHERE CITY = 'Paris'). What could such an expression possibly mean? It certainly couldn't mean anything like "All suppliers are in Paris," because—loosely speaking—the argument to which the hypothetical operator is applied isn't all suppliers, it's all suppliers in Paris.

In fact, however, we don't need the quantifiers anyway if the system supports the aggregate operator COUNT, thanks to the following equivalences:

- `EXISTS x ( p )  ≡  COUNT ( X WHERE p ) > 0`
- `FORALL x ( p )  ≡  COUNT ( X WHERE p ) = COUNT ( X )`
- `UNIQUE x ( p )  ≡  COUNT ( X WHERE p ) = 1`

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.

ASIDE

Although this book generally has little to say on performance, I should at least point out that the foregoing equivalences could lead to performance problems. For example, consider the following expression, which is an SQL formulation of the query "Get suppliers who supply at least one part":

```
SELECT *
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.SNO = S.SNO )
```

Now, here's another formulation that's logically equivalent to the foregoing:

```
SELECT *
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) > 0
```

But we don't really want the system to perform the complete count that's apparently being requested here and then check to see whether that count is greater than one; rather, we want it to stop counting as soon as it finds the second shipment. In other words, we'd really like some optimization to be done. Writing code that relies on optimization is usually not a good idea! So be careful over the use of COUNT; in particular, don't use it where EXISTS would be more logically correct.

Relational Completeness

Every operator of the relational algebra has a precise definition in terms of logic. (I didn't state this fact explicitly before, but it's easy to see that the definitions I gave in Chapters 6 and 7 for join and the rest can be reformulated in terms of logic as described in the present chapter.) It follows as a direct consequence that, for every expression of the relational algebra, there exists an expression of the relational calculus that's logically equivalent (i.e. , has the same semantics). In other words, the relational calculus is at least as "powerful" (meaning, more precisely, that it's at least as *expressive*) as the relational algebra: Any problem that can be expressed in the algebra can be expressed in the calculus.

Now, it might not be obvious, but actually the opposite is true too; that is, the algebra is at least as expressive as the calculus. It other words, the two formalisms are logically equivalent: Both are what's called *relationally complete*. I'll have a little more to say about this concept in Appendix A.

The Importance of Consistency

I have a small piece of unfinished business to attend to. Recall my claim in Chapter 8 that any proposition whatsoever (even obviously false ones) can be shown to be "true" in an inconsistent system. Now I can elaborate on that claim.

I'll start with a really simple example. Suppose that (a) relvar S is currently nonempty; (b) there's a constraint to the effect that there must always be at least one part, but relvar P is in fact currently empty (there's the inconsistency). Now consider the relational calculus query:

```
SX WHERE EXISTS PX ( TRUE )
```

Or if you prefer SQL:

```
SELECT *
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   P )
```

Now, if this expression is evaluated directly, the result will be empty. Alternatively, if the system (or the user) observes that there's a constraint that says that EXISTS PX (TRUE) must evaluate to TRUE, the WHERE clause can be reduced to one saying simply WHERE TRUE, and the result will then be the entire suppliers relation. At least one of these results must be wrong! In a sense, in fact, they're both wrong; given an inconsistent database, there simply isn't—there can't be—any well defined notion of correctness, and any answer is as good (or as bad) as any other. Indeed, this state of affairs should be self-evident: If I tell you some proposition *p* is both true and false, and then ask you whether *p* is true, there's simply no right answer you can give me.

In case you're still not convinced, consider the following slightly more realistic SQL example (under the same assumptions as before):

```
SELECT DISTINCT
       CASE WHEN EXISTS ( SELECT * FROM P ) THEN x ELSE y END
FROM   S
```

This expression will return either *x* or *y*—more precisely, it will return a table containing either *x* or *y*—depending, in effect, on whether or not the EXISTS invocation is replaced by just TRUE. Now consider that *x* and *y* can each be essentially anything at all ... For example, *x* might be an SQL expression denoting the total weight of all parts, while *y* might be the literal 0—in which case executing the query could easily lead to the erroneous conclusion that the total part weight is null instead of zero.

Concluding Remarks

It's my strong belief that database professionals in general, and SQL practitioners in particular, should have at least a small degree of familiarity with the basic concepts of predicate logic (or relational calculus—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."

Now, 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 chapter. And the benefits are so huge! I made essentially the same point in a chapter in another book—*Logic and Databases: The Roots of Relational Theory* (Trafford, 2007)—and I'd like to quote the concluding remarks from that chapter here:

> Surely it's worth investing a little effort up front in becoming familiar with [the material in this chapter] in order to avoid the problems associated with ambiguous business rules. 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—as I'll try to demonstrate in the next chapter.

Exercises

Exercise 10-1. As noted in the body of the chapter, there are exactly 16 dyadic connectives. Show the corresponding truth tables. How many monadic connectives are there?

Exercise 10-2. *(Repeated from the body of the chapter, but reworded here.)* (a) Prove that all of the monadic and dyadic connectives can be expressed in terms of suitable combinations of NOT and either AND or OR; (b) prove also that they can all be expressed in terms of just a single connective.

Exercise 10-3. Consider the predicate "*x* is a star." If the argument *the sun* is substituted for *x*, does the predicate become a proposition? If not, why not? And what about the argument *the moon*?

Exercise 10-4. Again consider the predicate "*x* is a star." If the argument *the sun* is substituted for *x*, is the predicate satisfied? If not, why not? And what about the argument *the moon*?

Exercise 10-5. Here's constraint CX1 once again from Chapter 8:

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

The expression IS_EMPTY (...) here is clearly a predicate. Now, in Chapter 8, I said the relvar name "S" in that predicate was acting as a *designator*. But isn't it actually a parameter? If not, what's the difference?

Exercise 10-6. *(Repeated from the body of the chapter.)* What does the following expression mean?

```
{ SX.SNAME } WHERE EXISTS SPX ( FORALL PX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

Exercise 10-7. *(Repeated from the body of the chapter.)* Give SQL analogs of the relational calculus expressions in the subsection "More Sample Queries" in the body of the chapter.

Exercise 10-8. Prove that AND and OR are associative.

Exercise 10-9. Let *p*(*x*) and *q* be predicates in which *x* does and does not appear, respectively, as a free variable. Which of the following statements are valid? (I remind you that the symbol "⇒" means *implies*; the symbol "≡" means *is equivalent to*. Note too that *A* ⇒ *B* and *B* ⇒ *A* are together the same as *A* ≡ *B*.)

a. EXISTS x (q) ≡ q

b. FORALL x (q) ≡ q

c. EXISTS x ($p(x)$ AND q) ≡ EXISTS x ($p(x)$) AND q

d. FORALL x ($p(x)$ AND q) ≡ FORALL x ($p(x)$) AND q

e. FORALL x ($p(x)$) ⇒ EXISTS x ($p(x)$)

f. EXISTS x (TRUE) ≡ TRUE

g. FORALL x (FALSE) ≡ FALSE

h. UNIQUE x ($p(x)$) ⇒ EXISTS x ($p(x)$)

i. UNIQUE x ($p(x)$) ⇒ FORALL x ($p(x)$)

j. FORALL x ($p(x)$) AND EXISTS x ($p(x)$) ⇒ UNIQUE x ($p(x)$)

k. FORALL x ($p(x)$) AND UNIQUE x ($p(x)$) ⇒ EXISTS x ($p(x)$)

Exercise 10-10. Let *p*(*x*,*y*) be a predicate with free variables *x* and *y*. Which of the following statements are valid?

a. EXISTS x EXISTS y ($p(x,y)$) ≡ EXISTS y EXISTS x ($p(x,y)$)

b. FORALL x FORALL y ($p(x,y)$) ≡ FORALL y FORALL x ($p(x,y)$)

c. FORALL x ($p(x,y)$) ≡ NOT EXISTS x (NOT $p(x,y)$)

d. EXISTS x ($p(x,y)$) ≡ NOT FORALL x (NOT $p(x,y)$)

e. `EXISTS` x `FORALL` y ($p(x,y)$) $\equiv$ `FORALL` y `EXISTS` x ($p(x,y)$)

f. `EXISTS` y `FORALL` x ($p(x,y)$) $\Rightarrow$ `FORALL` x `EXISTS` y ($p(x,y)$)

Exercise 10-11. Let $p(x)$ and $q(y)$ be predicates with free variables x and y, respectively. Which of the following statements are valid?

a. `EXISTS` x ($p(x)$) `AND EXISTS` y ($q(y)$) $\equiv$
`EXISTS` x `EXISTS` y ($p(x)$ `AND` $q(y)$)

b. `EXISTS` x (`IF` $p(x)$ `THEN` $q(x)$) $\equiv$
`IF FORALL` x ($p(x)$) `THEN EXISTS` x ($q(x)$)

Exercise 10-12. Where possible and reasonable, give relational calculus solutions to exercises from Chapters 6–9.

Exercise 10-13. Consider this query: "Get cities in which either a supplier or a part is located." Can this query be done in the relational calculus? If not, why not?

Exercise 10-14. Show that SQL is relationally complete (i.e., prove that, for every expression of the relational algebra or the relational calculus, there exists a semantically equivalent expression in SQL).

Exercise 10-15. Here's an excerpt from Chapter 8:

- 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.

Are these two statements logically equivalent? In other words, is there some duplication here?

Exercise 10-16. Is prenex normal form always achievable?

CHAPTER ELEVEN

Using Logic to Formulate SQL Expressions

IN CHAPTER 6, I DESCRIBED THE PROCESS OF EXPRESSION TRANSFORMATION AS IT APPLIED TO relational algebra expressions specifically; I showed how one such expression could be transformed into another logically equivalent one, using various transformation laws. The laws I considered included such things as:

a. Restrict distributes over union, intersect, and difference

b. Project distributes over union but not over intersect and difference

and several others. (As you might expect, analogous laws apply to expressions of the relational calculus also, though I didn't say much about such laws in Chapter 10.)

Now, the purpose of such transformations, as I discussed them earlier, was essentially optimization; the aim was to come up with an expression with the same semantics as the original one but better performance characteristics. However, the concept of expression transformation—or *query rewrite*, as it's sometimes (not very appropriately) known—has application in other areas, too. In particular, and very importantly, it can be used to transform precise logical expressions representing queries and the like into SQL equivalents. And that's what this chapter is all about: It shows how to take the logical or relational calculus formulation of some query or constraint (for example) and map it systematically into an SQL equivalent. And while the SQL formulation so obtained can sometimes be hard to understand, we know it's correct, because of the systematic manner in which it's been obtained. Hence the subtitle of this book: *How to Write Accurate SQL Code*.

Some Transformation Laws

Laws of transformation like the ones mentioned above are also known variously as:

- *Equivalences*, because they take the general form *exp1* ≡ *exp2* (recall from Chapter 10 that the symbol "≡" means "is equivalent to")
- *Identities*, because a law of the form *exp1* ≡ *exp2* can be read as saying that *exp1* and *exp2* are "identically equal," meaning they have identical semantics
- *Rewrite rules*, because a law of the form *exp1* ≡ *exp2* implies that an expression containing an occurrence of *exp1* can be rewritten as one containing an occurrence of *exp2* without changing the meaning

I'd like to expand on this last point, because it's crucial to what we're going to be doing in the present chapter. So: Let *X1* be an expression containing an occurrence of *x1* as a subexpression; let *x2* be equivalent to *x1*; and let *X2* be the expression obtained by substituting *x2* for the occurrence of *x1* in question in *X1*. Then *X1* and *X2* are logically and semantically equivalent; hence, *X1* can be rewritten as *X2*.

Here's a simple example. Consider the SQL expression

```
SELECT  SNO
FROM    S
WHERE ( STATUS > 10 AND CITY = 'London' )
OR    ( STATUS > 10 AND CITY = 'Athens' )
```

The boolean expression in the WHERE clause here is clearly equivalent to the following:

```
STATUS > 10 AND ( CITY = 'London' OR CITY = 'Athens' )
```

Hence the overall expression can be rewritten as:

```
SELECT    SNO
FROM      S
WHERE     STATUS > 10
AND     ( CITY = 'London' OR CITY = 'Athens' )
```

Here then are some of the transformation laws we'll be using in this chapter:

- *The implication law:*

  ```
  IF p THEN q  ≡  ( NOT p ) OR q
  ```

  I did state this law in Chapter 10, but I didn't illustrate its use there. Take a moment (if you need to) to check the truth tables and convince yourself that the law is valid. *Note:* The symbols *p* and *q* stand for arbitrary boolean expressions or predicates. In this chapter, I'll favor the term *boolean expression* over *predicate*, since the emphasis throughout is on expressions as such—i.e., pieces of program text, in effect—rather than on logic. Logic in general, and predicates in particular, are more abstract than pieces of program text (or an argument can be made to that effect, at least). As noted in the previous chapter, a boolean expression can always be taken as a concrete representation of some predicate.

- *The double negation law:*

  ```
  NOT ( NOT p )  ≡  p
  ```

  This law is obvious (but it's important).

- *De Morgan's laws:*

  ```
  NOT ( p AND q )  ≡  ( NOT p ) OR  ( NOT q )

  NOT ( p OR q  )  ≡  ( NOT p ) AND ( NOT q )
  ```

  I didn't discuss these laws in the previous chapter, but I think they make good intuitive sense. For example, the first one says, loosely, that if it's not the case that *p* and *q* are both true, then it must be the case that either *p* isn't true or *q* isn't true (or both). Be that as it may, the validity of both laws follows immediately from the truth tables. Here, for example, is the truth table corresponding to the first law:

p *q*	*p* AND *q*	NOT (*p* AND *q*)	(NOT *p*) OR (NOT *q*)
T T	T	F	F
T F	F	T	T
F T	F	T	T
F F	F	T	T

  Since the columns for NOT (*p* AND *q*) and (NOT *p*) OR (NOT *q*) are identical, the validity of the first of De Morgan's laws follows. Proof of the validity of the second is analogous.

- *The distributive laws:*

  ```
  p AND ( q OR  r )  ≡  ( p AND q ) OR  ( p AND r )

  p OR  ( q AND r )  ≡  ( p OR  q ) AND ( p OR  r )
  ```

  I'll leave the proof of these two to you. Note, however, that I was using the first of these laws in the SQL example near the beginning of this section. You might also note that these distributive laws are a little more general, in a sense, than the ones we saw in Chapter 6. In that chapter we saw examples of a monadic operator, such as restrict, distributing over a dyadic operator, such as union; here, by contrast, we see *dyadic* operators (AND and OR) each distributing over the other.

- *The quantification law:*

  ```
  FORALL x ( p ( x ) )  ≡  NOT EXISTS x ( NOT p ( x ) )
  ```

  I discussed this one in the previous chapter as well. In fact, I hope you can see it's really just an application of De Morgan's laws to EXISTS and FORALL expressions specifically (you'll recall from the previous chapter that EXISTS and FORALL are basically just iterated OR and iterated AND, respectively).

One further remark on these laws: Because De Morgan's laws in particular will often be applied to the result of a prior application of the implication law, it's convenient to restate the first of them, at least, in the following form (in which *q* is replaced by NOT *q* and the double negation law has been tacitly applied):

```
NOT ( p AND NOT q )  ≡  ( NOT p ) OR q
```

Or rather (but it's the same thing, logically):

```
( NOT p ) OR q  ≡  NOT ( p AND NOT q )
```

Equivalently:

```
IF p THEN q  ≡  NOT ( p AND NOT q )
```

Most of the references to one of De Morgan's laws in what follows will be to this restated formulation.

The remainder of this chapter offers practical guidelines on the use of these laws to help in the formulation of "complex" SQL expressions. I'll start with some very simple examples and build up gradually to ones that are quite complex.

Example 1: Logical Implication

Consider again the constraint from the previous chapter to the effect that all red parts must be stored in London. For a given part, this constraint corresponds to a business rule that might be stated (more or less formally) like this:

```
IF COLOR = 'Red' THEN CITY = 'London'
```

In other words, it's a logical implication. Now, SQL doesn't support logical implication as such, but the implication law tells us that the foregoing expression can be transformed into this one:

```
( NOT ( COLOR = 'Red' ) ) OR CITY = 'London'
```

(I've added some parentheses for clarity). And this expression involves only operators that SQL does support, so it can be formulated directly as a base table constraint:

```
CONSTRAINT BTCX1 CHECK ( NOT ( COLOR = 'Red' ) OR CITY = 'London' )
```

Or perhaps a little more naturally (making use of the fact that NOT ($a = b$) can be transformed into $a \neq b$—in SQL, *a* <> *b*—and dropping some parentheses):

```
CONSTRAINT BTCX1 CHECK ( COLOR <> 'Red' OR CITY = 'London' )
```

Example 2: Universal Quantification

Now, I was practicing a tiny deception in Example 1, inasmuch as I was pretending that the specific part to which the constraint applied was understood. But that's effectively just what happens with base table constraints in SQL; they're understood to apply to each and every row of the base table whose definition they're part of. However, suppose we wanted to be more explicit—i.e., suppose we wanted to state explicitly that the constraint applies to every part that happens to be represented in table P. In other words, for all such parts PX, if the color of part PX is red, then the city for part PX is London:

```
FORALL PX ( IF PX.COLOR = 'Red' THEN PX.CITY = 'London' )
```

NOTE

The name PX and others like it in this chapter are deliberately chosen to be reminiscent of the range variables used in examples in the previous chapter. In fact, I'm going to assume from this point forward that names of the form PX, PY, PZ, etc., denote variables that range over table P; names of the form SX, SY, SZ, etc., denote variables that range over table S; and so on. Details of how such variables are defined—in logic, I mean, not in SQL—aren't important for present purposes and are therefore omitted. In SQL, they're defined by means of AS clauses, which I'll show when we get to the SQL formulations as such.

Now, SQL doesn't support FORALL, but the quantification law tells us that the foregoing expression can be transformed into this one:

```
NOT EXISTS PX ( NOT ( IF PX.COLOR = 'Red'
                      THEN PX.CITY = 'London' ) )
```

(Again I've added some parentheses for clarity. From this point forward, in fact, I'll feel free to introduce extra parentheses or drop existing ones for clarity without further comment.) Now applying the implication law:

```
NOT EXISTS PX ( NOT ( NOT ( PX.COLOR = 'Red' )
                      OR PX.CITY = 'London' ) )
```

This expression could now be mapped directly into SQL, but it's probably worth tidying it up a little first. Applying De Morgan:

```
NOT EXISTS PX ( NOT ( NOT ( ( PX.COLOR = 'Red' )
                AND NOT ( PX.CITY = 'London' ) ) ) )
```

Applying the double negation law and dropping some parentheses:

```
NOT EXISTS PX ( PX.COLOR = 'Red' AND NOT ( PX.CITY = 'London' ) )
```

Finally:

```
NOT EXISTS PX ( PX.COLOR = 'Red' AND PX.CITY ≠ 'London' )
```

Now, the transformations so far have all been very simple; you might even have found them rather tedious. But mapping this final logical expression into SQL isn't quite so straightforward. Here are the details of that mapping:

- First of all (and unsurprisingly), NOT maps to NOT.
- Second, "EXISTS PX (*bx*)" maps to "EXISTS (SELECT * FROM P AS PX WHERE (*sbx*))," where *sbx* is the SQL analog of the boolean expression *bx*.
- Third, the parentheses surrounding *sbx* can be dropped, though they don't have to be.
- Last, the entire expression needs to be wrapped up inside some suitable CREATE ASSERTION syntax.

So here's the final version:

```
CREATE ASSERTION ... CHECK
     ( NOT EXISTS ( SELECT *
                    FROM   P AS PX
                    WHERE  PX.COLOR = 'Red'
                    AND    PX.CITY <> 'London' ) ) ;
```

Example 3: Implication and Universal Quantification

A query example this time—"Get part names for parts whose weight is different from that of every part in Paris." Here's a straightforward logical (i.e., relational calculus) formulation:

```
{ PX.PNAME } WHERE FORALL PY ( IF PY.CITY = 'Paris'
                               THEN PY.WEIGHT ≠ PX.WEIGHT )
```

This expression can be interpreted as follows: "Get PNAME values from parts PX such that, for all parts PY, if PY is in Paris, then PY and PX have different weights." Note that I use the terms *where* and *such that* interchangeably—whichever seems to reads best in the case at hand—when I'm giving natural language interpretations like the one under discussion.

As a first transformation, let's apply the quantification law:

```
{ PX.PNAME } WHERE NOT EXISTS PY ( NOT ( IF PY.CITY = 'Paris'
                                   THEN PY.WEIGHT ≠ PX.WEIGHT ) )
```

Next, apply the implication law:

```
{ PX.PNAME } WHERE
             NOT EXISTS PY ( NOT ( NOT ( PY.CITY = 'Paris' )
                             OR ( PY.WEIGHT ≠ PX.WEIGHT ) ) )
```

Apply De Morgan:

```
{ PX.PNAME } WHERE
             NOT EXISTS PY ( NOT ( NOT ( ( PY.CITY = 'Paris' )
                             AND NOT ( PY.WEIGHT ≠ PX.WEIGHT ) ) ) )
```

Tidy up, using the double negation law, plus the fact that NOT ($a \neq b$) is equivalent to $a = b$:

```
{ PX.PNAME } WHERE NOT EXISTS PY ( PY.CITY = 'Paris' AND
                                   PY.WEIGHT = PX.WEIGHT )
```

Map to SQL:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PY
       WHERE  PY.CITY = 'Paris'
       AND    PY.WEIGHT = PX.WEIGHT )
```

Incidentally, that DISTINCT is really needed in the opening SELECT clause here! Here's the result:*

PNAME
Screw
Cog

Unfortunately, there's a fly in the ointment in this example. Suppose there's at least one part in Paris, but all such parts have a null weight. Then we simply don't know—we can't possibly say—whether there are any parts whose weight is different from that of every part in Paris (i.e., the original query is strictly unanswerable). In SQL, however, the subquery following the keyword EXISTS will evaluate to an empty table for every possible part PX represented in P; the NOT EXISTS will therefore evaluate to TRUE for every such part PX; and the expression overall will therefore incorrectly return all part names in table P.

ASIDE

This is probably the biggest practical problem with nulls—they lead to wrong answers. What's more, of course, we don't know in general which answers are right and which wrong! For further elaboration of such matters, see the paper "Why Three- and Four-Valued Logic Don't Work" (mentioned in Appendix D).

What's more, not only is the foregoing SQL result incorrect, but *any* definite result would represent, in effect, a lie on the part of the system. To say it again, the only logically correct result is "I don't know"—or, to be more precise and a little more honest, "The system doesn't have enough information to give a definitive response to this query."

What makes matters even worse is that under the same conditions as before (i.e., if there's at least one part in Paris and those parts all have a null weight), the SQL expression

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT NOT IN ( SELECT PY.WEIGHT
                          FROM   P AS PY
                          WHERE  PY.CITY = 'Paris' )
```

—which looks as if it ought to be logically equivalent to the one shown previously (and indeed *is* so, in the absence of nulls)—will return an empty result: a different, though equally incorrect, result.

The moral is obvious: Avoid nulls!—and then the transformations all work properly.

* All query results shown in this chapter are based on the usual sample data values, of course. *Note:* According to reviewers, at least two SQL products gave the same result regardless of whether DISTINCT was specified. If so, then the products in question would seem to have a bug in this area.

Example 4: Correlated Subqueries

Consider the query "Get names of suppliers who supply both part P1 and part P2." Here's a logical formulation:

```
{ SX.SNAME } WHERE
           EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P1' ) AND
           EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = 'P2' )
```

An equivalent SQL formulation is straightforward:

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS ( SELECT *
                FROM   SP AS SPX
                WHERE  SPX.SNO = SX.SNO
                AND    SPX.PNO = 'P1' )
AND    EXISTS ( SELECT *
                FROM   SP AS SPX
                WHERE  SPX.SNO = SX.SNO
                AND    SPX.PNO = 'P2' )
```

Here's the result:

SNAME
Smith
Jones

As you can see, however, this SQL expression involves two *correlated* subqueries. (In fact, Example 3 involved a correlated subquery also. See Chapter 12 for further discussion.) But correlated subqueries are often contraindicated from a performance point of view, because—conceptually, at any rate—they have to be evaluated repeatedly, once for each row in the outer table, instead of just once and for all. The possibility of eliminating them thus seems worth investigating. Now, in the case at hand (where the correlated subqueries appear within EXISTS invocations), there's a simple transformation that can be used to achieve precisely that effect. The resulting expression is:

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  SX.SNO IN ( SELECT SPX.SNO
                   FROM   SP AS SPX
                   WHERE  SPX.PNO = 'P1' )
AND    SX.SNO IN ( SELECT SPX.SNO
                   FROM   SP AS SPX
                   WHERE  SPX.PNO = 'P2' )
```

More generally, the SQL expression

```
SELECT sic    /* "SELECT item commalist" */
FROM   T1
WHERE  [ NOT ] EXISTS ( SELECT *
                        FROM   T2
                        WHERE  T2.C = T1.C
                        AND    bx )
```

can be transformed into

```
SELECT sic
FROM   T1
WHERE  T1.C [ NOT ] IN ( SELECT T2.C
                         FROM   T2
                         WHERE  bx )
```

In practice, this transformation is probably worth applying whenever it can be. (Of course, it would be better if the optimizer could perform the transformation automatically; unfortunately, however, we can't always count on the optimizer to do what's best.) But there are many situations where the transformation simply doesn't apply. As Example 3 showed, nulls can be one reason it doesn't apply—by the way, are nulls a consideration in Example 4?—but there are cases where it doesn't apply even if nulls are avoided. As an exercise, you might like to try deciding which of the remaining examples in this chapter it does apply to.

Example 5: Naming Subexpressions

Another query: "Get full supplier details for suppliers who supply all purple parts." *Note:* This query, or one very like it, is often used to demonstrate a flaw in the relational divide operator as originally defined. See the further remarks on this topic at the end of this section.

Here first is a logical formulation:

```
{ SX } WHERE FORALL PX ( IF PX.COLOR = 'Purple' THEN
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

("names of suppliers SX such that, for all parts PX, if PX is purple, there exists a shipment SPX with SNO equal to the supplier number for supplier SX and PNO equal to the part number for part PX"). First we apply the implication law:

```
{ SX } WHERE FORALL PX ( NOT ( PX.COLOR = 'Purple' ) OR
             EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

Next De Morgan:

```
{ SX } WHERE
       FORALL PX ( NOT ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) )
```

Apply the quantification law:

```
{ SX } WHERE
       NOT EXISTS PX ( NOT ( NOT ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) ) )
```

Double negation:

```
{ SX } WHERE
       NOT EXISTS PX ( ( PX.COLOR = 'Purple' ) AND
       NOT EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) )
```

Drop some parentheses and map to SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  PX.COLOR = 'Purple'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

The result is the entire suppliers relation:*

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

Now, you might have had some difficulty in following the transformations in the foregoing example, and you might also be having some difficulty in understanding the final SQL formulation. Well, a useful technique, when the expressions start getting a little complicated as in this example, is to abstract a little by introducing symbolic names for subexpressions. Let's use *exp1* to denote the subexpression

```
PX.COLOR = 'Purple'
```

and *exp2* to denote the subexpression

```
EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
```

(note that both of these subexpressions can be directly represented, more or less, in SQL). Then the original relational calculus expression becomes:

```
{ SX } WHERE FORALL PX ( IF exp1 THEN exp2 )
```

Now we can see the forest as well as the trees, as it were, and we can start to apply our usual transformations—though now it seems to make more sense to apply them in a different sequence, precisely because we do now have a better grasp of the big picture. First, then, the quantification law:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( IF exp1 THEN exp2 ) )
```

Implication law:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 ) OR exp2 ) )
```

* Recall from Chapter 7 that since there aren't any purple parts, every supplier supplies all of them—even supplier S5, who supplies no parts at all. See the discussion of empty ranges in Chapter 10 for further explanation.

De Morgan:

```
{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 AND NOT ( exp2 ) ) )
```

Double negation:

```
{ SX } WHERE NOT EXISTS PX ( exp1 AND NOT ( exp2 ) )
```

Finally, expand *exp1* and *exp2* and map to SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  PX.COLOR = 'Purple'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

As I think this example demonstrates, SQL expressions obtained by the techniques under discussion are often quite hard to understand directly; as I've said before, however, we know they're correct, because of the systematic manner in which they've been derived.

As an aside, I can't resist showing a **Tutorial D** version of the example by way of comparison:

```
S WHERE ( !! SP ) { PNO } ⊇ ( P WHERE COLOR = 'Purple') { PNO }
```

Now let me explain the remark I made at the beginning of this section, regarding divide. For simplicity, let's denote the restriction P WHERE COLOR = 'Purple' by the symbol PP. Let's also simplify the query at hand—"Get full supplier details for suppliers who supply all purple parts"—so that it asks for supplier numbers only, instead of full supplier details. Then it might be thought that the query could be represented by the following expression:

```
SP { SNO , PNO } DIVIDEBY PP { PNO }
```

NOTE

DIVIDEBY here represents the divide operator as originally defined. See Chapter 7 if you need an explanation of this point.

With our usual sample data values, however, relation PP, and therefore the projection of relation PP on {PNO}, are both empty (because there aren't any purple parts), and this expression returns the supplier numbers S1, S2, S3, and S4. But if there aren't any purple parts, then every supplier supplies all of them (see the discussion of empty ranges in the previous chapter)—*even supplier S5*, who supplies no parts at all. And the foregoing division can't possibly return supplier number S5, because it extracts supplier numbers from SP instead of S, and supplier S5 isn't represented in SP. So the informal characterization of that division as "Get supplier numbers for suppliers who supply all purple parts" is incorrect; it should be, rather, "Get supplier numbers for suppliers who supply *at least one part* *and also supply* all purple parts." As this example demonstrates, therefore (and to repeat something I said in Chapter 7), the divide operator doesn't really solve the problem it was originally, and specifically, designed to address.

Example 6: More on Naming Subexpressions

I'll give another example to illustrate the usefulness of introducing symbolic names for subexpressions. The query is "Get suppliers such that every part they supply is in the same city as that supplier." Here's a logical formulation:

```
{ SX } WHERE FORALL PX
     ( IF EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
       THEN PX.CITY = SX.CITY )
```

("suppliers SX such that, for all parts PX, if there's a shipment of PX by SX, then PX.CITY = SX.CITY").

This time I'll just show the transformations without naming the transformation laws involved at each step (I'll leave that as an exercise for you):

```
{ SX } WHERE FORALL PX ( IF exp1 THEN exp2 )

{ SX } WHERE NOT EXISTS PX ( NOT ( IF exp1 THEN exp2 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 ) OR exp2 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT ( exp1 AND NOT ( exp2 ) ) ) )

{ SX } WHERE NOT EXISTS PX ( exp1 AND NOT ( exp2 ) )
```

Now expand *exp1* and *exp2* and map to SQL:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO )
              AND    PX.CITY <> SX.CITY )
```

Result:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

By the way, if you find this result a little surprising, note that supplier S3 supplies just one part, part P2, and supplier S5 supplies no parts at all; logically speaking, therefore, both of these suppliers do indeed satisfy the requirement that “every part they supply” is in the same city.

Here for interest is a **Tutorial D** version of the same example:

```
S WHERE RELATION { TUPLE { CITY CITY } } = ( ( !! SP ) JOIN P ) { CITY }
```

Example 7: Dealing with Ambiguity

Natural language is often ambiguous. For example, consider the following query: “Get suppliers such that every part they supply is in the same city.” First of all, notice the subtle (?) difference between this example and the previous one. Second, and more important, note that this natural language formulation is indeed ambiguous! For the sake of definiteness, I’m going to assume it means the following:

> Get suppliers SX such that for all parts PX and PY, if SX supplies both of them, then PX.CITY = PY.CITY.

Observe that a supplier who supplies just one part will qualify under this interpretation. (So will a supplier who supplies no parts at all, incidentally.) Alternatively, the query might mean:

> Get suppliers SX such that for all parts PX and PY, if SX supplies both of them *and they’re distinct*, then PX.CITY = PY.CITY.

Now a supplier who supplies just one part or no parts at all won’t qualify.

As I’ve said, I’m going to assume the first interpretation, just to be definite. But note that ambiguities of this kind are quite common with complex queries and complex business rules, and another advantage of logic in the context at hand is precisely that it can help pinpoint and resolve such ambiguities.

Here then is a logical formulation for the first interpretation:

```
{ SX } WHERE FORALL PX ( FORALL PY
     ( IF EXISTS SPX ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO )
       AND EXISTS SPY ( SPY.SNO = SX.SNO AND SPY.PNO = PY.PNO )
       THEN PX.CITY = PY.CITY ) )
```

And here are the transformations (again I’ll leave it to you to decide just which law is being applied at each stage):

```
{ SX } WHERE FORALL PX ( FORALL PY
         ( IF exp1 AND exp2 THEN exp3 ) )

{ SX } WHERE NOT EXISTS PX ( NOT FORALL PY
         ( IF exp1 AND exp2 THEN exp3 ) )

{ SX } WHERE NOT EXISTS PX ( NOT ( NOT EXISTS PY ( NOT
         ( IF exp1 AND exp2 THEN exp3 ) ) ) )
```

```
{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( IF exp1 AND exp2 THEN exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( NOT ( exp1 AND exp2 ) OR exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY ( NOT
           ( NOT ( exp1 ) OR NOT ( exp2 ) OR exp3 ) ) )

{ SX } WHERE NOT EXISTS PX ( EXISTS PY (
           ( exp1 AND exp2 AND NOT ( exp3 ) ) ) )
```

SQL equivalent:

```
SELECT *
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PY
              WHERE  EXISTS
                   ( SELECT *
                     FROM   SP AS SPX
                     WHERE  SPX.SNO = SX.SNO
                     AND    SPX.PNO = PX.PNO )
              AND    EXISTS
                   ( SELECT *
                     FROM   SP AS SPY
                     WHERE  SPY.SNO = SX.SNO
                     AND    SPY.PNO = PY.PNO )
              AND    PX.CITY <> PY.CITY ) )
```

By the way, I used two distinct range variables SPX and SPY, both ranging over SP, in this example purely for reasons of clarity; I could perfectly well have used the same one (say SPX) twice over—it would have made no logical difference at all. Here's the result:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

At this point, I'd like to mention another transformation law that's sometimes useful: *the contrapositive law*. Consider the implication IF NOT *q* THEN NOT *p*. By definition, this expression is equivalent to NOT (NOT *q*) OR NOT *p*—which is the same as *q* OR NOT *p*—which is the same as NOT *p* OR *q*—which is the same as IF *p* THEN *q*. So we have:

```
IF p THEN q  ≡  IF NOT q THEN NOT p
```

Note that this law does make intuitive sense: If the truth of *p* implies the truth of *q*, then the falsity of *q* must imply the falsity of *p*. For example, if "It's raining" implies "The streets are wet," then "The streets aren't wet" must imply "It isn't raining."

In the example at hand, then, another possible way of stating the interpretation previously assumed ("Get suppliers SX such that for all parts PX and PY, if SX supplies both of them, then PX.CITY = PY.CITY") is:

> Get suppliers SX such that for all parts PX and PY, if PX.CITY ≠ PY.CITY, then SX doesn't supply both of them. (Incidentally, is it obvious that this version is equivalent to the previous one?)

This perception of the query will very likely lead to a different (though logically equivalent) SQL formulation. I'll leave the details as an exercise.

Example 8: Using COUNT

Now, there's more to be said about the previous example. Let me state the query again: "Get suppliers such that every part they supply is in the same city." Here's yet another possible natural language interpretation of this query:

> Get suppliers SX such that the number of cities for parts supplied by SX is less than or equal to one.

Note that "less than or equal to," by the way—"equal to" alone would correspond to a different interpretation of the query. Logical formulation:

```
{ SX } WHERE COUNT ( PX.CITY WHERE EXISTS SPX
                   ( SPX.SNO = SX.SNO AND SPX.PNO = PX.PNO ) ) ≤ 1
```

This is the first example in this chapter to make use of an aggregate operator. As I think you can see, however, the mapping is quite straightforward. An equivalent SQL formulation is:

```
SELECT *
FROM   S AS SX
WHERE  ( SELECT COUNT ( DISTINCT PX.CITY )
         FROM   P AS PX
         WHERE  EXISTS ( SELECT *
                         FROM   SP AS SPX
                         WHERE  SPX.SNO = SX.SNO
                         AND    SPX.PNO = PX.PNO ) ) <= 1
```

The result is as shown under Example 7. However, I remind you from the previous chapter that as a general rule it's wise, for performance reasons, to be careful over the use of COUNT; in particular, don't use it where EXISTS would be more logically correct.

Here are some questions for you: First, given the foregoing SQL formulation of the query, is that DISTINCT in the COUNT invocation really necessary? Second, try to formulate the query in terms of GROUP BY and HAVING. If you succeed, what were the logical steps you went through to construct that formulation? (See Example 12 for further discussion of GROUP BY and HAVING.)

Example 9: Join Queries

This time, for practice, I'll just present the query and the SQL formulation and leave you to give the logical formulation and the derivation process. The query is "Get suppliers such that every part they supply is in the same city (as in Examples 7 and 8), *together with the city in question*." Here's the SQL formulation:

```
SELECT DISTINCT SX.* , PX.CITY
FROM   S AS SX , P AS PX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  SPX.SNO = SX.SNO
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPY
              WHERE  SPY.SNO = SPX.SNO
              AND    EXISTS
                   ( SELECT *
                     FROM   P AS PY
                     WHERE  PY.PNO = SPY.PNO
                     AND    PY.CITY <> PX.CITY ) ) )
```

Result:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris

Exercise: Is the DISTINCT necessary in this example? And why is this section called "Join Queries"?

Example 10: UNIQUE Quantification

Recall this example from Chapter 10—

```
CONSTRAINT CX6 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

—which is a logical formulation of the constraint that there's exactly one supplier for each shipment. Now, recall that the logic expression "EXISTS SX (*bx*)" maps to the SQL expression "EXISTS (SELECT * FROM S AS SX WHERE (*sbx*))," where *sbx* is the SQL analog of the boolean expression *bx*. However, the logic expression "UNIQUE SX (*bx*)" does *not* map to the SQL expression "UNIQUE (SELECT * FROM S AS SX WHERE (*sbx*))"! (There's a trap for the unwary here.) Instead, it maps to:

```
UNIQUE ( SELECT * FROM S AS SX WHERE ( sbx ) )
AND
EXISTS ( SELECT * FROM S AS SX WHERE ( sbx ) )
```

So the boolean expression portion of constraint CX6 maps to:

```
NOT EXISTS
  ( SELECT *
    FROM   SP AS SPX
```

```
      WHERE  NOT UNIQUE
           ( SELECT *
             FROM   S AS SX
             WHERE  SX.SNO = SPX.SNO ) )
      OR     NOT EXISTS
           ( SELECT *
             FROM   S AS SX
             WHERE  SX.SNO = SPX.SNO ) )
```

Now, there's another equivalence we might appeal to here—the logic expression UNIQUE SX (*bx*) is clearly equivalent to:

```
   COUNT ( SX WHERE ( bx ) ) = 1
```

As a result we can simplify the foregoing SQL formulation to:

```
   NOT EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  ( SELECT COUNT ( * )
                FROM   S AS SX
                WHERE  SX.SNO = SPX.SNO ) ) <> 1
```

Here for interest is another SQL formulation that uses neither UNIQUE nor COUNT. Try to convince yourself it's correct.

```
   NOT EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   S AS SX
              WHERE  SX.SNO = SPX.SNO
              AND    NOT EXISTS
                   ( SELECT *
                     FROM   S AS SY
                     WHERE  SY.SNO = SX.SNO
                     AND  ( SY.SNAME <> SX.SNAME OR
                            SY.STATUS <> SX.STATUS OR
                            SY.CITY <> SX.CITY ) ) ) )
```

Note carefully, however, that this formulation relies on the fact that duplicate rows are prohibited (in table S in particular); it doesn't work otherwise. Avoid duplicate rows!

Example 11: ALL or ANY Comparisons

You probably know that SQL supports what are called generically *ALL or ANY comparisons* (or, more formally, *quantified* comparisons, but I prefer to avoid this term because of possible confusion with SQL's EXISTS and UNIQUE operators). An ALL or ANY comparison is an expression of the form *rx* θ *sq*, where *rx* is a row expression, *sq* is a subquery, and θ is any of the usual scalar comparison operators supported in SQL ("=", "<>", "<", "<=", ">", ">=") followed by one of the keywords ALL, ANY, or SOME. (SOME is just a different spelling for ANY, and table subqueries are discussed further in Chapter 12.) The semantics are as follows:

- An ALL comparison returns TRUE if and only if the corresponding comparison condition without the ALL returns TRUE for all of the rows in the table represented by *sq*. (If that table is empty, the comparison returns TRUE.)
- An ANY comparison returns TRUE if and only if the corresponding comparison condition without the ANY returns TRUE for at least one of the rows in the table represented by *sq*. (If that table is empty, the comparison returns FALSE.)

Here's an example ("Get part names for parts whose weight is greater than that of every blue part"):

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' )
```

Result:

PNAME
Bolt
Screw
Cog

As this example suggests, the "row expression" *rx* in the ALL or ANY comparison *rx* θ *sq* is often—almost always, in fact—just a simple scalar expression, in which case the scalar value denoted by that expression is effectively coerced to a row that contains just that scalar value. (Incidentally, note that even if *rx* doesn't consist of a simple scalar expression but actually does denote a row of degree greater than one, θ can still be something other than "=" or "<>", though the practice isn't recommended. See Chapter 3 for further discussion of this point.)

Recommendation: Don't use ALL or ANY comparisons—they're error prone, and in any case their effect can always be achieved by other methods. As an illustration of the first point, consider the fact that a natural language formulation of the foregoing query might very well use *any* in place of *every*—"Get part names for parts whose weight is greater than that of *any* blue part"—which could lead to the incorrect use of >ANY in place of >ALL. As another example, illustrating both points, consider the following SQL expression:

```
SELECT DISTINCT SNAME
FROM   S
WHERE  CITY <>ANY ( SELECT CITY FROM P )
```

This expression could easily be read as "Get names of suppliers whose city isn't equal to any part city"—but that's not what it means. Instead, it's logically equivalent* to the following ("Get names of suppliers where there's at least one part in a different city"):

* Or is it? What if supplier or part cities could be null?

```
SELECT DISTINCT SNAME
FROM   S
WHERE  EXISTS ( SELECT *
                FROM   P
                WHERE  P.CITY <> S.CITY )
```

Result:

SNAME
Smith
Jones
Blake
Clark
Adams

In fact, ALL or ANY comparisons can always be transformed into equivalent expressions involving EXISTS, as the foregoing example suggests. They can also usually be transformed into expressions involving MAX or MIN—because (e.g.) a value is greater than all of the values in some set if and only if it's greater than the maximum value in that set—and expressions involving MAX and MIN are often easier to understand, intuitively speaking, than ALL or ANY comparisons. The following table summarizes the possibilities in this regard (I'll give an example in a moment):

	ANY	ALL
=	IN	
<>		NOT IN
<	< MAX	< MIN
<=	<=MAX	<=MIN
>	> MIN	> MAX
>=	>=MIN	>=MAX

Note in particular that =ANY and <>ALL are equivalent to IN and NOT IN, respectively.*
By contrast, =ALL and <>ANY have no analogous equivalents, but expressions involving those operators can always be replaced by expressions involving EXISTS instead, as already noted.

Caveat: Unfortunately, the foregoing transformations (using MAX and MIN) aren't guaranteed to work if the MAX or MIN argument happens to be an empty set. The reason is that SQL incorrectly defines the MAX and MIN of an empty set to be null.† For example, here again is the formulation shown earlier for the query "Get part names for parts whose weight is greater than that of every blue part":

* So these two are important exceptions to the overall recommendation to avoid ALL and ANY comparisons in general; i.e., you can use =ANY and IN interchangeably, and you can use <>ALL and NOT IN interchangeably too. (Personally, I think IN and NOT IN are much clearer than their alternatives, but it's your choice.)

† For the record, the MAX of an empty set should be the minimum value and the MIN of an empty set should be the maximum value (of the pertinent type in each case).

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' )
```

And here's a transformed "equivalent":

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT > ( SELECT MAX ( PY.WEIGHT )
                     FROM   P AS PY
                     WHERE  PY.COLOR = 'Blue' )
```

Now suppose there are no blue parts. Then the first of the foregoing expressions will return all part names in table P, but the second will return an empty result.*

Anyway, to make the transformation in the example valid after all, use COALESCE—e.g., as follows:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT > ( SELECT COALESCE ( MAX ( PY.WEIGHT ) , 0.0 )
                     FROM   P AS PY
                     WHERE  PY.COLOR = 'Blue' )
```

By way of another example, consider the query "Get part names for parts whose weight is less than that of some part in Paris." Here's a logical formulation:

```
{ PX.PNAME } WHERE EXISTS PY ( PY.CITY = 'Paris' AND
                               PX.WEIGHT < PY.WEIGHT )
```

Here's a corresponding SQL formulation:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  EXISTS ( SELECT *
                FROM   P AS PY
                WHERE  PY.CITY = 'Paris'
                AND    PX.WEIGHT < PY.WEIGHT )
```

But this query too could have been expressed in terms of an ALL or ANY comparison, thus:

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT <ANY ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.CITY = 'Paris' )
```

* Note that both expressions involve some coercion. As a slightly nontrivial exercise, you might like to try figuring out exactly what coercions are involved in each case.

Result:

PNAME
Nut
Screw
Cam

As this example suggests (and indeed as already stated), expressions involving ALL and ANY comparisons can always be transformed into equivalent expressions involving EXISTS instead. Some questions for you:

- Are you sure "<ANY" is the correct comparison operator in this example? (Was "less than any" the phrase used in the natural language version? Should it have been? Recall too that "less than any" maps to "<ALL"—right?)
- Which of the various formulations do you think is the most "natural"?
- Are the various formulations equivalent if the database permits nulls? Or duplicates?

Example 12: GROUP BY and HAVING

As promised earlier, there's a little more I want to say about the GROUP BY and HAVING clauses. Consider this query: "For each part supplied by no more than two suppliers, get the part number and city and the total quantity supplied of that part." Here's a possible logical (relational calculus) formulation:

```
{ PX.PNO , PX.CITY ,
           SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO , QTY ) AS TPQ }
                    WHERE COUNT ( SPY WHERE SPY.PNO = PX.PNO ) ≤ 2
```

SQL formulation:

```
SELECT PX.PNO , PX.CITY ,
              ( SELECT COALESCE ( SUM ( SPX.QTY ) , 0 )
                FROM   SP AS SPX
                WHERE  SPX.PNO = PX.PNO ) AS TPQ
FROM   P AS PX
WHERE  ( SELECT COUNT ( * )
         FROM   SP AS SPY
         WHERE  SPY.PNO = PX.PNO ) <= 2
```

Result:

PNO	CITY	TPQ
P1	London	600
P3	Oslo	400
P4	London	500
P5	Paris	500
P6	London	100

As the opening to this section suggests, however, the interesting thing about this example is that it's one that might appear to be more easily (certainly more succinctly) expressed using GROUP BY and HAVING, thus:

```
SELECT PX.PNO , PX.CITY , COALESCE ( SUM ( SPX.QTY ) , 0 ) AS TPQ
FROM   P AS PX , SP AS SPX
WHERE  PX.PNO = SPX.PNO
GROUP  BY PX.PNO
HAVING COUNT ( * ) <= 2
```

But:

- In that GROUP BY / HAVING formulation, is the appearance of PX.CITY in the SELECT item commalist legal?
- Do you think the GROUP BY / HAVING formulation is easier to understand?
- Does the GROUP BY / HAVING formulation work correctly for parts that aren't supplied by any suppliers at all? (No, it doesn't.)
- Are the formulations equivalent if the database permits nulls? Or duplicates?

As a further exercise, give SQL formulations (a) using GROUP BY and HAVING, (b) not using GROUP BY and HAVING, for the following queries:

- Get supplier numbers for suppliers who supply *N* different parts for some *N* > 3.
- Get supplier numbers for suppliers who supply *N* different parts for some *N* < 4.

What do you conclude from this exercise?

Exercises

Exercise 11-1. If you haven't already done so, complete the exercises included inline in the body of the chapter.

Exercise 11-2. Take another look at the various SQL expressions in the body of the chapter. From those SQL formulations alone (i.e., without looking at the problem statements), see if you can come up with a natural language interpretation of what the SQL expressions mean. Then compare your interpretations with the problem statements as given in the chapter.

Exercise 11-3. Try applying the techniques described in this chapter to some genuine SQL problems from your own work environment.

Exercise 11-4. Some of the examples discussed in the present chapter—or others very much like them—were also discussed in earlier chapters, but the SQL formulations I gave in those chapters were often more "algebra like" than "calculus like." Can you come up with any transformation laws that would allow the calculus formulations to be mapped into algebraic ones or vice versa?

CHAPTER TWELVE

Miscellaneous SQL Topics

THIS LAST CHAPTER IS SOMETHING OF A POTPOURRI; it discusses a few SQL features that, for one reason or another, don't fit very neatly into any of the previous chapters. It also gives a BNF grammar for SQL table expressions, for purposes of reference.

Also, this is as good a place as any to define two terms that you need to watch out for. The terms in question are *implementation defined* and *implementation dependent*, and they're heavily used in the SQL standard. Here are the definitions:

> **Definition:** An *implementation defined* feature is one whose semantics can vary from one implementation to another, but do at least have to be specified for any individual implementation. In other words, the implementation is free to decide how it will implement the feature in question, but the result of that decision must be documented. An example is the maximum length of a character string.

> **Definition:** An *implementation dependent* feature, by contrast, is one whose semantics can vary from one implementation to another and don't even have to be specified for any individual implementation. In other words, the term effectively means *undefined*; the implementation is free to decide how it will

implement the feature in question, and the result of that decision need not be documented (it might even vary from release to release). An example is the full effect of an ORDER BY clause if the specifications in that clause fail to specify a total ordering.

SELECT *

Use of the "SELECT *" form of the SQL SELECT clause is acceptable in situations where the specific columns involved, and their left to right ordering, are both irrelevant—for example, in an EXISTS invocation. It can be dangerous in other situations, however, because the meaning of that "*" can change if (e.g.) new columns are added to an existing table. **Recommendation:** Be on the lookout for such situations and try to avoid them. In particular, don't use "SELECT *" at the outermost level in a cursor definition—instead, always name the pertinent columns explicitly. A similar remark applies to view definitions also. *Note:* However, if you adopt the strategy suggested under the discussion of column naming in Chapter 3 of always accessing the database via views, then it might be safe to use "SELECT *" anywhere you like apart from the definitions of those views themselves.

Explicit Tables

An "explicit table" in SQL is an expression of the form TABLE *T*, where *T* is the name of a base table or view or an "introduced name" (see the discussion of WITH in Chapter 6). It's logically equivalent to the following:

(SELECT * FROM *T*)

Here's a fairly complicated example that uses explicit tables ("Get all parts—but if the city is London, show it as Oslo and show the weight as double"):

```
WITH T1 AS ( SELECT PNO , PNAME , COLOR , WEIGHT , CITY
             FROM   P
             WHERE  CITY = 'London' ) ,
     T2 AS ( SELECT PNO , PNAME , COLOR , WEIGHT , CITY ,
                    2 * WEIGHT AS NEW_WEIGHT , 'Oslo' AS NEW_CITY
             FROM T1 ) ,
     T3 AS ( SELECT PNO , PNAME , COLOR ,
                    NEW_WEIGHT AS WEIGHT , NEW_CITY AS CITY
             FROM   T2 ) ,
     T4 AS ( TABLE P EXCEPT CORRESPONDING TABLE T1 )
TABLE T4 UNION CORRESPONDING TABLE T3
```

Name Qualification

Column names in SQL can usually be dot qualified by the name of the applicable range variable (see the next section). However, SQL allows that qualifier to be omitted in many situations, in which case an implicit qualifier is assumed by default. But:

- The SQL rules regarding implicit qualification aren't always easy to understand. As a result, it isn't always obvious what a particular unqualified name refers to.

- What's unambiguous today might be ambiguous tomorrow (e.g., if new columns are added to an existing table).
- In Chapter 3 I recommended, strongly, that columns that represent the same kind of information be given the same name whenever possible. If that recommendation is followed, then unqualified names will often be ambiguous anyway, and dot qualification will therefore be required.

So a good general rule is: When in doubt, qualify. Unfortunately, however, there are certain contexts in which qualification isn't allowed. The contexts in question are, loosely, ones in which the name serves as a reference to the column per se, rather than to the data contained in that column. Here's a partial list of such contexts (note the last two in particular):

- A column definition in a base table definition
- A key or foreign key specification
- The column name commalist, if specified, in CREATE VIEW
- The column name commalist, if specified, following the definition of a range variable (again, see the next section)
- The "sort keys" in ORDER BY
- The column name commalist in JOIN ... USING
- The column name commalist, if specified, on INSERT
- The left side of a SET assignment on UPDATE

Range Variables

As we saw in Chapter 10, a range variable in the relational model is a variable—a variable in the sense of logic, that is, not the usual programming language sense—that "ranges over" the set of rows in some table (or the set of tuples in some relation, to be more precise). In SQL, such variables are defined by means of AS specifications in the context of either FROM or JOIN, as in the following example:

```
SELECT SX.SNO
FROM   S AS SX
WHERE  SX.STATUS > 15
```

SX here is a range variable that ranges over table S; in other words, its permitted values are rows of table S. You can think of the SELECT expression overall as being evaluated as follows. First, the range variable takes on one of its permitted values, say the row for supplier S1. Is the status value in that row greater than 15? If it is, then supplier number S1 appears in the result. Next, the range variable moves on to another row of table S, say the row for supplier S2; again, if the status value in that row is greater than 15, then the relevant supplier number appears in the result. And so on, exhaustively, until variable SX has taken on all of its permitted values.

NOTE

SQL calls a name such as SX in the example a *correlation name*, but it doesn't seem to have a term for the thing that such a name names; certainly there's no such thing in SQL as a "correlation." (Note in particular that the term doesn't necessarily have anything to do with correlated subqueries, which are discussed in the next section.) I prefer the term *range variable*.

As a matter of fact, SQL requires SELECT expressions always to be formulated in terms of range variables; if no such variables are specified explicitly, it assumes the existence of implicit ones with the same names as the corresponding tables. For example, the SELECT expression

```
SELECT SNO
FROM   S
WHERE  STATUS > 15
```

—arguably a more "natural" SQL formulation of the example discussed above—is treated as shorthand for this expression:

```
SELECT S.SNO
FROM   S AS S
WHERE  S.STATUS > 15
```

In this latter formulation, the "S" dot qualifiers and the "S" in the AS specification "AS S" do *not* denote table S; rather, they denote a range variable called S that ranges over the table with the same name.*

Now, in the SQL standard, the items in the commalist in a FROM clause (immediately following the keyword FROM itself) are called *table references*,† and so too are the table operands to an explicit JOIN (I mentioned this latter fact, but not the former, in Chapter 6). Let *tr* be such a reference. Then, if the table expression in *tr* is a table subquery (see the next section), then *tr* must also include an AS clause—even if the range variable defined by that AS clause is never explicitly mentioned anywhere else in the overall expression. For example:

```
( SELECT SNO , CITY FROM S ) AS TEMP1
  NATURAL JOIN
( SELECT PNO , CITY FROM P ) AS TEMP2
```

Here's another example (repeated from Chapter 7):

* Here I might admit (if pressed) to a sneaking sympathy with a remark an old friend once made to me in connection with this very point: "You mathematicians are all alike—you spend hours agonizing over things that are perfectly obvious to everybody else."

† The term isn't very apt. In most languages, a variable reference is a special case of an expression; syntactically, it's just a variable name, used to denote either the variable as such or the value of that variable, as the context demands. But an SQL table reference isn't a table expression—not in the sense in which this latter term is used in this book and (perhaps more to the point) not in the sense in which it's used in SQL, either.

```
SELECT PNO , GMWT
FROM ( SELECT PNO , WEIGHT * 454 AS GMWT
       FROM   P ) AS TEMP
WHERE  GMWT > 7000.0
```

For interest, here's the latter example repeated with all implicit qualifiers made explicit:

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT P.PNO , P.WEIGHT * 454 AS GMWT
       FROM   P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

NOTE

A range variable definition in SQL can always optionally include a column name commalist (defining column names for the table the range variable ranges over), as in this example:

```
SELECT *
FROM ( S JOIN P ON S.CITY > P.CITY ) AS TEMP
     ( A , B , C , D , E , F , G , H , I )
```

The column names A, B, ..., I here effectively rename columns SNO, SNAME, STATUS, S.CITY, PNO, PNAME, COLOR, WEIGHT, and P.CITY, respectively (see the explanation of JOIN ... ON in Chapter 6). However, it shouldn't be necessary to exercise this option very often if other recommendations in this book are followed.

Recommendation: Favor the use of explicit range variables, especially in "complex" expressions—they can aid clarity, and sometimes they can save keystrokes. Be aware, however, that SQL's name scoping rules for such variables can be hard to understand (but this is true regardless of whether the variables in question are explicit or implicit).

Caveat: Many SQL texts refer to range variable names (or correlation names) as *aliases*, and describe them as if they were just alternative names for the tables they range over. But such a characterization seriously misrepresents the true state of affairs—indeed, it betrays a serious lack of understanding of what's really going on—and is strongly deprecated on that account.

Subqueries

A *subquery* in SQL is a table expression, *tx* say, enclosed in parentheses; if the table denoted by *tx* is *t*, the table denoted by the subquery is *t* also. The expression *tx* can't be an explicit JOIN expression—so, e.g., "(SELECT * FROM *A* NATURAL JOIN *B*)" is a legal subquery, but "(*A* NATURAL JOIN *B*)" isn't.

Subqueries fall into three categories (though the syntax is the same in every case):

- A *table subquery* is a subquery that's neither a row subquery nor a scalar subquery.
- A *row subquery* is a subquery appearing in a position where a row expression is expected. Let *rsq* be such a subquery; then *rsq* must denote a table with just one row.

Let the table in question be *t*, and let the single row in *t* be *r*; then *rsq* behaves as if it denoted that row *r* (in other words, *t* is coerced to *r*).

- A *scalar subquery* is a subquery appearing in a position where a scalar expression is expected. Let *ssq* be such a subquery; then *ssq* must denote a table with just one row and just one column. Let the table in question be *t*, let the single row in *t* be *r*, and let the single value in *r* be *v*; then *ssq* behaves as if it denoted that value *v* (in other words, *t* is coerced to *r*, and then *r* is coerced to *v*).

The following examples involve, in order, a table subquery, a row subquery, and a scalar subquery:

```
SELECT SNO
FROM   S
WHERE  CITY IN ( SELECT CITY
                 FROM   P
                 WHERE  COLOR = 'Red' )

UPDATE S
SET  ( STATUS , CITY ) = ( SELECT DISTINCT STATUS , CITY
                           FROM   S
                           WHERE  SNO = 'S1' )
WHERE  CITY = 'Paris' ;

SELECT SNO
FROM   S
WHERE  CITY = ( SELECT CITY
                FROM   P
                WHERE  PNO = 'P1' )
```

Next, a *correlated* subquery is a special kind of (table, row, or scalar) subquery; to be specific, it's a subquery that includes a reference to some "outer" table. In the following example, the parenthesized expression following the keyword IN is a correlated subquery, because it includes a reference to the outer table S (the query is "Get names of suppliers who supply part P1"):

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  'P1' IN ( SELECT PNO
                 FROM   SP
                 WHERE  SP.SNO = S.SNO )
```

As noted in Chapter 11, correlated subqueries are often contraindicated from a performance point of view, because—conceptually, at any rate—they have to be evaluated once for each row in the outer table instead of just once and for all. (In the example, if the overall expression is evaluated as stated, the subquery will be evaluated *N* times, where *N* is the number of rows in table S.) For that reason, it's a good idea to avoid correlated subqueries if possible. In the case at hand, it's very easy to reformulate the query to achieve this goal:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  SNO IN ( SELECT SNO
                FROM   SP
                WHERE  PNO = 'P1' )
```

Finally, a "lateral" subquery is a special kind of correlated subquery. To be specific, it's a correlated subquery that (a) appears in a FROM clause and (b) includes a reference to an "outer" table that's defined by a table reference appearing earlier in that same FROM clause. For example, consider the query "For each supplier, get the supplier number and the number of parts supplied by that supplier." Here's one possible formulation of that query in SQL:

```
SELECT S.SNO , TEMP.PCT
FROM   S , LATERAL ( SELECT COUNT ( ALL PNO ) AS PCT
                     FROM   SP
                     WHERE  SP.SNO = S.SNO ) AS TEMP
```

The purpose of the keyword LATERAL is to tell the system that the subquery to which it's prefixed is correlated with something previously mentioned in the very same FROM clause (in the example, that subquery yields exactly one value—namely, the applicable count—for each SNO value in table S). Given the sample values above the result looks like this:

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

By the way, there's something going on here that you might find a little confusing. The items in a FROM item commalist are table references, and so they denote tables. In the example, though, the particular table reference that begins with the keyword LATERAL—more precisely, what remains of that table reference if the keyword LATERAL is removed—looks more like what might be called a *scalar* reference, or more precisely a scalar subquery; certainly it could used as such, should the context demand such an interpretation. In fact, however, it's a table subquery—even though it does effectively evaluate (for a given supplier number, through a double coercion) to a single scalar value, which then becomes its contribution to the applicable result row.

Following on from the previous point, it's not clear why "lateral" subqueries are required in any case. Certainly the foregoing example can easily be reformulated to avoid the "need" (?) for any such thing:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

The subquery has moved from the FROM clause to the SELECT clause; it still refers to something else in the same clause (S.SNO, to be specific), but now the keyword LATERAL is no longer required (?). However, do note what's happened to the specification AS PCT, which appeared inside the subquery in the LATERAL formulation but has now moved outside.

Finally: I've defined the term *subquery*; perhaps it's time to define the term *query*, too!—even though I've used that term ubiquitously throughout previous chapters. So here goes: A query is a retrieval request; in other words, it's a table expression—though such expressions can also be used in contexts other than queries per se—or a statement, such as a SELECT statement in "direct (i.e., interactive) SQL," that asks for such an expression to be evaluated. The term is sometimes used, though not in this book, to refer to an update request also. It's also used to refer to the natural language version of some retrieval or update request.

"Possibly Nondeterministic" Expressions

Recall from Chapter 2 that a "possibly nondeterministic" expression in SQL is a table expression that might give different results on different evaluations, even if the database hasn't changed in the interim. I remind you in particular that such expressions aren't allowed to appear in SQL integrity constraints.

The standard's rules for labeling a given table expression "possibly nondeterministic" are as follows. Let *tx* be a table expression. Then *tx* is considered to be "possibly nondeterministic" if and only if any of the following is true:

- *tx* is a union (without ALL), intersection, or difference, and the operand tables include a column of type character string.
- *tx* is a SELECT expression, the SELECT item commalist in that SELECT expression includes an item (*C* say) of type character string, and at least one of the following is true:

  — The SELECT item commalist is preceded by the keyword DISTINCT.

  — *C* involves a MAX or MIN invocation.

  — *tx* directly includes a GROUP BY clause and *C* is one of the grouping columns.
- *tx* is a SELECT expression that directly includes a HAVING clause and the boolean expression in that HAVING clause includes *either* a reference to a grouping column of type character string *or* a MAX or MIN invocation in which the argument is of type character string.

On the face of it, however, these rules seem to be neither accurate nor complete. For example:

- A union *with* ALL is surely still "possibly nondeterministic" if one of its operands is a "possibly nondeterministic" SELECT expression.
- An explicit NATURAL JOIN or JOIN USING has exactly the same potential for indeterminacy as (e.g.) an intersection.
- A difference with ALL can't possibly be "nondeterministic" if its first operand isn't.

Furthermore, the rules are certainly stronger than they need to be in some cases; for example, suppose NO PAD applies to the collation in effect and that collation is one in which there are no characters that are "equal but distinguishable."

Empty Sets

The empty set is the set containing no elements. This concept is ubiquitous, and extremely important, in the relational world, but SQL commits a number of errors in connection with it. Unfortunately there isn't much you can do about most of those errors, but you should at least be aware of them. Here they are (this is probably not a complete list):

- A VALUES expression isn't allowed to include an empty row expression commalist.
- The SQL "set functions" all return null if their argument is empty (except for COUNT(*) and COUNT, which return zero).
- If a scalar subquery evaluates to an empty table, that empty table is coerced to a null.
- If a row subquery evaluates to an empty table, that empty table is coerced to a row of all nulls.
- If the set of grouping columns and the table being grouped are both empty, GROUP BY produces a result containing just one, necessarily empty, group, whereas it should produce a result containing no groups at all.
- A key can't be an empty set of columns (nor can a foreign key, a fortiori).
- A table can't have an empty heading.
- A SELECT item commalist can't be empty.
- A FROM item commalist can't be empty.
- The set of common columns in UNION, INTERSECT, and EXCEPT can't be empty.
- A row can't have an empty set of components.

A BNF Grammar for SQL Table Expressions

For purposes of reference, it seems appropriate to close this chapter, and the main part of the book, with a BNF grammar for SQL table expressions. For reasons that aren't important here, some of the terms used in the grammar differ from their counterparts in the SQL standard; also, constructs that I've previously advised you not to use are deliberately omitted, as are certain somewhat esoteric features (e.g., recursion).* The following simplifying abbreviations are used:

exp	for	*expression*
ref	for	*reference*
spec	for	*specification*

And the following are assumed to be <*identifier*>s and are defined no further here:

```
<table name>
<column name>
<range variable name>
```

The following are also left undefined (though it might help to recall in particular that one form of scalar expression is a scalar subquery):

```
<scalar exp>
<boolean exp>
```

NOTE

As you can see, the grammar begins with a production for <*top level table exp*>, a term not mentioned in the body of the book. I introduce this syntactic category in order to capture the fact that join expressions in particular can't appear without being nested inside some other table expression†—but it does mean that a <*table exp*> as defined in the grammar doesn't correspond 100 percent to what the rest of the book calls a table expression! (To be specific, a <*top level table exp*> is a table expression but not a <*table exp*>.) I apologize if you find this state of affairs confusing, but it's the kind of thing that happens when you try to define a grammar for a language that violates orthogonality.

* In other words, the grammar is deliberately somewhat conservative, in that it fails to define as valid certain expressions that are so, according to the standard. But I don't believe it defines any expressions to be valid that aren't so according to the standard.

† This limitation (which was mentioned at various earlier points in the book without further explanation) was introduced with the 2003 version of the standard; it didn't apply to the 1999 version.

```
<top level table exp>
   ::=   [ <with spec> ] <nonjoin exp>

<with spec>
   ::=   WITH <name intro commalist>

<name intro>
   ::=   <table name> AS <table subquery>

<table exp>
   ::=   [ <with spec> ] <nonjoin or join exp>

<nonjoin or join exp>
   ::=   <nonjoin exp> | <join exp>

<nonjoin exp>
   ::=   <nonjoin term>
       | <nonjoin exp> UNION
                   [ DISTINCT ] [ CORRESPONDING ] <table term>
       | <nonjoin exp> EXCEPT
                   [ DISTINCT ] [ CORRESPONDING ] <table term>

<table term>
   ::=   <nonjoin term>
       | <join exp>

<nonjoin term>
  ::=   <table term> INTERSECT
                   [ DISTINCT ] [ CORRESPONDING ] <table primary>
      | <nonjoin primary>

<table primary>
   ::=   <nonjoin primary>
       | <join exp>

<nonjoin primary>
   ::=   ( <nonjoin exp> )
       | TABLE <table name>
       | <table selector>
       | <select exp>

<table selector>
   ::=   VALUES <row exp commalist>

<row exp>
   ::=   <scalar exp>
       | <row selector>
       | <row subquery>
```

```
<row selector>
   ::=   ( <scalar exp commalist> )

<row subquery>
   ::=   <subquery>

<subquery>
   ::=   ( <top level table exp> )

<select exp>
   ::=   SELECT [ DISTINCT ] [ * | <select item commalist> ]
           FROM <table ref commalist>
            [ WHERE <boolean exp> ]
              [ GROUP BY <column name commalist> ]
                [ HAVING <boolean exp> ]

<select item>
   ::=   <scalar exp> [ AS <column name> ]
       | <range variable name>.*

<table ref>
   ::=   <table name> [ AS <range variable name> ]
       | [ LATERAL ] <table subquery> AS <range variable name>
       | <join exp>

<table subquery>
   ::=   <subquery>

<join exp>
   ::=   <table ref> CROSS JOIN <table ref>
       | <table ref> NATURAL JOIN <table ref>
       | <table ref> JOIN <table ref>
                     USING ( <column name commalist> )
       | ( <join exp> )
```

Exercises

Exercise 12-1. According to the foregoing grammar, which of the following SQL expressions are legal <*top level table exp*>s and which not, syntactically speaking? (*A* and *B* are table names.)

```
A NATURAL JOIN B
A INTERSECT B
TABLE A NATURAL JOIN TABLE B
TABLE A INTERSECT TABLE B
SELECT * FROM A NATURAL JOIN SELECT * FROM B
SELECT * FROM A INTERSECT SELECT * FROM B
( SELECT * FROM A ) NATURAL JOIN ( SELECT * FROM B )
( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )
( TABLE A ) NATURAL JOIN ( TABLE B )
( TABLE A ) INTERSECT ( TABLE B )
( TABLE A ) AS AA NATURAL JOIN ( TABLE B ) AS BB
( TABLE A ) AS AA INTERSECT ( TABLE B ) AS BB
( ( TABLE A ) AS AA ) NATURAL JOIN ( ( TABLE B ) AS BB )
( ( TABLE A ) AS AA ) INTERSECT ( ( TABLE B ) AS BB )
```

Perhaps I should remind you that, relationally speaking, intersection is a special case of natural join. What do you conclude from this exercise?

Exercise 12-2. Consider any SQL product available to you. Does that product support (a) UNIQUE expressions, (b) explicit tables, (c) lateral subqueries, (d) "possibly nondeterministic" expressions?

Exercise 12-3. Throughout this book I've taken the term *SQL* to refer to the official standard version of that language specifically (though my treatment of the standard has deliberately not been exhaustive). Every product on the market, however, departs from the standard in various ways, both by omitting some standard features and by introducing proprietary features of its own. Again, consider any SQL product available to you. Identify as many departures from the standard in that product as you can. (The previous exercise might give you some clues in this regard.)

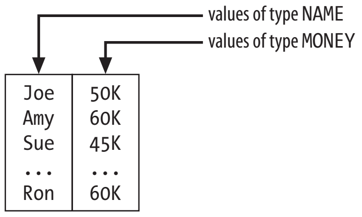
APPENDIX A

The Relational Model

I BELIEVE QUITE STRONGLY THAT, IF YOU THINK ABOUT THE ISSUE AT THE APPROPRIATE LEVEL OF ABSTRACTION, YOU'RE INEXORABLY LED TO THE POSITION THAT DATABASES MUST BE RELATIONAL. Let me immediately try to justify this strong claim! My argument goes like this:

- As I explained in Chapter 5, a database, despite the name, isn't really just a collection of data; rather, it's a collection of facts, or in other words *true propositions*—for example, the proposition "Joe's salary is 50K."
- Propositions like "Joe's salary is 50K" are easily encoded as *ordered pairs*—e.g., the ordered pair (Joe,50K), in the case at hand (where "Joe" is a value of type NAME, say, and "50K" is a value of type MONEY, say).
- But we don't want to record just any old propositions; rather, we want to record all propositions that are true instantiations of certain *predicates*. In the case of "Joe's salary is 50K," for example, the predicate is "*x*'s salary is *y*," where *x* is a value of type NAME and *y* is a value of type MONEY.
- In other words, we want to record the *extension* of the predicate "*x*'s salary is *y*," which we can do in the form of a set of ordered pairs.

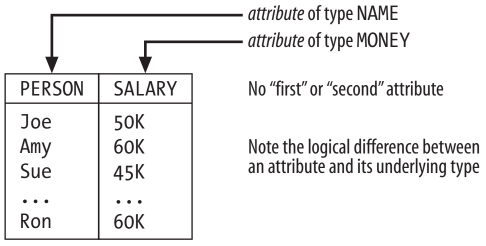
- But a set of ordered pairs is, precisely, a binary relation, in the mathematical sense of that term. Here's the definition: A binary relation over two sets *A* and *B* is a subset of the cartesian product of *A* and *B*; in other words, it's a set of ordered pairs (*a,b*), such that the first element *a* is a value from *A* and the second element *b* is a value from *B*.
- A binary relation in the foregoing sense can be depicted as a *table*. Here's an example:



  (Incidentally, this particular example is not just a relation but a *function*, because each person has just one salary. A function is a special case of a binary relation.) So we can regard this picture as depicting a subset of the cartesian product of the set of all names ("type NAME") and the set of all money values ("type MONEY"), in that order.

Given the argument so far, we can see that we're talking about some fairly humble (but very solid) beginnings. However, in 1969, Codd realized that:

- We need to deal with *n-adic*, not just dyadic, predicates and propositions (e.g., "Joe has salary 50K, works in department D4, and was hired in 1993"). So we need to deal with *n-ary* relations, not just binary ones, and *n-tuples* (*tuples* for short), not just ordered pairs.
- Left to right ordering might be acceptable for pairs but soon gets unwieldy for $n > 2$; so let's replace that ordering concept by the concept of *attributes* (identified by name), and let's redefine the relation concept accordingly. The example now looks like this:


attribute of type NAME
attribute of type MONEY
PERSON SALARY
Joe 50K
Amy 60K
Sue 45K
... ...
Ron 60K
No "first" or "second" attribute
Note the logical difference between an attribute and its underlying type

  From this point forward, then, you can take the term *relation* to mean a relation in the above sense, barring explicit statements to the contrary.

- Data representation alone isn't the end of the story—we need *operators* for deriving further relations from the given ("base") ones, so that we can do queries and the like (e.g., "Get all persons with salary 60K"). But since a relation is both a logical construct (the extension of a predicate) and a mathematical one (a special kind of set), we can apply both logical and mathematical operators to it. Thus, Codd was able to define both a *relational calculus* (based on logic) and a *relational algebra* (based on set theory). And the relational model was born.

The Relational Model vs. Others

Perhaps you can now see why it's my opinion that (to repeat something I said in Chapter 5) 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 2000 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 are just not 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, 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; I suspect it does, but 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 also. (Actually, some people might claim there isn't a unique relational model, either. I'll deal with that argument in a few moments.)

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 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 have to do with matters of implementation; 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.

* 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.

So what of the claims to the effect that there are several relational models, too? For example, in his book *Joe Celko's Data and Databases: Concepts in Practice* (Morgan Kaufmann, 1999), the author, Joe Celko, says 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, he goes on to identify what he claims are six "different relational models."

Now, I wrote an immediate response to these claims when I first encountered them. Here's an edited version of what I said at the time:

> 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 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 (in his introduction to *Readings in Database Systems*, Second Edition, Morgan Kaufmann, 1994) 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 *Databases, Types, and the Relational Model: The Third Manifesto*,* 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.

To all of the above I'd now like to add another point, which I think clearly refutes Celko's original argument. I agree there are several different geometries. But the reason why those geometries are all different is: *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, it 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 logical difference between relations and relvars.
- 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 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 and semijoin).
- We've added the concept of image relations.
- We have a better understanding of updating, including view updating 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).

* See Appendix D of the present book.

The Relational Model Defined

Now I'd like to give a precise definition 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.*) Now, 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 reasonably succinct, as well as being 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, divide, nulls, the entity integrity rule, the idea of being forced to choose one key and make it primary, and the idea that domains and types might somehow be 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 ten years without any notion of nulls at all. What's more, early implementations managed perfectly well without them, too.
- The original 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 key and foreign key constraints), relation variables, relational comparisons, relation type inference and associated features, image relations, 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 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 key and foreign key constraints. Relational comparisons too were at least implicitly required.)

Without further ado, then, let me give my own definition.

> **Definition:** The relational model consists of five components:
>
> - An open-ended collection of scalar types, including type BOOLEAN in particular
> - A relation type generator and an intended interpretation for relations of types generated thereby
> - Facilities for defining relation variables of such generated relation types

- A relational assignment operator for assigning relation values to such relation variables
- 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 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 set of system defined scalar type is required to include 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 "=" must be available in connection with every type, nonscalar as well as scalar, for without it we couldn't even say what the values are 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, then *v1* = *v2* returns TRUE if *v1* and *v2* are the very same value and FALSE otherwise.

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. The intended interpretation for a given relation of a given type, in a given context, is as a set of propositions; each such proposition (a) constitutes an instantiation of some predicate that corresponds to the relation heading, (b) is represented by a tuple in the relation body, and (c) is assumed to be true. 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. Relvars in particular are subject to *The Closed World Assumption* (see later in this appendix).

Let *RT* be some relation type. Associated with *RT*, then, there's a relation selector operator, with the properties that (a) every invocation of that operator returns a relation of type *RT* and (b) every relation of type *RT* is returned by some invocation of that operator. Also, since the equality comparison operator "=" is available in connection with every type, it's available in connection with type *RT* in particular. So too is the relational inclusion

* I should remind you that (a) the distinction between scalar and nonscalar is necessarily a somewhat informal one, and (b) the relational model therefore certainly doesn't rely on it in any formal sense. See Chapter 2 for further discussion.

operator ("⊆"); if relations *r1* and *r2* are of the same type, then *r1* is included in *r2* if and only if the body of *r1* is a subset of that of *r2*.

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. Such a variable is the only kind permitted in a relational database; in particular, scalar and tuple variables are prohibited. (In programs that access such a database, by contrast, they're not prohibited—in fact, they're probably required.)

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. Any violation of it thus has to be seen as serious. Database tables that involve top to bottom row ordering or left to right column ordering, or contain duplicate rows, or pointers, or nulls, or have anonymous columns or duplicate column names, all constitute such violations. But why is the principle so important? The answer is bound up with the observations I made in Chapter 5 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.

In fact, 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 (and choose among). 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, *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)—*of spartan simplicity*: It has no ordering to its tuples, it has no ordering to its attributes, it has no duplicate tuples, it has no pointers, and (at least as far as I'm concerned) it has 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 the relational model has just one.

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.)

There's one more point I should mention under the heading of "Relation Variables." As Darwen and I demonstrate in our book on *The Third Manifesto*, the database isn't really just "a container for relvars," despite the fact that we talk about it most of the time as if it were. Rather, the database itself is, logically, one (possibly rather large) variable in itself, which we might call a *dbvar*. Thus, "relation variables" aren't really variables as such; instead they're what in that same book we call *pseudovariables*. Their purpose is to provide users with the illusion that they can update the database—or the dbvar, rather—in a piecemeal fashion instead of en bloc (much as we talk about updating an individual tuple within a relvar instead of updating that entire relvar en bloc).

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 values 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 (a) must include support for *multiple* relational assignment in particular and (b) must abide by both *The Assignment Principle* and **The Golden Rule**.

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 able to define additional operators of their own, if desired). Precisely which operators are included isn't specified, but they're required to provide, in their totality, at least the expressive power of the relational calculus; in other words, they're required to be *relationally complete* (see further discussion below).

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,

* What's more, that eight ought by rights 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. What makes matters worse is that SQL doesn't support the true multiset union operator anyway; the operator it calls "multiset union" corresponds to what's called "union plus" in the literature. See the paper "The Theory of Bags: An Investigative Tutorial" (mentioned in Appendix D) for further explanation.

including query but certainly not limited to query alone. 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 will be just a subexpression of some boolean expression, frequently though not invariably an IS_EMPTY invocation)
- Serving as a basis for investigations into other areas, such as optimization and database design

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 (or the calculus—it comes to the same thing), 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 appendix 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 database administrator
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 collect them together for ease of reference. *Note:* The list isn't meant to be exhaustive. In particular, I've omitted the principles of normalization and other database design principles, because they weren't discussed in the body of the book (but see Appendix B).

The Information Principle (also known as *The Principle of Uniform Representation* or *The Principle of Uniformity of Representation*)

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* appears in relvar *R* at some given time, then the proposition *p* corresponding to *t* is assumed to be true at that time. Conversely, 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 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 this principle earlier in the book, but I appealed to it tacitly on

* So here we have another reason—a philosophical reason, if you like—for rejecting the notion of duplicates.

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* above)—and numerous benefits accrue from this simple fact.

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 the body of the book that it's being extremely charitable to SQL to describe it as a relational language. It follows 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 I wrote this book, and it's also 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 below 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

* In this connection, we'd also like to see an implementation that's more sophisticated in certain respects than current SQL implementations typically are. To be specific, we'd like to see an implementation based on what's called *The TransRelational™ Model* (see the section "Some Remarks on Physical Design" in Appendix B).

difficult than doing it right. 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 24th, 1997).

Foundations

There's still much interesting work to be done on theoretical foundations (in other words, 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. To what extent 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?
- Can we come up with a good way—that is, a way that's robust, logically sound, and ergonomically satisfactory—of dealing with the "missing information" problem?

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 5 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 belong naturally together. In a sense, they make it easier to see the forest as well as the trees.

By way of another illustration, consider the relational algebra. I showed in Chapters 6 and 7 that many of the operators of the algebra—including ones we use all the time, even if we don't realize it, like semijoin—are really shorthand for certain combinations of other operators.* Indeed, there are other useful operators that I didn't discuss in those chapters 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 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* (see Appendix D), Hugh Darwen, Nikos Lorentzos, and I—building on original work by

* As a matter of fact, Darwen and I show in our *Manifesto* book that every algebraic operator discussed in this book can be expressed in terms of just two primitives, *remove* (which is basically project) and either *nand* or *nor*.

Lorentzos—introduce *interval types* as a basis for supporting temporal data in a relational framework. For example, consider the "temporal relation" in Figure A-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*]

FIGURE A-1. A relation with an interval attribute

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 the relational model, and that's by means of various kinds of applications that run on top of 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 front end. For the users of such applications, the relational model will disappear 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 front end, 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 front end applications. Which would you prefer as a target?—a relational DBMS, or some other kind (an object oriented DBMS, say)? And if you opt for the former, as I obviously think you should, which would you prefer?—a DBMS that supports the relational model as such, 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 front end 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.

To repeat, 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; suffice it to say, however, that we believe we can simulate various nonrelational features of SQL, such as duplicates and nulls, without having to support such concepts directly in the underlying relational language.

APPENDIX B

Database Design Theory

THE GOAL OF PHYSICAL DATA INDEPENDENCE, DISCUSSED IN CHAPTER 1, has the direct consequence that logical and physical database design are different disciplines—logical design is concerned with what the database looks like to the user, physical design is concerned with how the logical design maps to physical storage. But the term *database design theory* is used almost exclusively in connection with logical design, not physical design (the point being that physical design is necessarily dependent on details of the target DBMS, whereas logical design is or should be DBMS independent). For that reason, in this appendix 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" (total) relvar constraint for relvar *R* can be regarded as the system's approximation to the relvar 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 (as noted in Chapter 8, 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 E/R ("entity/relationship") 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 admittedly important, but limited, constraints. Thus, while it might be acceptable 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 want to make up front. Recall from Chapter 9 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, the user interacts not with a database that contains base relvars only (the "real" database), but rather with what might be called a *user database* that contains some mixture of base relvars and views. But that user database is supposed to look and feel like the real database to that user; thus, all of the design principles to be discussed in this appendix 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 draft versions of this appendix 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 it's not my intent to explain the design process as it's actually carried out in practice; rather, my intent 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 want to spend a lot of time covering what should be familiar territory. For example, I deliberately don't plan to 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* be discussing).

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 relational 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 is a property of suppliers, not shipments). Figure B-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 B-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 tells us also what the consequences of such redundancy are. 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 appendix 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 *n*th normal form.
- It's possible for a relvar to be in *n*th 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'm not going to 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 be describing.

To repeat, I assume you already know something about these matters. As noted earlier, however, I want to focus on aspects of the subject that you might perhaps not be so familiar with; I want to try and 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 of this concept:

> **Definition:** Let *A* and *B* be subsets of the heading of relvar *R*. Then relvar *R* satisfies the *functional dependency* (FD) *A* → *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* → *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 B-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.

As the example indicates, the fact that some relvar *R* satisfies a given FD constitutes a database constraint in the sense of Chapter 8; more precisely, it constitutes a (single-relvar)

* The terms *dependence* and *dependency* are used interchangeably in the literature, and in this book.

RS

SNO	SNAME	STATUS	CITY	
S1	Smith	20	London	
S2	Jones	30	Paris	← *note the change*
S3	Blake	30	Paris	
S4	Clark	20	London	
S5	Adams	30	Athens	

FIGURE B-2. Revised suppliers relvar RS—sample value

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* → *B*, it necessarily satisfies the FD *A′* → *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 an FD that's satisfied by the relvar in question.

At this point I need to remind you of one term and introduce another. The first is *superkey*. Recall from Chapter 5 that a superkey is basically just a superset of a key (not necessarily a proper superset);* equivalently, 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*. Recall too that if *SK* is a superkey for *R* and *A* is any subset of the heading of *R*, *R* necessarily satisfies the FD *SK* → *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 must have the same value for *A*.

The term I need to introduce is *trivial FD*. Basically, an FD is *trivial* if and only 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 STATUS and CITY:

```
{ CITY , STATUS } → { CITY }
{ CITY , STATUS } → { STATUS }
{ CITY }          → { CITY }
```

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). Now, 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.

* Recall too that I use the term *key* to mean a candidate key specifically.

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 I can now 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 B-2 satisfies the FD {SNO} → {STATUS}; but if we decompose it—as I'm going to recommend we do in just a moment—into relvars SNC and CS (where SNC has attributes SNO, SNAME, and CITY and CS has attributes CITY and STATUS), then 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? The answer is: 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 appendix.

Finally, as I'm sure you know, the normalization discipline says: If relvar *R* isn't in BCNF, then decompose it into smaller ones that are (where "smaller" means, basically, having fewer attributes). For example:

- Relvar STP (see Figure B-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 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. *Exercise:* Show sample values for relvars SP and SS corresponding to the STP value in Figure B-1; convince yourself that SP and SS are in BCNF and that the decomposition eliminates the redundancy.
- Similarly, relvar RS (see Figure B-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). *Exercise:* Show sample values for SNC and CS corresponding to the RS value in Figure B-2; convince yourself that SNC and CS are in BCNF and that the decomposition eliminates the redundancy.

Nonloss Decomposition

If some relvar isn't in BCNF, it can and 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 (Figure B-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 B-3. (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 in that decomposition 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 B-3. Relvars SNS and CS—sample values

What exactly is it that makes some decompositions nonloss and others lossy? Well, note first 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 (and I'll use the term *decomposition* in this sense exclusively, until further notice). *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 back together again, we get back to the original relvar*. Observe in particular that, with reference to Figure B-3, relvar RS isn't equal to the join of its projections SNS and CS, and that's why the decomposition is lossy. With reference to Figure B-2, by contrast, it is equal to the join of its projections SNC and CS, and so 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, albeit incomplete, 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 theory union of *A*, *B*, and *C* is equal to that heading. Let *AB* denote the set theory union of *A* and *B*, and similarly for *AC*. If *R* satisfies the FD *A* → *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 B-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 answer to the original question was "incomplete," let me explain in terms of the foregoing example. Basically, the theorem does tell us that the decomposition of Figure B-2 is nonloss; by contrast, however, it doesn't tell us that the decomposition of Figure B-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 B-18 at the end of this appendix.)

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 or *n*-adic 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 basically all just common sense. Consider relvar STP again, for example (Figure B-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 explicit knowledge of BCNF whatsoever. But what does "naturally" mean here? What principles are being applied by the designer 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 always 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 earlier, in fact, I won't even 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. Now, 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. I don't want to get into a lot of detail on this issue here; let me just say that relvars with RVAs tend to look very much like the hierarchic structures 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.
- 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.
- 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.

Exercises 7-10, 7-12, and 7-13 at the end of Chapter 7 provide examples of some of these points. To repeat, however, RVAs can occasionally be useful, even in base relvars. See Exercise B-14 at the end of this appendix.

* Strictly speaking, it's relations, not relvars, that are always in 1NF (see Chapters 1 and 2); i.e., first normal form is a property of relations, not relvars. But no harm is done if we extend the concept to apply to relvars as well.

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*, ..., *C* be subsets of the heading of relvar *R*. Then *R* satisfies the *join dependency* (JD)
>
> ☆ { *A* , *B* , ... , *C* }
>
> (pronounced "star *A*, *B*, ..., *C*") if and only if every relation that's a legal value for *R* is equal to the join of its projections on *A*, *B*, ..., *C*.

Points arising from this definition:

- It's immediate that *R* can be nonloss decomposed into its projections on *A*, *B*, ..., *C* if and only if it satisfies the JD ☆{*A,B,...,C*}.
- It's also immediate that every FD is a JD, because (as Heath's theorem tells us) 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 B-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, if relvar *R* satisfies the JD ☆{*A,B,...,C*}, then that JD is *implied by superkeys* if and only if each of *A*, *B*, ..., *C* is a superkey for *R*.* For example, consider our usual suppliers relvar S. That relvar can be nonloss decomposed into its projections on SNO and SNAME, on SNO and STATUS, and on SNO and CITY; in other words, it satisfies the JD

```
☆ { SN , SS , SC }
```

where SN is {SNO,SNAME}, SS is {SNO,STATUS}, and SC is {SNO,CITY}. Since each of these is clearly a superkey for S, the JD is indeed implied by superkeys. (Whether we

* This definition of what it means for a JD to be implied by superkeys is valid so long as there's just one key. It can be generalized to take care of the case of two or more, but the details are a little messy; I therefore choose to omit them from this discussion. A precise and complete definition can be found in *The Relational Database Dictionary, Extended Edition* (see Appendix D).

would actually want to perform the corresponding decomposition is another matter. We know we could if we wanted to, that's all.)

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 ☆{*A,B,...,C*} is *trivial* if and only if at least one of *A*, *B*, ..., *C* is 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 }
```

NOTE

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 to the identity projection of the relvar S, i.e., the projection of that relvar on all of its attributes). By the way, I hope it's clear 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, as such, 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 the 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

Now, I'm sure you noticed that I didn't show an example, in the foregoing discussion of JDs and 5NF, of a relvar that was in BCNF but not in 5NF (and so could be nonloss decomposed to advantage). The reason I didn't is this: While JDs do exist that aren't just simple FDs, (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, though); 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, 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 in at least BCNF, are in fact 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, if you can get to BCNF (which is easy enough), and if there aren't any composite keys in your BCNF relvar (which is often but not always the case), then you don't have to worry about the complexities of JDs and 5NF in general—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 interest 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.

Furthermore, there's another simple theorem that addresses the same issue:

Let *R* be a BCNF relvar and let *R* have at least one nonkey attribute (that is, an attribute not participating in any key of *R*). Then *R* is in 5NF.

This theorem, like the previous one, describes a situation that arises extremely frequently in practice.

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 the 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 current design theory simply doesn't address. By way of example, consider Figure B-4, which shows a sample value for a relvar, SPJ, that's in 5NF and yet suffers from redundancy. For instance, 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

* Actually, what I'm claiming as a "consequence" here might not be valid if *R* has more than one key. The argument can be generalized to take care of this case as well, but again the details are a little messy; I therefore choose to omit them from this discussion. Again, further specifics can be found in *The Relational Database Dictionary, Extended Edition* (see Appendix D).

that project J1 is supplied by supplier S2. (The predicate is *Supplier SNO supplies part PNO to project JNO in quantity QTY*, and the sole key is {SNO,PNO,JNO}.) The only nontrivial JD 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 so it can't appear in a relvar with anything less than all three; thus, there's no nonloss decomposition that can remove the redundancies.

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 B-4. The 5NF relvar SPJ—sample value

There are a few further points I need to make here. First, I didn't mention the fact previously, but you probably know that 5NF is always achievable; that is, it's always possible to decompose a non5NF relvar (in a nonloss way) into 5NF projections.

Second, every 5NF relvar is in BCNF; 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 appendix (and I'll 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 it's certainly possible to consider other operators and therefore, perhaps, 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. These developments turn out to be 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 (see Appendix D). However, all I want to do here is give a definition of 6NF that works for "regular" (that is, nontemporal) relvars. Here it is:

* Equivalently, using the name SPJQ to refer to the set of all attributes of SPJ, the relvar satisfies the JD ☆{SPJQ}, which is both trivial and implied by the superkeys of *r*. (Recall from Chapter 6 that the join of a single relation *r* is just *r* itself.)

Definition: Relvar *R* is in *6NF* if and only if it satisfies no nontrivial JDs at all.

In other words, a relvar in 6NF can't be nonloss decomposed at all, other than trivially.* In particular, a "regular" relvar is in 6NF if and only if it's in 5NF, is of degree *n*, and has no key of degree less than *n* – 1 (note that every 6NF relvar is necessarily in 5NF). Our usual shipments relvar SP is in 6NF, as is relvar SPJ (see Figure B-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, other than trivially.

To close this subsection, observe that it does *not* follow that a relvar, just because it's in 5NF or 6NF, is necessarily well designed. For example, if relvar RS (see Figure B-2) satisfies the FD {CITY} → {STATUS}, its projection on {SNO,STATUS} is clearly in 6NF, but it certainly isn't a good choice for a base relvar. See the discussion of what's called "dependency preservation" in the section "Two Cheers for Normalization," later, for a more detailed explanation.

More on 5NF

Consider Figure B-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? Well:

SPJ

SNO	PNO	JNO
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

FIGURE B-5. Simplified relvar SPJ—sample value

- First of all, the JD means by definition that the relvar is equal to the join of, and so can be nonloss decomposed into, its projections SP, PJ, and SJ. (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 this 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 B-5, for example, the tuples (S1,P1), (P1,J1), and (S1,J1) appear in SP, PJ, and SJ, respectively, and the tuple

* A decomposition is trivial if and only if it's based on dependencies that are themselves trivial in turn.

(S1,P1,J1) appears in SPJ. *Note:* 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 "[is] in" or "appears in."

- Now, the tuple (*s,p*) 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 B-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 as well:

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 if such a cyclic constraint occurs that we might have a relvar that's in BCNF and not in 5NF.* In my experi-

* If the constraint takes the slightly simpler form "if *s* is connected to *p* and *j* is connected to *s*, then *s* and *p* and *j* must all be directly connected," then we might have a relvar that's in BCNF but not in *4NF* (and hence not in 5NF either, a fortiori). See Exercise B-16 at the end of the Appendix.

ence, however, such 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 multivalued 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 the present discussion (see Exercise B-19 at the end of the appendix)—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 taking projections, and we've seen that not all redundancies can be removed by taking projections; indeed, there are some kinds of redundancy, not discussed in this appendix 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 first of these FDs implies that {STREET,CITY,STATE} is a key; the second implies that the relvar isn't in BCNF. However, if we apply Heath's theorem and decompose it, in a nonloss way, into BCNF projections as follows (take *A* as {ZIP}, *B* as {CITY,STATE}, and *C* as {STREET})—

```
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 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 B-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 foregoing 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 B-6. Projections ZCS and ZS—sample values

ASIDE

As Chapter 9 shows, however, if we do in fact perform the foregoing decomposition after all, we could at least define the join of projections ZCS and ZS as a view and then define {ZIP,CITY,STATE} as a key for that view; in other words, we might at least be able to state the pertinent constraint in a comparatively straightforward manner.

Avoid update anomalies: This point is effectively just the previous one ("reduce redundancy") by another name. It's well known that designs that are less than fully normalized can be subject to certain update anomalies, precisely because of the redundancies they entail. In relvar STP, for example (see Figure B-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 constraint enforcement. 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). Yet another way to characterize it is: More single-tuple updates are logically acceptable if the design is fully normalized than would be the case if it isn't (because unnormalized designs imply redundancy, and redundancy implies that sometimes we have to update several things at once).

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. 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, except perhaps by way of documentation). 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.

Here on the other hand are some further reasons, over and above those already given, 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 nonloss 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. 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.

All of that being said, I must make it clear that I don't want my comments in this section to be seen as any sort of 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, that is, 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 failed, somehow, 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, in 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 certain 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 execute, 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 B-2), with its FD {CITY} → {STATUS}. That relvar can be regarded as the result of denormalizing relvars SNC (with attributes SNO, SNAME, and CITY) and CS (with attributes CITY and STATUS). Now consider the query "Get the average city status." Given the sample values in Figure B-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 then are some attempts at formulating this query in SQL:

1.
```
SELECT AVG ( 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 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 CITY, AVG ( STATUS ) AS RESULT
FROM   RS
GROUP BY 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 CITY, AVG ( AVG ( STATUS ) ) AS RESULT
FROM   RS
GROUP BY CITY
```

   Result: Syntax error. The SQL standard quite rightly doesn't allow "set function" invocations to be nested in this manner.* One more attempt:

* I say "quite rightly" only because we're in the SQL context specifically; a more orthodox language, such as **Tutorial D**, 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 surrounding expression are part of the AVG argument and which are part of the SUM argument.

5.
```
SELECT AVG ( TEMP.STATUS ) AS RESULT
FROM ( SELECT DISTINCT RS.CITY, RS.STATUS
       FROM   RS ) AS TEMP
```

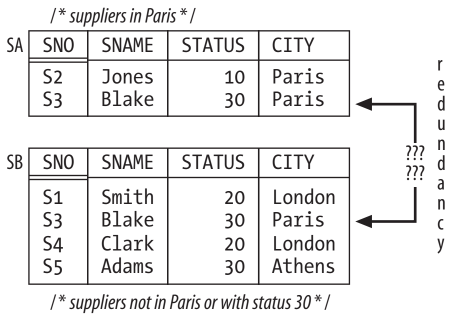
Result (correct at last): 26.667. But note how complicated this expression is compared to its analog on the fully normalized design of Figure B-3:

```
SELECT AVG ( STATUS ) AS RESULT
FROM   CS
```

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 B-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 can lead to update anomalies. (Redundancy of any kind can *always* lead to update anomalies.)



SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

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

FIGURE B-7. Relvars SA and SB—sample values

By the way, note that the tuple for supplier S3 certainly 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 (and we know from Chapter 8 the problems that can cause).

Well, the problem with the design of Figure B-7 is obvious: It's precisely the fact that the very same tuple has to appear in two distinct relvars, or in other words the fact that the meanings or predicates for the relvars concerned "overlap," as it were. Indeed, we can and probably should write a constraint to that effect:

```
CONSTRAINT SA_AND_SB_OVERLAP
  ( SA WHERE STATUS = 30 ) =
  ( SB WHERE STATUS = 30 AND CITY = 'Paris' ) ;
```

So an obvious rule is:

> *The Principle of Orthogonal Design (first version):* No two distinct relvars should be such that if some tuple *t* appears in one of them, then their relvar constraints require that tuple *t* to appear in the other one as well.

The term *orthogonal* here derives from the fact that what the principle effectively says is that relvars should be independent of one another—which they won't be, if their meanings overlap in the foregoing sense.

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 B-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 certainly 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:

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 B-8. Relvars SX and SY—sample values

> *The Principle of Orthogonal Design (second version):* Let *A* and *B* be distinct relvars. Then there must not exist nonloss decompositions of *A* and *B* into (say) *A1*, *A2*, ..., *Am* and *B1*, *B2*, ..., *Bn*, respectively, such that some projection *Ai* in the set *A1*, *A2*, ..., *Am* and some projection *Bj* in the set *B1*, *B2*, ..., *Bn* have relvar constraints that require that, if some tuple *t* appears in *Ai*, then that tuple *t* must appear in *Bj* as well.

This second version of the principle subsumes the first, because one "nonloss decomposition" that's always available for any relvar *R* is the one that consists of just the identity projection of *R*. But it's still not watertight. For the record, I'll give what I believe to be a watertight definition here, but I don't propose to elaborate on that definition any further in this appendix.

* 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 as well.

The Principle of Orthogonal Design (final version): Let *A* and *B* be distinct relvars. Replace *A* and *B* by nonloss decompositions into projections *A1*, *A2*, ..., *Am* and *B1*, *B2*, ..., *Bn*, respectively, such that every *Ai* (*i* = *1*, ..., *m*) and every *Bj* (*j* = *1*, ..., *n*) is in 6NF. Let some *i* and *j* be such that there exists a sequence of zero or more attribute renamings with the property that (a) when applied to *Ai*, it produces *Ak*, and (b) *Ak* and *Bj* are of the same type. Then there must not exist a constraint to the effect that, at all times, (*Ak* WHERE *ax*) = (*Bj* WHERE *bx*), where *ax* and *bx* are restriction conditions, neither of which is identically FALSE.

Several points arise from the foregoing discussion:

- 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 also. Again consider the decomposition of relvar S into its projections SX and SY, as illustrated in Figure B-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.
- 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 should ever appear in more than one of them. (Also, the union of those restrictions—which in fact will be a disjoint union—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 deliberately has nothing to say about physical design. But there are still some things that can usefully be said about that topic 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).

* 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,STATUS}, {SNO,SNAME}, and {SNO,STATUS,CITY} as a sensible design, even though S is certainly equal to the join of those three projections, because the first of those three isn't needed in the reconstruction process.

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 in a few moments.)

As for my second point: I claimed 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 regard; in those products, what the user sees and what's physically stored are essentially identical. In other words, what's physically stored in these products is effectively just a *direct image* of what the user logically sees, as Figure B-9 suggests. (I realize these remarks are oversimplified, but they're true enough for present purposes.)

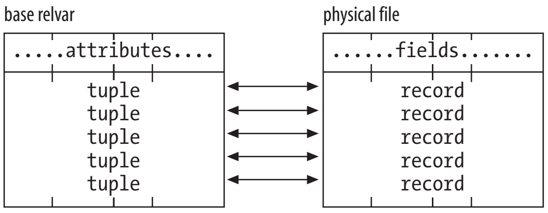


FIGURE B-9. Direct image implementation (deprecated)

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 (typically for performance reasons), 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* (see Appendix D). 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 whatsoever. The logical design really would have no performance implications at all.

Second, 6NF in particular 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 B-10.

SN

SNO	SNAME
S1	Smith
S2	Jones

SS

SNO	STATUS
S1	20

SC

SNO	CITY
S1	London
S2	Paris

FIGURE B-10. The status for supplier S2 is unknown

Of course, there's a lot more that needs to be said about this approach to missing information, but this appendix isn't the place. (The paper "The Closed World Assumption," mentioned in Appendix D, goes into more details.) Here I just want to stress the point that with this design, we don't have a "tuple" showing that supplier S2's status "is null"—we don't have a tuple showing supplier S2's status *at all*.

Last: 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.

Concluding Remarks

The main focus of this appendix 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, 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 is that logical design should generally aim to be application independent, too, as well as being 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

* 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.

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 it's constraints that 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. (Some of those constraints will be FDs, MVDs, or JDs in particular.)

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, it makes more single-tuple updates logically acceptable than would otherwise be the case. I explained that normalization is really formalized common sense. I also gave a logical and possibly unfamiliar argument, having to 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, we really need 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 here about "denormalizing for performance"). Well, the same is true of orthogonality also—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 appendix, database design usually involves tradeoffs and compromises.

Exercises

Exercise B-1. Give definitions, as precise as you can, of *functional dependency* and *join dependency*.

Exercise B-2. List all of the FDs, trivial as well as nontrivial, satisfied by the shipments relvar SP.

Exercise B-3. The concept of FD relies on the notion of tuple equality: True or false?

Exercise B-4. 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*, or "spurious," tuples. Since we have no way in general of knowing which tuples in the join are spurious and which genuine, we've lost information.)

Exercise B-5. Prove Heath's theorem. Prove also that the converse of that theorem isn't valid.

Exercise B-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 B-7. Keys are supposed to be unique and irreducible. Now, the system is clearly capable of enforcing uniqueness; but what about irreducibility?

Exercise B-8. What's (a) a trivial FD, (b) a trivial JD? Is the former a special case of the latter?

Exercise B-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 B-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 B-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 B-12. Most of the examples of nonloss decomposition in the body of the appendix showed a relvar being decomposed into exactly two projections. Is it ever necessary to decompose into three or more?

Exercise B-13. Many database designers recommend the use of artifical or *surrogate* keys in base relvars in place of what are sometimes called "natural" 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 key for that relvar. (Note, however, that {SNO,PNO} would still be a key; it just wouldn't be the only one 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 (i.e., 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. Two questions: Are surrogate keys the same thing as tuple IDs? And do you think they're a good idea?

Exercise B-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 B-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 B-16. Let *A*, *B*, and *C* be subsets of the heading of relvar *R* such that the set theory union of *A*, *B*, and *C* is equal to that heading. Let *AB* denote the set theory union of *A* and *B*, and similarly for *AC*. Then *R* satisfies the *multivalued dependencies* (MVDs)

$$A \rightarrow\!\!\!\rightarrow B$$
$$A \rightarrow\!\!\!\rightarrow C$$

(where $A \rightarrow\!\!\!\rightarrow B$ is pronounced "*A* double arrow *B*" or "*A* multidetermines *B*" or "*B* is multidependent on *A*," and similarly for $A \rightarrow\!\!\!\rightarrow C$) if and only if *R* satisfies the JD ☆{*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 B-17. Show that if *R* satisfies the FD $A \rightarrow B$, it also satisfies the MVD $A \rightarrow\!\!\!\rightarrow B$.

Exercise B-18. *(Fagin's Theorem.)* Let *R* be as in Exercise B-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 B-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 B-20. Give an example of a relvar that's in BCNF and not 4NF.

Exercise B-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 B-22. Write a **Tutorial D** CONSTRAINT statement to express the JD satisfied by relvar SPJ of Figure B-5.

Exercise B-23. *(Modified version of Exercise B-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*, and product *p* is sold in area *a*, and representative *r* sells product *p*, then *r* sells *p* in *a*.

Exercise B-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 B-25. There's a lot of discussion in the industry at the time of writing of 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 appendix apply to XML databases? Justify your answer.

Exercise B-26. Draw E/R diagrams for the databases from Exercises B-11, B-14, B-21, and B-23. What do you conclude from this exercise?

APPENDIX C

Answers to Exercises

Chapter 1

Exercise 1-1. Here are a few examples of statements from the early part of the chapter where the term *relation* should be replaced by the term *relvar*:

- "Every relation has at least one candidate key."
- "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 the same relation)."
- "[The] relational assignment operator…allows the value of some relational expression…to be assigned to some relation."
- "A view (also known as a virtual relation) is a named relation whose value at any given time *t* is the result of evaluating a certain relational expression at that time *t*."

And so on.

Exercise 1-2. E. F. Codd (1923–2003) was the original inventor of the relational model, among many other things. In December 2003 I published a brief tribute to him and his achievements, which you can find on the ACM SIGMOD website *http://www.acm.org/sigmod* and elsewhere. An expanded version of that tribute appears in my book *Date on Database: Writings 2000-2006* (Apress, 2006).

Exercise 1-3. A domain can be thought of as a conceptual pool of values from which actual attributes in actual relations take their actual values. In other words, a domain is a type, and the terms *domain* and *type* are effectively interchangeable—but personally I much prefer *type*, as having a longer pedigree (in the computing world, at least). *Domain* is the term used in most of the older database literature, however. *Note:* Don't confuse domains as understood in the relational world with the construct of the same name in SQL, which can be regarded at best as a very weak kind of type. See Chapter 2 (in particular, the answer to Exercise 2-1).

Exercise 1-4. A database satisfies the referential integrity rule if and only if for every tuple containing a *reference* (in other words, a foreign key value) there exists a *referent* (in other words, a tuple in the pertinent referenced relvar with that same value for the pertinent candidate key). Loosely: If *B* references *A*, then *A* must exist. See Chapters 5 and 8 for further discussion.

Exercise 1-5. Let *R* be a relvar and let *r* be the relation that's the value of *R* at some particular time. Then the heading, attributes, and degree of *R* are defined to be identical to the heading, attributes, and degree of *r*, respectively. Likewise, the body, tuples, and cardinality of *R* are defined to be identical to the body, tuples, and cardinality of *r*, respectively. Note, however, that the body, tuples, and cardinality of *R* vary over time, while the heading, attributes, and degree don't.

By the way, it follows from the foregoing that if we use SQL's ALTER TABLE to add a column to or drop a column from some base table *T*, the effect is to replace that table *T* by some logically distinct table *T′* (the term *table* being, in such contexts, SQL's counterpart to the relational term *relvar*). *T′* is *not* "the same table as before"—speaking purely from a logical point of view, that is. But it's convenient to overlook this nicety in informal contexts.

Exercise 1-6. See the section "Model vs. Implementation" in the body of the chapter.

Exercise 1-7. Physical data independence is the independence of users and application programs from the way the data is physically stored and accessed. It's a logical consequence of keeping a rigid separation between the model and its implementation. To the extent that such separation is observed, and hence to the extent that physical data independence is achieved, we have the freedom to make changes to the way the data is physically stored and accessed—probably for performance reasons—without at the same time having to make corresponding changes in queries and application programs. Such independence is desirable because it translates into protecting investment in training and applications.

The model is the abstract machine with which users interact; the implementation is the realization of that abstract machine on some physical computer system. Users have to understand the model, since it defines the interface they have to deal with; they don't have to understand the implementation, because that's under the covers (at least, it should be). The following analogy might help: In order to drive a car, you don't have to know what goes on under the hood—all you have to know is how to steer, how to shift gear, and so on. So the rules for steering, shifting, and the rest are the model, and what's under the hood is the implementation. (It's true that you might drive better if you have some understanding of what goes on under the hood, but you don't have to know. Analogously, you might use a data model better if you have some knowledge of how it's implemented—but ideally, at least, you don't have to know.) *Note:* The term *architecture* is sometimes used with a meaning very similar to that of *model* as defined above.

Exercise 1-8. Rows in tables are ordered top to bottom but tuples in relations aren't; columns in tables are ordered left to right but attributes in relations aren't; tables might have duplicate rows but relations never have duplicate tuples. Also, relations contain values, but tabular pictures don't (they don't even contain "occurrences" or "appearances" of such values); rather, they contain symbols that denote such values—for example, the symbol (or numeral) 5, which denotes the value five. See the answer to Exercise 3-5 in Chapter 3 for several further differences.

Exercise 1-9. *No answer provided.*

Exercise 1-10. Throughout this book I use the term *relational model* to mean the abstract machine originally defined by Codd (though that abstract machine has been refined, clarified, and extended somewhat since Codd's original vision). I don't use the term to mean just a relational design for some particular database. There are lots of relational models in the latter sense but only one in the former sense. (As noted in the body of the chapter, you can find quite a lot more on this issue in Appendix A.)

Exercise 1-11. Here are some:

- The relational model has nothing to say about "stored relations" at all; in particular, it categorically doesn't say which relations are stored and which not. In fact, it doesn't even say that relations as such have to be stored at all—there might be a better way to do it (and indeed there is, though the specifics are beyond the scope of this book).
- Even if we agree that the term "stored relation" might make some kind of sense—meaning a user visible relation that's represented in storage in some direct and efficient manner, without getting too specific on just what *direct* and *efficient* might mean—which relations are "stored" should be of no significance whatsoever at the relational (user) level of the system. In particular, the relational model categorically does *not* say that "tables" (or "base tables," or "base relations") are stored and views aren't.
- The extract quoted doesn't mention the crucial logical difference between relations and relvars.
- The extract also seems to assume that *table* and *base table* are interchangeable terms and concepts (a very serious error, in my opinion).
- The extract also seems to distinguish between tables and relations (and/or relvars). If "table" means, specifically, an SQL table, then I certainly agree there are some important distinctions to be observed, but they're not the ones the extract seems to be interested in.

- "[It's] important to make a distinction between stored relations…and virtual relations": Actually, it's extremely important from the user's perspective (and from the perspective of the relational model, come to that) *not* to make any such distinction at all!

Exercise 1-12. Here are a few things that are wrong with it:

- The relational model as such doesn't "define tables" at all, in the sense meant by the extract quoted. It doesn't even "define" relations (or relvars, rather). Instead, such definitions are supplied by some user. And anyway: What's a "simple" table? Are there any complex ones?
- What does the phrase "each relation and many to many relationships" mean? What does it mean to "define tables" for such things?
- The following concepts aren't part of the model, so far as I know: entities, relationships between entities, linking tables, "cross-reference keys." (It's true that Codd's original model had a rule called "entity integrity," but that name was only a name, and I reject that rule in any case.) It's also true that it's possible to put some charitable interpretations on all of these terms, but the statements that result from such interpretations are usually wrong. For example, relations don't always represent "entities" (what "entity" is represented by the relation that's the projection of suppliers on STATUS and CITY?).
- Primary and secondary indexes and rapid access to data are all implementation notions—they're nothing to do with the model. In particular, primary or candidate keys shouldn't be equated with "primary indexes."
- "Based upon qualifications"? Would it be possible to be a little more precise? It's truly distressing, in the relational context above all others (where precision of thought and articulation was always a key objective), to find such dreadfully sloppy phrasing. Well, yeah, you know, a relation is kind of like a table, or a kind of a table, or something…if you know what I mean.
- Finally, *what about the operators*? It's an all too common error to think the relational model has to do with structure only and to forget about the operators. But the operators are crucial! As Codd himself once remarked: "Structure without operators is…like anatomy without physiology."

Exercise 1-13. Here are some possible CREATE TABLE statements. Regarding the column data types, see Chapter 2. *Note:* These CREATE TABLE statements, along with their **Tutorial D** counterparts, are repeated in Chapter 5, where further pertinent discussion can also be found.

```
CREATE TABLE S
  ( SNO    VARCHAR(5)   NOT NULL ,
    SNAME  VARCHAR(25)  NOT NULL ,
    STATUS INTEGER      NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( SNO ) ) ;
```

```
CREATE TABLE P
  ( PNO    VARCHAR(6)   NOT NULL ,
    PNAME  VARCHAR(25)  NOT NULL ,
    COLOR  CHAR(10)     NOT NULL ,
    WEIGHT NUMERIC(5,1) NOT NULL ,
    CITY   VARCHAR(20)  NOT NULL ,
    UNIQUE ( PNO ) ) ;

CREATE TABLE SP
  ( SNO    VARCHAR(5)   NOT NULL ,
    PNO    VARCHAR(6)   NOT NULL ,
    QTY    INTEGER      NOT NULL ,
    UNIQUE ( SNO , PNO ) ,
    FOREIGN KEY ( SNO ) REFERENCES S ( SNO ) ,
    FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ) ;
```

Note that SQL encloses the column definitions and the key and foreign key specifications all inside the same set of parentheses (contrast this with what **Tutorial D** does—see Chapter 2). Note too that by default SQL columns permit nulls; if we want to prohibit them, therefore (and I do), we have to specify an explicit constraint to that effect. There are various ways of defining such a constraint; specifying NOT NULL as part of the column definition is probably the easiest.

Exercise 1-14. **Tutorial D** (I can't show this in SQL, because SQL doesn't support relational assignment):

```
SP := SP UNION
         RELATION { TUPLE { SNO 'S5' , PNO 'P6' , QTY 250 } } ;
```

The text between the keyword UNION and the closing semicolon is a *relation selector invocation* (see Chapter 3), and it denotes the relation that contains just the tuple to be inserted. *Note:* In practice, that UNION might better be D_UNION. See Chapter 5.

Exercise 1-15. I'll give an answer here for completeness, but I'll defer detailed explanations to Chapter 7:

```
S := WITH ( S WHERE CITY = 'Paris' ) AS R1 ,
          ( EXTEND R1 ADD ( 25 AS NEW_STATUS ) ) AS R2 ,
          R2 { ALL BUT STATUS } AS R3 ,
          R3 RENAME ( NEW_STATUS AS STATUS ) AS R4 :
     ( S MINUS R1 ) UNION R4 ;
```

Exercise 1-16. First consider the general assignment:

```
R := rx ;
```

Here *R* is a relvar name and *rx* is a relational expression, denoting the relation to be assigned to relvar *R*. An SQL analog might look like this:

```
DELETE FROM T ;
INSERT INTO T tx ;
```

Here *T* is an SQL table corresponding to relvar *R* and *tx* is an SQL table expression corresponding to the relational expression *rx*. Note the need for the preliminary DELETE; note too that anything could happen, loosely speaking, between that DELETE and the subsequent INSERT, whereas there's no notion in the relational case of there *being* anything "between the DELETE and the INSERT" (the assignment is a semantically atomic operation).

So now I've answered the question "Can all relational assignments be expressed in terms of INSERT and/or DELETE and/or UPDATE?" But note that relational assignment is a single operation, whereas the foregoing DELETE/INSERT combination is a *sequence* of two distinct statements. One implication of this fact is that a failure could occur between those two statements, a failure that couldn't occur with the assignment as such.

There's another point I need to clear up here, too. In the body of the chapter, I said that SQL doesn't support relational assignment directly, and that's true. However, one reviewer objected that, for example, the following SQL expression "could be thought of as relational assignment" (I've simplified the reviewer's example somewhat):

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

In effect, the reviewer was suggesting that this expression is assigning some table value to a table variable called LS. But it isn't. In particular, it isn't possible to go on and do further queries or updates on LS; LS isn't an independent table in its own right, it's just a temporary table that's conceptually materialized as part of the process of evaluating the specified expression. That expression is not a relational assignment. (In any case, assignment of any kind is a statement, not an expression.)

And one further point: The SQL standard supports a variant of CREATE TABLE, "CREATE TABLE AS," that allows the base table being created to be initialized to the result of some query, thereby not only creating the table in question but also assigning an initial value to it. Once initialized, however, the table in question behaves just like any other base table; thus, CREATE TABLE AS doesn't really constitute support for relational assignment either.

Exercise 1-17. The discussions that follow are based on more extensive ones to be found in my book *An Introduction to Database Systems* (see Appendix D).

Duplicate tuples: Essentially, the concept makes no sense. Suppose for simplicity that the suppliers relation had just two attributes, SNO and CITY, and suppose it contained a tuple showing that it's a "true fact" that supplier S1 is located in London. Then if it also contained a duplicate of that tuple (if that were possible), it would simply be informing us of that same "true fact" a second time. But (as Chapter 4 observes) if something is true, saying it twice doesn't make it more true! For further discussion, see Chapter 4, or the paper "Double Trouble, Double Trouble" mentioned in Appendix D.

Tuple ordering: The lack of tuple ordering means there's no such thing as "the first tuple" or "the fifth tuple" or "the 97th tuple" of a relation, and there's no such thing as "the next tuple"; in other words, there's no concept of positional addressing, and no concept of "nextness." If we did have such concepts, we would need certain additional operators as well—for example, "retrieve the *n*th tuple," "insert this tuple *here*," "move this tuple from *here* to *there*," and so on. As a matter of fact (to lift some text from Appendix A), it's axiomatic that if we have *n* different ways to represent information, then we need *n* different sets of operators. And if $n > 1$, then we have more operators to implement, document, teach, learn, remember, and use. But those extra operators add complexity, not power! There's nothing useful that can be done if $n > 1$ that can't be done if $n = 1$.

By the way, another good argument against ordering (of any kind) is that positional addressing is fragile—the addresses change as insertions and deletions are performed. And yet another is that the meaning of a given data item (e.g., a tuple) can depend on that item's position.

Attribute ordering: The lack of attribute ordering means there's no such thing as "the first attribute" or "the second attribute" (and so on), and there's no "next attribute" (there's no concept of "nextness")—attributes are always referenced by name, never by position. As a result, the scope for errors and obscure programming is reduced. For example, there's no way to subvert the system by somehow "flopping over" from one attribute into another. This situation contrasts with that found in many programming systems, where it often is possible to exploit the physical adjacency of logically discrete items, deliberately or otherwise, in a variety of subversive ways. *Note:* Many other negative consequences of attribute ordering (or column ordering, rather, in SQL) are discussed in subsequent chapters. See also the paper "A Sweet Disorder," mentioned in Appendix D.

In the interest of accuracy, I should add that for reasons that don't concern us here, relations in mathematics, unlike their counterparts in the relational model, do have a left to right ordering to their attributes. A similar remark applies to tuples also.

Chapter 2

Exercise 2-1. A type is a finite, named 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 and so forth). There's no difference between a domain and a type. *Note:* SQL does make a distinction between domains and types, however. In particular, it supports both a CREATE TYPE statement and a CREATE DOMAIN statement. To a first approximation, CREATE TYPE is SQL's counterpart to the TYPE statement of **Tutorial D**, which I'll be discussing in Chapter 8 (though there are many, many differences, not all of them trivial in nature, between the two). CREATE DOMAIN might be regarded, very charitably, as SQL's attempt to provide a tiny part of the total functionality of CREATE TYPE (it was added to the language in 1992, while CREATE TYPE wasn't added until 1999); now that CREATE TYPE exists, there seems to be no good reason to use, or even support, CREATE DOMAIN at all.

Exercise 2-2. Every type has at least one associated selector; a selector is an operator that allows us to select, or specify, an arbitrary value of the type in question. Let *T* be a type and let *S* be a selector for *T*; then every value of type *T* must be returned by some invocation of *S*, and every invocation of *S* must return some value of type *T*. See Chapter 8 for further discussion. *Note:* Selectors are provided "automatically" in **Tutorial D** (since they're at least implicitly required by the relational model) but not (in general) in SQL. In fact, although the selector concept necessarily exists, SQL doesn't really have a term for it; certainly *selector* as such isn't an SQL term. Further details are beyond the scope of this book.

A literal is a "self-defining symbol"; it denotes a value that can be determined at compile time. More precisely, a literal is a symbol that denotes a value that's fixed and determined by the symbol in question (and the type of that value is therefore also fixed and determined by the symbol in question). Here are some **Tutorial D** examples:

```
4                          /* a literal of type INTEGER  */
'XYZ'                      /* a literal of type CHAR     */
FALSE                      /* a literal of type BOOLEAN  */
5.0                        /* a literal of type FIXED    */
POINT(5.0,2.5)             /* a literal of type POINT    */
```

Every value of every type, tuple and relation types included, must be denotable by means of some literal. A literal is a special case of a selector invocation; to be specific, it's a selector invocation all of whose arguments are themselves specified by literals (implying in particular that a selector invocation with no arguments at all, like the INTEGER selector invocation 95, is a literal by definition). Note that there's a logical difference between a literal as such and a constant; a constant is a value, a literal is a symbol that denotes a value.

Exercise 2-3. A THE_ operator is an operator that provides access to some component of some "possible representation," or *possrep*, of some specified value of some specified type. See Chapter 8 for further discussion. *Note:* THE_ operators are effectively provided "automatically" in both **Tutorial D** and SQL, to a first approximation. However, although the THE_ operator concept necessarily exists, SQL doesn't exactly have a term for it; certainly *THE_ operator* as such isn't an SQL term. Further details are beyond the scope of this book.

Exercise 2-4. True in principle; might not be completely true in practice (but to the extent it isn't, we're talking about a confusion over model vs. implementation).

Exercise 2-5. A *parameter* is a formal operand in terms of which some operator is defined. An *argument* is an actual operand, provided to be substituted for some parameter in some invocation of the operator in question. (People often use these terms as if they were interchangeable; much confusion is caused that way, and you need to be on the lookout for it.) *Note:* There's also a logical difference between an argument as such and the expression that's used to specify it. For example, consider the expression (2+3)–1, which is an invocation of the arithmetic operator "–". The first argument to that invocation is the value five, but that argument is specified by the expression 2+3 (which is an invocation of the arithmetic operator "+"; in fact, *every* expression represents some operator invocation).

A *database* is a repository for data. (*Note:* Much more precise definitions are possible; one such can be found in Chapter 5 of this book.) A *DBMS* is a software system for managing databases; it provides recovery, concurrency, integrity, query/update, and other services.

A *foreign key* is a subset of the heading of some relvar, values of which must be equal to values of some key of some relvar. A *pointer* is a value (an *address*, essentially) for which certain special operators—notably referencing and dereferencing operators—are (and must be) defined.* *Note:* Brief definitions of the referencing and dereferencing operators are given in a footnote in the body of the chapter.

A *generated* type is a type obtained by executing some type generator such as ARRAY or RELATION; specific array and relation types are thus generated types. A *nongenerated* type is a type that's not a generated type.

A *relation* is a value; it has a type, but isn't itself a type. A *type* is a named set of values (all possible values of some particular kind).

A *scalar* type is a type that has no user visible components; a *nonscalar* type is a type that's not a scalar type. Values, variables, operators and so forth are scalar or nonscalar according as their type is scalar or nonscalar. Be aware, however, that these terms are neither very formal nor very precise, in the final analysis. In particular, we'll meet a couple of important relations in Chapter 3 called TABLE_DUM and TABLE_DEE that are "scalar" by the foregoing definition!—or so it might be argued, at least.

Type is a model concept; types have semantics that must be understood by the user. *Representation* is an implementation concept; representations are supposed to be hidden from the user. In particular (and as noted in the body of the chapter), if *X* is a value or variable of type *T*, then the operators that apply to *X* are the operators defined for *T*, not the operators defined for the representation for *T*. For example, just because the representation for type ENO ("employee numbers") happens to be CHAR, say, it doesn't follow that we can concatenate two employee numbers; we can do that only if concatenation ("||") is an operator that's defined for type ENO.

A *system defined* (or *built in*) type is a type that's available for use as soon as the system is installed (it "comes in the same box the system comes in"). A *user defined* type is a type whose definition and implementation are provided by some suitably skilled user after the system is installed. (To the user of such a type, however—as opposed to the designer and implementer of that type—that type should look and feel just like a system defined type.)

A *system defined* (or *built in*) operator is an operator that's available for use as soon as the system is installed (it comes in the same box the system comes in). A *user defined*

* A much more extensive discussion of the logical difference between foreign keys and pointers can be found in the paper "Inclusion Dependencies and Foreign Keys" (see Appendix D).

operator is an operator whose definition and implementation are provided by some suitably skilled user after the system is installed. (To the user of such an operator, however—as opposed to the designer and implementer of that operator—that operator should look and feel just like a system defined operator.) User defined operators can take arguments of either user or system defined types (or a mixture), but system defined operators can take arguments of system defined types only.

Exercise 2-6. Coercion is implicit type conversion. It's deprecated because it's error prone (but note that this is primarily a pragmatic issue; whether or not coercions are permitted has little to do with the relational model as such).

Exercise 2-7. Because it confuses type and representation.

Exercise 2-8. A type generator is an operator that returns a type instead of a value (and is invoked at compile time instead of run time). The relational model requires support for two such: namely, TUPLE and RELATION. *Note:* Types generated by these particular type generators are nonscalar, but there's no reason in principle why generated types have to be nonscalar. REF in SQL is an example of a scalar type generator.

Exercise 2-9. A relation is in first normal form (1NF) if and only if every tuple contains a single value, of the appropriate type, in every attribute position; in other words, *every* relation is in first normal form. Given this fact, you might be forgiven for wondering why we bother to talk about the concept at all (and in particular why it's called "first"). The reason, as I'm sure you know, is that (a) we can extend it to apply to relvars as well as relations, and then (b) we can define a series of "higher" normal forms for relvars that turn out to be important in database design. In other words, 1NF is the base on which those higher normal forms build. But it really isn't all that important as a notion in itself.

NOTE

I should add that 1NF is one of those concepts whose definition has evolved somewhat over time. It used to be defined to mean that every tuple had to contain a single "atomic" value in every attribute position. As we've come to realize, however (and as I tried to show in the body of the chapter), the concept of data value atomicity actually has no objective meaning. An extensive discussion of 1NF can be found in the paper "What First Normal Form Really Means" (see Appendix D).

Exercise 2-10. The type of *X* is the type, *T* say, specified as the type of the result of the operator to be executed last—"the outermost operator"—when *X* is evaluated. That type is significant because it means *X* can be used in exactly (that is, in all and only) those positions where a literal value of type *T* can appear.

Exercise 2-11.

```
OPERATOR CUBE ( I INTEGER ) RETURNS INTEGER ;
   RETURN I * I * I ;
END OPERATOR ;
```

Exercise 2-12.

```
OPERATOR FGP ( P POINT ) RETURNS POINT ;
   RETURN POINT ( F ( THE_X ( P ) ) , G ( THE_Y ( P ) ) ) ;
END OPERATOR ;
```

Exercise 2-13. The following relation type is the type of the suppliers relvar S:

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

The suppliers relvar S itself is a variable of this type. And every legal value of that variable—for example, the value shown in Figure 1-3 in Chapter 1—is a value of this type.

Exercise 2-14. SQL definitions are given in the answer to Exercise 1-13 in Chapter 1. **Tutorial D** definitions:

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

VAR SP BASE RELATION
  { SNO CHAR , PNO CHAR , QTY INTEGER }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES S
  FOREIGN KEY { PNO } REFERENCES P ;
```

Some differences between the SQL and **Tutorial D** definitions:

- As noted in the answer to Exercise 1-13 in Chapter 1, SQL specifies keys and foreign keys, along with table columns,* all inside the same set of parentheses—a fact that makes it hard to determine exactly what the pertinent *type* is. (As a matter of fact, SQL doesn't really support the concept of a relation type at all, as we'll see in Chapter 3.)
- The left to right order in which columns are listed matters in SQL. See Chapter 5 for further discussion.
- SQL tables don't have to have keys at all.

The significance of the fact that relvar P, for example, is of a certain relation type is as follows:

- The only values that can ever be assigned to relvar P are relations of that type.
- A reference to relvar P can appear wherever a literal of that type can appear (as in, for example, the expression P JOIN SP), in which case it denotes the relation that happens to be the current value of that relvar at the pertinent time. (In other words, a *relvar reference* is a valid relational expression in **Tutorial D**; note, however, that an analogous remark does *not* apply to SQL, at least not 100 percent.) See Chapter 6 for further discussion.

Exercise 2-15. a. Not valid; LOCATION = CITY('London'). b. Valid; BOOLEAN. c. Presumably valid; MONEY (I'm assuming that multiplying a money value by an integer

* And certain other items, too, beyond the scope of the present discussion.

returns another money value). d. Not valid; BUDGET+MONEY(50000). e. Not valid; ENO > ENO('E2'). f. Not valid; NAME(THE_C(ENAME)||THE_C(DNAME)) (I'm assuming that type NAME has a single "possrep component" called C, of type CHAR). g. Not valid; CITY(THE_C(LOCATION)||'burg') (I'm assuming that type CITY has a single "possrep component" called C, of type CHAR). *Note:* I'm also assuming throughout these answers that a given type *T* always has a selector with the same name. See Chapter 8 for further discussion.

Exercise 2-16. Such an operation logically means replacing one type by another, not "updating a type" (types aren't variables). Consider the following. First of all, the operation of defining a type doesn't actually create the corresponding set of values; conceptually, those values already exist, and always will exist (think of type INTEGER, for example). All the "define type" operation (the TYPE statement in **Tutorial D**—see Chapter 8) really does is introduce a name by which that set of values can be referenced. Likewise, dropping a type doesn't actually drop the corresponding values, it just drops the name that was introduced by the corresponding "define type" operation. It follows that "updating a type" really means dropping the type name and then reintroducing that name to refer to a different set of values. Of course, there's nothing to preclude support for some kind of "alter type" shorthand to simplify matters—and SQL does support such an operator, in fact—but using such a shorthand isn't really "updating the type."

Exercise 2-17. The empty type is certainly a valid type; however, it wouldn't make much sense to define a variable to be of such a type, because no value could ever be assigned to such a variable! Despite this fact, the empty type turns out to be crucially important in connection with type inheritance—but that's a topic that's (sadly) beyond the scope of this book. Refer to the book *Databases, Types, and the Relational Model: The Third Manifesto*, by Hugh Darwen and myself (see Appendix D), if you want to know more.

Exercise 2-18. Let *T* be an SQL type for which "=" is not defined and let *C* be a column of type *T*. Then *C* can't be part of a candidate key or a foreign key, nor can it be part of the argument to DISTINCT or GROUP BY or ORDER BY, nor can restrictions or joins or unions or intersections or differences be defined in terms of it. And what about implementation constructs such as indexes? There are probably other implications as well.

Second, let *T* be an SQL type for which the semantics of "=" are user defined and let *C* be a column of type *T*. Then the effects of making *C* part of a candidate key or foreign key or applying DISTINCT or GROUP BY (etc., etc.) to it will be user defined as well, at best, and unpredictable at worst.

Exercise 2-19. Here's a trivial example of such violation. Let X be the character string 'AB ' (note the trailing space), let Y be the character string 'AB', and let PAD SPACE apply to the pertinent collation. Then the comparison X = Y gives TRUE, and yet the operator invocations CHAR_LENGTH(X) and CHAR_LENGTH(Y) give 3 and 2, respectively. I leave the detailed implications for you to think about, but it should be clear that problems are likely to surface in connection with DISTINCT, GROUP BY, and ORDER BY operations among others (as well as in connection with certain implementation constructs, such as indexes).

Exercise 2-20. Because (a) they're logically unnecessary, (b) they're error prone, (c) end users can't use them, (d) they're clumsy—in particular, they have a direction to them, which nonpointer values don't—and (e) they undermine type inheritance. (Details of this last point are beyond the scope of this book.) There are other reasons too. See the paper cited earlier, "Inclusion Dependencies and Foreign Keys," for further discussion.

Exercise 2-21. One answer has to do with nulls; if we "set X to null" (which isn't really assigning a value to X, because nulls aren't values, but never mind), the comparison X = NULL certainly doesn't give TRUE. There are many other examples too, not involving reliance on nulls. Again I leave the implications for you to think about.

Exercise 2-22. No! (Which database does type INTEGER belong to?) In an important sense, the whole subject of types and type management is orthogonal to the subject of databases and database management. We might even imagine the need for a "type administrator," whose job it would be to look after types in a manner analogous to that in which the database administrator looks after databases.

Exercise 2-23. An expression denotes a value; it can be thought of as a rule for computing or determining the value in question. A statement doesn't denote a value; instead, it causes some action to occur, such as assigning a value to some variable or changing the flow of control. In SQL, for example,

```
X + Y
```

is an expression, but

```
SET Z = X + Y ;
```

is a statement.

Exercise 2-24. An RVA is an attribute whose type is some relation type, and whose values are therefore relations of that type. A repeating group is an "attribute" of some type *T* whose values aren't values of type *T*—note the contradiction in terms here!—but, rather, bags or sets or sequences or (etc.) of values of type *T*. *Note:* The "attribute" in question is often itself of some tuple type (or something approximating a tuple type). For example, a file, in a system that allows repeating groups, might be such that each record consists of an ENO field (employee number), an ENAME field (employee name), and a repeating group JOBHIST, in which each entry consists of a JOB field (job title), a FROM field, and a TO field (where FROM and TO are dates).

Exercise 2-25. In SQL, a subquery is, loosely, a table expression in parentheses. Later chapters will elaborate (especially Chapter 12).

Exercise 2-26. See Chapter 3.

Chapter 3

Exercise 3-1. See the body of the chapter.

Exercise 3-2. Two values of any kind are equal if and only if they're the very same value! In particular, (a) two tuples *tx* and *ty* are equal if and only if they have the same attributes *A1*, *A2*, ..., *An* and for all *i* (*i* = 1, 2, ..., *n*), the value *vx* of *Ai* in *tx* is equal to the value *vy* of *Ai* in *ty*; (b) two relations *rx* and *ry* are equal if and only if they have the same heading and the same body.

Exercise 3-3. **Tutorial D** tuple selector invocations:

```
TUPLE { PNO 'P1' , PNAME 'Nut' ,
                   COLOR 'Red' , WEIGHT 12.0 , CITY 'London' }

TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 }
```

SQL analogs ("row value constructor" invocations):

```
ROW ( 'P1' , 'Nut' , 'Red' , 12.0 , 'London' )

ROW ( 'S1' , 'P1' , 300 )
```

Observe the lack of column names (or "field" names, to use the SQL term) and reliance on left to right ordering in these SQL expressions. The keyword ROW can be omitted without changing the meanings.

Exercise 3-4. The following selector invocation denotes a relation of two tuples:

```
RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } ,
           TUPLE { SNO 'S1' , PNO 'P2' , QTY 200 } }
```

SQL analog (a "table value constructor" invocation):

```
VALUES ROW ( 'S1' , 'P1' , 300 ) ,
       ROW ( 'S1' , 'P2' , 200 )
```

Either or both of the two "row value constructor" invocations here can omit the ROW keyword if desired. By the way, the fact that there are no parentheses enclosing those row value constructor invocations isn't an error. In fact, the following SQL expression—

```
VALUES ( ROW ( 'S1' , 'P1' , 300 ) ,
         ROW ( 'S1' , 'P2' , 200 ) )
```

(which is certainly legal, syntactically speaking)—denotes something entirely different! See Exercise 3-10.

Exercise 3-5. The list that follows is based on one in my book *An Introduction to Database Systems* (see Appendix D).

- Each attribute in the heading of a relation involves a type name, but those type names are usually omitted from tables (where by *tables* I mean tabular pictures of relations).
- Each component of each tuple in the body of a relation involves a type name and an attribute name, but those type and attribute names are usually omitted from tabular pictures.

- Each attribute value in each tuple in the body of a relation is a value of the applicable type, but those values (or literals denoting those values, rather) are usually shown in some abbreviated form—for example, S1 instead of 'S1'—in tabular pictures.
- The columns of a table have a left to right ordering, but the attributes of a relation don't. One implication of this point is that columns can have duplicate names, or even no names at all. For example, consider the SQL expression

  ```
  SELECT DISTINCT S.CITY, S.STATUS * 2, P.CITY
  FROM   S, P
  ```

  What are the column names in the result of this expression?
- The rows of a table have a top to bottom ordering, but the tuples of a relation don't.
- A table might contain duplicate rows, but a relation never contains duplicate tuples.
- Tables are usually regarded as having at least one column, while relations are allowed to have no attributes at all (see the section "TABLE_DUM and TABLE_DEE" in the body of the chapter).
- Tables (at least in SQL) are allowed to include nulls, but relations certainly aren't.
- Tables are "flat" or two-dimensional, but relations are *n*-dimensional.

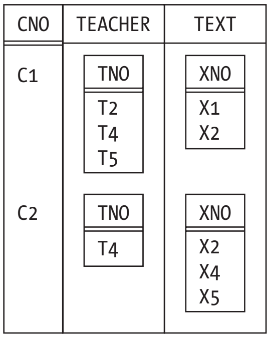
Exercise 3-6. One exception is as follows: Since no database relation can have an attribute of any pointer type, no tuple in such a relation can have an attribute of any pointer type either. The other exception is a little harder to state, but what it boils down to is that if tuple *t* has heading {*H*}, then no attribute of *t* can be defined in terms of any tuple or relation type with that same heading {*H*}, at any level of nesting.

Here are (a) a tuple with a tuple valued attribute and (b) a tuple with a relation valued attribute:

```
TUPLE { NAME 'Superman' ,
        ADDR TUPLE { STREET '1600 Pennsylvania Ave.' ,
                     CITY   'Washington' ,
                     STATE  'DC' ,
                     ZIP    '20500' } }

TUPLE { SNO 'S2' ,
        PNO_REL RELATION { TUPLE { PNO 'P1' } ,
                           TUPLE { PNO 'P2' } } }
```

Exercise 3-7. For a relation with one RVA, see relation R4 in Figure 2-2 in Chapter 2; for an equivalent relation with no RVA, see relation R1 in Figure 2-1 in Chapter 2. Here's one with two RVAs:

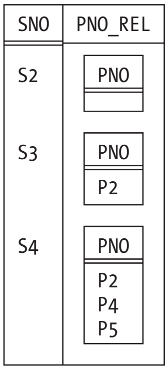


CNO	TEACHER	TEXT
C1	TNO T2 T4 T5	XNO X1 X2
C2	TNO T4	XNO X2 X4 X5

The intended meaning is: *Course CNO can be taught by every teacher TNO in TEACHER and uses every textbook XNO in TEXT.* Here's a relation without RVAs that conveys the same information:

CNO	TNO	XNO
C1	T2	X1
C1	T2	X2
C1	T4	X1
C1	T4	X2
C1	T5	X1
C1	T5	X2
C2	T4	X2
C2	T4	X4
C2	T4	X5

As for a relation with an RVA such that there's no relation without an RVA that represents precisely the same information, one simple example can be obtained from Figure 2-2 in Chapter 2 by just replacing the PNO_REL value for (say) supplier S2 by an empty relation:



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

However, it isn't necessary to invoke the notion of an empty relation in order to come up with an example of a relation with an RVA such that there's no relation without an RVA that represents precisely the same information. (*Subsidiary exercise:* Justify this remark! If you give up, take a look at the answer to Exercise B-14, near the end of this appendix.)

Perhaps I should elaborate on what it means for two relations to represent the same information. Basically, relations *r1* and *r2* represent the same information if and only if it's possible to map *r1* into *r2* and vice versa by means of operations of the relational algebra, without introducing any additional information into the mapping in either direction. With reference to relations R4 in Figure 2-2 in Chapter 2 and R1 in Figure 2-1 in Chapter 2, for example, we have:

```
R4 = R1 GROUP ( { PNO } AS PNO_REL )

R1 = R4 UNGROUP ( PNO_REL )
```

Each relation can thus be defined in terms of the other, and the two therefore do represent the same information.* See Chapter 7 for further discussion of the GROUP and UNGROUP operators.

Exercise 3-8. TABLE_DEE and TABLE_DUM (DEE and DUM for short) are the only relations with no attributes; DEE contains exactly one tuple (the 0-tuple), DUM contains no tuples at all. SQL doesn't support them because tables in SQL are always required to have at least one column. (In other words, SQL's version of the relational algebra is like an arithmetic that has no zero.)

Exercise 3-9. (*Note:* You might want to come back and take another look at this answer after reading Chapter 10.) We need the concept of relations in general before we can have the concept of relations of degree zero in particular. The concept of relations in general depends on predicate logic. Predicate logic depends on propositional logic. Propositional logic depends on the truth values TRUE and FALSE. So if we tried to replace TRUE and FALSE by DEE and DUM, we would be going round in circles!

Also, it would be a little odd (to say the least) if all boolean expressions suddenly became relational expressions, and host languages thus suddenly all had to support relational data types.

Would it make sense to define a relvar of degree zero? It's hard but not impossible to imagine a situation in which such a relvar might be useful—but that's not the point. Rather, the point is that the system shouldn't include a prohibition against defining such a relvar. If it did, then that fact would constitute a violation of orthogonality, and such violations always come back to bite us eventually.

Exercise 3-10. The first denotes an SQL table of four rows (three distinct ones and one duplicate). The second denotes an SQL table of one row, that row consisting of four "field" values all of which are rows in turn. Note that none of the fields involved (in either case) is named.

Exercise 3-11. The given expression is semantically equivalent to this one:

* Another useful informal characterization is this: Relations *r1* and *r2* represent the same information if and only if, for any query *q1* that can be addressed to *r1*, there's a corresponding query *q2* that can be addressed to *r2* that produces the same result (and vice versa).

```
SELECT SNO
FROM   S
WHERE  STATUS > 20
OR   ( STATUS = 20 AND SNO > 'S4' )
OR     STATUS IS NULL
OR     SNO IS NULL
```

Exercise 3-12. See the body of the chapter.

Exercise 3-13. See the body of the chapter.

Exercise 3-14. EXISTS (*tx*), where *tx* is the SQL analog of the relational expression *rx* (so *tx* is an SQL table expression).

Exercise 3-15. See the body of the chapter.

Exercise 3-16. AS is used in SELECT clauses (to introduce column names); CREATE VIEW; FROM clauses (to introduce range variable names—by contrast, the syntax used to introduce *column* names in this context doesn't use AS); WITH; and other contexts not discussed in this book.

You were also asked (a) in which cases the keyword was optional; (b) in which cases the AS clause took the form "<*something*> AS *name*"; and (c) in which cases it took the form "*name* AS <*something*>": *No answer provided.*

Chapter 4

Exercise 4-1. To deal with this argument properly would take more space than we have here, but it all boils down to what's sometimes called *The Principle of Identity of Indiscernibles* (see Appendix A). Let *a* and *b* be any two entities—for example, two pennies. Well, if there's absolutely no way whatsoever of distinguishing between *a* and *b*, then there aren't two entities but only one! It might be true for certain purposes that the two entities can be interchanged, but that fact isn't sufficient to make them indiscernible (there's a logical difference between interchangeability and indiscernibility, in fact, and arguments to the effect that "duplicates occur naturally in the real world" tend to be based on a muddle over this difference). A detailed analysis of this whole issue can be found in the paper "Double Trouble, Double Trouble" (see Appendix D).

Exercise 4-2. Strictly speaking, before we can answer this question, we need to pin down what WHERE and UNION mean in the presence of duplicates. The paper "The Theory of Bags: An Investigative Tutorial" (see Appendix D) goes into details; here let me just say that if we adopt the SQL definitions, then the law certainly doesn't apply. In fact, it doesn't apply to either UNION ALL or UNION DISTINCT! By way of example, let *r* be an SQL table with just one column—*C*, say—containing just two rows, each of them containing just the value *v*. Then the following expressions produce the indicated results:

```
SELECT C
FROM   r
WHERE  TRUE
OR     TRUE

Result:  v * 2.
```

```
SELECT C
FROM   r
WHERE  TRUE
UNION  DISTINCT
SELECT C
FROM   r
WHERE  TRUE

Result:  v * 1.

SELECT C
FROM   r
WHERE  TRUE
UNION  ALL
SELECT C
FROM   r
WHERE  TRUE

Result:  v * 4.
```

NOTE

If the various (implicit or explicit) ALLs in the foregoing expressions were all replaced by DISTINCT, it would be a different story. What do you conclude?

Exercise 4-3. Remarks similar to those in the answer to the previous exercise apply here also. Again I'll skip the details, but the net is that the law almost certainly doesn't apply. I leave development of a counterexample to you.

Exercise 4-4. As far as I can see, the only way to resolve the ambiguity is by effectively defining a mapping from each of the (multiset) argument tables to a proper set, and likewise defining a mapping of the (multiset) result table—i.e., the desired cartesian product—to a proper set. (The mappings involve attaching a unique identifier to each row.) It seems to me, in fact, that the standard's failed attempt at a definition here serves only to emphasize the point that one of the most fundamental concepts in the entire SQL language (viz., the idea that tables should permit duplicate rows) is fundamentally flawed—and cannot be repaired without, in effect, dispensing with the concept altogether.

Exercise 4-5. I don't think the problem can be fixed.

Exercise 4-6. *No answer provided!*

Exercise 4-7. The question was: Do you think nulls occur naturally in the real world? Only you can answer this question—but if your answer is *yes*, I think you should examine your reasoning very carefully. For example, consider the statement "Joe's salary is $50,000." That statement is either true or false. Now, you might not know whether it's true or false; but your not knowing has nothing to do with whether it actually is true or false. In particular, your not knowing is certainly not the same as saying Joe's salary is null! "Joe's salary is $50,000" is a statement about the real world. "Joe's salary is null" is a statement about your knowledge (or lack of knowledge, rather) about the real world. We certainly shouldn't keep a mixture of these two very different kinds of statements in the same relation, or in the same relvar! *Note:* The discussion of *relvar predicates* in the next chapter should give you further food for thought in this connection.

Suppose you had to represent the fact that you don't know Joe's salary on some paper form. Would you enter a null, as such, into that form? I don't think so! Rather, you would leave the box blank, or put a question mark, or write "unknown," or something along those lines. And that blank, or question mark, or "unknown," or whatever, is a value, not a null (recall that the one thing we can be definite about regarding nulls is that they aren't values). Speaking for myself, therefore, no, I don't think nulls do "occur naturally in the real world."

Exercise 4-8. True (though not in SQL!). Null is a marker that represents the absence of information. UNKNOWN is a value, just as TRUE and FALSE are values. There's a logical difference between the two, and to confuse them as SQL does is a logical mistake (I'd like to say a big logical mistake, but all logical mistakes are big by definition).

Exercise 4-9. Yes, it does; SQL's analog of MAYBE *p* is *p* IS UNKNOWN.

Exercise 4-10. In 2VL there are 4 monadic connectives and 16 dyadic connectives, corresponding to the 4 possible monadic truth tables and 16 possible dyadic truth tables. Here are those truth tables (I've indicated the ones that have common names, such as NOT, AND, and OR):

T	T
F	T

T	T
F	F

NOT	
T	F
F	T

T	F
F	F

	T	F
T	T	T
F	T	T

IF	T	F
T	T	F
F	T	T

NAND	T	F
T	F	T
F	T	T

	T	F
T	F	F
F	T	T

OR	T	F
T	T	T
F	T	F

	T	F
T	T	F
F	T	F

XOR	T	F
T	F	T
F	T	F

	T	F
T	F	F
F	T	F

	T	F
T	T	T
F	F	T

IFF	T	F
T	T	F
F	F	T

	T	F
T	F	T
F	F	T

NOR	T	F
T	F	F
F	F	T

	T	F
T	T	T
F	F	F

AND	T	F
T	T	F
F	F	F

	T	F
T	F	T
F	F	F

	T	F
T	F	F
F	F	F

In 3VL, there are 27 (3 to the power 3) monadic connectives and 19,683 (3 to the power 3^2) dyadic connectives. (In general, in fact, *n*VL has *n* to the power *n* monadic connectives and *n* to the power n^2 dyadic connectives.) Many conclusions might be drawn from these facts; one of the most immediate is that 3VL is vastly more complex than 2VL (much more so, probably, than most people, including those who think nulls are a good thing, realize, or at least admit to).

Exercise 4-11. 2VL is truth functionally complete if it supports NOT and either AND or OR (see the answer to Exercise 10-2 for further discussion). And it turns out that SQL's 3VL—under an extremely charitable interpretation of that term!—is also truth functionally complete. The paper "Is SQL's Three-Valued Logic Truth Functionally Complete?" (see Appendix D) gives further explanation.

Exercise 4-12. It's not a tautology in 3VL, because if *bx* evaluates to UNKNOWN, the whole expression also evaluates to UNKNOWN. But there does exist an analogous tautology in 3VL: viz., *bx* OR NOT *bx* OR MAYBE *bx*. *Note:* This state of affairs explains why, in SQL, if you execute the query "Get all suppliers in London" and then the query "Get all suppliers not in London," you don't necessarily get (in combination) all suppliers; you have to execute the query "Get all suppliers who may be in London" as well. Note the implications for query rewrite ... not to mention the potential for serious mistakes (on the part of both users and the system, I might add). To spell the point out: It's very natural to assume that expressions that are tautologies in 2VL are also tautologies in 3VL, but such is not always the case.

Exercise 4-13. It's not a contradiction in 3VL, because if *bx* evaluates to UNKNOWN, the whole expression also evaluates to UNKNOWN. But there does exist an analogous (slightly tricky!) contradiction in 3VL: viz., *bx* AND NOT *bx* AND NOT MAYBE *bx*. *Note:* As you might expect, this state of affairs has implications similar to those noted in the answer to the previous exercise.

Exercise 4-14. In 3VL, (a) *r* JOIN *r* isn't necessarily equal to *r*, and (b) INTERSECT isn't a special case of JOIN. The reason is that (a) for join two nulls aren't considered to "compare equal" but (b) for intersection they are (just another one of the vast—infinite?—number of absurdities that nulls inevitably seem to lead us into). However, TIMES is still a special case of JOIN, as it is in 2VL.

Exercise 4-15. Here are the rules: Let *x* be an SQL row. Suppose for simplicity that *x* has just two components, *x1* and *x2* (in left to right order, of course!). Then *x* IS NULL is defined to be equivalent to *x1* IS NULL AND *x2* IS NULL, and *x* IS NOT NULL is defined to be equivalent to *x1* IS NOT NULL AND *x2* IS NOT NULL. It follows that the given row is neither null nor nonnull ... What do you conclude from this state of affairs?

By the way: At least one reviewer commented here that he'd never thought of a row being null. But rows are values (just as tuples and relations are values), and hence the idea of some row being unknown makes exactly as much sense as, say, the idea of some salary being unknown. Thus, if the concept of representing an unknown value by a "null" makes any sense at all—which of course I don't think it does—then it surely applies to rows (and tables, and any other kind of value you can think of) just as much as it applies to scalars. And as this exercise demonstrates, SQL tries to support this position (for rows, at least, though not for tables), but fails.[*]

Exercise 4-16. No. Here are the (3VL) truth tables:

NOT	
T	F
U	U
F	T

IS NOT TRUE	
T	F
U	T
F	T

Exercise 4-17. No. For definiteness, consider the case in which *x* is an SQL row. Suppose for simplicity (as in the answer to Exercise 4-15) that *x* has just two components, *x1* and *x2*. Then *x* IS NOT NULL is defined to be equivalent to *x1* IS NOT NULL AND *x2* IS NOT NULL, and NOT (*x* IS NULL) is defined to be equivalent to *x1* IS NOT NULL OR *x2* IS NOT NULL. What do you conclude from this state of affairs?

Exercise 4-18. The transformation isn't valid, as you can see by considering what happens if EMP.DNO is null (were you surprised?). The implications, once again, are that users and the system are both likely to make mistakes (and there's some history here, in fact).

Exercise 4-19. The query means "Get suppliers who are known not to supply part P2" (note that *known not*, and note too the subtle difference between that phrase and *not known*); it does *not* mean "Get suppliers who don't supply part P2." The two formulations aren't equivalent (consider, e.g., the case where the only supplier number "value" matching the part number value P2 in SP is null).

Exercise 4-20. No two of the three statements are equivalent. Statement a. follows the rules of SQL's 3VL; statement b. follows the definition of SQL's UNIQUE operator; and statement c. follows SQL's definition of duplicates. In particular, if *k1* and *k2* are both null, then a. gives UNKNOWN, b. gives FALSE, and c. gives TRUE. *Note:* Here for the record are the rules I was referring to:

- In SQL's 3VL, the comparison *k1* = *k2* gives TRUE if *k1* and *k2* are both nonnull and are equal, FALSE if *k1* and *k2* are both nonnull and are unequal, and UNKNOWN otherwise.
- With SQL's UNIQUE operator, the comparison *k1* = *k2* gives TRUE if and only if *k1* and *k2* are both nonnull and are equal, and FALSE otherwise.
- In SQL, *k1* and *k2* are duplicates if and only if (a) they're nonnull and equal or (b) they're both null.

> **NOTE**
> **Throughout the foregoing, "equal" refers to SQL's own, somewhat idiosyncratic definition of the "=" operator (see Chapter 2).** ***Subsidiary exercise:*** **Do you think these rules are reasonable? Justify your answer.**

[*] It ought logically to support it for tables, too, but doesn't.

Exercise 4-21. The output from INTERSECT ALL and EXCEPT ALL can contain duplicates, but only if duplicates are present in the input.

Exercise 4-22. Yes! (We don't want duplicates in the database, but that doesn't mean we never want duplicates anywhere else.)

Exercise 4-23. A very good question.

Chapter 5

Exercise 5-1. In some ways a tuple does resemble a record and an attribute a field—but these resemblances are only approximate. A relvar shouldn't be regarded as just a file, but rather as a "file with discipline," as it were. The discipline in question is one that results in a considerable simplification in the structure of the data as seen by the user, and hence in a corresponding simplification in the operators needed to deal with that data, and indeed in the user interface in general. What is that discipline? Well, it's that there's no top to bottom ordering to the records; and no left to right ordering to the fields; and no nulls; and no repeating groups; and no pointers; and no anonymous fields (and so on and so forth). Partly as a consequence of these facts, it really is much better to think of a relvar like this: The heading represents some predicate, and the body at any given time represents the extension of that predicate at that time.

Exercise 5-2. Loosely, the remark means "Update the STATUS attribute in tuples for suppliers in London." But tuples (and, a fortiori, attribute values within tuples) are values and simply can't be updated, by definition. Here's a more precise version of the remark:

- Let relation *s* be the current value of relvar S.
- Let *ls* be that restriction of *s* for which the CITY value is London.
- Let *ls′* be that relation that's identical to *ls* except that the STATUS value in each tuple is as specified in the given UPDATE operation.
- Let *s′* be the relation denoted by the expression (*s* MINUS *ls*) UNION *ls′*.
- Then *s′* is assigned to S.

Exercise 5-3. Because relational operations are fundamentally set level and SQL's "positioned update" operations are fundamentally tuple level (or row level, rather). Although set level operations for which the set in question is of cardinality one are sometimes—perhaps even frequently—acceptable, they can't always work. In particular, tuple level update operations might work for a while and then cease to work when integrity constraint support is improved.

Exercise 5-4. The statements aren't equivalent. The source for the first is the table *t1* denoted by the specified *table* subquery; the source for the second is the table *t2* containing just the row denoted by the specified *row* subquery (i.e., the VALUES argument). If table S does include a row for supplier S6, then *t1* and *t2* are identical. But if table S doesn't include such a row, then *t1* is empty while *t2* contains a row of all nulls.

Exercise 5-5. *The Assignment Principle* states that after assignment of the value *v* to the variable *V*, the comparison *V* = *v* must return TRUE. SQL violates this principle if "*v* is null"; it also violates it on certain character string assignments; and it certainly also violates it for any type for which the "=" operator isn't defined, including type XML in particular, and (in general) certain user defined types as well. *Negative consequences:* Too many to list here.

Exercise 5-6. As in the body of the chapter, I assume the availability of certain user defined types in the following definitions:

```
CREATE TABLE TAX_BRACKET
  ( LOW        MONEY   NOT NULL ,
    HIGH       MONEY   NOT NULL ,
    PERCENTAGE INTEGER NOT NULL ,
    UNIQUE ( LOW ) ,
    UNIQUE ( HIGH ) ,
    UNIQUE ( PERCENTAGE ) ) ;

CREATE TABLE ROSTER
  ( DAY   DAY_OF_WEEK NOT NULL ,
    HOUR  TIME_OF_DAY NOT NULL ,
    GATE  GATE        NOT NULL ,
    PILOT NAME        NOT NULL ,
    UNIQUE ( DAY, HOUR, GATE ) ,
    UNIQUE ( DAY, HOUR, PILOT ) ) ;

CREATE TABLE MARRIAGE
  ( SPOUSE_A         NAME NOT NULL ,
    SPOUSE_B         NAME NOT NULL ,
    DATE_OF_MARRIAGE DATE NOT NULL ,
    UNIQUE ( SPOUSE_A, DATE_OF_MARRIAGE ) ,
    UNIQUE ( DATE_OF_MARRIAGE, SPOUSE_B ) ,
    UNIQUE ( SPOUSE_B, SPOUSE_A ) ) ;
```

Exercise 5-7. Because keys represent constraints and constraints apply to variables, not values. (That said, it's certainly possible, and sometimes useful, to think of subset *k* of the heading of relation *r* as if it were "a key for *r*" if it's unique and irreducible with respect to the tuples of *r*. But thinking this way is strictly incorrect, and potentially confusing, and certainly much less useful than thinking about keys for relvars as opposed to relations.)

Exercise 5-8. Here's one: Suppose relvar *A* has a "reducible key" consisting of the disjoint union of *K* and *X*, say, where *K* and *X* are both subsets of the heading of *A* and *K* is a genuine key. Then relvar *A* satisfies the functional dependence $K \rightarrow X$. Suppose now that relvar *B* has a foreign key referencing that "reducible key" in *A*. Then *B* too satisfies the functional dependency $K \rightarrow X$; as a result, *B* probably displays some redundancy (in fact, it's probably not in Boyce/Codd normal form).

Exercise 5-9. Keys are sets of attributes—in fact, every key is a subset of the pertinent heading—and key values are thus tuples by definition, even when the tuples in question have exactly one attribute. Thus, for example, the key for the parts relvar P is

{PNO} and not just PNO, and the key value for the parts tuple for part P1 is TUPLE {PNO 'P1'} and not just 'P1'.

Exercise 5-10. Let m be the smallest integer greater than or equal to $n/2$. R will have the maximum possible number of keys if either (a) every distinct set of m attributes is a key or (b) n is odd and every distinct set of m-1 attributes is a key. Either way, it follows that the maximum number of keys in R is $n!/(m!*(n-m)!)$. *Note:* The expression $n!$ is read as either "n factorial" or "factorial n" and is defined as the product $n * (n-1) * ... * 2 * 1$. Relvars TAX_BRACKET and MARRIAGE (see Exercise 5-6) are both examples of relvars with the maximum possible number of keys; so is any relvar of degree zero. (If $n = 0$, the formula becomes $0!/(0!*0!)$, and $0!$ is 1.)

Exercise 5-11. A superkey is a subset of the heading with the uniqueness property; a key is a superkey with the irreducibility property. All keys are superkeys, but "most" superkeys aren't keys.

The concept of a *subkey* can be useful in studying normalization. Here's a definition: Let X be a subset of the heading of relvar R; then X is a subkey for R if and only if there exists some key K for R such that K is a superset of X. For example, the following are all of the subkeys for relvar SP: {SNO,PNO}, {SNO}, {PNO}, and {}. Note that the empty set {} is necessarily a subkey for all possible relvars R.

Exercise 5-12. Sample data:

EMP

ENO	MNO
E4	E2
E3	E2
E2	E1
E1	E1

I'm using the trick here of pretending that a certain employee (namely, employee E1) acts as his or her own manager, which is one way of avoiding the use of nulls in this kind of situation. Another and probably better approach is to separate the reporting structure relationships out into a relvar of their own, excluding from that relvar any employee who has no manager:

EMP

ENO	...
E4	...
E3	...
E2	...
E1	...

EM

ENO	MNO
E4	E2
E3	E2
E2	E1

Subsidiary exercise: What are the predicates for relvars EM and the two versions of EMP here? (Thinking carefully about this exercise should serve to reinforce the suggestion that the second design is preferable.)

Exercise 5-13. Because it doesn't need to, on account of the fact that column correspondences are established on the basis of ordinal position rather than name. See the discussion in the body of the chapter.

Exercise 5-14. Note that such a situation must represent a one to one relationship, by definition. One obvious case arises if we decide to split some relvar "vertically," as in the following example (suppliers):

```
VAR SNT BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES SC ;

VAR SC BASE RELATION
  { SNO SNO, CITY CHAR }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES SNT ;
```

One implication is that we probably need a mechanism for updating two or more relvars at the same time (and probably one for defining two or more relvars at the same time also). See the discussion of multiple assignment in Chapter 8.

Exercise 5-15. Tutorial D definitions (I'll assume for the sake of this exercise that **Tutorial D** supports the referential actions CASCADE and NO CASCADE):

```
VAR P BASE RELATION { PNO ... , ... } KEY { PNO } ;

VAR PP BASE RELATION { MAJOR_PNO ... , MINOR_PNO ... , QTY ... }
    KEY { MAJOR_PNO , MINOR_PNO }
    FOREIGN KEY { MAJOR_PNO } REFERENCES P
            RENAME ( PNO AS MAJOR_PNO ) ON DELETE CASCADE
    FOREIGN KEY { MINOR_PNO } REFERENCES P
            RENAME ( PNO AS MINOR_PNO ) ON DELETE NO CASCADE ;
```

With these definitions, deleting a part will cascade to those tuples where the pertinent part number appears as a MAJOR_P# value (meaning the part in question contains other parts as components), but not to those tuples where the same part number appears as a MINOR_P# value (meaning the part in question is a component of other parts).

SQL definitions:

```
CREATE TABLE P ( PNO ... , ... , UNIQUE ( PNO ) ) ;

CREATE TABLE PP ( MAJOR_PNO ... , MINOR_P# ... , QTY ... ,
                  UNIQUE ( MAJOR_PNO , MINOR_PNO ) ,
                  FOREIGN KEY ( MAJOR_PNO ) REFERENCES P ( PNO )
                          ON DELETE CASCADE ,
                  FOREIGN KEY ( MINOR_PNO ) REFERENCES P
                          ON DELETE RESTRICT ) ;
```

Exercise 5-16. It's obviously not possible to give a definitive answer to this exercise. I'll just mention the referential actions supported by the standard, which are NO ACTION (the default), CASCADE, RESTRICT, SET DEFAULT, and SET NULL. *Subsidiary exercise:* What's the difference between NO ACTION and RESTRICT?

Exercise 5-17. A predicate is a truth valued function; a proposition is a predicate with no parameters. See the body of the chapter for some examples, and Chapter 10 for more examples and an extended discussion of these concepts in general.

Exercise 5-18. Relvar P: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY.* Relvar SP: *Supplier SNO supplies part PNO in quantity QTY.*

Exercise 5-19. The intension of relvar *R* is the intended interpretation of *R*. The extension of relvar *R* at a given time is the set of tuples appearing in *R* at that time.

Exercise 5-20. *No answer provided.*

Exercise 5-21. *The Closed World Assumption* says (loosely) that everything stated or implied by the database is true and everything else is false. And *The Open World Assumption*—yes, there is such a thing—says that everything stated or implied by the database is true and everything else is unknown. As you can probably see, therefore, *The Open World Assumption* leads directly to a requirement for nulls and three-valued logic, and is strongly deprecated for that reason. The paper "The Closed World Assumption" (see Appendix D) gives more information.

Exercise 5-22. To say relvar *R* has an empty key is to say *R* can never contain more than one tuple (because every tuple has the same value for the empty set of attributes—namely, the empty tuple; thus, if *R* were to contain two or more tuples, we would have a key uniqueness violation on our hands). And constraining *R* never to contain more than one tuple could certainly be useful. I'll leave finding an example of such a situation as a subsidiary exercise.

Exercise 5-23. See the answer to Exercise 5-17.

Exercise 5-24. The question certainly makes sense, insofar as every relvar has an associated predicate. However, just what the predicate is for some given relvar is in the mind of the definer of that relvar (and in the user's mind too, I trust). For example, if I define a relvar C as follows—

```
VAR C BASE RELATION { CITY CHAR } KEY { CITY } ;
```

—the corresponding predicate might be almost anything! It might, for example, be *CITY is a city in California*; or *CITY is a city in which at least one supplier is located*; or *CITY is a city that's the capital of some country*;* and so on. In the same way, the predicate for a relvar of degree zero—

```
VAR Z BASE RELATION { } KEY { } ;
```

—might also be "almost anything," except that (since the relvar has no attributes and the corresponding predicate therefore has no parameters) the predicate in question must in fact degenerate to a proposition. That proposition will evaluate to TRUE if the value of Z is TABLE_DEE and FALSE if the value is TABLE_DUM.

By the way, observe that relvar Z has an empty key. It's obvious that every degree zero relvar must have an empty key; however, you shouldn't conclude that degree zero relvars are the only ones with empty keys (see the answer to the previous exercise but one).

* Or even *CITY is the name of somebody's favorite teddy bear*. There's nothing in the relvar definition to say that CITY has to denote a city.

Exercise 5-25. Of course not. In fact, "most" relations aren't values of some relvar. As a trivial example, the relation denoted by S{CITY}, the projection of S on CITY, isn't a value of any relvar in the suppliers-and-parts database. Note, therefore, that throughout this book, when I talk about some relation, I certainly don't necessarily mean a relation that's the value of some relvar.

Chapter 6

First of all, here are answers to a couple of exercises that were stated inline in the body of the chapter. The first asked what the difference was, given our usual sample data, between the expressions S JOIN (P{PNO}) and (S JOIN P){PNO}. *Answer:* Since S and P{PNO} have no common attributes, the first expression represents a cartesian product; given our usual sample data, the result contains all possible combinations of a supplier tuple and a part number (loosely speaking), and has cardinality 30. The second yields part numbers for parts that are in the same city as some supplier (again, loosely speaking); given our usual sample data, the result contains all part numbers except P3.

The second exercise asked what the difference was between an equijoin and a natural join. *Answer:* Let the relations to be joined be *r1* and *r2*, and assume for simplicity that *r1* and *r2* have just one common attribute, *A*. Before we can perform the equijoin, then, we need to do some renaming. For definiteness, suppose we apply the renaming to *r2*, to yield *r3* = *r2* RENAME (*A* AS *B*). Then the equijoin is defined to be equal to (*r1* TIMES *r3*) WHERE *A* = *B*. Note in particular that *A* and *B* are both attributes of the result, and every tuple in that result will have the same value for those two attributes. Projecting attribute *B* away from that result yields the natural join *r1* JOIN *r2*.

Exercise 6-1. a. The result has duplicate column names (as well as left to right column ordering). b. The result has left to right column ordering. c. The result has an unnamed column (as well as left to right column ordering). d. The result has duplicate rows. e. SQL syntax error: S NATURAL JOIN P has no column called S.CITY.* f. Nothing wrong. g. The result has duplicate rows and left to right column ordering; it also has no SNO column, a fact that might come as a surprise.

Exercise 6-2. No! In particular, certain relational divides that you might expect to fail don't. Here are some examples (which won't make much sense until you've read the relevant section of Chapter 7):

a. Let relation PZ be of type RELATION {PNO PNO} and let its body be empty. Then the expression

```
SP { SNO , PNO } DIVIDEBY PZ { PNO }
```

reduces to the projection SP{SNO} of SP on SNO.

* It doesn't really have a column called S.SNO, either (it has a column called SNO, unqualified, instead); however, there's a bizarre syntax rule to the effect that the column can be referred to by that qualified name anyway. (When I say the rule is bizarre, I mean it's extremely difficult to state precisely, as well as being both counterintuitive and logically incorrect.)

b. Let *z* be either TABLE_DEE or TABLE_DUM. Then the expression

```
r DIVIDEBY z
```

reduces to *r* JOIN *z*.

c. Let relations *r* and *s* be of the same type. Then the expression

```
r DIVIDEBY s
```

gives TABLE_DEE if *r* is nonempty and every tuple of *s* appears in *r*, TABLE_DUM otherwise.

d. Finally, *r* DIVIDEBY *r* gives TABLE_DUM if *r* is empty, TABLE_DEE otherwise.

Exercise 6-3. The joining attributes are SNO, PNO, and CITY. The result looks like this:

SNO	SNAME	STATUS	CITY	PNO	QTY	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12.0
S1	Smith	20	London	P4	200	Screw	Red	14.0
S1	Smith	20	London	P6	100	Cog	Red	19.0
S2	Jones	10	Paris	P2	400	Bolt	Green	17.0
S3	Blake	30	Paris	P2	200	Bolt	Green	17.0
S4	Clark	20	London	P4	200	Screw	Red	14.0

The predicate is: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY; part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY; and supplier SNO supplies part PNO in quantity QTY.* Note that both appearances of SNO in this predicate refer to the same parameter, as do both appearances of PNO and both appearances of CITY.

The simplest SQL formulation is just

```
S NATURAL JOIN SP NATURAL JOIN P
```

(though it might be necessary to prefix this expression with "SELECT * FROM," depending on context).

Exercise 6-4. In 2-dimensional cartesian geometry, the points $(x,0)$ and $(0,y)$ are the projections of the point (x,y) on the X axis and the Y axis, respectively; equivalently, (x) and (y) are the projections into certain 1-dimensional spaces of the point (x,y) in 2-dimensional space. These notions are readily generalizable to *n* dimensions (recall from Chapter 3 that relations are indeed *n*-dimensional).

Exercise 6-5.

a. SQL analog:

```
SELECT DISTINCT CITY
FROM   S NATURAL JOIN SP
WHERE  PNO = 'P2'
```

NOTE

Here and throughout these answers, I show SQL expressions that aren't necessarily direct transliterations of their algebraic counterparts but are, rather, "more natural" formulations of the query in SQL terms.

Predicate: *City CITY is such that some supplier who supplies part P2 is located there.*

CITY
London
Paris

b. SQL analog:

```
SELECT *
FROM   P
WHERE  PNO NOT IN
     ( SELECT PNO
       FROM   SP
       WHERE  SNO = 'S2' )
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, is stored in city CITY, and isn't supplied by supplier S2.*

PNO	PNAME	COLOR	WEIGHT	CITY
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

c. SQL analog:

```
SELECT CITY
FROM   S
EXCEPT CORRESPONDING
SELECT CITY
FROM   P
```

Predicate: *City CITY is such that some supplier is located there but no part is stored there.*

CITY
Athens

d. SQL analog:

```
SELECT SNO , PNO
FROM   S NATURAL JOIN P
```

NOTE

There's no need to do the projections of S and P in the Tutorial D version, either. Do you think the optimizer might ignore them?

Predicate: *Supplier SNO and part PNO are colocated.*

SNO	PNO
S1	P1
S1	P4
S1	P6
S2	P2
S2	P5
S3	P2
S3	P5
S4	P1
S4	P4
S4	P6

e. SQL analog:

```
SELECT S.CITY AS SC , P.CITY AS PC
FROM   S , P
```

Predicate: *Some supplier is located in city SC and some part is stored in city PC.*

SC	PC
London	London
London	Paris
London	Oslo
Paris	London
Paris	Paris
Paris	Oslo
Athens	London
Athens	Paris
Athens	Oslo

Exercise 6-6. Intersection and cartesian product are both special cases of join, so we can ignore them here. The fact that union and join are commutative is immediate from the fact that the definitions are symmetric in the two relations concerned. I now show that union is associative. Let *t* be a tuple. Using "≡" to stand for "if and only if" and "∈" to stand for "appears in," we have:

```
t ∈ r UNION (s UNION u)  ≡  t ∈ r OR t ∈ (s UNION u)
                         ≡  t ∈ r OR (t ∈ s OR t ∈ u)
                         ≡  (t ∈ r OR t ∈ s) OR t ∈ u
                         ≡  t ∈ (r UNION s) OR t ∈ u
                         ≡  t ∈ (r UNION s) UNION u
```

Note the appeal in the third line to the associativity of OR. The proof that join is associative is analogous.

As for SQL, well, let's first of all ignore nulls and duplicate rows (what happens if we don't?). Then:

- SELECT A, B FROM T1 UNION CORRESPONDING SELECT B, A FROM T2 and SELECT B, A FROM T2 UNION CORRESPONDING SELECT A, B FROM T1 aren't equivalent, because they produce results with different left to right column orderings. Thus, union in general isn't commutative in SQL (and the same goes for intersection).

- T1 JOIN T2 and T2 JOIN T1 aren't equivalent (in general), because they produce results with different left to right column orderings. Thus, join in general isn't commutative in SQL (and the same goes for product).

The operators are, however, all associative.

Exercise 6-7. RENAME is the only one.

Exercise 6-8. The cartesian product of a single table *t* is defined to be just *t*. But the question of what the product of *t1* and *t2* is if *t1* and *t2* both contain duplicate rows is a tricky one! See the answer to Exercise 4-4 for further discussion.

Exercise 6-9. **Tutorial D** on the left, SQL on the right, as usual:*

a.
```
SP                             |  SELECT * FROM SP
                               |  or
                               |  TABLE SP /* see Chapter 12 */
```

b.
```
( SP WHERE PNO = 'P1' )        |  SELECT SNO
                  { SNO }      |  FROM   SP
                               |  WHERE  PNO = 'P1'
```

c.
```
S WHERE STATUS ≥ 15            |  SELECT *
    AND STATUS ≤ 25            |  FROM   S
                               |  WHERE  S.STATUS BETWEEN
                               |         15 AND 25
```

d.
```
( ( S JOIN SP )                |  SELECT DISTINCT PNO
  WHERE CITY = 'London' )      |  FROM   SP, S
                  { PNO }      |  WHERE  SP.SNO = S.SNO
                               |  AND    S.CITY = 'London'
```

e.
```
P { PNO } MINUS                |  SELECT PNO
( ( S JOIN SP )                |  FROM   P
  WHERE CITY = 'London' )      |  EXCEPT CORRESPONDING
                  { PNO }      |  SELECT PNO
                               |  FROM   SP , S
                               |  WHERE  SP.SNO = S.SNO
                               |  AND    S.CITY = 'London'
```

f.
```
WITH SP { SNO , PNO } AS Z :   |  SELECT DISTINCT XX.PNO AS X ,
( ( Z RENAME ( PNO AS X ) )    |                  YY.PNO AS Y
  JOIN                         |  FROM   SP AS XX , SP AS YY
  ( Z RENAME ( PNO AS Y ) ) )  |  WHERE  XX.SNO = YY.SNO
{ X , Y }                      |
```

* The solutions aren't unique, in general. The **Tutorial D** solutions in particular could often be improved by using operators described in Chapter 7.

g.
```
( S WHERE STATUS <                 |  SELECT SNO
  STATUS FROM ( TUPLE FROM ( S     |  FROM   S
  WHERE SNO = 'S1' ) ) ) { SNO }   |  WHERE  STATUS <
                                   |       ( SELECT STATUS
                                   |         FROM   S
                                   |         WHERE  SNO = 'S1' )
```

As you'll recall from Chapter 3, the **Tutorial D** expression STATUS FROM (TUPLE FROM *r*) extracts the STATUS value from the single tuple in relation *r* (that relation *r* must have cardinality one). By contrast, the SQL version of the query effectively does a double coercion: First, it coerces a table of one row to that row; second, it coerces that row to the single scalar value it contains.

h.
```
WITH ( S WHERE CITY =              |  SELECT DISTINCT SPX.PNO
         'London' ) AS RX ,        |  FROM   SP AS SPX
     ( SP RENAME ( PNO AS Y ) )    |  WHERE  NOT EXISTS (
                    AS RY :        |  SELECT S.SNO FROM S
( P WHERE RX { SNO } ⊆             |  WHERE  S.CITY = 'London'
    ( RY WHERE Y = PNO )           |  AND    NOT EXISTS (
               { SNO } ) { PNO }   |  SELECT SPY.*
                                   |  FROM   SP AS SPY
                                   |  WHERE  SPY.SNO = S.SNO
                                   |  AND    SPY.PNO =
                                   |         SPX.PNO ) )
```

Note the use of a relational comparison in the **Tutorial D** expression here. The SQL version uses EXISTS (see Chapter 10).

i.
```
( S { SNO } JOIN P { PNO } )  |  SELECT SNO , PNO
  MINUS SP { SNO , PNO }      |  FROM   S, P
                              |  EXCEPT CORRESPONDING
                              |  SELECT SNO , PNO
                              |  FROM   SP
```

j.
```
WITH ( SP WHERE SNO = 'S2' )       |  SELECT S.SNO FROM S
                     AS RA ,       |  WHERE  NOT EXISTS (
     ( SP RENAME ( SNO AS X ) )    |  SELECT P.* FROM P
                     AS RB :       |  WHERE  NOT EXISTS (
S WHERE ( RB WHERE X = SNO )       |  SELECT SP.* FROM SP
         { PNO } ⊇ RA { PNO }      |  WHERE  SP.PNO = P.PNO
                                   |  AND    SP.SNO = 'S2' )
                                   |  OR     EXISTS (
                                   |  SELECT SP.*
                                   |  FROM   SP
                                   |  WHERE  SP.PNO = P.PNO
                                   |  AND    SP.SNO = S.SNO ) )
```

Exercise 6-10. It's intuitively obvious that all three statements are true. *No further answer provided.*

Exercise 6-11. Union isn't idempotent in SQL, because the expression SELECT *R*.* FROM *R* UNION CORRESPONDING SELECT *R*.* FROM *R* isn't identically equal to SELECT *R*.* FROM *R*. That's because if *R* contains any duplicates, they'll be eliminated from the result of the union. (And what happens if *R* contains any nulls? Good question!)

Join is idempotent, and therefore so are intersection and cartesian product—in all cases, in the relational model but not in SQL, thanks again to duplicates and nulls.

Exercise 6-12. The expression *r*{} denotes the projection of *r* on no attributes; it returns TABLE_DUM if *r* is empty and TABLE_DEE otherwise. The answer to the question "What's the corresponding predicate?" depends on what the predicate for *r* is. For example, the predicate for SP{} is (a trifle loosely): *There exists a supplier SNO, there exists a part PNO, and there exists a quantity QTY such that supplier number SNO supplies part PNO in quantity QTY*. Note that this predicate is in fact a proposition; if SP is empty (in which case SP{} is TABLE_DUM) it evaluates to FALSE, otherwise (in which case SP{} is TABLE_DEE) it evaluates to TRUE.

The expression *r*{ALL BUT} denotes the projection of *r* on all of its attributes (in other words, the identity projection of *r*); it returns *r*.

Exercise 6-13. I believe DB2 and Ingres both perform this kind of optimization. Other products might do so too.

Exercise 6-14. The expression means "Get suppliers who supply all purple parts." Of course, the point is that (given our usual sample data values) there aren't any purple parts. The expression correctly returns all five suppliers. For further explanation (in particular, of the fact that this is indeed the correct answer), see Chapter 11.

Exercise 6-15. A rough equivalent in SQL might look like this:

```
SELECT CITY
FROM ( SELECT CITY FROM S
       UNION  CORRESPONDING
       SELECT CITY FROM P ) AS TEMP
WHERE  NOT EXISTS
     ( SELECT    CITY FROM S
       INTERSECT CORRESPONDING
       SELECT    CITY FROM P )
```

This SQL expression isn't precisely equivalent to the relational version, however. To be specific, if supplier cities and part cities aren't disjoint, then the SQL expression won't fail at run time but will simply return an empty result. *Note:* The CORRESPONDING specifications could safely be omitted in this example—why, exactly?—but it's easier, and shouldn't hurt, always to specify CORRESPONDING, even when it's logically unnecessary.

Exercise 6-16. Two relations *r1* and *r2* can be joined if and only if they're joinable—i.e., if and only if attributes with the same name are of the same type (equivalently, if and only if the set theory union of their headings is a legal heading). That's the dyadic case. Extending the definition to the *n*-adic case is easy: Relations *r1*, *r2*, ..., *rn* ($n > 0$) are joinable if and only if, for all *i*, *j* ($1 \le i \le n$, $1 \le j \le n$), relations *ri* and *rj* are joinable.

Exercise 6-17. The fact that JOIN, UNION, and D_UNION are all both commutative and associative is what makes it possible to define *n*-adic versions.

SQL does effectively support *n*-adic join and union, though not for *n* < 2. The syntax is:

```
t1 NATURAL JOIN t2
   NATURAL JOIN t3
    ..........
   NATURAL JOIN tn

SELECT * FROM t1 UNION CORRESPONDING SELECT * FROM t2
                 UNION CORRESPONDING SELECT * FROM t3
                   ..............................
                 UNION CORRESPONDING SELECT * FROM tn
```

An *n*-adic version of MINUS makes no sense because MINUS is neither commutative nor associative.

Exercise 6-18. For a brief justification, see the answer to Exercise 6-12. A longer one follows. Consider the projection of the suppliers relvar S on SNO, S{SNO}. Let's refer to the result of this projection as *r*; given our usual sample data values, *r* contains five tuples. Now consider the projection of that relation *r* on the empty set of attributes, *r*{}. Clearly, projecting any *tuple* on no attributes at all yields an empty tuple; thus, every tuple in *r* produces an empty tuple when *r* is projected on no attributes. But all empty tuples are duplicates of one another; thus, projecting the 5-tuple relation *r* on no attributes yields a relation with no attributes and one (empty) tuple, or in other words TABLE_DEE.

Now recall that every relvar has an associated predicate. For relvar S, that predicate looks like this:

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

For the projection *r* = S{SNO}, it looks like this:

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

And for the projection *r*{}, it looks like this:

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

Observe now that this last predicate is in fact a proposition: It evaluates to TRUE or FALSE, unconditionally. In the case at hand, *r*{} is TABLE_DEE and the predicate (proposition) evaluates to TRUE. But suppose no suppliers at all were represented in the database at this time. Then S{SNO} would yield an empty relation *r*, *r*{} would be TABLE_DUM, and the predicate (proposition) in question would evaluate to FALSE.

Chapter 7

Exercise 7-1.

a . SQL analog:

```
SELECT *
FROM   S
WHERE  SNO IN
     ( SELECT SNO
       FROM   SP
       WHERE  PNO = 'P2' )
```

> **NOTE**
> **Here and throughout these answers, I show SQL expressions that aren't necessarily direct transliterations of their algebraic counterparts but are, rather, "more natural" formulations of the query in SQL terms.**

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, and supplies part P2.*

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

b. SQL analog:

```
SELECT *
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM   SP
       WHERE  PNO = 'P2' )
```

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, and doesn't supply part P2.*

SNO	SNAME	STATUS	CITY
S5	Adams	30	Athens

c. SQL analog:

```
SELECT *
FROM   P AS PX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   S AS SX
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, is stored in city CITY, and is supplied by all suppliers.*

PNO	PNAME	COLOR	WEIGHT	CITY

d. SQL analog:

```
SELECT *
FROM   P
WHERE  ( SELECT COALESCE ( SUM ( QTY ) , 0 )
         FROM   SP
         WHERE  SP.PNO = P.PNO ) < 500
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, is stored in city CITY, and is supplied in a total quantity, taken over all suppliers, less than 500.*

PNO	PNAME	COLOR	WEIGHT	CITY
P3	Screw	Blue	17.0	Oslo
P6	Cog	Red	19.0	London

e. SQL analog:

```
SELECT *
FROM   P
WHERE  CITY IN
     ( SELECT CITY
       FROM   S )
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, is stored in city CITY, and is located in the same city as some supplier.*

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

f. SQL analog:

```
SELECT S.* , 'Supplier' AS TAG
FROM   S
```

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, supplies part P2, and has a TAG of 'Supplier'.*

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

g. SQL analog:

```
SELECT SNO , 3 * STATUS AS TRIPLE_STATUS
FROM   S
```

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, supplies part P2, and has a TRIPLE_STATUS value of three times the value of STATUS.*

SNO	SNAME	STATUS	CITY	TRIPLE_STATUS
S1	Smith	20	London	60
S2	Jones	10	Paris	30
S3	Blake	30	Paris	90
S4	Clark	20	London	60
S5	Adams	30	Athens	90

h. SQL analog:

```
SELECT PNO , PNAME, COLOR , WEIGHT , CITY , SNO , QTY
       WEIGHT * QTY AS SHIPWT
FROM   P NATURAL JOIN SP
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, is stored in city CITY, is supplied by supplier SNO in quantity QTY, and that shipment (of PNO by SNO) has total weight SHIPWT equal to WEIGHT times QTY.*

SNO	PNO	QTY	PNAME	COLOR	WEIGHT	CITY	SHIPWT
S1	P1	300	Nut	Red	12.0	London	3600.0
S1	P2	200	Bolt	Green	17.0	Paris	3400.0
S1	P3	400	Screw	Blue	17.0	Oslo	6800.0
S1	P4	200	Screw	Red	14.0	London	2800.0
S1	P5	100	Cam	Blue	12.0	Paris	1200.0
S1	P6	100	Cog	Red	19.0	London	1900.0
S2	P1	300	Nut	Red	12.0	London	3600.0
S2	P2	400	Bolt	Green	17.0	Paris	6800.0
S3	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P4	300	Screw	Red	14.0	London	4200.0
S4	P5	400	Cam	Blue	12.0	Paris	4800.0

i. SQL analog:

```
SELECT P.* , WEIGHT * 454 AS GMWT , WEIGHT * 16 AS OZWT
FROM   P
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR, weight WEIGHT, weight in grams GMWT (= 454 times WEIGHT), and weight in ounces OZWT (= 16 times WEIGHT).*

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT	OZWT
P1	Nut	Red	12.0	London	5448.0	192.0
P2	Bolt	Green	17.0	Paris	7718.0	204.0
P3	Screw	Blue	17.0	Oslo	7718.0	204.0
P4	Screw	Red	14.0	London	6356.0	168.0
P5	Cam	Blue	12.0	Paris	5448.0	192.0
P6	Cog	Red	19.0	London	8626.0	228.0

j. SQL analog:

```
SELECT P.* , ( SELECT COUNT ( SNO )
               FROM   SP
               WHERE  SP.PNO = P.PNO ) AS SCT
FROM   P
```

Predicate: *Part PNO is used in the enterprise, is named PNAME, has color COLOR, weight WEIGHT, and city CITY, and is supplied by SCT suppliers.*

PNO	PNAME	COLOR	WEIGHT	CITY	SCT
P1	Nut	Red	12.0	London	2
P2	Bolt	Green	17.0	Paris	4
P3	Screw	Blue	17.0	Oslo	1
P4	Screw	Red	14.0	London	2
P5	Cam	Blue	12.0	Paris	2
P6	Cog	Red	19.0	London	1

k. SQL analog:

```
SELECT S.* , ( SELECT COUNT ( PNO )
               FROM   SP
               WHERE  SP.SNO = S.SNO ) AS NP
FROM   S
```

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, and supplies NP parts.*

SNO	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

l. SQL analog:

```
SELECT CITY , AVG ( STATUS ) AS AVG_STATUS
FROM   S
GROUP  BY CITY
```

Predicate: *The average status of suppliers in city CITY is AVG_STATUS.*

CITY	AVG_STATUS
London	20
Paris	20
Athens	30

m. SQL analog:

```
SELECT COUNT ( SNO ) AS N
FROM   S
WHERE  CITY = 'London'
```

Predicate: *There are N suppliers in London.*

N
2

The lack of double underlining here is *not* a mistake.

n. SQL analog:

```
( SELECT *
  FROM   SP
  EXCEPT CORRESPONDING
  SELECT *
  FROM   SP
  WHERE  SNO = 'S1' )
UNION  CORRESPONDING
SELECT 'S7' AS SNO , PNO , QTY * 0.5 AS QTY
FROM   SP
WHERE  SNO = 'S1'
```

Predicate: *Either (a) supplier SNO supplies part PNO in quantity QTY (and SNO isn't S1), or (b) SNO is S7 and supplier S1 supplies part PNO in quantity twice QTY.* By the way: What happens if SP already includes any tuples for supplier S7?

SNO	PNO	QTY
S7	P1	150
S7	P2	100
S7	P3	200
S7	P4	100
S7	P5	50
S7	P6	50
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Exercise 7-2. The expressions *r1* MATCHING *r2* and *r2* MATCHING *r1* are equivalent if and only if *r1* and *r2* are of the same type, in which case both expressions reduce to just JOIN{*r1*,*r2*}, which reduces in turn to INTERSECT{*r1*,*r2*}.

Exercise 7-3. Rename isn't primitive because (for example) the expressions

```
S RENAME ( CITY AS SCITY )
```

and

```
( EXTEND S ADD ( CITY AS SCITY ) ) { ALL BUT CITY }
```

are equivalent.

Exercise 7-4. `EXTEND S { SNO } ADD ( COUNT ( !! SP ) AS NP )`

Exercise 7-5. You can determine which of the expressions are equivalent from the following results of evaluating them. Note that SUM(1), evaluated over *n* tuples, is equal to *n*.

a.

r empty:

CT

r has *n* tuples ($n > 0$):

CT
n

b.

r empty:

CT
0

r has *n* tuples ($n > 0$):

CT
n

c.

r empty:

CT

r has *n* tuples ($n > 0$):

CT
n

d.

r empty:

CT
0

r has *n* tuples ($n > 0$):

CT
n

In other words, the result is a relation of degree one in every case. If *r* is nonempty, all four expressions are equivalent; otherwise a. and c. are equivalent, and b. and d. are equivalent. SQL analogs:

a.
```
SELECT COUNT ( * ) AS CT
FROM   r
EXCEPT CORRESPONDING
SELECT 0 AS CT
FROM   r
```

b.
```
SELECT COUNT ( * ) AS CT
FROM   r
```

NOTE

We could harmlessly append INTERSECT CORRESPONDING SELECT 0 AS CT FROM *r* to the foregoing, if we liked.

c. Same as a.

d. Same as b.

Exercise 7-6. SQL returns null in all cases except COUNT, where it does correctly return zero. As to why, your guess is as good as mine.

Exercise 7-7. Here's one reasonably straightforward formulation: *Supplier SNO supplies part PNO if and only if part PNO is mentioned in relation PNO_REL.* That "and only if" is important, by the way (right?).

Exercise 7-8. Relation *r* has the same cardinality as SP and the same heading, except that it has one additional attribute, X, which is relation valued. The relations that are values of X have degree zero; furthermore, each is TABLE_DEE, not TABLE_DUM, because every tuple *sp* in SP effectively includes the 0-tuple as its value for that subtuple of *sp* that corresponds to the empty set of attributes. Thus, each tuple in *r* effectively consists of the corresponding tuple from SP extended with the X value TABLE_DEE, and the original GROUP expression is logically equivalent to the following:

```
EXTEND SP ADD ( TABLE_DEE AS X )
```

The expression *r* UNGROUP (X) yields the original SP relation again.

Exercise 7-9.

e.
```
N := COUNT ( SP           |  SET N = ( SELECT COUNT ( * )
     WHERE SNO = 'S1' ) ; |           FROM   S
                          |           WHERE  SNO = 'S1' ) ;
```

f.
```
( S WHERE CITY =             |  SELECT *
  MIN ( S , CITY ) ) { SNO } |  FROM   S
                             |  WHERE  CITY =
                             |       ( SELECT MIN ( CITY )
                             |         FROM   S )
```

g.
```
S { CITY }             |  SELECT DISTINCT CITY
WHERE COUNT ( !!S ) > 1 |  FROM   S AS SX
                       |  WHERE  ( SELECT COUNT ( * )
                       |           FROM   S AS SY
                       |           WHERE  SY.CITY = SX.CITY )
                       |         > 1
```

h.
```
RESULT :=                    |  SET RESULT = CASE WHEN
IF AND ( S , SNAME < CITY )  |       ( SELECT COALESCE ( EVERY
THEN 'Y' ELSE 'N' END IF ;   |       ( SNAME < CITY ) , TRUE ) )
                             |         FROM   S ) THEN 'Y'
                             |                     ELSE 'N'
                             |              END ;
```

NOTE

EVERY is SQL's analog of Tutorial D's AND aggregate operator, but it returns null, not TRUE, if its argument is empty.

Exercise 7-10. Supplier numbers for suppliers who supply part P2 and part numbers for parts supplied by supplier S2, respectively. Note that these natural language formulations are symmetric, while their formal counterparts certainly aren't; that's because R4 is itself asymmetric, in the sense that it treats supplier numbers and part numbers very differently. *Note:* It's precisely because relations with RVAs are (usually) asymmetric that they're (usually) contraindicated.

Exercise 7-11. It denotes a relation looking like this (in outline):

SNO	SNAME	STATUS	CITY	PNO_REL
S1	Smith	20	London	PNO P1 P2 ... P6
S2	Jones	10	Paris	PNO P1 P2
...	.....	..	......	.....
S5	Adams	30	Athens	PNO

Attribute PNO_REL here is an RVA. Note in particular that the empty set of parts supplied by supplier S5 is represented by an empty set, not—as it would be if we were to form the outer join of S and SP—by null. To represent an empty set by an empty set seems like such an obviously good idea; in fact, there would be no need for outer join at all if RVAs were properly supported!

Note, incidentally, that if *r* is the foregoing relation, then the expression

```
( r UNGROUP ( PNO_REL ) ) { ALL BUT PNO }
```

will *not* return our usual suppliers relation. To be precise, it will return a relation that's identical to our usual suppliers relation, except that it'll have no tuple for supplier S5.

Exercise 7-12. The first is straightforward: It inserts a new tuple, with SNO S6, SNAME Lopez, STATUS 30, CITY Madrid, and PNO_REL value a relation containing just one tuple, containing in turn the PNO value P5. As for the second, I think it would be helpful to show the relevant portion—at least, a simplified version of it—of the **Tutorial D** grammar for relational assignment (the names of the syntactic categories are meant to be self-explanatory):

```
<relation assign>
   ::=    <relvar ref> := <relation exp>
        | <relation insert>
        | <relation delete>
        | <relation update>

<relation insert>
    ::=   INSERT <relvar ref> <relation exp>

<relation delete>
    ::=   DELETE <relvar ref> [ WHERE <boolean exp> ]

<relation update>
    ::=   UPDATE <relvar ref> [ WHERE <boolean exp> ] :
                 { <attribute assign commalist> }
```

And an <*attribute assign*>, if the attribute in question is relation valued, is basically just a <*relation assign*>, and that's where we came in. Thus, in the exercise, what the second update does is replace the tuple for supplier S2 by another in which the PNO_REL value additionally includes a tuple for supplier S5.

Observe, therefore, that the two updates are a formal representation of the following natural language updates:

a. Add the fact that supplier S6 supplies part P5 to the database.

b. Add the fact that supplier S2 supplies part P5 to the database.

With our usual suppliers-and-parts design (without RVAs), there's no qualitative difference between these two—both involve the insertion of a single tuple into relvar SP, like this (let's ignore attribute QTY, for simplicity):

a. `INSERT SP RELATION { TUPLE { SNO 'S6' , PNO 'P5' } } ;`

b. `INSERT SP RELATION { TUPLE { SNO 'S2' , PNO 'P5' } } ;`

But with SSP, these symmetric updates are treated asymmetrically. As noted in the answer to Exercise 7-10, it's this lack of symmetry that's the problem (usually but not always) with relvars that include RVAs.

Exercise 7-13. Query a. is easy:

```
WITH ( SSP RENAME ( SNO AS XNO ) ) { XNO, PNO_REL } AS X ,
     ( SSP RENAME ( SNO AS YNO ) ) { YNO, PNO_REL } AS Y :
( X JOIN Y ) { XNO, YNO }
```

Note that the join here is being done on an RVA (and is thus implicitly performing relational comparisons).

Query b., by contrast, is not so straightforward. Query a. was easy because SSP "nests parts within suppliers," as it were; for Query b. we would really like to have suppliers nested within parts instead. So let's do that:*

```
WITH ( SSP UNGROUP ( PNO_REL ) ) GROUP ( { SNO } AS SNO_REL )
                                                 AS PPS ,
     ( PPS RENAME ( PNO AS XNO ) ) { XNO, SNO_REL } AS X ,
     ( PPS RENAME ( PNO AS YNO ) ) { YNO, SNO_REL } AS Y :
( X JOIN Y ) { XNO, YNO }
```

Exercise 7-14.

```
WITH ( P RENAME ( WEIGHT AS WT ) ) AS R1 ,
     ( EXTEND P ADD ( COUNT ( R1 WHERE WT > WEIGHT )
                                    AS N_HEAVIER ) AS R2 :
( R2 WHERE N_HEAVIER < 2 ) { ALL BUT N_HEAVIER }
```

* This example thus points up an important difference between RVAs in a relational system and hierarchies in a system like IMS (or XML?). In IMS, the hierarchies are "hard wired" into the database, as it were; in other words, we're stuck with whatever hierarchies the database designer has seen fit to give us. In a relational system, by contrast, we can dynamically construct whatever hierarchies we want, by means of appropriate operators of the relational algebra.

```
SELECT *
FROM   P AS PX
WHERE ( SELECT COUNT ( * )
        FROM   P AS PY
        WHERE  PX.WEIGHT > PY.WEIGHT ) < 2
```

The query returns the tuples for parts P2, P3, and P6 (i.e., a relation of cardinality three, even though the specified quota was two). Quota queries can also return fewer tuples than requested; e.g., consider the query "Get the ten heaviest parts."

NOTE

Quota queries are quite common in practice. In our book *Databases, Types, and the Relational Model: The Third Manifesto* (see Appendix D), therefore, Hugh Darwen and I suggest a "user friendly" shorthand for expressing them, according to which the foregoing query could be expressed thus:

```
( ( RANK P BY ( DESC WEIGHT AS W ) ) WHERE W ≤ 2 ) { ALL BUT W }
```

SQL has something similar.

Exercise 7-15. This query highlights a weakness of the SUMMARIZE operator as such. Here's the requested formulation:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUMD ( QTY ) AS SDQ )
```

The trick is in that "SUMD"—the "D" here stands for *distinct*, and it means "Eliminate redundant duplicates before computing the sum." But this trick is indeed nothing but a trick (a pretty ad hoc trick at that), and it's one of many reasons why the following expression, using EXTEND and image relations, is to be preferred:

```
EXTEND S { SNO } ADD ( SUM ( !!SP , QTY ) AS SDQ )
```

Exercise 7-16.

```
EXTEND S ADD ( COUNT ( !!SP ) AS NP , COUNT ( !!SJ ) AS NJ )

( SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS NP ) )
  JOIN
( SUMMARIZE SJ PER ( S { SNO } ) ADD ( COUNT ( JNO ) AS NJ ) )

SELECT SNO , ( SELECT COUNT ( PNO )
               FROM   SP
               WHERE  SP.SNO = S.SNO ) AS NP ,
             ( SELECT COUNT ( JNO )
               FROM   SJ
               WHERE  SJ.SNO = S.SNO ) AS NJ
FROM   S
```

Exercise 7-17. It's easy to see that "!!" is idempotent; thus, !!(!!SP) is the same as !!SP, and the expression is semantically equivalent to:

```
S WHERE (!!SP) { PNO } = P { PNO }
```

("Get suppliers who supply all parts").

Exercise 7-18. No, there's no logical difference.

Exercise 7-19. S JOIN SP isn't a semijoin; S MATCHING SP isn't a join (it's a projection of a join). The expressions *r1* JOIN *r2* and *r1* MATCHING *r2* are equivalent if and only if relations *r1* and *r2* are of the same type (when the final projection becomes an identity projection, and the expression overall degenerates to *r1* INTERSECTION *r2*).

Exercise 7-20. If *r1* and *r2* are of the same type and *t1* is a tuple in *r1*, the expression !!*r2* (for *t1*) denotes a relation of degree zero: TABLE_DEE if *t1* appears in *r2*, TABLE_DUM otherwise. And if *r1* and *r2* are the same relation, !!*r2* denotes TABLE_DEE for every tuple in *r1*.

Exercise 7-21. They're the same unless table S is empty, in which case the first yields a one-column, one-row table containing a zero and the second yields a one-column, one-row table "containing a null."

Chapter 8

Exercise 8-1. A type constraint is a definition of the set of values that constitute a given type. The type constraint for type *T* is checked when some selector for type *T* is invoked; if the check fails, the selector invocation fails on a type constraint violation.

A database constraint is a constraint on the values that can appear in a given database. Database constraints are checked "at semicolons"—more specifically, at the end of any statement that assigns a value to (any of) the pertinent relvar(s). If the check fails, the assignment fails on a database constraint violation. *Note:* Database constraints must also be checked when they're defined. If that check fails, the constraint definition must be rejected.

Exercise 8-2. The Golden Rule states that no update operation must ever cause any database constraint to evaluate to FALSE, and hence, a fortiori, that no update operation must ever cause any relvar constraint to evaluate to FALSE either. However, a (total) relvar constraint might evaluate to FALSE, not because some single-relvar constraint is violated, but rather because some multi-relvar constraint is violated. The point is somewhat academic, however, given that which individual relvar constraints are single-relvar and which multi-relvar is somewhat arbitrary anyway.

Exercise 8-3. "Assertion" is SQL's term for a constraint specified via CREATE ASSERTION. An attribute constraint is a statement to the effect that a certain attribute is of a certain type. A base table constraint is an SQL constraint that's specified as part of a base table definition (and not as part of a column definition within a base table definition). A column constraint is an SQL constraint that's specified as part of a column definition. A multi-relvar constraint is a database constraint that mentions two or more distinct relvars. A referential constraint is a constraint to the effect that if *B* references *A*, then *A* must exist. A relvar constraint for relvar *R* is a database constraint that mentions *R*. A row constraint is an SQL constraint with the property that it can be checked for a given row by examining just that row in isolation. A single-relvar constraint is a database constraint that mentions just one relvar. A state constraint is a database constraint that isn't a transition constraint. "The" (total) database constraint for database *DB* is the logical AND of all of the relvar constraints for relvars in *DB* and TRUE. "The" (total) relvar constraint for relvar *R* is the logical AND of all of the database constraints that mention *R* and TRUE. A transition constraint is a constraint on the legal transitions a database can make from one "state" (i.e., value) to another. A tuple constraint is a relvar constraint with the property that it can be checked for a given tuple by examining just that tuple in isolation. Which of these categories if any do (a) key constraints, (b) foreign key constraints, fall into? *No answers provided.*

Exercise 8-4. See the body of the chapter.

Exercise 8-5. a. 345. b. QTY.

Exercise 8-6. See the body of the chapter.

Exercise 8-7.

```
TYPE CITY POSSREP { C CHAR CONSTRAINT C = 'London'
                                   OR C = 'Paris'
                                   OR C = 'Rome'
                                   OR C = 'Athens'
                                   OR C = 'Oslo'
                                   OR C = 'Stockholm'
                                   OR C = 'Madrid'
                                   OR C = 'Amsterdam' } ;
```

Now we can define the CITY attribute in relvars S and P to be of type CITY instead of just type CHAR.

Exercise 8-8. By definition, there's no way to impose a constraint that's exactly equivalent to the one given in the previous answer without defining an explicit type. But we could impose the constraint that supplier cities in particular are limited to the same eight values by means of a suitable database constraint, and similarly for part cities. For example, we could define a base table as follows:

```
CREATE TABLE C ( CITY CHAR , UNIQUE ( CITY ) ) ;
```

We could then "populate" this table with the eight city values:

```
INSERT INTO C VALUES 'London'    ,
                     'Paris'     ,
                     'Rome'      ,
                     'Athens'    ,
                     'Oslo'      ,
                     'Stockholm' ,
                     'Madrid'    ,
                     'Amsterdam' ;
```

Now we could define some foreign keys:

```
CREATE TABLE S ( ... ,
                 FOREIGN KEY ( CITY ) REFERENCES C ( CITY ) ) ;

CREATE TABLE P ( ... ,
                 FOREIGN KEY ( CITY ) REFERENCES C ( CITY ) ) ;
```

This approach has the advantage that it makes it easier to change the set of valid cities, if the requirement should arise.

Another approach would be to define an appropriate set of base table (or column) constraints as part of the definitions of base tables S and P. Another would be to use an appropriate set of CREATE ASSERTION statements. Yet another would be to define some appropriate triggered procedures.

All of these approaches are somewhat tedious, with the first perhaps being the least unsatisfactory.

Exercise 8-9.

```
TYPE SNO POSSREP
{ C CHAR CONSTRAINT
         CHAR_LENGTH ( C ) ≥ 2 AND CHAR_LENGTH ( C ) ≤ 5
         AND SUBSTR ( C, 1, 1 ) = 'S'
         AND CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ) ≥ 0
         AND CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ) ≤ 9999 } ;
```

I'm assuming that operators CHAR_LENGTH, SUBSTR, and CAST_AS_INTEGER are available and have the obvious semantics.

Exercise 8-10. `TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;`

I'm assuming the existence of a user defined type called POINT as defined in the body of the chapter.

Exercise 8-11. Type POINT is an example, but there are many others—for example, you might like to think about type PARALLELOGRAM, which can "possibly be represented" in numerous different ways (how many can you think of?). As for type constraints for such a type: Conceptually, each possrep specification *must* include a type constraint; moreover, those constraints must all be logically equivalent. For example:

```
TYPE POINT
     POSSREP CARTESIAN { X FIXED , Y FIXED
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 }
     POSSREP POLAR { R FIXED , THETA FIXED
             CONSTRAINT R ≤ 100.0 } ;
```

Whether some shorthand could be provided that would effectively allow us to specify the constraint just once is a separate issue, beyond the scope of this book.

Exercise 8-12. A line segment can possibly be represented by its begin and end points or by its midpoint, length, and slope (angle of inclination).

Exercise 8-13. I'll give answers in terms of the INSERT, DELETE, and UPDATE shorthands, not relational assignment as such:

CX1: INSERT into S, UPDATE of STATUS in S

CX2: INSERT into S, UPDATE of CITY or STATUS in S

CX3: INSERT into S, UPDATE of SNO in S

CX4: INSERT into S, UPDATE of SNO or CITY in S

CX5: UPDATE of STATUS in S, INSERT into SP, UPDATE of SNO or PNO in SP (I'm assuming here that constraint CX6, the foreign key constraint from SP to S, is being enforced)

CX6: DELETE from S, UPDATE of SNO in S, INSERT into SP, UPDATE of SNO in SP

CX7: INSERT into LS or NLS, UPDATE of SNO in LS or NLS

CX8: INSERT into S or P, UPDATE of SNO or CITY in S, UPDATE of PNO or CITY in P

CX9: UPDATE of SNO or STATUS in S

Exercise 8-14. No, though there's no particular reason why it shouldn't if it were thought desirable.

Exercise 8-15. (The following answer is a little simplified but captures the essence of what's going on.) Let *c* be a base table constraint on table *T*; then the CREATE ASSERTION counterpart to *c* is logically of the form FORALL *r* (*c*)—or, in terms a little closer to concrete SQL syntax, NOT EXISTS *r* (NOT *c*)—where *r* stands for a row in *T*. In other words, the logically necessary universal quantification is implicit in a base table constraint but has to be explicit in an assertion.

Exercise 8-16. The formal reason has to do with the definition of FORALL when the applicable "range" is an empty set; see Chapter 10 for further explanation. **Tutorial D** has nothing directly analogous to base table constraints and thus doesn't display any analogous behavior.

Exercise 8-17.

```
CREATE TABLE S
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( STATUS >= 20 OR SNO NOT IN ( SELECT SNO
                                     FROM   SP
                                     WHERE  PNO = 'P6' ) ) ;

CREATE TABLE P
  ( ... ,
    CONSTRAINT CX5 CHECK
      ( NOT EXISTS ( SELECT *
                     FROM   S NATURAL JOIN SP
                     WHERE  STATUS < 20
                     AND    PNO = 'P6' ) ) ;
```

Observe that in this latter formulation, the constraint specification makes no reference to the base table whose definition it forms part of. Thus, the same specification could form part of the definition of absolutely any base table whatsoever. (It's essentially identical to the CREATE ASSERTION version, anyway.)

Exercise 8-18. The boolean expression in constraint CX1 is a simple restriction condition; the one in constraint CX5 is more complex. One implication is that a tuple presented for insertion into S can be checked against constraint CX1 without even looking at any of the values currently existing in the database, whereas the same is not true for constraint CX5.

Exercise 8-19. Yes, of course it's possible; constraint CX3 does the trick. But note that, in general, neither a constraint like CX3 nor an explicit KEY specification can guarantee that the specified attribute combination satisfies the irreducibility requirement. (Though it would at least be possible to impose a syntax rule to the effect that if two distinct keys are specified for the same relvar, then neither is allowed to be a proper subset of the other. Such a rule would help, but it still wouldn't do the whole job.)

Exercise 8-20.

```
CREATE ASSERTION CX8 CHECK
   ( ( SELECT COUNT ( * )
       FROM ( SELECT CITY
              FROM   S
              WHERE  SNO = 'S1'
              UNION  CORRESPONDING
              SELECT CITY
              FROM   P
              WHERE  PNO = 'P1' ) AS POINTLESS ) < 2 ) ;
```

Exercise 8-21. Space reasons make it too difficult to show **Tutorial D** and SQL formulations side by side here, so in each case I'll show the former first and the latter second. I omit details of which operations might cause the constraints to be violated.

a.
```
CONSTRAINT CXA IS_EMPTY
   ( P WHERE COLOR = 'Red' AND WEIGHT ≥ 50.0 ) ;

CREATE ASSERTION CXA CHECK ( NOT EXISTS (
   SELECT *
   FROM   P
   WHERE  COLOR = 'Red'
   AND    WEIGHT >= 50.0 ) ) ;
```

b.
```
CONSTRAINT CXB IS_EMPTY (
   ( S WHERE CITY = 'London' )
       WHERE TUPLE { PNO 'P2' } ∉ (!! SP) { PNO } ) ;

CREATE ASSERTION CXB CHECK (
   NOT EXISTS ( SELECT * FROM S
                WHERE  CITY = 'London'
                AND    NOT EXISTS
                     ( SELECT * FROM SP
                       WHERE  SP.SNO = S.SNO
                       AND    SP.PNO = 'P2' ) ) ) ;
```

c.
```
CONSTRAINT CXC COUNT ( S ) = COUNT ( S { CITY } ) ;

CREATE ASSERTION CXC CHECK ( UNIQUE ( SELECT CITY FROM S ) ) ;
```

d.
```
CONSTRAINT CXD COUNT ( S WHERE CITY = 'Athens' ) < 2 ;

CREATE ASSERTION CXD CHECK
     ( UNIQUE ( SELECT CITY FROM S WHERE CITY = 'Athens' ) ) ;
```

e.
```
CONSTRAINT CXE COUNT ( S WHERE CITY = 'London' ) > 0 ;

CREATE ASSERTION CXE CHECK
     ( EXISTS ( SELECT * FROM S WHERE CITY = 'London' ) ) ;
```

f.
```
CONSTRAINT CXF COUNT ( P WHERE COLOR = 'Red'
                         AND WEIGHT < 50.0 ) > 0 ;

CREATE ASSERTION CXF CHECK
     ( EXISTS ( SELECT * FROM P
                WHERE  COLOR = 'Red'
                AND    WEIGHT < 50.0 ) ) ;
```

g.
```
CONSTRAINT CXG CASE
                  WHEN IS_EMPTY ( S ) THEN TRUE
                  ELSE AVG ( S , STATUS ) > 10
               END CASE ;

CREATE ASSERTION CXG CHECK
     ( CASE
          WHEN NOT EXISTS ( SELECT * FROM S ) THEN TRUE
          ELSE ( SELECT AVG ( STATUS ) FROM S ) > 10
       END ) ;
```

h.
```
CONSTRAINT CXH
   CASE
      WHEN IS_EMPTY ( SP ) THEN TRUE
      ELSE IS_EMPTY ( SP WHERE QTY > 2 * AVG ( SP , QTY ) )
   END CASE ;

CREATE ASSERTION CXH CHECK
     ( CASE
          WHEN NOT EXISTS ( SELECT * FROM SP ) THEN TRUE
          ELSE NOT EXISTS ( SELECT * FROM SP
                            WHERE  QTY > 2 *
                                 ( SELECT AVG ( QTY )
                                   FROM   SP ) )
       END ) ;
```

i.
```
CONSTRAINT CXI CASE
    WHEN COUNT ( S ) < 2 THEN TRUE
    ELSE IS_EMPTY ( JOIN
      { ( S WHERE STATUS = MAX ( S { STATUS } ) ) { CITY } ,
        ( S WHERE STATUS = MIN ( S { STATUS } ) ) { CITY } } )
               END CASE ;
```

```
CREATE ASSERTION CXI CHECK ( CASE
   WHEN ( SELECT COUNT ( * ) FROM S ) < 2 THEN TRUE
   ELSE NOT EXISTS
      ( SELECT * FROM S AS X , S AS Y
        WHERE  X.STATUS = ( SELECT MAX ( STATUS ) FROM S )
        AND    Y.STATUS = ( SELECT MIN ( STATUS ) FROM S )
        AND    X.CITY = Y.CITY )
                             END ) ;
```

j.
```
CONSTRAINT CXJ P { CITY } ⊆ S { CITY } ;

CREATE ASSERTION CXJ CHECK ( NOT EXISTS
     ( SELECT * FROM P
       WHERE  NOT EXISTS
            ( SELECT * FROM S
              WHERE  S.CITY = P.CITY ) ) ) ;
```

k.
```
CONSTRAINT CXK IS_EMPTY (
       ( ( EXTEND P ADD ( (!! SP) JOIN S ) { CITY } AS SC )
         WHERE TUPLE { CITY CITY } ∉ SC ) ) ;

CREATE ASSERTION CXK CHECK ( NOT EXISTS
     ( SELECT * FROM P
       WHERE  NOT EXISTS
            ( SELECT * FROM S
              WHERE  S.CITY = P.CITY
              AND    EXISTS
                   ( SELECT * FROM SP
                     WHERE  S.SNO = SP.SNO
                     AND    P.PNO = SP.PNO ) ) ) ) ;
```

l.
```
CONSTRAINT CXL
   COUNT ( ( ( S WHERE CITY = 'London' ) JOIN SP ) { PNO } ) >
   COUNT ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ) { PNO } ) ;

CREATE ASSERTION CXL CHECK (
 ( SELECT COUNT ( DISTINCT PNO ) FROM S , SP
   WHERE  S.SNO = SP.SNO
   AND    S.CITY = 'London' ) >
 ( SELECT COUNT ( DISTINCT PNO ) FROM S , SP
   WHERE  S.SNO = SP.SNO
   AND    S.CITY = 'Paris' ) ) ;
```

m.
```
CONSTRAINT CXM
   SUM ( ( ( S WHERE CITY = 'London' ) JOIN SP ) , QTY ) >
   SUM ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ) , QTY ) ;

CREATE ASSERTION CXM CHECK (
 ( SELECT COALESCE ( SUM ( QTY ) , 1 ) FROM S , SP
   WHERE  S.SNO = SP.SNO
   AND    S.CITY = 'London' ) >
 ( SELECT COALESCE ( SUM ( QTY ) , 0 ) FROM S , SP
   WHERE  S.SNO = SP.SNO
   AND    S.CITY = 'Paris' ) ) ;
```

n.
```
CONSTRAINT CXN IS_EMPTY
   ( ( SP JOIN P ) WHERE QTY * WEIGHT > 20000.0 ) ;

CREATE ASSERTION CXN CHECK
     ( NOT EXISTS ( SELECT * FROM SP NATURAL JOIN P
                    WHERE  QTY * WEIGHT > 20000.0 ) ) ;
```

Exercise 8-22. It guarantees that the constraint is satisfied by an empty database (i.e., one containing no relvars).

Exercise 8-23. Suppose we were to define a relvar SC with attributes SNO and CITY and predicate *Supplier SNO has no office in city CITY*. Suppose further that supplier S1 has an office in just ten cities. Then *The Closed World Assumption* would imply that relvar SC must have *n*-10 tuples for supplier S1, where *n* is the total number of valid cities (possibly in the entire world)!

Exercise 8-24. We need a multiple assignment (if we are to do the delete in a single statement as requested):

```
DELETE S WHERE SNO = x , DELETE SP WHERE SNO = x ;
```

Exercise 8-25. These constraints can't be expressed declaratively (neither SQL nor **Tutorial D** currently has any direct support for transition constraints; triggered procedures can be used, but the details are beyond the scope of this book). However, here are formulations using the "primed relvar name" convention discussed briefly in the section "Miscellaneous Issues":

a.
```
CONSTRAINT CXA
    IS_EMPTY ( ( ( S' WHERE CITY = 'Athens' ) { SNO } ) JOIN S )
               WHERE CITY ≠ 'Athens'
               AND   CITY ≠ 'London'
               AND   CITY ≠ 'Paris' )
AND IS_EMPTY ( ( ( S' WHERE CITY = 'London' ) { SNO } ) JOIN S )
               WHERE CITY ≠ 'London'
               AND   CITY ≠ 'Paris' ) ;
```

b.
```
CONSTRAINT CXB IS_EMPTY
   ( P WHERE SUM ( !! SP , QTY ) > SUM ( !! SP' , QTY ) ) ;
```

c.
```
CONSTRAINT CXC IS_EMPTY
   ( S WHERE SUM ( !! SP' , QTY ) < 0.5 * SUM ( !! SP , QTY ) ) ;
```

The qualification "in a single update" is important because we aren't trying to outlaw the possibility—and in fact we can't—of reducing the total shipment quantity by, say, one third in one update and then another third in another.

Exercise 8-26. See the body of the chapter.

Exercise 8-27. *No answer provided.*

Exercise 8-28. SQL fails to support type constraints for a slightly complicated reason having to do with type inheritance. The details are beyond the scope of this book; if you're interested, you can find a detailed discussion in the book *Databases, Types, and the Relational Model: The Third Manifesto*, by Hugh Darwen and myself (see Appendix D). As for consequences, one is that when you define a type in SQL, you can't even specify the values that make up that type!—except for the a priori constraint imposed by the representation—and so, absent further controls, you can wind up with incorrect data in the database (even nonsensical data, like a shoe size of 1000).

Exercise 8-29. In principle they all apply—though **Tutorial D** in particular deliberately provides no way of defining type constraints, other than a priori ones, for either nonscalar or system defined types.

Exercise 8-30. See the discussion of possibly nondeterministic expressions in Chapter 12. *No further answer provided.*

Chapter 9

Exercise 9-1.

```
VAR NON_COLOCATED VIRTUAL
  ( S { SNO } JOIN P { PNO } ) MINUS ( S JOIN P ) { SNO , PNO }
    KEY { SNO , PNO } ;

CREATE VIEW NON_COLOCATED AS
       SELECT SNO , PNO
       FROM   S , P
       WHERE  S.CITY <> P.CITY
        /* UNIQUE ( SNO , PNO ) */ ;
```

Exercise 9-2. Substituting the view definition for the view reference in the outer FROM clause (and showing all name qualifiers explicitly), we obtain:

```
SELECT DISTINCT LSSP.STATUS , LSSP.QTY
FROM ( SELECT S.SNO , S.SNAME , S.STATUS , SP.PNO , SP.QTY
       FROM   S NATURAL JOIN SP
       WHERE  S.CITY = 'London' ) AS LSSP
WHERE  LSSP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'London' )
```

This simplifies to:

```
SELECT DISTINCT STATUS , QTY
FROM   S NATURAL JOIN SP
WHERE  CITY = 'London'
AND    PNO IN
     ( SELECT PNO
       FROM   P
       WHERE  CITY <> 'London' )
```

Exercise 9-3. The sole key is {SNO,PNO}. The predicate is: *Supplier SNO is under contract, is named SNAME, has status STATUS, has some city, and supplies part PNO in quantity QTY.*

Exercise 9-4.

a.
```
( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                                             WHERE COLOR = 'Green'
```

b.
```
( EXTEND ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
         ADD ( WEIGHT + 5.3 AS WTP ) ) { PNO , WTP }
```

c.
```
INSERT ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
  RELATION { TUPLE { PNO 'P99' , WEIGHT 12.0 , COLOR 'Purple' } } ;
```

Observe that this INSERT is logically equivalent to a relational assignment in which the target is specified as something other than a simple relvar reference. The ability to update views implies that such assignments must indeed be legitimate, both syntactically and semantically. Similar remarks apply to parts d. and e. below.

d.
```
DELETE ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                                         WHERE WEIGHT < 9.0 ;
```

e.
```
UPDATE ( ( P WHERE WEIGHT > 14.0 ) { PNO , WEIGHT , COLOR } )
                   WHERE WEIGHT = 18.0 : { COLOR := 'White' } ;
```

As a subsidiary exercise, what further simplifications can be applied to the foregoing expansions?

Exercise 9-5.

a.
```
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
             { PNO , WT , COLOR } ) WHERE COLOR = 'Green'
```

b.
```
( EXTEND
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
{ PNO , WT , COLOR } ) ADD ( WT + 5.3 AS WTP ) ) { PNO , WTP }
```

c.
```
INSERT
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
    { PNO , WT , COLOR } )
RELATION { TUPLE { PNO 'P99' , WT 12.0 , COLOR 'Purple' } } ;
```

d.
```
DELETE
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
    { PNO , WT , COLOR } ) WHERE WT < 9.0 ;
```

e.
```
UPDATE
( ( ( EXTEND P ADD ( WEIGHT * 454 AS WT ) ) WHERE WT > 6356.0 )
{ PNO , WT , COLOR } ) WHERE WT = 18.0 : { COL := 'White' } ;
```

Exercise 9-6. Here first is an SQL version of the view definition from Exercise 9-4:

```
CREATE VIEW HEAVYWEIGHT AS
       SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ;
```

For parts a.-e., I first show an SQL analog of the **Tutorial D** formulation, followed by the expanded form:

a.

```
SELECT *
FROM   HEAVYWEIGHT
WHERE  COL = 'Green'

SELECT *
FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ) AS POINTLESS
WHERE  COL = 'Green'
```

I leave further simplification, here and in subsequent parts, as a subsidiary exercise (barring explicit statements to the contrary).

b.

```
SELECT PNO , WT + 5.3 AS WTP
FROM   HEAVYWEIGHT

SELECT PNO , WT + 5.3 AS WTP
FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
       FROM   P
       WHERE  WEIGHT > 14.0 ) AS POINTLESS
```

c.

```
INSERT INTO HEAVYWEIGHT ( PNO , WT , COL )
            VALUES ( 'P99' , 12.0 , 'Purple' ) ;

INSERT INTO ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
              FROM   P
              WHERE  WEIGHT > 14.0 ) ( PNO , WT , COL )
              VALUES ( 'P99' , 12.0 , 'Purple' ) ;
```

Interestingly, this INSERT violates SQL's syntax rules (because the target isn't specified as a simple table reference); thus, explaining SQL's view processing in terms of substitution alone doesn't work here. So the foregoing INSERT has to be transformed into something else—though exactly what is rather hard to say.

d.

```
DELETE FROM ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
              FROM   P
              WHERE  WEIGHT > 14.0 ) WHERE WT < 9.0 ;
```

Again the transformed version isn't valid SQL syntax, but this time a valid equivalent is a little easier to find:

```
DELETE FROM P WHERE WEIGHT > 14.0 AND WEIGHT < 9.0 ;
```

(A "no op," as you'll observe. Do you think the optimizer will be able to recognize this fact?)

e.
```
UPDATE ( SELECT PNO , WEIGHT AS WT , COLOR AS COL
         FROM   P
         WHERE  WEIGHT > 14.0 )
SET    COL = 'White'
WHERE  WT = 18.0 ;
```

Syntactically valid equivalent:

```
UPDATE P
SET    COLOR = 'White'
WHERE  WEIGHT = 18.0 AND WEIGHT > 14.0 ;
```

SQL answers to Exercise 9-5: *None provided.*

Exercise 9-7. Probably only HEAVYWEIGHT (first version), since that's probably the only one that will be regarded as updatable.

Exercise 9-8. The sole key is {PNO} in both cases. The predicate for the view in Exercise 9-4 is: *Part PNO is used in the enterprise, has some name and some city, and has color COL and weighs WT pounds (and WT is greater than 14).* The predicate for the view in Exercise 9-5 is: *Part PNO is used in the enterprise, has some name and some city, and has color COL and weighs WT grams (and WT is greater than 6356).* As for the total relvar constraints: For the view in Exercise 9-4, the constraint is basically the same as that for relvar P, modified to take account of the renaming and projection, AND the constraint that the weight must be greater than 14. The constraint for the view in Exercise 9-5 is analogous.

Exercise 9-9. Here are some:

- If users are to operate on views instead of base relvars, it's clear that those views should look to the user as much like base relvars as possible. In accordance with *The Principle of Interchangeability*, in fact, the user shouldn't have to know they're views at all but should be able to treat them as if they were base relvars. And just as the user of a base relvar needs to know what keys that base relvar has (in general), so the user of a view needs to know what keys that view has (again, in general). Explicitly declaring those keys is one way to make that information available.
- The DBMS might be unable to deduce keys for itself (this is almost certainly the case with SQL products on the market today). Explicit declarations are thus likely to be the only means available (to the DBA, that is) of informing the DBMS—as well as the user—of the existence of such keys.
- Even if the DBMS were able to deduce keys for itself, explicit declarations would at least enable the system to check that its deductions and the DBA's explicit specifications were consistent.
- The DBA might have some knowledge that the DBMS doesn't, and might thus be able to improve on the DBMS's deductions.

- As noted in the body of the chapter, such a facility could provide a simple and convenient way of stating certain important constraints that could otherwise be stated only in a very circumlocutory fashion.

  Subsidiary exercise: Which if any of the foregoing points do you think apply not just to key constraints in particular but to integrity constraints in general?

Exercise 9-10. One example is as follows: The suppliers relvar is equal to the join of its projections on {SNO,SNAME}, {SNO,STATUS}, and {SNO,CITY}—just so long as appropriate constraints are in force, that is (right?). So we could make the projections base relvars and make the join a view.

Exercise 9-11. Here are some pertinent observations. First, the replacement process itself involves several steps, which might be summarized as follows:

```
/* define the new base relvars */

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

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

/* copy the data to the new base relvars */

INSERT LS  ( S WHERE CITY = 'London' ) ;

INSERT NLS ( S WHERE CITY ≠ 'London' ) ;

/* drop the old relvar */

DROP VAR S ;

/* create the desired view */

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

Now we must do something about the foreign key in relvar SP that references the old base relvar S. Clearly, it would be best if that foreign key could now be taken as referring to the view S instead;* if this is impossible (as it typically is in today's products), then we might want to define another base relvar as follows:

```
VAR SS BASE RELATION { SNO CHAR } KEY { SNO } ;
```

And copy the data (obviously this must be done before dropping relvar S):

```
INSERT SS S { SNO } ;
```

* Indeed, logical data independence is a strong argument in favor of allowing constraints in general to be defined for views as well as base relvars. See the subsidiary exercise at the end of the answer to Exercise 9.9.

Now we need to add the following foreign key specification to the definitions of relvars LS and NLS:

```
FOREIGN KEY { SNO } REFERENCES SS
```

Finally, we must change the specification for the foreign key {SNO} in relvar SP to refer to SS instead of S.

Exercise 9-12. a. *No answer provided*—except to note that if it's difficult to answer the question for some product, then that very fact is part of the point of the exercise in the first place. b. As for part a. c. *No answer provided.*

Exercise 9-13. For the distinction, see the body of the chapter. SQL doesn't support snapshots at the time of writing. (It does support CREATE TABLE AS—see the answer to Exercise 1-16 in Chapter 1—which allows a base table to be initialized when it's created, but CREATE TABLE AS has no REFRESH option.)

Exercise 9-14. "Materialized view" is a deprecated term for a snapshot. The term is deprecated because it muddies concepts that are logically distinct and ought to be kept distinct—by definition, views simply aren't materialized, so far as the model is concerned—and it's leading us into a situation in which we no longer have a clear term for a concept we did have a clear term for, originally. It should be firmly resisted.* In fact, I'm tempted to go further; it seems to me that people who advocate use of the term "materialized view" are betraying their lack of understanding of the relational model in particular and the distinction between model and implementation in general.

Exercise 9-15. First, here's a definition of Design b. in terms of Design a.:

```
VAR SSP VIRTUAL ( S JOIN SP )
    KEY { SNO , PNO } ;

VAR XSS VIRTUAL ( S NOT MATCHING SP )
    KEY { SNO } ;
```

And here's a definition of Design a. in terms of Design b.:

```
VAR S VIRTUAL ( XSS D_UNION SSP { ALL BUT PNO , QTY } )
    KEY { SNO } ;

VAR SP VIRTUAL ( SSP { SNO , PNO , QTY } )
    KEY { SNO , PNO } ;
```

The applicable database constraints for the two designs can be stated as follows:

```
CONSTRAINT DESIGN_A IS_EMPTY ( SP { SNO } MINUS S { SNO } ) ;

CONSTRAINT DESIGN_B IS_EMPTY ( SSP { SNO } JOIN XSS { SNO } ) ;
```

Given these constraints, the designs are information equivalent. But Design a. is superior, because the relvars in that design are both fully normalized; by contrast, relvar SSP in Design b. isn't fully normalized (in fact, it isn't even in second normal form), meaning it displays much redundancy and is thereby potentially subject to a variety of

* I realize I've probably already lost this battle, but I'm an eternal optimist.

"update anomalies." Consider also what happens with Design b. if some supplier ceases to supply any parts, or used not to but now does. Further discussion of the problems with Design b. is beyond the scope of this book, but several of the publications listed in Appendix D discuss such problems in more detail (see also Appendix B).

Incidentally, I note in passing that—given that {SNO} is a key for relvar S—constraint DESIGN_A here exemplifies another way of formulating a referential constraint.

Exercise 9-16. (You might want to review the section "The Reliance on Attribute Names" in Chapter 6 before reading this answer.) Yes, views should indeed have been sufficient to solve the logical data independence problem. But the trouble with views as conventionally understood is that a view definition specifies both the application's perception of some portion of the database *and* the mapping between that perception and the database "as it really is." In order to achieve the kind of data independence I'm talking about here, those two specifications need to be kept separate.

Chapter 10

Exercise 10-1. See the answer to Exercise 4-10 in Chapter 4.

Exercise 10-2. First of all, it's easy to see that we don't need both OR and AND, because (e.g.)

```
p AND q  ≡  NOT ( NOT ( p ) OR NOT ( q ) )
```

(This equivalence is easily established by means of truth tables.) It follows that we can freely use both OR and AND in what follows.

Now consider the connectives involving a single proposition *p*. Let *c*(*p*) be the connective under consideration. Then the possibilities are as follows:

```
c(p)  ≡  p OR NOT ( p )    /* always TRUE  */
c(p)  ≡  p AND NOT ( p )   /* always FALSE */
c(p)  ≡  p                 /* identity     */
c(p)  ≡  NOT ( p )         /* NOT          */
```

Now consider the connectives involving two propositions *p* and *q*. Let *c*(*p*,*q*) be the connective under consideration. Then the possibilities are as follows:

```
c (p,q)  ≡  p OR NOT ( p ) OR q OR NOT ( q )
c (p,q)  ≡  p AND NOT ( p ) AND q AND NOT ( q )
c (p,q)  ≡  p
c (p,q)  ≡  NOT ( p )
c (p,q)  ≡  q
c (p,q)  ≡  NOT ( q )
c (p,q)  ≡  p OR q
c (p,q)  ≡  p AND q
c (p,q)  ≡  p OR NOT ( q )
c (p,q)  ≡  p AND NOT ( q )
c (p,q)  ≡  NOT ( p ) OR q
c (p,q)  ≡  NOT ( p ) AND q
c (p,q)  ≡  NOT ( p ) OR NOT ( q )
c (p,q)  ≡  NOT ( p ) AND NOT ( q )
c (p,q)  ≡  ( NOT ( p ) OR q ) AND ( NOT ( q ) OR p )
c (p,q)  ≡  ( NOT ( p ) AND q ) OR ( NOT ( q ) AND p )
```

As a subsidiary exercise, and in order to convince yourself that the foregoing definitions do indeed cover all of the possibilities, you might like to construct the corresponding truth tables.

Turning to part (b) of the exercise: Actually there are two such primitives, NOR and NAND, often denoted by a down arrow, "↓" (the *Peirce arrow*) and a vertical bar, "|" (the *Sheffer stroke*), respectively. Here are the truth tables:

NOR	T	F
T	F	F
F	F	T

NAND	T	F
T	F	T
F	T	T

As these tables suggest, *p*↓*q* ("*p* NOR *q*") is equivalent to NOT (*p* OR *q*) and *p*|*q* ("*p* NAND *q*") is equivalent to NOT (*p* AND *q*). In what follows, I'll concentrate on NOR (I'll leave NAND to you). Observe that this connective can be thought of as "neither nor" ("neither the first operand nor the second is true"). I now show how to define NOT, OR, and AND in terms of this operator:

```
NOT ( p )    ≡  p ↓ p
p OR q       ≡  ( p ↓ q ) ↓ ( p ↓ q )
p AND q      ≡  ( p ↓ p ) ↓ ( q ↓ q )
```

For example, let's take a closer look at the "*p* AND *q*" case:

p	*q*	*p*↓*p*	*q*↓*q*	(*p*↓*p*) ↓ (*q*↓*q*)
T	T	F	F	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	F

This truth table shows that the expression (*p*↓*p*)↓(*q*↓*q*) is equivalent to *p* AND *q*, because its first, second, and final columns are identical to the pertinent columns in the truth table for AND:

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

Since we've already seen that all of the other connectives can be expressed in terms of NOT, OR, and AND, the overall conclusion follows.

Exercise 10-3. "The sun is a star" and "the moon is a star" are both propositions, though the first is true and the second false.

Exercise 10-4. *The sun* satisfies the predicate, *the moon* doesn't.

Exercise 10-5. A parameter can be replaced by any argument whatsoever, just so long as it's of the right type. A designator isn't, and in fact can't be, replaced by anything at all; instead—just like a variable reference in a programming language—it simply "designates," or denotes, the value of the pertinent variable at the pertinent time (i.e., when the constraint is checked, in the case at hand).

Exercise 10-6. "Get names of suppliers such that there exists a shipment—a *unique* shipment, that is—linking them to all parts." Note that this query will return either (a) all supplier names, if the cardinality of relvar P is less than two, or (b) an empty result, otherwise.

Exercise 10-7. The following SQL expressions are deliberately meant to be as close to their relational counterparts as possible.

Example 1: Get all pairs of supplier numbers such that the suppliers concerned are colocated.

```
SELECT SX.SNO AS SA , SY.SNO AS SB
FROM   S AS SX , S AS SY
WHERE  SX.CITY = SY.CITY
AND    SX.SNO < SY.SNO
```

Example 2: Get supplier names for suppliers who supply at least one red part.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PX
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = PX.PNO
              AND    PX.COLOR = 'Red' ) )
```

Example 3: Get supplier names for suppliers who supply at least one part supplied by supplier S2.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  EXISTS
     ( SELECT *
       FROM   SP AS SPX
       WHERE  EXISTS
            ( SELECT *
              FROM   SP AS SPY
              WHERE  SX.SNO = SPX.SNO
              AND    SPX.PNO = SPY.PNO
              AND    SPY.SNO = 'S2' ) )
```

Example 4: Get supplier names for suppliers who don't supply part P2.

```
SELECT DISTINCT SX.SNAME
FROM   S AS SX
WHERE  NOT EXISTS
          ( SELECT *
            FROM   SP AS SPX
            WHERE  SPX.SNO = SX.SNO
            AND    SPX.PNO = 'P2' )
```

Example 5: For each shipment, get full shipment details, including total shipment weight.

```
SELECT SPX.* , PX.WEIGHT * SPX.QTY AS SHIPWT
FROM   P AS PX , SP AS SPX
WHERE  PX.PNO = SPX.PNO
```

Example 6: For each part, get the part number and the total shipment quantity.

```
SELECT PX.PNO ,
     ( SELECT COALESCE ( SUM ( ALL SPX.QTY ) , 0 )
       FROM   SP AS SPX
       WHERE  SPX.PNO = PX.PNO ) AS TOTQ
FROM   P AS PX
```

Example 7: Get part cities that store more than five red parts.

```
SELECT DISTINCT PX.CITY
FROM   P AS PX
WHERE  ( SELECT COUNT ( * )
         FROM   P AS PY
         WHERE  PY.CITY = PX.CITY
         AND    PY.COLOR = 'Red' ) > 5
```

Exercise 10-8. The following truth table shows that AND is associative; the proof for OR is analogous.

p	*q*	*r*	*p* AND *q*	(*p* AND *q*) AND *r*	(*q* AND *r*)	*p* AND (*q* AND *r*)
T	T	T	T	T	T	T
T	T	F	T	F	F	F
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	F	F	T	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Exercise 10-9. a. Not valid (suppose *x* ranges over an empty set and *q* is TRUE; then EXISTS *x* (*q*) is FALSE). b. Not valid (suppose *x* ranges over an empty set and *q* is FALSE; then FORALL *x* (*q*) is TRUE). c. Valid. d. Valid. e. Not valid (suppose *x* ranges over an empty set; then FORALL *x* (*p*(*x*)) is TRUE but EXISTS *x* (*p*(*x*)) is FALSE, and TRUE ⇒ FALSE is FALSE). f. Not valid (suppose *x* ranges over an empty set; then EXISTS *x* (TRUE) is FALSE). g. Not valid (suppose *x* ranges over an empty set; then FORALL *x* (FALSE) is TRUE). h. Valid. i. Not valid (e.g., saying that exactly one integer is equal to zero isn't the same as saying that all integers are equal to zero). j. Not valid (e.g., the fact that all days are 24 hours long and the fact there exists at least one day that's 24 hours long don't together imply that exactly one day is 24 hours long). k. Valid. Note that (valid!) equivalences and implications like those under discussion here can be used as a basis for a set of calculus expression transformation rules, much like the algebraic expression transformation rules discussed in Chapter 6.

Exercise 10-10. a. Valid. b. Valid. c. Valid. d. Valid. e. Not valid (e.g., saying that for every integer *y* there exists a greater integer *x* isn't the same as saying there exists an integer *x* that's greater than all integers *y*). f. Valid.

Exercise 10-11. a. Valid. b. Valid.

Exercise 10-12. I give solutions only where there's some significant point to be made regarding the solution in question. *Exercises from Chapter 6*:

6-12. The following relational calculus expressions denote TABLE_DEE and TABLE_DUM, respectively:

```
{ } WHERE TRUE

{ } WHERE FALSE
```

And this expression denotes the projection of the current value of relvar S over no attributes:

```
{ } WHERE EXISTS ( SX )
```

The relational calculus isn't normally considered as having a direct counterpart to **Tutorial D**'s *r*{ALL BUT ...}, but there's no reason in principle why it shouldn't.

6-15. The relational calculus isn't normally considered as having a direct counterpart to **Tutorial D**'s D_UNION, but there's no reason in principle why it shouldn't.

Exercise 10-12. (cont.) *Exercises from Chapter 7*:

7-1.

d. `{ PX } WHERE SUM ( SPX WHERE SPX.PNO = PX.PNO , QTY ) < 500`

e. `{ PX } WHERE EXISTS ( SX WHERE SX.CITY = PX.CITY )`

j. `{ PX , COUNT ( SPX WHERE SPX.PNO = PX.PNO ) AS SCT }`

7-8. A relational calculus analog of the **Tutorial D** expression SP GROUP ({} AS X) is:

```
{ SPX , { } AS X }
```

7-10. Here's a relational calculus analog of the **Tutorial D** expression (R4 WHERE TUPLE {PNO 'P2'} ∈ PNO_REL){SNO}:

```
RANGEVAR RX RANGES OVER R4 ,
RANGEVAR RY RANGES OVER RX.PNO_REL ;

RX.SNO WHERE EXISTS ( RY WHERE RY.PNO = 'P2' )
```

Note that the definition of RY here depends on that of RX (the two definitions are separated by a comma, not a semicolon, and are thereby bundled into a single operation).

And here's a relational calculus analog of the **Tutorial D** expression ((R4 WHERE SNO = 'S2') UNGROUP (PNO_REL)){PNO}:

```
RY.PNO WHERE RX.SNO = 'S2'
```

7-11. `{ SX , { SPX.PNO WHERE SPX.SNO = SX.SNO } AS PNO_REL }`

7-12. In practice we need analogs of the conventional INSERT, DELETE, and UPDATE (and relational assignment) operators that are in keeping with a calculus style rather than an algebraic one (and this observation is true regardless of whether we're talking about relvars with RVAs, as in the present context, or without them). Further details are beyond the scope of the present book, but in any case are straightforward. *No further answer provided.*

Exercise 10-12. (cont.) *Exercises from Chapter 8: No answers provided.*

Exercise 10-12. (cont.) *Exercises from Chapter 9: No answers provided.*

Exercise 10-13. Recall from the body of the chapter that the set over which a range variable ranges is always the body of some relation—usually *but not always* the relation that's the current value of some relvar (emphasis added). In this example, the range variable ranges over what is, in effect, a union:

```
RANGEVAR CX RANGES OVER { SX.CITY } , { PX.CITY } ;

{ CX } WHERE TRUE
```

Note that the definition of range variable CX makes use of range variables SX and PX, which I assume to have been previously defined.

Exercise 10-14. In order to show that SQL is relationally complete, it's sufficient* to show that (a) there exist SQL expressions for each of the algebraic operators restrict, project, product, union, and difference, and that (b) the operands to those SQL expressions can be arbitrary SQL expressions in turn.

First of all, as we know, SQL effectively does support the relational algebra RENAME operator, thanks to the availability of the optional AS specification on items in the SELECT clause.† We can therefore ensure that tables do all have proper column names, and hence that the operands to product, union, and difference in particular satisfy the requirements of the algebra with respect to column naming. Furthermore—provided those column naming requirements are indeed satisfied—the SQL column name inheritance rules in fact coincide with those of the algebra as described in Chapter 6.

Here then are SQL expressions corresponding approximately to the five primitive operators:

```
Algebra                  SQL

R WHERE p                 SELECT * FROM R WHERE p

R { A , B , ... , C }     SELECT DISTINCT A , B , ... , C FROM R

R1 TIMES R2               SELECT * FROM R1 , R2

R1 UNION R2               SELECT * FROM R1
                          UNION  CORRESPONDING
                          SELECT * FROM R2
```

* Sufficient, yes—but is it necessary?

† To state the matter a little more precisely: An SQL analog of the algebraic expression T RENAME (A AS B) is the (extremely inconvenient!) SQL expression SELECT A AS B, X, Y, …, Z FROM T (where X, Y, ..., Z are all of the columns of T apart from A, and I choose to overlook the fact that the SQL expression results in a table with a left to right ordering to its columns).

```
R1 MINUS R2                SELECT * FROM R1
                           EXCEPT CORRESPONDING
                           SELECT * FROM R2
```

Moreover, (a) each of *R1* and *R2* in the SQL expressions shown above is in fact a table reference, and (b) if we take any of the five SQL expressions shown and enclose it in parentheses, what results is a table reference in turn.[*] It follows that SQL is indeed relationally complete.

NOTE

Actually there's a glitch in the foregoing—SQL fails to support projection over no columns at all (because it doesn't support empty SELECT clauses). As a consequence, it doesn't support TABLE_DEE or TABLE_DUM, and it therefore isn't 100 percent relationally complete after all.

Exercise 10-15. Let *TP* and *DC* denote the propositions "the database contains only true propositions" and "the database is consistent," respectively. Then the first bullet item says:

```
IF TP THEN DC
```

The second says:

```
IF NOT ( DC ) THEN NOT ( TP )
```

Appealing to the definition of logical implication in terms of NOT and OR, it's easy to see that these two propositions are indeed equivalent. (In fact, the second is the *contrapositive* of the first. See Chapter 11 for further explanation.)

Exercise 10-16. No—though textbooks on logic usually claim the opposite, and in practice it's "usually" achievable. The paper "A Remark on Prenex Normal Form" (see Appendix D) goes into details.

Chapter 11

Exercise 11-1. Here first is an SQL formulation of the query "Get suppliers SX such that for all parts PX and PY, if PX.CITY ≠ PY.CITY, then SX doesn't supply both of them." (How different is this formulation from the one shown in the body of the chapter?)

```
SELECT SX.*
FROM   S AS SX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   P AS PY
              WHERE  PX.CITY <> PY.CITY
              AND    EXISTS
```

* I ignore the fact that SQL would require such a table reference to include a pointless range variable definition.

```
                    ( SELECT *
                      FROM   SP AS SPX
                      WHERE  SPX.SNO = SX.SNO
                      AND    SPX.PNO = PX.PNO )
               AND    EXISTS
                    ( SELECT *
                      FROM   SP AS SPX
                      WHERE  SPX.SNO = SX.SNO
                      AND    SPX.PNO = PY.PNO ) ) )
```

Next, you were asked to give SQL formulations (a) using GROUP BY and HAVING, (b) not using GROUP BY and HAVING, of the following queries:

- Get supplier numbers for suppliers who supply *N* different parts for some *N* > 3.
- Get supplier numbers for suppliers who supply *N* different parts for some *N* < 4.

Here are GROUP BY and HAVING formulations:

```
SELECT SNO
FROM   SP
GROUP  BY SNO
HAVING COUNT ( * ) > 3

SELECT SNO
FROM   SP
GROUP  BY SNO
HAVING COUNT ( * ) < 4
UNION  CORRESPONDING
SELECT SNO
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM   SP )
```

And here are non GROUP BY, non HAVING formulations:

```
SELECT SNO
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) > 3

SELECT SNO
FROM   S
WHERE  ( SELECT COUNT ( * )
         FROM   SP
         WHERE  SP.SNO = S.SNO ) < 4
```

You were also asked: What do you conclude from this exercise? Well, one thing I conclude is that we need to be very circumspect in our use of GROUP BY and HAVING. Observe in particular that the natural language queries were symmetric, which the GROUP-BY/HAVING formulations clearly aren't. By contrast, the non GROUP BY, non HAVING formulations *are* symmetric.

Exercise 11-2. *No answer provided.*

Exercise 11-3. *No answer provided.*

Exercise 11-4. I'm certainly not going to give anything like a complete answer to this exercise, but I will at least observe that the following equivalences allow certain algebraic expressions to be converted into calculus ones and vice versa:

- `r WHERE bx1 AND bx2  ≡  ( r WHERE bx1 ) JOIN ( r WHERE bx2 )`
- `r WHERE bx1 OR bx2   ≡  ( r WHERE bx1 ) UNION ( r WHERE bx2 )`
- `r WHERE NOT ( bx )   ≡  r MINUS ( r WHERE bx )`

Other transformations were discussed in passing throughout the body of the book (from Chapter 6 on).

Chapter 12

Exercise 12-1.

`A NATURAL JOIN B` : Legal
`A INTERSECT B` : Illegal
`TABLE A NATURAL JOIN TABLE B` : Illegal
`TABLE A INTERSECT TABLE B` : Legal
`SELECT * FROM A NATURAL JOIN SELECT * FROM B` : Illegal
`SELECT * FROM A INTERSECT SELECT * FROM B` : Legal
`( SELECT * FROM A ) NATURAL JOIN ( SELECT * FROM B )` : Illegal
`( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )` : Legal
`( TABLE A ) NATURAL JOIN ( TABLE B )` : Illegal
`( TABLE A ) INTERSECT ( TABLE B )` : Legal
`( TABLE A ) AS AA NATURAL JOIN ( TABLE B ) AS BB` : Legal
`( TABLE A ) AS AA INTERSECT ( TABLE B ) AS BB` : Illegal
`( ( TABLE A ) AS AA ) NATURAL JOIN ( ( TABLE B ) AS BB )` : Illegal
`( ( TABLE A ) AS AA ) INTERSECT ( ( TABLE B ) AS BB )` : Illegal

One thing I conclude from this exercise is that the rules are very difficult to remember (to say the least). In particular, SQL expressions involving INTERSECT can't always be transformed straightforwardly into their JOIN counterparts.

Exercise 12-2. *No answer provided.*

Exercise 12-3. *No answer provided.*

Appendix B

Exercise B-1. *No answer provided.*

Exercise B-2. The complete set of FDs—what's known, formally, as the *closure*, though it's nothing to do with the closure property of the relational algebra—for relvar SP is as follows:

```
{ SNO , PNO , QTY } → { SNO , PNO , QTY }
{ SNO , PNO , QTY } → { SNO , PNO }
{ SNO , PNO , QTY } → { PNO , QTY }
{ SNO , PNO , QTY } → { SNO , QTY }
{ SNO , PNO , QTY } → { SNO }
{ SNO , PNO , QTY } → { PNO }
{ SNO , PNO , QTY } → { QTY }
{ SNO , PNO , QTY } → { }

{ SNO , PNO }       → { SNO , PNO , QTY }
{ SNO , PNO }       → { SNO , PNO }
{ SNO , PNO }       → { PNO , QTY }
{ SNO , PNO }       → { SNO , QTY }
{ SNO , PNO }       → { SNO }
{ SNO , PNO }       → { PNO }
{ SNO , PNO }       → { QTY }
{ SNO , PNO }       → { }

{ PNO , QTY }       → { PNO , QTY }
{ PNO , QTY }       → { PNO }
{ PNO , QTY }       → { QTY }
{ PNO , QTY }       → { }

{ SNO , QTY }       → { SNO , QTY }
{ SNO , QTY }       → { SNO }
{ SNO , QTY }       → { QTY }
{ SNO , QTY }       → { }

{ SNO }             → { SNO }
{ SNO }             → { }

{ PNO }             → { PNO }
{ PNO }             → { }

{ QTY }             → { QTY }
{ QTY }             → { }

{ }                 → { }
```

Exercise B-3. True ("whenever two tuples have the same value for *A*, they also have the same value for *B*" implies a comparison between the projections of the tuples in question on {*A*,*B*}). *Note:* As I pointed out in the body of the appendix, it does make sense to talk about projections of tuples as well as of relations, and a similar remark applies to several other relational operators as well (e.g., rename, extend). Technically, we say the operators in question are *overloaded*.

Exercise B-4. *No answer provided* (the exercise is easy).

Exercise B-5. Heath's theorem says: If *R*{*A*,*B*,*C*} satisfies the FD *A* → *B*, then *R* is equal to the join of its projections *R1* on {*A*,*B*} and *R2* on {*A*,*C*}. In the following simple proof of this theorem, I use the same informal shorthand for tuples that I used in the body of the appendix.

First I show that no tuple of *R* is lost by taking the projections and then joining those projections back together again. Let (*a*,*b*,*c*) ∈ *R*. Then (*a*,*b*) ∈ *R1* and (*a*,*c*) ∈ *R2*, and so (*a*,*b*,*c*) ∈ *R1* JOIN *R2*.

Next I show that every tuple of the join is indeed a tuple of *R* (in other words, the join doesn't generate any "spurious" tuples). Let (*a*,*b*,*c*) ∈ *R1* JOIN *R2*. In order to generate such a tuple in the join, we must have (*a*,*b*) ∈ *R1* and (*a*,*c*) ∈ *R2*. Hence there must exist a tuple $(a,b',c) \in R$ for some b', in order to generate the tuple $(a,c) \in R2$. We therefore must have $(a,b') \in R1$. Now we have $(a,b) \in R1$ and $(a,b') \in R1$; hence we must have $b = b'$, because $A \rightarrow B$. Hence $(a,b,c) \in R$.

The converse of Heath's theorem would say that if *R*{*A*,*B*,*C*} is equal to the join of its projections on {*A*,*B*} and on {*A*,*C*}, then *R* satisfies the FD $A \rightarrow B$. This statement is false. To show this is so, it's sufficient to exhibit a single counterexample; I'll leave the question of finding such a counterexample to you. (If you give up, you can find one in the answer to Exercise B-20. Consider also relvar SPJ as illustrated in Figure B-5 in the body of the appendix.)

Exercise B-6. See the body of the appendix.

Exercise B-7. See Chapter 5.

Exercise B-8. For the definitions, see the body of the appendix. The former *is* a special case of the latter. For example, relvar S satisfies the trivial FD {CITY,STATUS} → {STATUS}. Applying Heath's theorem, therefore, we see that S satisfies the trivial JD ☆{*AB*,*AC*}, where *A* is {CITY,STATUS}, *B* is {STATUS}, and *C* is {SNO,SNAME}.

Exercise B-9. An FD is basically a statement (actually a proposition) of the form $A \rightarrow B$, where *A* and *B* are each subsets of the heading of *R*. Given that a set of *n* elements has 2^n possible subsets, it follows that each of *A* and *B* has 2^n possible values, and hence an upper limit on the number of possible FDs in *R* is 2^{2n}. Thus, for example, the upper limit for a relvar of degree five is 1,024.

Exercise B-10. Let the specified FD be satisfied by relvar *R*. Now, every tuple *t* of *R* has the same value (namely, the 0-tuple) for that subtuple of *t* that corresponds to the empty set of attributes. If *B* is empty, therefore, the FD $A \rightarrow B$ is trivially true for all possible sets *A* of attributes of *R*; in fact, it's a *trivial* FD (and it isn't very interesting). On the other hand, if *A* is empty, the FD $A \rightarrow B$ means all tuples of *R* have the same value for *B* (since they certainly all have the same value for *A*). And if *B* in turn is "all of the attributes of *R*"—in other words, if *R* has an empty key—then *R* is constrained to contain at most one tuple.

Exercise B-11. Suppose we start with a relvar R with attributes D, P, S, L, T, and C (attribute names corresponding to parameters of the predicate in the obvious way). Then R satisfies the following nontrivial FDs:

```
{ L }         → { D , P , C , T }
{ D , P , C } → { L , T }
{ D , P , T } → { L , C }
{ D , P , S } → { L , C }
```

Thus, a possible set of BCNF relvars (in outline) is:

```
SCHEDULE { L , D , P , C , T }
         KEY { L }
         KEY { D , P , C }
         KEY { D , P , T }

STUDYING { S , L }
         KEY { S , L }
```

STUDYING is in 6NF; SCHEDULE is in 5NF but not 6NF. The FD {D,P,T} → {L,C} has been "lost" in this decomposition; moreover, if we decomposed SCHEDULE into its 6NF projections on {L,D}, {L,P}, {L,C}, and {L,T}—which we could certainly do if we wanted—we would additionally "lose" the FDs {D,P,C} → {L,T} and {D,P,T} → {L,C}. Thus, it's probably not a good idea to perform that further decomposition.

Exercise B-12. Yes, sometimes, though probably not very often; in fact, such a possibility is the whole *raison d'être* for 5NF as opposed to 4NF. The subsection "More on 5NF" in the body of the appendix gives an example of a relvar that can be nonloss decomposed into three projections and not into two. Moreover, that decomposition is probably desirable, because (once again) it reduces redundancy and thereby avoids certain update anomalies that might otherwise occur.

Exercise B-13. Surrogate keys are *not* the same thing as tuple IDs. For one thing (to state the obvious), surrogates identify entities and tuple IDs identify tuples, and there's certainly nothing like a one to one correspondence between entities and tuples. (Think of derived tuples in particular—for example, tuples in the result of some query. In fact, it's not at all clear that derived tuples will have tuple IDs anyway.) Furthermore, tuple IDs usually have performance connotations, but surrogates don't (access to a tuple via its tuple ID is usually assumed to be fast, but no such observation applies to surrogates). Also, tuple IDs are usually concealed from the user, but surrogates mustn't be (because of *The Information Principle*—see Appendix A); in other words, it's probably (and desirably!) not possible to store a tuple ID in a database relvar, while it certainly is possible, and desirable, to store a surrogate in a database relvar. In a nutshell: Surrogate keys have to do with logical design, tuple IDs have to do with physical design.

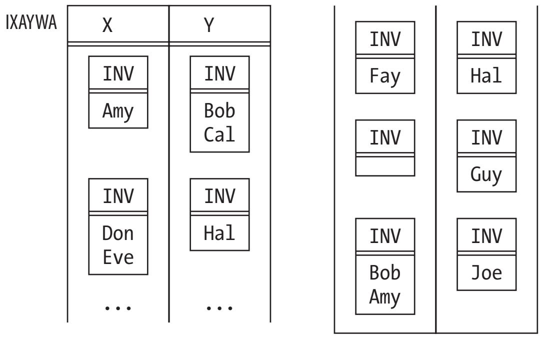
Are surrogate keys a good idea? Well, observe first that the relational model has nothing to say on this question; like the whole business of database design, in fact, whether or not to use surrogate keys has to do with *how to apply* the relational model, not with the relational model as such.

That said, I now have to say that the question of whether surrogate keys are good or bad is far from straightforward. There are strong arguments on both sides: so many such, in fact, that I can't possibly do justice to them all here. Instead, I'll simply refer you to a detailed article of my own on the subject, "Composite Keys" (see Appendix D). *Note:* The article is called "Composite Keys" because surrogate keys are most likely to be useful in practice in situations in which existing keys (and corresponding foreign keys) are composite keys specifically.

One last point on surrogates: Given that a BCNF relvar with no composite keys is "automatically" in 5NF, many people seem to think that simply introducing a surrogate key into a BCNF relvar "automatically" means the relvar is now in 5NF—but it doesn't mean that at all. In particular, if the relvar had a composite key before the introduction of the surrogate, it still has one afterward.

Exercise B-14. For the moment, I'll use the notation $X \rightarrow Y$ to mean that if everyone in the set of people X attends, then everyone in the set of people Y will attend as well (my choice of notation here isn't exactly arbitrary). Until further notice, I'll refer to expressions such as $X \rightarrow Y$ as *statements*.

The first and most obvious design thus consists of a single relvar, IXAYWA ("if X attends, Y will attend"), containing a tuple for each of the given statements (see the following figure, where INV stands for "invitee"). Observe that X and Y in that relvar are RVAs.



But we can do better than this. Of course, I chose the FD notation deliberately. Indeed, the statement "if X attends, Y will attend" is clearly isomorphic to the statement "the FD $X \rightarrow Y$ is satisfied" (imagine a relvar with a boolean attribute for each of Amy, Bob, Cal, and so on, in which each tuple represents a kind of "bitmapped" attendance list that's consistent with the specified conditions). Hence we can use the theory of FDs to reduce the amount of information we need to record explicitly in the relvar. In fact, we can use that theory to find what's called an *irreducible equivalent* to the given set of statements—that is, a (usually small) set of statements that includes no redundant information and has the property that all of the original statements are implied by statements in that set.

Detailed consideration of how to find an irreducible equivalent is beyond the scope of this book; all I want to do here is give a sketch of what's involved. First, I pointed out in Appendix B that if $X \rightarrow Y$ is satisfied, then $X' \rightarrow Y'$ is satisfied for all supersets X' of X and all subsets Y' of Y. We can therefore reduce the amount of information we need to record explicitly by ensuring that every statement $X \rightarrow Y$ we do record is such that the set X is as small as possible and the set Y is as large as possible—meaning, more precisely, that:

- Nobody can be removed from the set X without the resulting statement no longer being true.
- Nobody can be added to the set Y without the resulting statement no longer being true.

Also, we know that the FD $X \rightarrow Y$ is necessarily satisfied for all sets Y such that Y is a subset of X (in particular, $X \rightarrow X$ is always satisfied; for example, "if Amy attends, Amy will attend" is clearly satisfied). Such cases are trivial, and we don't need to record them explicitly at all.

Note too that if relvar IXAYWA contains only those FDs that constitute an irreducible equivalent, then $\{X\}$ will be a key for the relvar.

For further discussion, I refer you to my book *An Introduction to Database Systems* (see Appendix D).

Exercise B-15. The simplest design (in outline) is:

```
EMP  { ENO , ENAME , SALARY }
     KEY { ENO }

PGMR { ENO , LANG }
     KEY { ENO }
     FOREIGN KEY { ENO } REFERENCES EMP
```

Every employee has a tuple in EMP (and EMP has no other tuples). Employees who happen to be programmers additionally have a tuple in PGMR (and PGMR has no other tuples). Note that the join of EMP and PGMR gives full information—employee number, name, salary, and language skill—for programmers (only).

The only significant difference if programmers could have an arbitrary number of language skills is that relvar PGMR would be "all key" (i.e., its sole key would be {ENO,LANG}).

Exercise B-16. Let the projections of R on $\{A,B\}$ and $\{A,C\}$ be $R1$ and $R2$, respectively, and let $(a,b1,c1) \in R$ and $(a,b2,c2) \in R$. Then $(a,b1) \in R1$ and $(a,b2) \in R1$, and $(a,c1) \in R2$ and $(a,c2) \in R2$; thus, $(a,b1,c2) \in J$ and $(a,b2,c1) \in J$, where $J = R1$ JOIN $R2$. But R satisfies the JD ☆$\{AB,AC\}$ and so $J = R$; thus, $(a,b1,c2) \in R$ and $(a,b2,c1) \in R$.

By the way, note that the definition of MVD is symmetric in B and C; thus, R satisfies the MVD $A \rightarrow\rightarrow B$ if and only if it satisfies the MVD $A \rightarrow\rightarrow C$. MVDs always come in pairs in this way. For this reason, it's usual to write such pairs as a single statement, thus:

$A \rightarrow\rightarrow B \mid C$

Exercise B-17. Let C be the attributes of R not included in A or B. By Heath's theorem, if R satisfies the FD $A \rightarrow B$, then it satisfies the JD ☆$\{AB,AC\}$. By definition, however (see Exercise B-16), if R satisfies the JD ☆$\{AB,AC\}$, then it satisfies the MVDs $A \rightarrow\rightarrow B$ and $A \rightarrow\rightarrow C$. Thus, $A \rightarrow B$ implies $A \rightarrow\rightarrow B$.

Exercise B-18. This result is immediate from the definition of multivalued dependency (see Exercise B-16).

Exercise B-19. Exercise B-17 shows that if $K \rightarrow A$, then certainly $K \rightarrow\rightarrow A$. But if K is a key, then $K \rightarrow A$, and the desired conclusion follows immediately.

Exercise B-20. Let C be a certain club, and let relvar $R\{A,B\}$ be such that the tuple (a,b) appears in R if and only if a and b are both members of C. Then R is equal to the cartesian product of its projections $R\{A\}$ and $R\{B\}$; thus, it satisfies the JD ☆$\{A,B\}$ and, equivalently, the following MVDs:

$\{\ \} \rightarrow\rightarrow A \mid B$

These MVDs aren't trivial, since they're certainly not satisfied by all binary relvars, and they're not implied by a superkey either. It follows that *R* isn't in 4NF. However, it is in BCNF, since it's "all key" (see Exercise B-24).

Exercise B-21. First let's introduce three relvars, with the obvious interpretations:

```
REP     { REPNO ,  ... } KEY { REPNO  }
AREA    { AREANO , ... } KEY { AREANO }
PRODUCT { PRODNO , ... } KEY { PRODNO }
```

Next, we can represent the relationships (a) between sales representatives and sales areas and (b) between sales representatives and products by relvars like this:

```
RA { REPNO , AREANO } KEY { REPNO , AREANO }
RP { REPNO , PRODNO } KEY { REPNO , PRODNO }
```

Every product is sold in every area. So if we introduce a relvar

```
AP { AREANO , PRODNO } KEY { AREANO , PRODNO }
```

to represent the relationship between areas and products, then we have the following constraint:

```
CONSTRAINT C1 AP = AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

Note that this constraint implies that AP isn't in 4NF. In fact, AP doesn't give us any information we can't obtain from the other relvars. To be precise, we have:

```
AP { AREANO } = AREA { AREANO }
AP { PRODNO } = PRODUCT { PRODNO }
```

But let's assume for the moment that relvar AP is included in our design anyway.

No two representatives sell the same product in the same area. In other words, given an {AREANO,PRODNO} combination, there's exactly one responsible sales representative, REPNO, so we can introduce a relvar

```
APR { AREANO , PRODNO , REPNO } KEY { AREANO , PRODNO }
```

in which (to make the FD explicit)

```
{ AREANO , PRODNO } → { REPNO }
```

(Specification of {AREANO,PRODNO} as a key is sufficient to express this FD.) Now, however, relvars RA, RP, and AP are all redundant, since they're all projections of APR; they can therefore all be dropped. In place of constraint C1 we now need constraint C2:

```
CONSTRAINT C2 APR { AREANO , PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

This constraint must be stated separately and explicitly (it isn't "implied by keys").

Also, since every representative sells all of that representative's products in all of that representative's areas, we have the additional constraint C3 on relvar APR:

```
{ REPNO } →→ { AREANO } | { PRODNO }
```

(These MVDs are nontrivial, and relvar APR isn't in 4NF.) Again the constraint must be stated separately and explicitly.

Thus the final design consists of the relvars REP, AREA, PRODUCT, and APR, together with the constraints C2 and C3:

```
CONSTRAINT C2 APR { AREANO , PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;

CONSTRAINT C3 APR = APR { REPNO , AREANO } JOIN
              APR { REPNO , PRODNO } ;
```

This exercise illustrates very nicely the point that, in general, normalization might be adequate to represent some of the semantic aspects of a given problem (basically, FDs, MVDs, and JDs that are implied by keys), but explicit statement of additional constraints is needed for other aspects. It also illustrates the point that it might not always be desirable to normalize "all the way" (relvar APR is in BCNF but not in 4NF).

NOTE

As a subsidiary exercise, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration. Might such a design mean that some of the comments in the previous paragraph no longer apply?

Exercise B-22.

```
CONSTRAINT SPJ_JD SPJ = JOIN { SPJ { SNO , PNO } ,
                               SPJ { PNO , JNO } ,
                               SPJ { JNO , SNO } ;
```

Exercise B-23. This is a "cyclic constraint" example. The following design is suitable:

```
REP     { REPNO ,  ... } KEY { REPNO  }
AREA    { AREANO , ... } KEY { AREANO }
PRODUCT { PRODNO , ... } KEY { PRODNO }

RA { REPNO ,  AREANO } KEY { REPNO ,  AREANO }
AP { AREANO , PRODNO } KEY { AREANO , PRODNO }
PR { PRODNO , REPNO  } KEY { PRODNO , REPNO  }
```

Also, the user needs to be informed that the join of RA, AP, and PR does *not* involve any "connection trap":

```
CONSTRAINT NO_TRAP
        ( RA JOIN AP JOIN PR ) { REPNO ,  AREANO } = RA AND
        ( RA JOIN AP JOIN PR ) { AREANO , PRODNO } = AP AND
        ( RA JOIN AP JOIN PR ) { PRODNO , REPNO  } = PR ;
```

NOTE

As with Exercise B-21, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration.

Exercise B-24. a. True. b. True. c. False, though "almost" true. Here's a counterexample: We're given a relvar USA {COUNTRY, STATE} ("STATE is part of COUNTRY"), where COUNTRY is the U.S. in every tuple. This relvar satisfies the FD {} → {COUNTRY}, which is neither trivial nor implied by a key, and so the relvar is not in BCNF (it can be nonloss decomposed into its two unary projections).

Exercise B-25. Yes, they do. What do you conclude?

Exercise B-26. I'm not going to attempt to show any E/R diagrams here, but the point of the exercise is simply to lend weight to the following remarks from Appendix B:

> 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, while it might be acceptable 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 [*italics added*].

APPENDIX D

Suggestions for Further Reading

AS ITS TITLE STATES, THIS APPENDIX GIVES SOME SUGGESTIONS FOR FURTHER READING. I apologize for the fact that the majority of the publications listed are ones for which I'm either the author or a coauthor... They're in alphabetical order by author name, and ascending chronological order within author. *Note:* This book isn't concerned with specific SQL products, and I therefore don't list any product-specific publications below. But many such publications exist, and you'll probably want to refer to one or more of them if you want to apply the ideas discussed in the present book to some individual product.

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

   Among other things, this paper strongly endorses one of the messages of the present book: viz., that (as I put it in the preface) "SQL is ... complicated, confusing, and error prone—much more so, I venture to suggest, than its apologists would have you believe." Here's an extended quote from the introduction to the paper:

   > *SQL is painful.* A big headache that comes with a database system is the SQL language. It is the union of all conceivable features (many of which are rarely used or should be dis-

couraged to use anyway) and is way too complex for the typical application developer. Its core, say selection-projection-join queries and aggregation, is extremely useful, but we doubt that there is wide and wise use of all the bells and whistles. Understanding semantics of SQL (not even of SQL-92), covering all combinations of nested (and correlated) subqueries, null values, triggers, etc. is a nightmare. Teaching SQL typically focuses on the core, and leaves the featurism as a "learning-on-the-job" life experience. Some trade magazines occasionally pose SQL quizzes where the challenge is to express a complicated information request in a single SQL statement. Those statements run over several pages, and are hardly comprehensible. When programmers adopt this style in real applications and given the inherent difficulty of debugging a very high-level "declarative" statement, it is extremely hard if not impossible to gain high confidence that the query is correct in capturing the users' information needs. In fact, good SQL programming in many cases decomposes complex requests into a sequence of simpler SQL statements.

2. 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 (the main one being that the 1969 paper permitted relation valued attributes while the 1970 one didn't). 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. I would like to suggest, politely, that every database professional should read one or both of these papers every year.

3. E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6*. Englewood Cliffs, N.J.: Prentice Hall (1972).

This is the paper in which Codd first formally defined the original relational algebra and relational calculus. Not an easy read, but it repays careful study.

4. 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).

Codd was perhaps the foremost advocate of nulls and three-valued logic 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 does at least touch on most of the arguments on both sides of the issue.

5. Hugh Darwen: "The Role of Functional Dependence in Query Decomposition," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

   This paper gives a set of inference rules by which functional dependences (FDs) satisfied by the relation *r* resulting from an arbitrary relational expression can be inferred from those holding for the relvar(s) mentioned in the expression in question. The set of FDs thus inferred can then be inspected to determine the key constraints satisfied by *r*, thus providing a basis for the key inference rules mentioned in passing in Chapter 4.

6. 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.

7. Hugh Darwen: "How to Handle Missing Information Without Using Nulls" (presentation slides), *http://www.thethirdmanifesto.com* (May 9th, 2003; revised May 16th, 2005).

   Presents one possible approach to the problem identified in the title.

8. C. J. Date: "Fifty Ways to Quote Your Query," *http://www.dbpd.com* (July 1998).

   This paper is referenced in Chapter 6.

9. C. J. Date: "Composite Keys," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

   This paper is referenced in Appendix B.

10. C. J. Date: *An Introduction to Database Systems* (8th edition). Boston, Mass.: Addison-Wesley (2004).

    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; in particular, they include a detailed discussion of SQL's "object/relational" features (REF types, reference values, and so on), explaining why they violate relational principles. *Note:* Other textbooks covering similar territory are references [32], [41], and [42].

11. C. J. Date: *The Relational Database Dictionary, Extended Edition*. Berkeley, Calif.: Apress (2008).

    Many of the definitions given in the body of the present book are based on ones in this reference.

12. C. J. Date: "Double Trouble, Double Trouble," in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

    An extensive and detailed treatment of the problems caused by duplicates. The discussion of duplicates in Chapter 4 is based on an example from this paper.

13. C. J. Date: "What First Normal Form Really Means," in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

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, is that the concept of *atomicity* (in terms of which first normal form was originally defined) has no absolute meaning.

14. C. J. Date: "A Sweet Disorder," in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

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 much less trivial than you might think.

15. 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," all in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

The titles say it all.

16. C. J. Date: "Two Remarks on SQL's UNION," in *Date on Database: Writings 2000–2006*. Berkeley, Calif.: Apress (2006).

This short paper describes some of the weirdnesses that arise in connection with SQL's UNION operator—and by implication its INTERSECT and EXCEPT operators as well—from (a) coercions and (b) duplicate rows.

17. C. J. Date: "A Cure for Madness," in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

A detailed examination of the fact that, very counterintuitively, the SQL expressions SELECT *sic* FROM (SELECT * FROM *t* WHERE *p*) WHERE *q* and SELECT *sic* FROM *t* WHERE *p* AND *q* aren't logically equivalent—even though they ought to be, and even though at least one current SQL product does sometimes transform the former into the latter.

18. C. J. Date: "Why Three- and Four-Valued Logic Don't Work," in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

As noted in the body of the present book, SQL's null support is based on three-valued logic. Actually its implementation of that logic is seriously flawed; but even if it weren't, it would still be advisable not to use it, and this paper explains why.

19. C. J. Date: "The Logic of View Updating," in *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). See *http://www.trafford.com/07-0690*.

This paper, together with a lengthy appendix on the same subject in reference [28], offers evidence in support of the claim that views are always logically updatable, modulo possible violations of either *The Assignment Principle* or **The Golden Rule**.

20. C. J. Date: "The Closed World Assumption," in *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). See *http://www.trafford.com/07-0690*.

The Closed World Assumption is seldom articulated, and yet it forms the basis of almost everything we do when we use a database. This paper examines the assumption in detail; in particular, it shows why it's preferred to its rival, *The Open World Assumption* (on which the "semantic web" is based, incidentally—or so it has been claimed). It also explains how we can still get "don't know" answers when we want them, even from a database without nulls.

21. C. J. Date: "The Theory of Bags: An Investigative Tutorial," in *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). See *http://www.trafford.com/07-0690*.

Among other things, this paper discusses what happens to operators like union when their operands are bags instead of sets.

22. C. J. Date: "Inclusion Dependencies and Foreign Keys" (to appear).

An alternative title for this paper might be *Rethinking Foreign Keys*; it demonstrates among other things that the foreign key notion encompasses far more than it's usually given credit for. It also includes a discussion of the logical differences between foreign keys and pointers. (As hinted in Chapter 2, some writers have claimed that foreign keys are nothing more than traditional pointers in sheep's clothing, but such is not the case.)

23. C. J. Date: "Image Relations" (to appear).

A detailed discussion of the semantics and usefulness of the image relation construct (see Chapter 7).

24. C. J. Date: "Is SQL's Three-Valued Logic Truth Functionally Complete?" (to appear).

Among other things, this paper includes a comprehensive and precise description of SQL's support for nulls and three-valued logic.

25. C. J. Date: "A Remark on Prenex Normal Form" (to appear).

A discussion of prenex normal form, with special reference to the database context.

26. C. J. Date: *Go Faster! The TransRelational™ Approach to DBMS Implementation* (to appear).

A detailed description of *The TransRelational™ Model*, a novel implementation technology mentioned briefly in passing in Appendixes A and B of the present book. *Note:* A brief introduction to that technology can also be found in Appendix A of reference [10].

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

This book provides thorough coverage of the SQL standard as of early 1997. A number of features have been added to the standard since that time (including the so called object/relational features (see reference [28]), but they're mostly irrelevant as far the goal of using SQL relationally is concerned. In my not unbiased opinion, therefore, the book is a good source for more detail on just about every aspect of SQL—at least, in its standard incarnation—touched on in the body of the present book.

28. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Boston, Mass.: Addison-Wesley (2006).

This book introduces and explains *The Third Manifesto*, a detailed proposal for the future of data and database management systems. It includes a precise though somewhat formal definition of the relational model; it also includes a detailed proposal for the necessary supporting type theory (including a comprehensive model of type inheritance) and a comprehensive description of **Tutorial D**.

29. C. J. Date and Hugh Darwen: "Multiple Assignment" in *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

See Chapter 8.

30. 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 Appendix A.

31. C. J. Date and David McGoveran: "Why Relational DBMS Logic Must Not Be Many-Valued," in C. J. Date, *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007). See *http://www.trafford.com/07-0690*.

This paper presents arguments in support of the position that database languages should be based (like the relational model, but unlike SQL) on two-valued logic.

32. Ramez Elmasri and Shamkant Navathe: *Fundamentals of Database Systems* (4th edition). Boston, Mass.: Addison-Wesley (2004).

33. Stéphane Faroult with Peter Robson: *The Art of SQL*. Sebastopol, Calif.: O'Reilly Media Inc. (2006).

A practitioner's guide on how to use SQL to get good performance in currently available products. The following lightly edited list of subtitles from the book's twelve chapters gives some idea of the scope:

1. Designing Databases for Performance
2. Accessing Databases Efficiently
3. Indexing
4. Understanding SQL Statements
5. Understanding Physical Implementation
6. Classic SQL Patterns
7. Dealing with Hierarchic Data
8. Difficult Cases
9. Concurrency
10. Large Data Volumes

11. Response Times
12. Monitoring Performance

The book doesn't deviate much from relational principles in its suggestions and recommendations—in fact, it explicitly advocates adherence to those principles, for the most part. But it also recognizes that today's optimizers are less than perfect; thus, it gives guidance on how to choose the specific SQL formulation for a given problem, out of many logically equivalent formulations, that's likely to perform best (and it explains why). It also describes a few coding tricks that can help performance, such as using MIN to determine that all entries in a yes/no column are *yes* (instead of doing an explicit existence test for *no*). On the question of hints to the optimizer (which many products do support), it includes the following wise words: "The trouble with hints is that they are more imperative than their name suggests, and every hint is a gamble on the future—a bet that circumstances, volumes, database algorithms, hardware and the rest will evolve in such a way that [the] forced execution path will forever remain, if not absolutely the best, at least acceptable ... Remember that you should heavily document anything that forces the hand of the DBMS."

34. 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. **Tutorial D** and the version of the relational algebra I've described in this book both have their roots in this paper.

35. Jim Gray and Andreas Reuter: *Transaction Processing: Concepts and Techniques*. San Mateo, Calif.: Morgan Kaufmann (1993).

The standard text on transaction management.

36. Lex de Haan and Toon Koppelaars: *Applied Mathematics for Database Professionals*. Berkeley, Calif.: Apress (2007).

Among other things, this book includes an extensive set of identities (here called *rewrite rules*) that can be used as in Chapter 11 to help with the formulation of complex SQL expressions. It also shows how to implement integrity constraints by means of procedural code (see Chapter 8). Recommended.

37. Wilfrid Hodges: *Logic*. London, England: Penguin Books (1977).

A gentle introduction to logic for the uninitiated.

38. International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:2003 (2003).

The official SQL standard. Note that it is indeed an international standard, not just (as so many people seem to think) an American or "ANSI" standard.

39. 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 ones available to cover 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.

40. Fabian Pascal: *Practical Issues in Database Management: A Reference for the Thinking Practitioner*. Boston, Mass.: Addison-Wesley (2000).

In some respects this book can be seen as a companion to the present book. It 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.

41. Raghu Ramakrishnan and Johannes Gehrke: *Database Management Systems* (3rd edition). New York, N.Y.: McGraw-Hill (2003).

42. Avi Silberschatz, Henry F. Korth, and S. Sudarshan: *Database System Concepts* (5th edition). New York, N.Y.: McGraw-Hill (2005).

43. Robert R. Stoll: *Sets, Logic, and Axiomatic Theories*. San Francisco, Calif.: W. H. Freeman and Company (1961).

The relational model is 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 [37].

44. Dave Voorhis: Rel. *http://db@builder.sourceforge.net/rel.html*.

Downloadable code for a prototype implementation for a dialect of **Tutorial D**.

45. Moshé M. Zloof: "Query-By-Example," Proc. NCC *44*, Anaheim, Calif. (May 1975). Montvale, N.J.: AFIPS Press (1977).

Query-By-Example (QBE) is a nice illustration of the fact that it's entirely possible to produce a very "user friendly" language based on relational calculus instead of relational algebra. Zloof was the original inventor and designer of QBE, and this paper was the first of many by Zloof on the subject.

INDEX

Symbols

| (the Sheffer stroke), 373
↓ (the Peirce arrow), 373
∈ (contained in), 62
≡ (equivalent to), 206
⇒ (implies), 206
⇔ (bi-implies), 206
→ (FD), 97, 168
→→ (MVD), 311
⊂ (properly included in), 63
⊃ (properly includes), 63
⊆ (included in), 63, 110
⊇ (includes), 63
!! (image relation reference),
  see image relation
θ-join, 123

Numbers

0-tuple, 57
1NF, *see* first normal form
2VL, 79
3VL, 79
4NF, *see* fourth normal form
5NF, *see* fifth normal form
6NF, *see* sixth normal form

A

Abbey, Edward, 101
ACID properties, 172
aggregate operator, 141–144
  see also summarize
"alias," 259
ALL vs. DISTINCT, *see* DISTINCT
ALL BUT, 114
ALL or ANY comparisons, 249–253
alternate key, 96
ambiguity, 245–247
AND (aggregate operator), 141, 354
antecedent, 207
argument, 210, 320
  vs. parameter, 40, 320
Aristotle, 271
arity, 17
assertion, 166
  see also constraint
assignment, 22, 38
  multiple, *see* multiple assignment
  relational, 10, 92–94
  row, 47
  sole update operator, 23

assignment (*continued*)
  SQL, 43
  tuple, 41
Assignment Principle, The, 51, 94, 279, 336
associativity, 127, 343
atomicity
  data value, 34–36
  semantic, 92, 176
  statement, 176
  syntactic, 176
  transaction, 172
  see also scalar
attribute, 7, 17, 54
  relation valued,
    see relation valued attribute
  vs. column, 3
attribute constraint, *see* constraint
attribute FROM, 56
attribute names, reliance on, 127–129
attribute value, 54
axiom, 104

B

bag, 36
  union, 277
base relvar, 20, 89–102
BCNF, *see* Boyce/Codd normal form
bi-implication, 206
  SQL, 43
bill of materials, 106, 338
body, 17, 60, 61–62
BOOLEAN, 28, 275
  vs. truth, 178
bound variable, 213–215
Boyce/Codd normal form, 290
Breazu-Tannen, Val, 128
Buneman, O. Peter, 128
Bush, George W., 211
business rule, 161, 230, 285

C

candidate key, 7
  definition, 94
cardinality, 17, 60
Carroll, Lewis, 64, 154
cartesian product, *see* product
CASCADE, 99
  SQL, 338
Celko, Joe, 272
Chaudhuri, Surajit, 389
CHECK option, 196–197
Closed World Assumption, The, 102, 106, 279, 339
closure, 10, 107, 110–113, 129, 191
  FDs, 380
COALESCE, 83
Codd, E. F., *passim*
coercion, 32
  SQL, 43–45, 62
collation, 45–47
column naming discipline, 45, 66–68, 99, 112
column ordinal position,
  see ordinal position
commalist, 7
commutativity, 126–127, 209, 344
connection trap, 299
connective, 79, 87, 206–209, 332
  3VL, 332
  *n*VL, 332–333
  see also truth tables
consequent, 207
consistency, 172
  logical system, 179, 229
constant, 189
constraint
  attribute, 165
  base table, 170, 220
  checking, 92, 171–177
  column, 171
  database, *see* database constraint
  foreign key, *see* foreign key
  key, *see* key
  multi-relvar, 169
  referential, 97
  relvar, 169
  single-relvar, 169
  total (database), 178
  total (relvar), 178
  transition, 180, 365
  tuple, 167
  type, *see* type constraint
  view, 192–195
  vs. predicate, 177–179
  vs. proposition, 179
contradiction, 83
  3VL, 333
contrapositive, 378
  law, 246
correctness, *see* consistency
correlated subquery, *see* subquery
correlation name, 109, 258
  see also range variable
CORRESPONDING, 119
CREATE DOMAIN, 319
CREATE TABLE AS, 318, 371
CREATE TYPE, 319
cursor, 91
CWA, *see Closed World Assumption, The*

D

D_UNION, *see* disjoint union
da Vinci, Leonardo, 5
Darwen, Hugh, *passim*
data independence
  logical, 128, 199–200, 370–371
  physical, 15, 314
data model, 13, 16
database vs. DBMS, 29
database constraint, 166–170
  SQL, 170–171
database design, 285–312
  physical, 306–308
database statistics, 124
Date, C.J., *passim*
DCO, *see* domain check override
de Haan, Lex, xix, 171, 395
De Morgan's laws, 235–236
declarative, 25
DEE, *see* TABLE_DEE
default values, 85
degree
  key, 94
  relation, 17
  tuple, 54
DELETE expansion, 93
denormalization, 302–304
dependency preservation, 301
dereferencing, 48
derived relvar, 20
designator, 179, 373–374
difference, 12, 120
  definition, 120
direct image, 307
disjoint union, 93, 119–120
  n-adic, 120
DISTINCT, 77–79, 119–120
distributive laws, 235
distributivity, 126
divide, 139–141, 340–341
  definition, 140
  flaw, 243–244
DIVIDEBY, *see* divide
domain, 6
  SQL, 43
  vs. type, 314
  see also type
domain check override, 30
dot qualification, 109, 215, 256–257
double negation law, 235
double underlining, 8
DUM, *see* TABLE_DUM
duplicates, 17, 57, 71–79
  and nulls, 334
  and projection, 114
  and restriction, 330
  and union, 119, 330
  generated by SQL, 77
durability, 172

E

E/R modeling, *see* entity/relationship modeling
Elmasri, Ramez, 394
empty key, 339
empty range, 222–223
empty relation, 61
empty set, 57
  SQL, 263
empty tuple, 57
empty type, 51, 324
entity integrity, 8, 9, 188
entity/relationship modeling, 286, 312
"equal but distinguishable," 46
equality, 33–34, 38
  SQL, 43
equijoin, 123, 340
equivalence, 206
  law of transformation, 234
EVERY, 354
EXCEPT, *see* difference
existential quantification, 211
  3VL, 216
  iterated OR,223–224
  SQL, 216, 227
EXISTS, *see* existential quantification
explicit table, 256
expression transformation, 72, 124–127, 233–254
  laws, 234–236
expression vs. statement, 29, 325
extend, 134–136
  definition, 136
extension (predicate), 102, 339

F

Fagin, Ron, 292
Fagin's Theorem, 311
  proof, 385
Faroult, Stpéhane, xvi, 394
FD, *see* functional dependence
FD preservation, *see* dependency preservation
field, 47
fifth normal form, 295
  significance, 295–300
first normal form, 18, 34, 322
"flat relation," 62–63
FORALL, *see* universal quantification
foreign key, 8, 97–100
  definition, 97
fourth normal form, 299, 300
free variable, 213–215
  see also parameter

functional dependence, 97, 151, 288, 391
  definition, 97
  implied by superkey, 97
  trivial, 288

G

Gehrke, Johannes, 396
generated type, 41
Gennick, Jonathan, 77
Golden Rule, The, 178, 279
Gray, Jim, 171, 395
group, 153, 354, 355
GROUP BY, 247, 253–254, 379
  redundant, 149

H

Hall, Patrick, 395
HAVING, 247, 253–254, 379
  redundant, 150
heading
  relation, 17, 60
  tuple, 54
Heath, Ian, 291
Heath's Theorem, 291, 294, 310
  proof, 381–382
Hitchcock, Peter, 395
Hodges, Wilfrid, 395

I

idempotence, 131, 346
identity
  law of transformation, 234
  projection, 114
  restriction, 113
  value, 116–117
identity (equality), *see* equality
image relation, 136–139, 145–146, 151–152
  definition, 136
implementation, 13
implementation defined, 255
implementation dependent, 255–256
implication, 206–209
  law, 234
  SQL (simulating), 236
inclusion, 63, 110
information equivalence, 329
Information Principle, The, 276, 279
INSERT expansion, 93
INSERT (SQL), 93–94
instantiation, 101
integrity constraint, *see* constraint
intended interpretation, 101
intension, 101, 339
intersect, 11, 120
  definition, 120
  n-adic, 116
  vs. join, 115
irreducibility
  key, *see* key
  relvar, 298
IS_EMPTY, 63
isolation, 172

J

JD, *see* join dependence
join, 12, 115–118
  definition, 115
  greater-than, 118
  n-adic, 116
  SQL, 117–118
join dependence, 294
  implied by superkeys, 294
  trivial, 295
joinability, 115, 131, 346

K

key, 7, 8, 94–96
  definition, 94
  irreducibility, 94–95, 310
  relvars not relations, 96
  uniqueness, 94
Koppelaars, Toon, xix, 171, 395
Korth, Henry F., 396

L

lateral subquery, *see* subquery
Lincoln, Abraham, 230
literal, 40, 320
  vs. constant, 189
logical data independence,
  see data independence
logical difference, passim
  see also Wittgenstein
logical operator, *see* connective
Lorentzos, Nikos A., 281, 282, 297, 394

M

Magritte, René, 19
MATCHING, *see* semijoin
materialization (view), 192
"materialized view," *see* snapshot
McGoveran, David, 394
Melton, Jim, 395
MINUS, *see* difference
missing information without nulls, 85, 308, 391, 393

model vs. implementation, 13
multiple assignment, 178–179
multirelvar constraint, *see* constraint
multiset, *see* bag
multivalued dependence, 300, 311, 385
  implied by superkey, 311
  trivial, 311
MVD, *see* multivalued dependence

N

n-ary relation, 7
natural join, *see* join
  definition, 115
Navathe, Shamkant, 394
NO PAD, 46
nonloss decomposition, 290–292
nonscalar, *see* scalar
normalization, 18, 169, 287–304
NOT MATCHING, *see* semidifference
NOT NULL, 83, 171
null, 9, 56, 79–85
  generated by SQL, 82
  vs. real world, 331–332
  vs. UNKNOWN, 332

O

"object/relational," 36–37
Ohori, Atsushi, 128
Open World Assumption, The, 106, 339
operating via views, *see* view strategy
operator, 3, 39
optimizer, 72, 124–126
OR (aggregate operator), 141
ORDER BY, 18, 155–156
ordering
  not for attributes, 18
  not for tuples, 18
ordinal position (SQL), 68
orthogonality (database design), 304–306
outer join, 84
OWA, *see Open World Assumption, The*

P

PAD SPACE, 46
parameter
  in predicates, 209
  vs. argument, *see* argument
Pascal, Fabian, 396
Peirce arrow, 373
physical data independence,
  see data independence
PJ/NF, *see* projection-join normal form
placeholder, *see* parameter
PNF, *see* prenex normal form
pointer, 48
"positioned update," 91, 335
possible representation, 162
"possibly nondeterministic," 46, 156, 170, 262–263
possrep, 162
predicate, 101, 210
  compound, 210
  instantiation, 210
  relational expression, 122–113
  relvar, *see* relvar predicate
  simple, 210
  view, 190
  vs. boolean expression, 234
predicate calculus, 205
predicate logic, *see* predicate calculus
prenex normal form, 217, 232
primary key, 7, 96
primitive operator
  2VL, 209
  relational algebra, 120
Principle of Identity of Indiscernibles, The, 279, 330
Principle of Interchangeability, The, 173, 187–189, 279
Principle of Orthogonal Design, The, 305–306
Principle of Uniform Representation, The, 277, 279
Principle of Uniformity of Representation, The, 277, 279
product, 11, 116
  definition, 116
  n-adic, 116
  vs. join, 116
project, 11, 114–115
  definition, 114
projection-join normal form, 297
proper inclusion, 63
proper subset, 56
proper superkey, 97
proper superset, 56
proposition, 101, 206
  compound, 207
  simple, 207
proto tuple, 215
public table, 128–129, 203
purple parts, 131, 141, 241–244, 346

Q

QBE, *see* Query-By-Example
quantification law, 235
quantifiers, 211–215
  don't need both, 221
  other kinds, 224
  vs. COUNT, 227
  see also existential quantification; UNIQUE;
    universal quantification
query, 262
Query-By-Example, 396
query expression, *see* table expression

query rewrite, 72, 233
quota query, 155, 158

R

Ramakrishnan, Raghu, 396
range variable, 215, 217–218, 377
    SQL, 216, 257–259
read-only, 108
redundancy, 287, 296, 300, 309
REF type, 48
reference value, 48
referenced relvar, 97
referencing, 48
    vs. foreign key, 49
referencing relvar, 97
referential action, 99
    SQL, 338
referential constraint, 97
referential integrity, 8, 9, 314
    metaconstraint, 180
    see also foreign key
relation, 7, 23
    definition, 60
    origin of term, 60
    vs. table, 3, 25, 69, 315, 326–327
    vs. type, 102–105
relation comparison, 63–64
relation constant, 189
relation equality, 63
relation selector, 60
relation type inference, 111
    SQL, 112
relation value, *see* relation
relation valued attribute, 36, 152–154
    contraindicated, 293, 354
relation variable, *see* relvar
relational algebra, 10, 107–159
relational assignment, *see* assignment
    Tutorial D grammar, 355
relational calculus, 109, 205, 215–230
relational completeness, 228, 278
    SQL, 377–378
relational model, 269–283
    definition, 274–278
    objectives, 278–279
relcon, 189
relvar, 22, 23, 89–90
relvar constraint, *see* constraint
relvar predicate, 100–102
relvar reference, 108, 323
rename, 111–112
    definition, 111
    not primitive, 352
repeating group, 35, 52, 325
restrict, 10, 113–114
    definition, 113
restriction condition, 113
Reuter, Andreas, 171, 395
rewrite rule, 234
Robson, Peter, 394
row assignment, 58
row comparison, 58–59
row ID, 4, 188
row subquery, *see* subquery
row type, 57
row type constructor, 57
row value constructor, 57
rule of inference, 104
Russell, Bertrand, 211
RVA, *see* relation valued attribute

S

scalar, 40–42
    SQL, 42–43
scalar subquery, *see* subquery
"SELECT *," 256
selector, 31, 163–164, 320
    SQL, 31, '166
    vs. literal, 40
self-referencing relvar, 106
semantic optimization, 173–174
semidifference, 134
    definition, 134
semijoin, 133–134
    definition, 133
"set function," 77–78, 148
    see also aggregate operator; summarize
Shakespeare, William, 101
Sheffer stroke, 373
Silberschatz, Avi, 396
Simon, Alan R., 395
single-relvar constraint, *see* constraint
sixth normal form, 297–298
snapshot, 200–201, 371
source, 22
SQL legacy, 283
statement
    logic, 102
    programming language, 29
    vs. expression, *see* expression vs. statement
Stoll, Robert R., 396
Stonebraker, Mike, 272
strong typing, 33
    SQL, 43
subkey, 337
subquery, 45
    correlated, 240–241, 260
    lateral, 261–262
    row, 259–260
    scalar, 260
    table, 259
substitution (view), 191

Sudarshan, S., 396
SUMD, 357
summarize, 146–152
  definition, 146
superkey, 96, 167, 289, 339
surrogate key, 310, 383

T

TABLE_DEE, 64–65, 117, 189
  vs. TRUE, 329
TABLE_DUM, 64–65
  vs. FALSE, 329
table expression, 4
  SQL grammar, 264–266
  evaluation, 123
table reference, 118
table subquery, *see* subquery
"TABLE *T*," *see* explicit table
table value constructor, 65
table vs. relation, *see* relation
"tables and views," 21, 186
target, 22
tautology, 87
  3VL, 333
temporal data, 281, 297
THE_ operator, 32, 163–164, 320
  SQL, 32, 166
theorem, 104
three-valued logic, 79
TIMES, *see* product
Todd, Stephen, 395
total database constraint, *see* constraint
total relvar constraint, *see* constraint
transaction, 171–172
transition constraint, *see* constraint
TransRelational™ Model, The, 280, 307, 393
trigger, 92, 100
trivial FD, *see* functional dependence
trivial JD, *see* join dependence
trivial MVD, *see* multivalued dependence
truth functional completeness, 87
truth tables
  2VL, 332
  3VL, 79, 87
tuple, 7, 53–57
  definition, 54
  vs. row, 3
tuple equality, 57, 96
TUPLE FROM, 62
tuple selector, 55
tuple type, 55
tuple value, *see* tuple
two-valued logic, 79
type, 6, 37–42
type constraint, 162–165
  checking, 165
  SQL, 165–166
type constructor
  not for tables, 47–48
  ROW, 47
type generator, 41
  RELATION, 41
  TUPLE, 41
"typed table," 48

U

U_ operators, 282
ungroup, 153, 354, 355
union, 11, 118–120
  definition, 118
  n-adic, 120
unique index, 14
UNIQUE (quantifier), 224–225
  SQL, *see* UNIQUE (SQL)
UNIQUE (SQL), 167–168, 225–228, 248–249, 334
uniqueness (key), *see* key
universal quantification, 211
  iterated AND, 223–224
  not in SQL, 217, 221–222, 227
  SQL (simulating), 236–238
UNKNOWN, 79, 81
  vs. null, 332
update anomalies, 301–302
update vs. UPDATE, 10
UPDATE expansion, 154–155
UPDATE (read-only), 154
updating is set level, 90–92

V

value, 23
  vs. variable, 23
VALUES, 65
variable, 23–24
  logic, *see* bound variable; free variable
view, 20
  definition, 186
  predicate, 190
  purpose, 199–200
  retrieval, 190–192
  update, *see* view updating
  vs. snapshot, *see* snapshot
view constraint, *see* constraint
view defining expression, 186
view strategy, 68, 129
view updating, 195–196
  SQL, 197–198
virtual relvar, *see* view
Voorhis, Dave, 396

W

Weikum, Gerhard, 389
"what if" queries, 155
WITH, 121–122
Wittgenstein, Ludwig J. J., 19, 144, 308

X

XML, 24, 36, 37, 38, 43, 312, 319, 356
XOR (aggregate operator), 141

Z

Zloof, Moshé M., 396

ABOUT THE AUTHOR

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold well over three quarters of a million copies at the time of writing and is used by several hundred colleges and universities worldwide. He is also the author of many other books on database management, including most recently:

- From Morgan Kaufmann: *Temporal Data and the Relational Model* (coauthored with Hugh Darwen and Nikos A. Lorentzos, 2003)
- From O'Reilly: *Database in Depth: Relational Theory for Practitioners* (2005)
- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, coauthored with Hugh Darwen, 2006)
- From O'Reilly: *The Relational Database Dictionary* (2006)
- From Apress: *Date on Database: Writings 2000-2006* (2006)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007)
- From Apress: *The Relational Database Dictionary, Extended Edition* (2008)

Another book, *Go Faster! The TransRelational™ Approach to DBMS Implementation*, is due for publication in the near future.

Mr. Date was inducted into the Computing Industry Hall of Fame in 2004. He enjoys a reputation that is second to none for his ability to explain complex technical subjects in a clear and understandable fashion.

COLOPHON

The cover image is from www.jupiterimages.com. The cover fonts are Akzidenz Grotesk and Orator. The text font is Adobe's Meridien; the heading font is ITC Bailey.