O'REILLY®

/theory/in/practice

SQL and Relational Theory

How to Write Accurate SQL Code

C. J. Date

3rd Edition

O'REILLY®

/ theory / in / practice

SQL and Relational Theory

SQL is full of difficulties and traps for the unwary. You can avoid them if you understand relational theory, but only if you know how to put that theory into practice. In this book, Chris Date explains relational theory in depth, and demonstrates through numerous examples and exercises how you can apply it to your use of SQL.

This third edition has been revised, extended, and improved throughout. Topics whose treatment has been expanded include data types and domains, table comparisons, image relations, aggregate operators and summarization, view updating, and subqueries. A special feature of this edition is a new appendix on NoSQL and relational theory.

- Could you write an SQL query to find employees who have worked at least once in every programming department in the company? And be sure it's correct?
- Why is proper column naming so important?
- Nulls in the database cause wrong answers. Why? What you can do about it?
- How can image relations help you formulate complex SQL queries?
- SQL supports "quantified comparisons," but they're better avoided. Why? And how?

Database theory and practice have evolved considerably since Codd first defined the relational model, back in 1969. This book draws on decades of experience to present the most up to date treatment of the material available anywhere. Anyone with a modest to advanced background in SQL can benefit from the insights it contains. The book is product independent.

Chris Date has a stature that is unique in the database industry. He is best known for his textbook *An Introduction to Database Systems* (Addison-Wesley). He enjoys a reputation that is second to none for his ability to explain complex technical issues in a clear and understandable fashion.

US $39.99 CAN $45.99
ISBN: 978-1-491-94117-1
9 781491 941171 53999

Twitter: @oreillymedia
facebook.com/oreilly
oreilly.com

Databases

SQL and Relational Theory

How to Write Accurate SQL Code

THIRD EDITION

C. J. Date

SQL and Relational Theory: How to Write Accurate SQL Code (3rd edition)
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 (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Printing History:
January 2009: First Edition.
December 2011: Second Edition.
October 2015: Third Edition.

Revision History:
2015-09-30 First release.
See *http://www.oreilly.com/catalog/errata.csp?isbn=0636920046158* for release details.

ISBN: 978-1-491-94117-1
[LSI]

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 (1877-1947)

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

—Anon.

These are my principles.
If you don't like them, I have others.

—Groucho Marx (1890-1977)

There is no royal road to geometry.

—Euclid (c. 365-275 BCE), attrib.

——— ♦♦♦♦♦ ———

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

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 some 900,000 copies at the time of writing and is used by several hundred colleges and universities worldwide. He is also the author of numerous other books on database management, including most recently:

- From Ventus: *Go Faster! The TransRelational™ Approach to DBMS Implementation* (2002, 2011)
- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, with Hugh Darwen, 2007)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007) and *Database Explorations: Essays on The Third Manifesto and Related Topics* (with Hugh Darwen, 2010)
- From Apress: *Date on Database: Writings 2000-2006* (2007)
- From O'Reilly: *Database Design and Relational Theory: Normal Forms and All That Jazz* (2012); *View Updating and Relational Theory: Solving the View Update Problem* (2013); *Relational Theory for Computer Professionals: What Relational Databases Are Really All About* (2013); and *The **New** Relational Database Dictionary* (2015, to appear)

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.

Contents

Preface to the First Edition

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 code that 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, the language 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, more or less, 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, over 45 years later, 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 in order to comply with 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 adherence 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 Media Inc., 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?
>
> 1. What exactly is first normal form?
> 2. What's the connection between relations and predicates?
> 3. What's semantic optimization?
> 4. What's an image relation?
> 5. Why is semidifference important?
> 6. Why doesn't deferred integrity checking make sense?
> 7. What's a relation variable?
> 8. What's prenex normal form?
> 9. Can a relation have an attribute whose values are relations?
> 10. Is SQL relationally complete?
> 11. Why is *The Information Principle* important?
> 12. How does XML fit with the relational model?
>
> This book provides answers to these and many related questions. Overall, it's meant to help database practitioners understand relational theory in depth and make good use of that understanding in their professional day-to-day activities.

As the final sentence in the foregoing extract indicates, it was my hope that readers of that earlier book would be able to apply its ideas for themselves, without further assistance from me. 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 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 book).

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 the present 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 (also to admit to former errors), 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. Indeed, 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 that's totally founded on one great theoretical idea. But I suppose the cynic's position would be "Yes, but what has theory done for me lately?" In other words, those of us who do think theory is important must continually be 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 explicitly 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 F.[1]

Finally, I'd like to mention that I have some live seminars available based on the material in this book. See *www.justsql.co.uk/chris_date/chris_date.htm* or *www.thethirdmanifesto.com* for further details. An online version of one of those seminars is available too, at *http://oreilly.com/catalog/0636920010005/*.

[1] In response to reader requests, in the third edition I've moved the answers that are specific to a given chapter to the end of the chapter in question and deleted the old Appendix F.

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 G of the present book), and very good it was, too. But what struck me most about that class was seeing at first hand the kinds of difficulties the attendees had in applying relational and logical principles to their use of SQL. Now, I do assume those attendees had some knowledge of those principles—they were database practitioners, after all—but it seemed to me they really needed some guidance in the application of those principles 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 and Andy Oram—for their encouragement, contributions, and support throughout the production of this book.

C. J. Date
Healdsburg, California
2008 (minor revisions 2011, 2015)

Preface to the Second Edition

The second edition differs from its predecessor in a number of ways. The overall objective remains the same, of course—using SQL relationally is still the emphasis—but the text has been revised throughout to reflect, among other things, experience gained from teaching live seminars based on the first edition.

One significant change is a deletion: The appendix on design theory has gone. There are two reasons for this change. First, design theory as such never really did have all that much to do with the book's main message, anyway; second, the appendix was getting so extensive that it threatened to overwhelm the rest of the text. (It was already longer than any chapter or any other appendix in the book. In fact, I've since expanded that appendix into a separate book in its own right. That book—*Database Design and Relational Theory: Normal Forms and All That Jazz*—

is due to be published soon by O'Reilly.[2] It can be seen as a companion, or perhaps a sequel, to the present book.)

On the positive side, a lot of new material has been added (including, importantly, a discussion of how to deal with missing information without using nulls); examples, exercises, and answers have been expanded and improved in various respects; and the treatment of SQL has been upgraded to cover recent changes to the SQL standard. A variety of corrections and numerous cosmetic improvements have also been made.[3] (In particular, the **Tutorial D** examples—**Tutorial D** being the language I use to illustrate relational concepts[4]—have been upgraded to reflect several recent improvements to that language. See Appendix D.) The net effect is to make the text rather more comprehensive—but, sadly, some 25 percent bigger—than its predecessor.

And talking of the text, this is clearly the place to say something about my use of footnotes! Frankly, I'm rather embarrassed at how many footnotes there are; I'm well aware how annoying they can be. Indeed, they can seriously impede readability. But any text dealing with SQL is more or less forced into a heavy use of footnotes, at least if it wants to be tutorial in nature and yet reasonably comprehensive at the same time. The reason is that SQL involves so many inconsistencies, exceptions, and special cases that treating everything "in line"—i.e., at the same level of description—makes it very difficult to see the forest for the trees. (Indeed, this is one reason why the SQL standard itself is so difficult to understand.) Thus, there are numerous places in the book where the major idea is described "in line" in the main body of the text, and exceptions and the like (which must at least be mentioned, for reasons of accuracy and completeness) are relegated to a footnote. It might be best simply to ignore all footnotes on a first reading.

C. J. Date
Healdsburg, California
2011 (minor revisions 2015)

---

[2] It was published in 2012.

[3] In this connection, I'd like to acknowledge the contribution of a reader of the first edition, Thomas Uhren, who found an embarrassingly large number of errors. I'll try harder in future. I promise.

[4] Note that the name **Tutorial D** is always set in boldface.

Preface to the Third Edition

Much of the justification for the second edition applies to this third edition as well, but more so (as it were). Again the book has benefited from experience gained in teaching the material in various live classes. There are revisions of substance to many of the chapters, based on any or all of the following: changes to the SQL standard, improvements to **Tutorial D**, improvements in my own understanding of relational theory, better ways of presenting some of the material, and (in a few cases) simple corrections of errors. Also, in response to requests from readers:

- I've increased the font size. (My own eyesight isn't what it used to be, either.)
- I've replaced all "smart quotes" in coding examples by "dumb quotes," in order to help with cut and paste operations.
- I've moved exercise answers to the chapter in which they belong (and set them in a different typeface), instead of putting them all in an appendix of their own at the back of the book.

I've also added an appendix on the new "NoSQL" systems, in which (as the title of that appendix indicates) I've tried to show what the relationship is between those systems and relational theory. Finally, textual and formatting improvements have been made throughout.

C. J. Date
Healdsburg, California
2015

Chapter 1

Setting the Scene

My soul, sit thou a patient looker-on;
Judge not the play before the play is done;
Her plot hath many changes; every day
Speaks a new scene; the last act crowns the play.

—Francis Quarles:
Emblems (1635)

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, I'll have comparatively little to say in this chapter about SQL as such; instead, 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, as it were: in other words, to lay some groundwork on which the rest of the book can build. But even though I do 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?[1]

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 large 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

[1] 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 ... Equijoins 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." *Your comment here.*

service to the idea of teaching the relational model—but that teaching seems mostly 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
- The importance of attribute names
- The crucial role of integrity constraints
- *The Information Principle* and its significance

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 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* (usually pronounced to rhyme with *couple*), and *attribute*. SQL doesn't use these terms, of course—it uses the more "user friendly" terms *table*, *row*, and *column* instead. And I'm generally sympathetic to the idea of 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 so myself—I would argue that it's acceptable only if all parties involved understand that those 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 formal terms. In this book, therefore, I'll tend to use those 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 say too that it does its best to complicate another. I refer to its use of the terms *operator*, *function*, *procedure*, *routine*, and *method*, all of which denote essentially the same thing (with, perhaps, very minor differences). In this book I'll use the term *operator* throughout; thus, for example, I'll refer to "=" (equality comparison), ":=" (assignment), "+" (addition), DISTINCT, JOIN, SUM, GROUP BY (etc., etc) all as operators specifically.

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 very few places where the context demands otherwise.[2] However:

- The standard's use of terminology is sometimes not very apt. In such situations, I generally prefer to use terminology of my own. For example, I use the term *table expression* in place of the standard term *query expression*, for the following reasons among others: First, the value such expressions denote is indeed a table and not a query; second, queries aren't the

[2] The standard has been through several versions, or editions, over the years. The version current at the time of writing is SQL:2011 (a formal reference for which can be found in Appendix G); the previous version was SQL:2008, the one before that was SQL:2003, the one before that was SQL:1999, and the one before that was SQL:1992. Most of the SQL features discussed in this book were present in SQL:1992, and often in even earlier versions.

only context in which such expressions are used anyway. (As a matter of fact the standard does use the term *table expression*, but this term too it uses quite inappropriately; to be specific, it uses it to refer to what comes after the SELECT clause in a SELECT – FROM – WHERE – GROUP BY – HAVING expression.)

- Following on from the previous point, I should add that not all table expressions—in either my sense or the standard's—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).[3] *Please note that these remarks apply to many of the individual discussions in the body of the book; it would be very tedious to keep on repeating them, however, 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 if they don't have much to do with relational processing as such. Examples here include the so called analytic or window (OLAP) functions; dynamic SQL; temporary tables; and details of user defined types.

- For reasons that aren't important here, 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 at all. 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, what I'm advocating is supposed to be a *relational* approach to SQL). For example, row IDs in particular are likely to violate either *The Principle of Interchangeability* (see Chapter 9) or *The Information Principle* (see Appendix A) or both; 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

[3] These particular limitations were added in SQL:2003; they didn't apply to SQL:1992, which is where explicit JOIN invocations were first introduced, nor to SQL:1999.

isn't product specific; instead, it's concerned with principles. What do I mean by principles? Well, here's a definition (from *Chambers Twentieth Century Dictionary*):

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

The point about principles is: 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 fashion or fads. But I do realize you sometimes have to make compromises and tradeoffs 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 make such a compromise.
- You should know what the theoretically correct situation is, and you should have strong reasons for departing from it.
- You should document those reasons, too, so that if they cease to be valid 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 part 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 other branches of mathematics, 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 prior to that time was all too deficient in any such qualities. 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 G.

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 in this subsection (and in the next two) 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 usual to depict relations on paper as tables (see Fig. 1.1 for a self-explanatory example). Relations are defined over *types* (also known as *domains*); a type is basically a conceptual pool of values from which actual attributes in actual relations take their actual values. With reference to Fig. 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 ("departments") relation and the attribute called DNO in the EMP ("employees") relation would both contain values from that conceptual pool. (By the way, it isn't necessary, though it's often a good idea, for attributes to have the same name as the corresponding type, and frequently they won't. We'll see plenty of counterexamples later.)

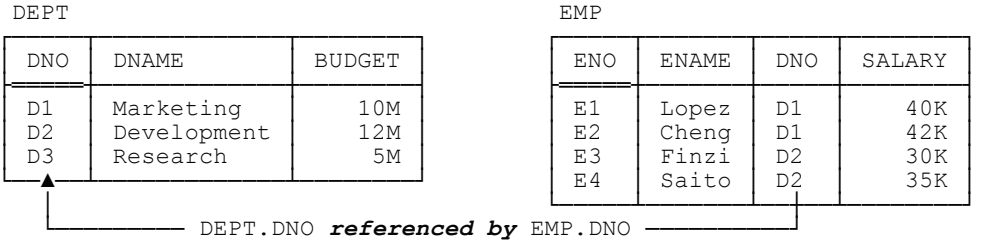


Fig. 1.1: The departments-and-employees database—sample values

As I've said, tables like those in Fig. 1.1 represent *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 represent *attributes* of the relation and the rows represent *tuples*. The value *n* can be any nonnegative integer. A 1-ary relation is said to be *unary*; a 2-ary relation, *binary*; a 3-ary relation, *ternary*; and so on.

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.[4] 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 a single attribute—such that every tuple in the relation has a unique value for the combination in question. In Fig. 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 mentioned the useful 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 (blank spaces appearing immediately before or after any comma are ignored). For example, if *A*, *B*, and *C* are attribute names, then the following are all attribute name commalists:
>
> ```
> A , B , C
> C , A , B
> ```

[4] 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 elsewhere in this chapter as well. Exercise 1.1 at the end of the chapter addresses this issue.

```
B

A , C
```

So too is the empty sequence of attribute names.

Moreover, when some commalist is enclosed in braces and thereby denotes a set, then (a) blank spaces appearing immediately after the opening brace or immediately before the closing brace are ignored, (b) the order in which the elements appear within the commalist is immaterial (because sets have no ordering to their elements), and (c) if an element appears more than once, it's treated as if it appeared just once (because sets don't contain duplicate elements). *End of aside.*

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 doesn't make any real difference if we decide to call that key primary. But if that relation has two or more candidate 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 department always has both a unique department number and a unique department name—not a very realistic example, perhaps, but good enough for present purposes—so that {DNO} and {DNAME} are both candidate keys for DEPT. Then we might choose {DNO}, say, to be the primary key.

Observe now that I said it's *usual* to choose a primary key. Indeed it is usual—but it's not 100 percent necessary. 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 Fig. 1.1 I'll indicate primary key attributes by double underlining[5]—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 that reason, from this point forward I'll use the term *key*, unqualified, to mean any candidate key, regardless of whether the candidate key in question has additionally been designated as primary. (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 combination, or set, of attributes *FK* in some relation *r2* such that each *FK* value is required to be equal to some value of some key *K* in some relation *r1* (*r1* and *r2* not necessarily distinct).[6] With reference to Fig. 1.1, for example, {DNO} is a foreign key in EMP whose values are required to match values of the 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 the DNO attribute has the value D2, then

[5] See the answers to Exercises 5.27 in Chapter 5 and 7.23 in Chapter 7 for further discussion of this convention.

[6] This definition is deliberately somewhat simplified. A better definition can be found in Chapter 5.

DEPT must also contain a tuple in which the DNO attribute 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* constraints—generic, in the sense that they apply to every possible database, loosely speaking. One has to do with primary keys and the other with foreign keys. Here they are:

1. *The entity integrity rule:* Primary key attributes don't permit nulls.

2. *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 pertinent candidate key (the "target 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 a DNO value of D2, say, 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 target 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's my very strong opinion that *nulls have no place in the relational model*. (Codd thought otherwise, obviously, but I have strong reasons for taking the position I do.) In order to explain the entity integrity rule, therefore, I need to suspend disbelief, as it were, at least for 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 (especially 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 employee E2 in relation EMP—we can't reasonably enter a real value, precisely because we don't know what that real value should be—we *mark* the SALARY component within that tuple as null, as indicated here:

ENO	ENAME	DNO	SALARY
E2	Cheng	D1	

Now, it's important to understand that this tuple contains *nothing at all* in its SALARY component. But it's very hard to draw pictures of nothing at all! In the picture, I've used shading to represent the fact that the SALARY component is empty, but it would really be more accurate not to show that component (or the value portion of that component, at any rate) at all.[7] Be that as it may, I'll use this same convention of representing empty positions by shading elsewhere in this book—but, to repeat, such shading mustn't be understood as representing any kind of value at all. You can think of it (the shading, that is) as constituting the null marker, or flag, if you like.

To get back to the entity integrity rule: In terms of relation EMP, then, that rule says, loosely, that a given tuple might have an unknown name, or an unknown department number, or an unknown salary, but it can't have an unknown employee number. The justification, such as it is, for this state of affairs is that if the employee number were unknown, we wouldn't even know which "entity" (i.e., which employee) we were talking about.

That's all I want to say about nulls for now. Please forget about them until further notice.

Manipulative Features

The manipulative part of the model in turn divides into two parts:

- The *relational algebra*, which is a collection of operators (e.g., difference, or MINUS) that can be applied to relations

- A *relational assignment* operator, which allows the value of some relational algebra 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 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 us to derive "new" relations from "old" ones. Each such operator takes one or more

[7] I'm sorry—I'm trying to suspend disbelief, but I do find it difficult ...The fact is, as we'll see in Chapters 3 and 4, a tuple with "nothing at all" as the "value" of some component simply isn't a tuple in the first place, and that really ought to be the end of the discussion. But let's soldier on.

relations as input and produces another relation as output; for example, difference (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. For example, 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);[8] I'll give more details in Chapter 6, and in Chapter 7 I'll describe a number of additional operators as well. Fig. 1.2 is a pictorial representation of those original operators. *Note:* If you're unfamiliar with these operators and find these brief descriptions a little 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.

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. *Note:* This operator is also known variously as *cartesian product* (sometimes more specifically *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.

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

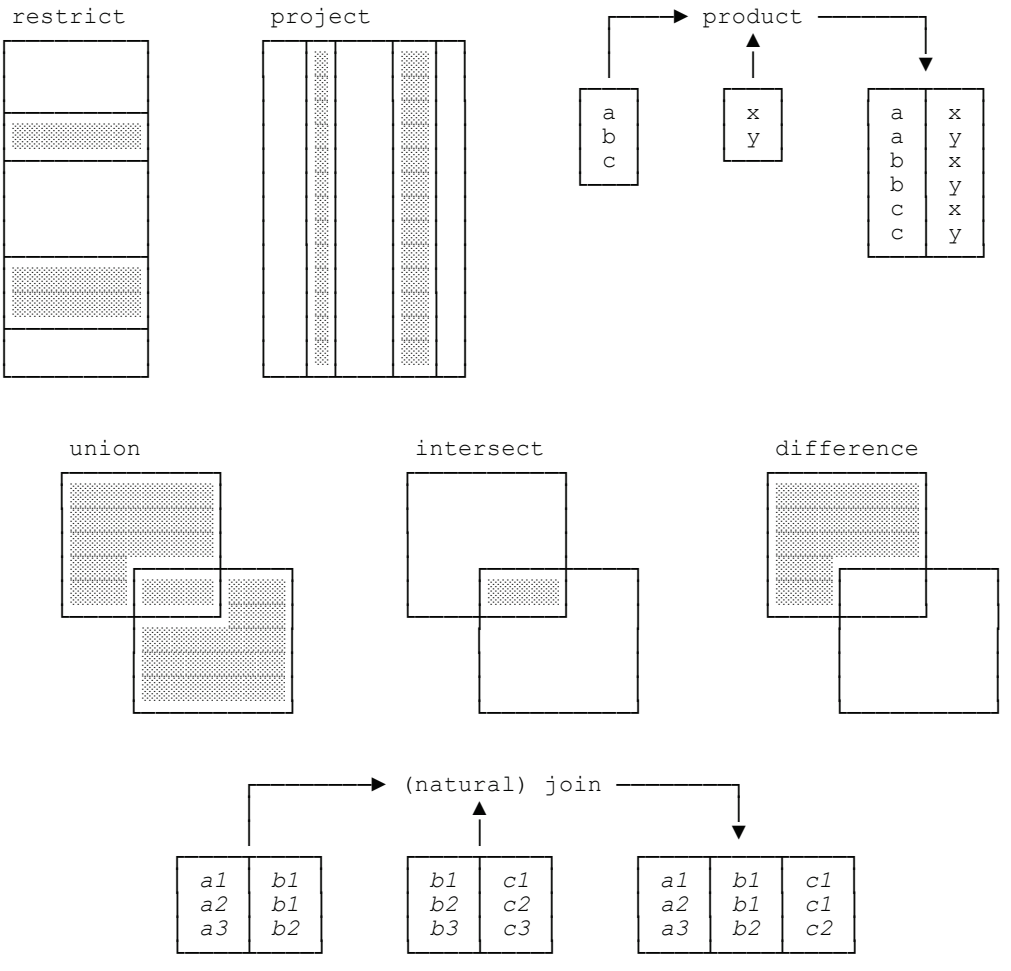


Fig. 1.2: The original relational algebra

Union

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

Intersect

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

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, to distinguish it from various other kinds to be discussed later in this book. 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 might 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

Fig. 1.3 shows a set of sample values for the database 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 books and other publications can help, not hinder, learning.) The semantics are as follows:

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 from the underlining in the figure, I've made {SNO} the primary key); one name (SNAME), not necessarily unique (though the SNAME values in Fig. 1.3 do happen to be unique); one status value (STATUS), representing some kind of ranking or preference level among available suppliers; and one location (CITY).

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

Fig. 1.3: The suppliers-and-parts database–sample values

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, corresponding to the primary keys of S and P, respectively). Notice that Fig. 1.3 shows one supplier, supplier S5, with no shipments at all.

MODEL vs. IMPLEMENTATION

Before going any further, there's an 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 one 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 (first sense) is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

Let me illustrate these definitions in terms of the relational model specifically. First of all, consider the concept *relation* itself. That concept 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 inside the computer, or how individual data values are physically encoded, or what indexes or other physical 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 physical access paths are used, or what 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 particular, they have to know that such keys have the property of uniqueness. Now, key uniqueness is typically enforced in today's systems 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 sense) is what the user has to know.
- The *implementation* is what the user doesn't have to know.

Please understand that I'm not saying here that 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, observe that everything to do with performance is fundamentally an implementation issue, not a model issue. This point is widely misunderstood! For example, we often hear remarks to the effect that "joins are slow." But such remarks simply 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 has a faster or slower implementation of some specific join, on some specific data, than some other specific product does—but that's about 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 you won't use it badly. If you write a relational algebra expression such as S JOIN SP, you're within your rights to expect the system to implement it efficiently. But if you insist on, in effect, hand coding that join yourself, perhaps like this (pseudocode)—

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

—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.[9]

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 for a moment on this matter of performance. 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.

[9] More than one reviewer observed that this sentence didn't seem to 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 for many years to mean a simple record level I/O facility, of one kind or another.

> However, it goes without saying that this objective hasn't yet been fully achieved, and so (as I've already said, more or less) 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. End of aside.*

Back to the distinction between model and implementation, and points arising from that distinction. 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 seem to be 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. Now, the reason we might want to change those storage and access details 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 designs also, I might add).

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, in principle, relational systems are capable of. I'll come back to this issue in the next section.

Now I 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 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 remainder 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, very specifically, an attribute-name : type-name pair, and where no two attributes in the same heading have the same attribute name), and the body is a set of tuples that conform to that heading. In the case of the suppliers relation in Fig. 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 Fig. 1.3 and hence to pretend the heading is just a set of attribute names. For example, the STATUS attribute does have a type—INTEGER, let's say—but I didn't show it in Fig. 1.3. But you should never forget it's there!

Next, the number of attributes in the heading is the *degree* (sometimes the *arity*), both of that heading as such and of any relation that has that heading. And the number of tuples in the body is the *cardinality*, both of the body itself and of the relation that contains it. For example, relation S in Fig. 1.3 has degree 4 and cardinality 5; likewise, relation P in that figure has degree 5 and cardinality 6, and relation SP in that figure has degree 3 and cardinality 12. *Note:* The

term *degree* is used in connection with tuples also.[10] 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 throughout this book I *always* use the term "relation" to mean a relation—without duplicate tuples, by definition—and not an SQL table. Please understand too that relational operations always produce a result without duplicate tuples, again by definition. For example, projecting the suppliers relation of Fig. 1.3 on attribute CITY produces the result shown on the left below 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 that 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 as well). 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 Fig. 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 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 do 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 Fig. 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

[10] It's also used in connection with keys (see Chapter 5).

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, these two pictures 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 differences between these two queries, and between these two tables, are hardly very significant; in fact, however, they have some serious consequences, some of which I'll be touching on in later chapters—e.g., see the discussion of SQL's explicit JOIN operator in Chapter 6.)

Finally, relations are always *normalized* (equivalently, they're in *first normal form,* 1NF).[11] 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.)

Before I finish with 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, on the one hand, and a picture of a relation as shown in, for example, Figs. 1.1 and 1.3, on the other. To say it one more time, the constructs in Figs. 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 somewhat of 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 usually 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, at least a little bit—then we do have to recognize that the two concepts are not identical.

In passing, 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 (see *en.wikipedia.org/wiki/The_Treachery_of_Images*) that beautifully illustrates the point I'm trying to make here. The painting is of an ordinary tobacco pipe, but underneath Magritte has written *Ceci n'est pas une pipe* ... the point being, of course, that *obviously* the painting isn't a pipe—instead, 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

[11] "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 discipline of database design.

on paper. Indeed, it's that simple representation on paper 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 the simple representation in question does suggest some things that aren't true (e.g., that there's a top to bottom ordering to the tuples).

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. For now, let me just point out that we've encountered quite a few important examples of such 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 we'll be meeting many more in the pages ahead.

Some Crucial Points

At this juncture I'd like to mention some crucial points that I'll be expanding on in later chapters (especially in Chapter 3). The points in question are these:

- *Every subset of a tuple is a tuple.* For example, consider the tuple for supplier S1 in Fig. 1.3. That tuple has four components, corresponding to the four attributes SNO, SNAME, STATUS, and CITY. And if we remove (say) the SNAME component, what's left is indeed still a tuple: viz., a tuple with three components (a tuple of degree three).
- *Every subset of a heading is a heading.* For example, consider the heading of the suppliers relation in Fig. 1.3. That heading has four attributes: SNO, SNAME, STATUS, and CITY. And if we remove (say) the SNAME and STATUS attributes, what's left is still a heading, a heading of degree two.
- *Every subset of a body is a body.* For example, consider the body of the suppliers relation in Fig. 1.3. That body has five tuples, corresponding to the five suppliers S1, S2, S3, S4,

and S5. And if we remove (say) the S1 and S3 tuples, what's left is still a body, a body of cardinality three.

Aside: Perhaps I should also state here for the record that throughout this book—in accordance with standard scientific practice—I take expressions of the form "*B* is a subset of *A*" to include the possibility that *A* and *B* might be equal. Thus, for example, every tuple is a subset of itself (and so is every heading, and so is every body). When I want to exclude such possibilities, I'll talk explicitly in terms of *proper* subsets. For example, our usual tuple for supplier S1 (see Fig. 1.3) is certainly a subset of itself, but it isn't a proper subset of itself. What's more, the foregoing remarks apply equally to supersets, mutatis mutandis; for example, that tuple for supplier S1 is a superset of itself, but not a proper superset of itself.[12] *End of aside*.

I'd also like to say something about the crucial notion of *equality*—especially as that notion applies to tuples and relations specifically. In general, two values are equal if and only if they're the very same value. For example, the integer 3 is equal to the integer 3, and not to anything else—in particular, not to any other integer. In exactly the same way, two tuples are equal if and only if they're the very same tuple. With reference to Fig. 1.3, for example, the tuple for supplier S1 is equal to the tuple for supplier S1, and not to anything else—in particular, not to any other tuple. In other words, two tuples are equal if and only if (a) they involve exactly the same attributes and (b) corresponding attribute values are equal in turn.

Moreover—this might seem obvious, but it needs to be said—two tuples are *duplicates* of each other if and only if they're equal: in other words, if and only if they're the very same tuple.

Turning now to relations: In exactly the same way, two relations are equal if and only if they're the very same relation. With reference to Fig. 1.3, for example, the suppliers relation is equal to the suppliers relation and not to anything else—in particular, not to any other relation. In other words, two relations are equal if and only if, in turn, their headings are equal and their bodies are equal.

As I've already said, I'll be returning to these matters in Chapter 3. Here let me just add that the notion of tuple equality in particular is absolutely fundamental—just about everything in the relational model is crucially dependent on it, as we'll see.

BASE vs. DERIVED RELATIONS

As I explained earlier, the operators of the relational algebra allow us to start with some given relations, such as the ones depicted in Fig. 1.3, and obtain further relations from those given ones

[12] What I've described in this paragraph is indeed in accordance with standard scientific practice; however, you might have encountered a different convention in less formal contexts. To be specific, some people use "*B* is a subset of *A*" to mean what I mean when I say *B* is a *proper* subset of *A*, and use "*B* is a subset of *or equal to A*" to mean what I mean when I say *B* is a subset of *A*. Similarly for supersets, of course, mutatis mutandis.

(thereby allowing us to do queries, for example). The given relations are referred to as *base* relations, the others are *derived* relations. In order to get started, therefore, we have to have some way of establishing, or defining, those base relations in the first place. In SQL, this task is performed by means of the CREATE TABLE statement (the SQL counterpart to a base relation being, of course, a base table, which is what CREATE TABLE creates). 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 just as if they were base relations,[13] 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. But I must emphasize that thinking of views as 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 the difference between base relations and views described like this (*warning! untruths coming up!*):

- 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 quite deliberately has nothing to say as to what's physically stored!—in fact, it has nothing to say about physical storage matters at all. In particular, it categorically does not say that base relations are physically stored and views aren't. The only requirement is that there must be some mapping between whatever *is* physically stored and the 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 from whatever's physically

[13] 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 SQL products are concerned; nevertheless, I still claim it's true in principle. See the section "Update Operations" in Chapter 9 for further discussion.

stored, then everything else can be, too. 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.[14] In other words: Base relations are no more (and no less!) "physical" than views are, so far as the relational model is concerned.

Of course, the fact that the relational model says nothing about physical storage is (as I said) quite deliberate. The idea was to give implementers the freedom to implement the model in whatever way they chose—in particular, in whatever way seemed likely to yield good performance—without compromising on physical data independence. The sad fact is, however, most SQL product vendors seem not to have understood this point (or not to have risen to the challenge, at any rate); instead, they map base tables fairly directly to physical storage,[15] 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—talks in terms of "tables and views." Clearly, anyone who talks this way is under the impression that tables and views are different things, and probably also that "tables" always means base tables specifically, and probably also that base tables are physically stored 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 use the term *relation* to mean a relation (possibly a base relation, possibly a view, possibly a query result, and so on); and if I want to mean 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. Likewise, don't fall into the common trap of thinking base relations (or base tables, in SQL) have to be physically stored.

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 material boring. Anyway, now I come to something you might not know already. The fact is, historically

---

[14] Of course, this specific scheme will be inadequate if there can exist suppliers (like supplier S5 in Fig. 1.3) who currently supply no parts.

[15] I say this knowing full well that the majority of today's SQL products do provide a variety of options for hashing, partitioning, indexing, clustering, and otherwise organizing the data as it appears in physical storage. Despite this state of affairs, I still consider the mapping from base tables to physical storage in those products to be fairly direct. (For that very reason, in fact, elsewhere I've labeled those products "direct image systems." For further explanation see my book *Go Faster! The TransRelational™ Approach to DBMS Implementation*, discussed briefly in Appendix G.)

there's been a lot of confusion over yet another logical difference: namely, that between relations as such, on the one hand, and relation *variables* on the other.

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

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

Note carefully that 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 ... ;
```

—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 duplicate rows and left to right column ordering) a relation variable, whose values are relations as such (different relations at different times).

Take another look at Fig. 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 Fig. 1.3, and suppose we delete the set of tuples (actually there's only one) for suppliers in Athens:

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

Here's the result:

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

Conceptually, what's happened here 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 MINUS ( S WHERE CITY = 'Athens' ) ;
```

As with all assignments, the effect here is that (a) the *source expression* on the right side is evaluated and then (b) the result of that evaluation—a relation value in the case at hand, since the source expression is a relational expression specifically—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** (note the boldface). **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 G)—and I'll use it in the present book too, when I'm explaining relational concepts.[16] But since my intended audience is SQL practitioners, I'll show SQL analogs as well, most of the time. *Note:* A BNF grammar for **Tutorial D** can be found in Appendix D. *End of aside.*

To repeat, DELETE is shorthand for a certain relational assignment—and, of course, an analogous remark applies to INSERT and UPDATE also: They too are 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.[17] 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* (three base relvars, to be precise).

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).

---

[16] Several reviewers complained about this fact—that is, they felt I should be using SQL itself, not some nonstandard language like **Tutorial D**, in order to illustrate relational ideas. (One even suggested the book be renamed "**Tutorial D** and Relational Theory"!) But SQL as such was never intended to be a vehicle for illustrating relational ideas, while **Tutorial D** explicitly was; and in any case, SQL simply isn't adequate to the task. Indeed, if it were, a book like this one wouldn't be necessary.

[17] So the relational community isn't perfect, either! But SQL makes essentially the same mistake, of course, because it too has just one term, *table*, that sometimes has to be understood as meaning a table value and sometimes a table variable.

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 here because clear thinking in this area can be such a great help, in so many ways.) Here then are some definitions:

> **Definition:** A *value* is an "individual constant" (this is the term used by logicians), such as 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 location in time and space. Indeed, distinct representations of the same value can appear at any number of distinct locations in time and space—meaning, loosely, that any number of different variables (see the definition immediately following) 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 value. (After all, that's what "variable" means—to be a variable is to be updatable, to be updatable is to be a variable; equivalently, to be a variable is to be assignable to, to be assignable to is to be a variable.)

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 chapter.

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

This brings us to the end of this preliminary chapter. For the most part, my aim in this chapter has been just to tell you what I rather hope you knew already (and you might have felt the chapter 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 *relation*, *tuple*, and *attribute* (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* (or *domain*), *n-ary relation*, *tuple*, *attribute*, *candidate key* (*key* for short), *primary key*, *foreign key*, *entity integrity*, *referential integrity*, *relational assignment*, and *the relational algebra*. (I also briefly mentioned *the relational calculus*.) With regard to the algebra, I mentioned the *closure* property and very briefly described the operators *restrict*, *project*, *product*, *union*, *intersection*, *difference*, and *join*.

- I discussed various properties of relations, introducing the terms *heading*, *body*, *cardinality*, and *degree*. Relations have no duplicate tuples, no top to bottom tuple ordering, and no left to right attribute ordering. I also discussed the difference between *base relations* (or base relvars, rather) and *views*. And I explained that every subset of a tuple is a tuple, every subset of a heading is a heading, and every subset of a body is a body.

- 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 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—and we'll see many more in the pages to come.

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 on and on.

Note: Sometime after writing the previous 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, but it doesn't do anything with those declarative statements; for example, it allows the user to state 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

Now it's your turn. Of course, it isn't possible to set any particularly searching exercises at this early point in the book, and the following are mostly little more than review questions. Nevertheless, I'd like to recommend that you have a go at them before going on to read the answers in the next section.

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.

1.2 Who was E. F. Codd?

1.3 What's a domain?

1.4 What do you understand by the term *referential integrity?*

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.

1.6 Distinguish between the two meanings of the term *data model*.

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

1.8 In the body of the chapter, I said that tables like those in Figs. 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?

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 keys 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 in general terms, of the running example. It's not so important to remember the actual data values in detail, though it certainly wouldn't hurt if you did.)

1.10 "There's only one relational model." Explain this remark.

1.11 The following is an excerpt from a certain 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.

1.12 The following is an excerpt from another 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?

1.13 Write CREATE TABLE statements for an SQL version of the suppliers-and-parts database.

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.

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 the discussion of "what if" queries in Chapter 7 for a detailed explanation.)

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?

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.)

ANSWERS

1.1 Here are a few examples of statements from the early part of the chapter in which every occurrence of the term *relation* (highlighted here in boldface) should be replaced by the term *relvar:*

- "[Every] **relation** has at least one candidate key."
- "[A] foreign key is a combination, or set, of attributes *FK* in some **relation** *r2* such that each *FK* value is required to be equal to some value of some key *K* in some **relation** *r1* (*r1*and *r2* not necessarily distinct)."
- "[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.

1.2 E. F. Codd (1923-2003) was the 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 *www.acm.org/sigmod* and elsewhere. (An expanded version of that tribute appears in my book *Date on Database: Writings 2000-2006*, Apress, 2006. See Appendix G.)

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), as well as being slightly more succinct.

Domain is the term used in most of the older database literature, however. *Caveat:* 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). Also, be aware that some older writings (including certain very early ones by myself) unfortunately and mistakenly use the term *domain* when what they really mean is *attribute*. Be on your lookout for confusion in this area.

1.4 A database satisfies the referential integrity rule if and only if for every tuple containing a *reference* (i.e., a foreign key value) there exists a *referent* (i.e., a tuple in the pertinent "target" relvar with that same value as a value for the pertinent target key). Loosely: If *B* references *A*, then *A* must exist. See Chapters 5 and 8 for further discussion.

1.5 Let *R* be a relvar. Then every relation *r* that can legally be assigned to *R* must have the same heading, and hence a fortiori the same attributes and same degree (see Chapters 2 and 3 for further discussion), and the heading, attributes, and degree of *R* are, respectively, the heading, attributes, and degree of every such relation *r*. They can therefore (and in practice always are) specified as part of the definition of *R*.

Now let the relation that's assigned to *R* at some particular time *t* be *r*. Then the body, tuples, and cardinality of *R* at that time *t* are, respectively, the body, tuples, and cardinality of *r*. Note, therefore, that the body, tuples, and cardinality of a relvar 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*, then the effect, logically speaking, is to replace that table *T* by some 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.

1.6 See the section "Model vs. Implementation" in the body of the chapter.

1.7 (a) 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—typically 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 the protection of investment in training, applications, and logical database designs.

(b) The model is the abstract machine with which users interact; the implementation is the concrete 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 do 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 shouldn't have to know.) *Note:* The term *architecture* is sometimes used with a meaning very similar to that of *model* as defined here.

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 appearances—for example, the symbol 5 (i.e., the *numeral* 5), which denotes an appearance of the value five. See the answer to Exercise 3.5 in Chapter 3 for several further differences.

1.9 *No answer provided.*

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.

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—there might be a better way to do it. (And indeed there is. See my book *Go Faster! The TransRelational™ Approach to DBMS Implementation*, discussed briefly in Appendix G.)

- 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 (i.e., user) level of the system. In particular, the relational model categorically does *not* say that "tables" (meaning, more specifically, base tables, or rather base relvars) 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.

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 on earth 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,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, more generally, 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 see 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 observed: "Structure without operators is ... like anatomy without physiology."

As a kind of postscript to the foregoing, I remark that the relational model certainly seems to have received more than its fair share of misunderstanding or misrepresentation in the literature over the years. Here are a few more quotes to illustrate the point:

- "**Relational model:** A scheme for defining databases in which data elements are organized into relations, typically viewed as rows in tables." As I wrote when I first had occasion to comment on this "definition" (which appears in a book on object technology): Never mind the inaccuracies—you mean that's *it?* What about the operators? What about integrity? What about declarative queries? What about views? What about the model's set level nature? What about optimization? And so on and so forth.

- "A newer form of database manager, the ***relational model***, ... [removes] information about complex relationships from the database ... Although the relational model is much more flexible than its predecessors, it pays a price for this flexibility. The information about complex relationships that was removed from the database must be expressed as procedures in every program that accesses the database, a clear violation of the independence required for modularity." I'll leave comments on this one to you (it's from that same book on object technology).

- "Consider a data relationship in which a part can have multiple suppliers and vice versa ... There are two base tables: a part table and a supplier table. Then there is a cross-reference table from part to supplier *and another cross-reference table from supplier to*

part" (italics added). This quote is from what has to be one of the worst textbooks I've ever read.

1.13 Here are some possible CREATE TABLE statements. Regarding the column data types, see Chapter 2. See also the answer to Exercise 2.15 in that chapter. *Note:* These CREATE TABLE statements, along with their **Tutorial D** counterparts, are repeated in Chapter 5, where further pertinent discussion can 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—again, see Chapters 2 and 5). 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.

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. See Chapter 5 for further discussion.

1.15 I'll give an answer here for completeness (**Tutorial D** again), but I'll defer detailed explanations to Chapters 6 and 7:

```
WITH ( t1 := S WHERE CITY = 'Paris' ,
       t2 := EXTEND t1 : { STATUS := 25 } ) :
S := ( S MINUS t1 ) UNION t2 ;
```

1.16 First consider the generic 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 ;
```

—where *T* is an SQL table corresponding to relvar *R* and *tx* is an SQL table expression corresponding to 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). In other words, the foregoing DELETE / INSERT combination, unlike the assignment it's trying to mimic, is a *sequence* of two distinct statements. One implication of this fact is that a failure could occur between those two statements, something that couldn't happen with the assignment as such.

As for the question "Can all relational assignments be expressed in terms of INSERT and/or DELETE and/or UPDATE?", the answer is *yes* (though in fact we don't need UPDATE as such at all). To be specific, the generic assignment

```
R := rx ;
```

is logically equivalent to:

```
R := ( R MINUS ( R MINUS rx ) ) UNION ( rx MINUS R ) ;
```

To elaborate slightly, let *d* and *i* be the relations denoted by the expressions *R* MINUS *rx* and *rx* MINUS *R*, respectively. Then the original assignment is logically equivalent to the following one:

```
R := ( R MINUS d ) UNION i ;
```

This latter assignment is effectively equivalent to deleting *d* from *R* and then inserting *i* into *R*. Do note, however, that the DELETE and INSERT in question are both being done as part of the same statement, not as two separate statements. See the discussion of *multiple assignment* in Chapter 8.

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 of that chapter 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 overall SELECT expression. That expression is not a relational assignment. (In any case, assignment of any kind is a statement, not an expression. *Statement vs. expression* is another of those important logical differences. See Exercise 2.24 in Chapter 2.)

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 populated with 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, as such, either.

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 G).

Duplicate tuples: Essentially, the concept makes no sense. Suppose for simplicity that the suppliers relvar 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 noted in Chapter 4, 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 G.

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.[18] And if $n > 1$, then 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 $n > 1$ 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.

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" (i.e., 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 to another. This situation contrasts with that found in certain programming systems, where it might be 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 (especially Chapters 6 and 7). See also the paper "A Sweet Disorder," mentioned in Appendix G.

In the interest of accuracy, I should add that for reasons that needn'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. See Appendix A for further discussion.

[18] Note that tuple ordering does indeed constitute a way of representing information—namely, by position; that is, the fact that a given tuple appears *here* and not *there* certainly does represent information, of some kind.

Chapter 2

Types and Domains

A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up.

—Luca Cardelli and Peter Wegner:
"On Understanding Types, Data Abstraction, and Polymorphism"
ACM Comp. Surv. 17, No. 4 (December 1985)

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, I'm going to assume throughout this book that attribute STATUS of the suppliers relvar S is defined to be of type INTEGER. Under that assumption, every relation that's a possible value for relvar S must also have a STATUS attribute of type INTEGER—which means in turn that every tuple in such a relation 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 the chapter. For now, let me just say that—with certain important exceptions, which I'll also be discussing later—a relational attribute (i.e., an attribute of a relation or relvar) 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 very 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, that is, 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.

- Few users will ever be faced with the job of defining a type (and even for those who are, the business of defining a type doesn't involve much in the way of 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). That type contains exactly two values: two truth values, to be specific, denoted by the literals TRUE and FALSE, respectively. However, real systems will support a variety of other system defined types as well, of course, and I'll assume for definiteness that types INTEGER (integers), RATIONAL (rational numbers), and CHAR (character strings of arbitrary length) are all system defined types. *Note:* I'll discuss the system defined types supported by SQL in particular later in the chapter.

> *Aside:* A rational number is a number that can be expressed as the ratio of two integers (e.g., 3/8, 593/370, −4/3); an irrational number is a number that can't be so expressed (e.g., π, √2). Rational numbers in turn fall into two categories: (a) those whose fractional part can be expressed in decimal notation by means of a finite sequence of digits followed by an infinite sequence of zeros, which can be ignored without loss (e.g., 3/8 = 0.375000...), and (b) those whose fractional part can be expressed in decimal notation by means of a possibly empty finite sequence of digits followed by another finite sequence of digits, the first of which is nonzero, that infinitely repeats (e.g., 593/370 = 1.60270270...). By contrast, the fractional part of an irrational number in decimal notation consists of an infinite, nonrepeating sequence of digits (e.g., π = 3.14159..., √2 = 1.41421...). Now, many programming languages support a numeric type they call REAL. A real number is a number that's either rational or irrational; computers being finite, however, the only real numbers they're capable of representing precisely are necessarily rational ones.[1] Hence **Tutorial D**'s choice of the keyword RATIONAL. *End of aside.*

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

[1] And then only rational numbers of the first kind, such as 3/8.

statement on my part, if you like, but I want to present a series of arguments in support of 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 I hope will convince you that domains really are types.

- *Data value atomicity and first normal form:* And this part I hope will convince you that the types in question 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, purely for the sake of the example, that the supplier number (SNO) attributes in relvars S and SP are of some user defined type—sorry, domain—which I'll assume for simplicity is called SNO as well. Likewise, I'm going to assume that the part number (PNO) attributes in relvars P and SP are also 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 the 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 */
```

But why isn't it? *Answer:* 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[2] 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, suppose some user wants to find suppliers (like supplier S5 in the sample values of Fig. 1.3 in Chapter 1) who currently supply no parts at all. The following might be an attempt to formulate this query in SQL:

---

[2] DBMS = database management system. Note that there's a logical difference between a DBMS and a database! Unfortunately, the industry very commonly uses the term *database* when what it means is either some DBMS product, such as Oracle, or the particular copy of such a product that happens to be installed on a particular computer. I do *not* follow this usage in this book. The problem is, if you call the DBMS a database, what do you call the database?

```
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 here because this is an expression, not a statement. See Exercise 2.24 at the end of the chapter.)

As the comment says, this formulation is certainly not OK. The reason is that, in the last line, the user presumably meant to say WHERE SP.SNO = S.SNO, but by mistake—probably just a slip of the typing fingers—he or she said WHERE SP.***PNO*** = S.SNO instead. And, given that we're indeed talking about a simple typo (probably), it would be a friendly act on the part of the DBMS to interrupt 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 of any SQL product that actually behaves in the way I've just suggested; in today's products, depending on exactly how you've set up the database, either the query will simply fail or it'll give the wrong answer. Well ... not exactly the wrong answer, perhaps, but the right answer to the wrong question. (Does that make you feel any better?)

To repeat, therefore, the DBMS should reject a comparison like SP.PNO = S.SNO if it isn't valid. 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 I frankly 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 start using 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, the system certainly mustn't prevent you from doing that comparison, because (of course) the system certainly 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 in this example:

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

And this new clause would be separately authorizable—most users wouldn't be allowed to use it (perhaps only the DBA[3] 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 SQL 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 both 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 checking in the first place, before we even get to domain check override. (In essence, in fact, Codd-style domain checking applies 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 it's at least plausible that the first two of these could be valid (and evaluate successfully, and possibly 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; as previously noted, in fact, I'm assuming for the sake of the present discussion that those particular domains SNO and PNO are actually user defined types. Now, it's possible (even likely) that those user defined types are both physically represented in terms of the system defined type CHAR; in fact, let's assume such is the case, just to be definite. However, those representations are part of the implementation,

---

[3] DBA = database administrator.

not the model—they're irrelevant to the user, and as we saw in Chapter 1 they're supposed to be hidden from the user. In particular, therefore, 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).

As the foregoing paragraph suggests, however, when we define a type, we do also have to define the operators that can be used in connection with values and variables of the type in question. 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.[4] 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')
```

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; in fact, 'X4' is a character string literal). Since they're of different types, they certainly can't be equal (recall from the beginning of the present section 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.

---

[4] 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. (Come to that, they also have nothing to do with the restriction operator of relational algebra, which people sometimes refer to as selection.)

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 as illustrated in the foregoing examples is often called *coercion* in the literature. In the first example, therefore, we can say 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—in the case at hand—effectively converts a given SNO or PNO value to the character string (or whatever else it is) that's used to represent it.[5] Assume for the sake of the example that the THE_ operators for types SNO and 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 corresponding 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 here, 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.[6] 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—

---

[5] 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 some types might have more than one associated THE_ operator. See Chapter 8 for further discussion.)

[6] If you're not familiar with orthogonality as an important language design principle, you can read about it in "A Note on Orthogonality" in my book *Relational Database Writings 1994-1997* (Addison-Wesley, 1998).

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, that they're coercible to those right types). Observe, moreover, that this mechanism works for all operations, not just for the equality comparisons I've been discussing; the emphasis in the literature, in discussions of domain checking, on equality and other comparison operations is sanctioned 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 possibly mean to add a weight and a quantity?).

I'd like to close this section by stressing the absolutely fundamental role played—not just in type theory!—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 specifically. 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. (As a matter of fact, I said exactly this in Chapter 1, as you might recall.) 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

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 stipulation raises the obvious 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; even at best, it's not very precise. For example, what about character strings? Are character strings atomic? Well, every database 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 into smaller pieces by the DBMS." So are such 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 allow as attribute values in tuples in relations:

- Bit strings
- Rational 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.

Before drawing any conclusions from the discussion so far, I'd like to consider another example, one that some might regard as more startling, in a way. Refer to Fig. 2.1 below. 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 pertinent {SNO,PNO} combination. Further, let's agree for the sake of the example 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}

Fig. 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 a case of "repeating groups," and repeating groups are the one thing

that just about 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!* For consider:

- First, note that I've renamed the attribute PNO_SET, and I've enclosed the groups of part numbers that are PNO_SET values 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.[7]

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 Fig. 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 Fig. 2.2, where I've changed the attribute name to PNO_REL). Why would it be more useful? Because relations, not general sets, are what the relational model is all about.[8] 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!

---

[7] Observe that I don't claim R3 is 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.

[8] 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

Fig. 2.2: Relation R4 (a revised version of R3)

Terminology: Attribute PNO_REL in Fig. 2.2 is 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.[9] 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 (multisets); 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).

---

[9] The individual elements in an SQL multiset don't have to be rows but can be values of any available SQL type—for example, integers. The same goes for arrays as well.

Incidentally, you might recall that a few years ago we were hearing a great deal about so called "object/relational" systems. Well, the foregoing paragraph goes a long way toward explaining why a true object/relational system would in fact 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 from the user's point of view is precisely that we can have attribute values in relations that are of arbitrary complexity. Perhaps a better way to say it 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 are pleased to 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). To elaborate briefly:

- The types we're interested in are always *finite* because we're dealing with computers, which (as pointed out in connection with type RATIONAL earlier in the chapter) are finite by definition.

- Note also that qualifier *named:* Types with different names are different types.

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.

  Aside: If every value is of exactly one type, then no value is of two or more types, and thus types are always disjoint (nonoverlapping). However, perhaps I need to elaborate on this point briefly. As one reviewer of this chapter 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* type inheritance. Since those reasons, and indeed the whole topic of inheritance as such, are independent of the context we're in (be it relational or something else), I'm not going to discuss them in this book. *End of aside*.

- Every *variable*, every *attribute*, every *operator* that returns a result, and every *parameter* of every operator is defined, or declared, to be of some type.[10] 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 in turn of type *T*.

- Every *expression* denotes some value and is therefore of some type: namely, the type of the value in question, which is to say the type of the value returned by the outermost operator in the expression (where by "outermost" I mean the operator that's executed last). For example, the type of the expression

  ```
  ( a / b ) + ( x - y )
  ```

  is the type declared for 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* basically just means that operator *Op* has a parameter of type *T*.[11] For example, integers have the usual arithmetic operators; dates and times have special calendar arithmetic operators; XML documents have what are called "XPath" and "XQuery" 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 the ability for 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 (or a mixture, of course), as you would surely expect.

Observe now that, by definition, values and variables of a given type *T* can be operated upon only by means of the operators associated with that type *T*. For example, in the case of the system defined type INTEGER:

- The system provides an assignment operator ":=" for assigning integer values to integer variables.

---

[10] Throughout this book I treat *declared* and *defined* as synonymous.

[11] 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* as such—*not* the operators associated with the representation of type *T*. For example, just because the representation for type SNO happens to be CHAR (say), 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.)

- It also provides a format for writing integer literals. (However, it doesn't provide any more general integer selector operators, nor does it provide any corresponding THE_ operators, because—as should be obvious if you think about it—such operators aren't needed for a system defined type like INTEGER.)

- 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 LIKE, SUBSTR (substring), "| |" (concatenate), 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 (still assuming it *is* user defined), we would certainly define the necessary selector and THE_ operators, and we would also define assignment (":=") and comparison operators ("=", "≠", possibly "<", and so on). However, we probably wouldn't define operators "+", "*", and so on, which would mean that arithmetic on supplier numbers wouldn't be supported (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. Defining a name for the type (obviously enough).

2. Defining the values that make up that type. I'll discuss this aspect in detail in Chapter 8.

3. Defining 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 (at least, not much).

4. Defining one or more selector operators for selecting, or specifying, values of that type. *Note:* Here's as good a place as any to point out in the interest of accuracy that the selectors for type *T* aren't "associated with" type *T* in the sense that they have a parameter of type *T*; rather, they return a result of type *T*.

5. Defining the operators, including in particular assignment (":="), equality comparison ("="), and THE_ operators, that apply to values and variables of that type (see below).

6. For those operators that return a result, defining the type of that result (again, see below).

Observe that points 4, 5, and 6 taken together imply that (a) the system knows precisely which expressions are legal, and (b) 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 (a) is called REFLECT, (b) takes a single parameter P, of type POINT, and (c) returns a result also of type POINT (so the type of the operator is declared to be 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 to be returned. Each of those arguments is defined by means of 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.[12]

- Line 3 marks the end of the definition.

Now, the discussions in this section so far have been framed in terms of user defined types, for the most part. But similar considerations apply to system defined types also, 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, defines legal integer values, defines the hidden representation, and—as we've already seen—defines a corresponding literal format, defines the corresponding operators ":=", "=", "+", and so on (though users can define additional operators, of course, if they want to).

There's one last point I want to make. I've mentioned selector operators several times; what I haven't said, however (at least not explicitly), is that selectors—more precisely, selector

---

[12] 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 SNO and PNO selectors discussed earlier, takes two arguments (because points are represented by pairs of values, not just by a single value). See Chapter 8 for further discussion.

invocations—are really just a generalization of the more familiar concept of a *literal*.[13] What I mean by this remark is that all literals are selector invocations, but 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 has (*must* have) an associated format for writing literals. And for completeness I should add that every value of every type must be denotable by means of some literal of the type in question.

SCALAR vs. NONSCALAR TYPES

Types are frequently said to be 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 really 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, however, I do rely on it informally; I mean, I do find it intuitively useful on occasion. To be specific, I occasionally 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.[14]

> *Aside:* Another term you'll sometimes hear used to mean "scalarness" is *encapsulation*. Be aware, however, that this term is also used—especially in object oriented contexts—to refer to the physical bundling, or packaging, of code and data (or operator definitions and data representation definitions, to be more precise). But this latter use of the term mixes model and implementation considerations; clearly the user shouldn't care, and shouldn't

[13] The concept might be familiar, but it seems to be quite difficult to find a good definition for it in the literature! See Exercise 2.2.

[14] This sentence is only an approximation to the truth. A more accurate statement would be: Nongenerated types—see later in the present section—are scalar; generated types (e.g., relation types) are typically nonscalar, but don't have to be. An example of a scalar generated type is the SQL type CHAR(25) (see the next section).

need to care, whether code and data are physically bundled together or are kept separate. *End of aside.*

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, and no two attributes in the same heading have the same attribute name). 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 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 a 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 present purposes, however, the only type generators we're interested in are TUPLE and RELATION. Here's an example involving the TUPLE type generator:

```
VAR STV /* tuple variable */
    TUPLE { STATUS INTEGER , SNO CHAR , CITY CHAR , SNAME CHAR } ;
```

The value of variable STV 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 order, just to show the order

doesn't matter).[15] Thus, we might imagine a code fragment that (a) extracts a one-tuple relation (perhaps the relation containing just the tuple for supplier S1) from the current value of relvar S, then (b) extracts the single tuple from that one-tuple relation, and finally (c) assigns that tuple to the variable STV. In **Tutorial D:**

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

Important: I don't want you to misunderstand me here. While a variable like STV 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). In other words, relvars are the *only* kind of variable allowed in a relational database. *Note:* This fact—i.e., that relvars are the only kind of variable allowed in a relational database—constitutes what's called *The Information Principle*. I'll have more to say about it in Appendix A.

By the way, note carefully that (as the foregoing example suggests) there's a logical difference between a 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, or in other words the same attributes and same degree).

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 the form in which they're seen by the user. In fact, the physical representation of tuples and relations should be hidden from the user, just as it is for scalar values (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—relational assignment, and tuple and relational equality comparisons—to the next chapter.

[15] Note that it does make sense to talk about the heading of a tuple—tuples have headings just as relations do, as will be explained in more detail in 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 (it also allows users to define their own types, but as I've already said I don't intend to say much about user defined types in this chapter):

```
BOOLEAN      INTEGER           CHARACTER(n)
             SMALLINT          CHARACTER VARYING(n)
             BIGINT            CHARACTER LARGE OBJECT(n)
             NUMERIC(p,q)      BINARY(n)
             DECIMAL(p,q)      BINARY VARYING(n)
             FLOAT(p)          BINARY LARGE OBJECT(n)
```

This isn't a complete list—other SQL system defined types include an "XML document" type (XML); a variety of "national character string types" (NATIONAL CHARACTER(*n*), etc.); and a variety of datetime types (DATE, TIME, TIMESTAMP, INTERVAL). However, I'll be ignoring such types, mostly, for the purposes of this book. Points arising:

- A number of defaults, abbreviations, and alternative spellings, including INT for INTEGER, CHAR for CHARACTER, VARCHAR for CHARACTER VARYING, VARBINARY for BINARY VARYING, CLOB for CHARACTER LARGE OBJECT, BLOB for BINARY LARGE OBJECT, are also supported.

- As you can see, SQL, unlike **Tutorial D**, requires its various character string types to have an associated length specification.

- The same goes for the various BINARY types. Note that BINARY doesn't mean binary numbers, it means *bit string* (or, perhaps more accurately, *byte* string, since the associated length specification gives the corresponding length in *octets*).[16]

- The *p* in NUMERIC, DECIMAL, and FLOAT is the associated *precision*, and the *q* in NUMERIC and DECIMAL is the associated *scale factor* ($p > 0$, $0 \leq q \leq p$).[17] Thus, e.g., the specification DECIMAL(5,2) denotes decimal numbers in the range −999.99 to +999.99, inclusive.

- Strictly speaking, CHAR (for example) isn't really a type as such—rather, it's a type *generator*. By contrast, CHAR(25), for example, *is* a type as such, and it's obtained by invoking that type generator with the value 25 as sole argument to that invocation. What's more, analogous remarks apply to every "scalar type" in the foregoing list except for type

[16] True bit string types—BIT(*n*) and BIT VARYING(*n*), where *n* was the length in bits—were introduced in SQL:1992 but dropped again in SQL:2003.

[17] SQL actually calls *q* simply the *scale*, but there are good reasons to prefer the term *scale factor*.

BOOLEAN and the various integer types (SMALLINT, INTEGER, BIGINT).[18] For simplicity, however, I'll overlook this point in what follows (most of the time, at any rate) and continue to refer to CHAR and the rest as if they were indeed types as such, much as SQL itself does.

- Literals of more or less conventional format are supported for all of these types.

- 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 convenient abbreviations for *reference* and *expression*, respectively.

- An explicit equality comparison operator is also supported for all of these types.[19] The syntax is:

  ```
  <scalar exp> = <scalar exp>
  ```

  Equality comparisons are also performed implicitly when numerous other operations (e.g., joins and unions, grouping and duplicate elimination operations, and many others) are executed.

- Regarding type BOOLEAN in particular, I should point out that although it's included in the 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 in question 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 (e.g., "yes/no columns") might sometimes be needed.

---

[18] SQL also supports a ROW type generator, as we'll see in the section "Row and Table Types in SQL," later. In fact, it also supports ARRAY, MULTISET, and REF (but, oddly enough, not TABLE) as type generators.

[19] Unfortunately, however, that support is severely flawed. First of all, SQL supports coercions (see the section immediately following this one), with the consequence that "=" can give TRUE even when the comparands are of different types. Second, in the case of character string types, it's possible for "=" to give TRUE even when the comparands are of the same type but clearly distinct (see the next section but one, "Collations in SQL"). And it's also possible—for all types, not just character string 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. Also, for certain types not discussed in detail in this book, including type XML and certain user defined types, "=" isn't defined at all. *Note:* This list of flaws in SQL's support for "=" is *not* complete.

- Finally, in addition to the foregoing scalar types, SQL also supports something it calls domains. However, 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 well 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).

  Note: One thing SQL's domains most definitely don't do is constrain comparisons. For example, suppose columns S.CITY and P.CITY are defined on SQL domains SCD and PCD, respectively. Then you might expect the comparison S.CITY = P.CITY to fail. However, it won't, not necessarily; rather, it'll fail if and only if the data types underlying those domains fail to satisfy the requirements for such comparisons as outlined in the section immediately following. In SQL, in other words, such comparisons are legal if and only if the columns involved have what might be called compatible data types, regardless of whether they're defined on the same SQL domain, and regardless even of whether any SQL domains as such are involved at all.

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.

- Numeric values can be assigned only to numeric variables and compared only with numeric values (where "numeric" means INTEGER, SMALLINT, BIGINT, NUMERIC, DECIMAL, or FLOAT).

- Character string values can be assigned only to character string variables and compared only with character string values (where "character string" means CHAR, VARCHAR, or CLOB).

- Bit string values can be assigned only to bit string variables and compared only with bit string values (where "bit string" means BINARY, VARBINARY, or BLOB).

Thus, for example, an attempt to compare a number and a character string is illegal. However, an attempt to compare (say) 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.

Now, it's a widely accepted principle in computing that coercions are generally 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! By way of example, consider the SQL tables T1 and T2 shown in Fig. 2.3 below. Let column X be of type INTEGER in table T1 but NUMERIC(5,1) in table T2, and let column Y be 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
```

The result is shown as the rightmost table (T3) in Fig. 2.3. As the figure suggests, columns X and Y in that result 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.[20]

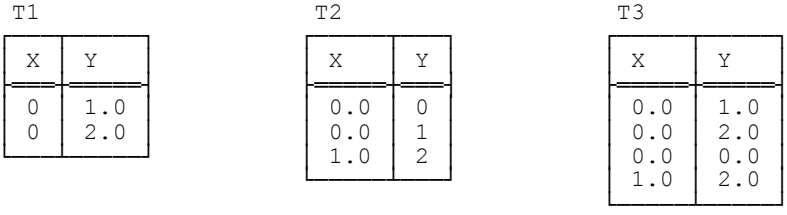


Fig. 2.3: A very strange "union"

Strong recommendations: 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 any 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 elsewhere 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
```

[20] One reviewer suggested that the "strangeness" of the union in this example 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; I don't want to argue the point. But if the SQL language designers want to define an operator that manifestly doesn't behave like the union operator of the relational model (or the union operator of set theory, come that), then it seems to me that, first, it doesn't help the cause of understanding to call that operator "union"; second (and rather more important), it isn't up to me to show that such a "union" can sometimes cause problems—rather, it's up to those language designers to show that it can't.

For completeness, however, I need to add that certain coercions are unfortunately built into the very fabric of SQL and so 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, albeit a trifle loosely, just a SELECT 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 (a) θ is a simple scalar comparison operator, such as "<" or ">",[21] followed by the keyword ALL or ANY, and (b) *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 (sadly) rather 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

[21] As I'm sure you know, the scalar comparison operators supported by SQL are "=", "<>" (not equals), "<", "<=" (less than or equals), ">", and ">=" (greater than or equals).

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*). *Note:* In early versions of SQL there was just one character set, that character set had just one collation, and that collation was based on the numerical order of the binary codes used to represent the characters in that character set. But there's no intrinsic reason why collating sequences should have to depend on internal coding schemes, and there are good practical reasons why they shouldn't.

So much for the basic idea. However, there are complications. One arises from the fact that any given collation can (or, rather, must) 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) have the same 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.[22]

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 defined 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 (possibly even if they're of different types, thanks to SQL's support for coercion). I'll use the term "distinct, considered equal" 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 such that "distinct, considered equal" values are treated as equal. 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 spaces in the foreign key value.

What's more, when evaluating expressions involving operators such as UNION, INTERSECT, EXCEPT, JOIN, GROUP BY, DISTINCT (and so on), the system sometimes has to decide which of several "distinct, considered equal" values is to be chosen as the value of 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

---

[22] As a historical note, I remark that in the original (i.e., IBM) version of SQL, the only available collation—which was based on the internal coding scheme, of course—supported PAD SPACE only, and did that only tacitly. The reason for this state of affairs was a desire on the part of the language designers in IBM to conform to the corresponding rules for PL/I.

evaluated; indeed, they might quite legitimately give different results on different occasions. For example, if collation CASE_INSENSITIVE applies to column *C* in table *T*, then SELECT MAX(*C*) 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" (see Chapter 12 for further discussion). It's important to note, however, that such expressions aren't allowed in integrity constraints (see Chapter 8), presumably 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.

> *Aside:* As I've just said, possibly nondeterministic expressions aren't allowed in constraints. Oddly enough, however, they *are* allowed to appear in queries and updates, where they can surely do just as much damage (?). *End of aside*.

ROW AND TABLE TYPES IN SQL

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

```
VAR STV 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 calls it a type *constructor*). Here's an SQL analog of the foregoing **Tutorial D** example:

```
DECLARE SRV /* 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.[23] In the case at hand, therefore, there are actually 4! = 4 * 3 * 2 * 1 = 24 different row types all consisting of the same four components (!).[24]

SQL also supports row assignment. Recall this **Tutorial D** tuple assignment:

---

[23] For some reason SQL refers to the components of row types produced by invocation of the ROW type constructor (and to the components of rows of such types) not as columns but as *fields*.

[24] More generally, the expression *n*! (which is read as either "*n* factorial" or "factorial *n*" and is often pronounced "*n* bang") is defined as the product *n* * (*n*−1) * ... * 2 * 1.

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

Here's an SQL row assignment analog:

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

The expression on the right side here is a *row subquery*—i.e., it's a table expression, syntactically speaking, but it's a table expression that's acting as a row expression. That's why there's no explicit counterpart in the example to **Tutorial D**'s TUPLE FROM (see the discussion of subqueries and coercion in the section "SQL Type Checking and Coercion" a couple of pages back).

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

Turning now to tables: Interestingly, SQL doesn't really have a TABLE type generator (or type constructor, as SQL would probably call it) 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 defining what by rights should be called table variables. For example, recall this **Tutorial D** 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), which defines a certain integrity constraint on suppliers, doesn't have to come after the column definitions but can appear almost anywhere—e.g., between the definitions of columns SNO and SNAME. Not to mention the NOT NULL specifications on the individual column definitions, which also define certain integrity constraints.) 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 said, I should say too 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 explicitly prohibited in the relational model.[25] In fact, if some table *T* has a column whose values are pointers to rows in some "target" table *T′*, then that table *T* can't possibly represent a relation in the relational model sense. (As a matter of fact, the target table *T′* can't do so either.) As I've just indicated, however, tables like table *T* 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.

> *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 target key value. Note, however, that foreign keys certainly aren't pointers!—there are many 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." Values, having no location, certainly don't have addresses.) *End of aside.*

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* (other than as noted in just a moment)—the relational model nowhere prescribes just what those types must be, and in fact they can be as complex as you like (they can even be relation types). 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

---

[25] 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 certain *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 located at the given address).

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 (both of which I'll simplify just slightly for present purposes). The first 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 (which in fact I've already touched on) 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 to those databases almost impossible. (These aren't the only problems with pointers, but they're among the more obvious ones.) Codd wanted to get away from such problems in his relational model, and of course he succeeded.

EXERCISES

2.1 What's a type? What's the difference between a domain and a type?

2.2 What do you understand by the term *selector?* And what exactly is a literal?

2.3 What's a THE_ operator?

2.4 Physical representations are always hidden from the user: True or false?

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
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?

2.6 Explain in your own words the difference between the concepts *scalar* and *nonscalar*.

2.7 What do you understand by the term *coercion?* Why is coercion a bad idea?

2.8 Why doesn't domain check override make sense?

2.9 What's a type generator?

2.10 Define *first normal form*. Why do you think it's so called?

2.11 Let *X* be an expression. What's the type of *X*? What's the significance of the fact that *X* is of some type?

2.12 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.

2.13 Let LENGTH be a user defined type, with the obvious semantics. Use **Tutorial D** to define an operator that, given the length of two adjacent sides of a rectangle, returns the corresponding area.

2.14 Give an example of a relation type. Distinguish between relation types, relation values, and relation variables.

2.15 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?

2.16 With reference to the departments-and-employees database from Chapter 1 (see Fig. 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 (or would-be expressions) are valid? For those that are, state the type of the result; for the others, give 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'`

2.17 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.

2.18 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?

2.19 In the relational world, the equality operator "=" applies to every type. By contrast, SQL doesn't require "=" to apply to every type, 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?

2.20 Following on from the previous exercise, we can say that if *v1* = *v2* evaluates to TRUE in the relational world, then 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?

2.21 Why are pointers excluded from the relational model?

2.22 *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?

2.23 Do you think that types "belong to" databases, in the same sense that relvars do?

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

2.25 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."

2.26 What's a subquery?

2.27 To repeat from Exercise 2.19: In the relational world, the equality operator "=" applies to every type. But what about type BOOLEAN? And what about SQL's row and table types?

ANSWERS

2.1 A type is 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 (etc., etc.). There's no difference between a domain and a type. *Note:* SQL does draw a distinction between domains and types, however. The distinction shows up most immediately in the fact that SQL supports both a CREATE TYPE statement and a CREATE DOMAIN statement. To a first approximation, CREATE TYPE can be thought of as 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, between the two). As for CREATE DOMAIN, it might be regarded, very charitably, as SQL's attempt to provide a tiny part of the total functionality of CREATE TYPE (it was introduced in SQL:1992, while CREATE TYPE wasn't introduced until SQL:1999); now that CREATE TYPE exists, there seems little reason to use, or even support, CREATE DOMAIN at all.

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 successful invocation of *S*, and every successful 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 required by the relational model, at least implicitly—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  */
2.5                        /* a literal of type RATIONAL */
POINT ( 5.0 , 2.5 )        /* a literal of type POINT    */
```

(The last of these involves the user defined type POINT from the body of the chapter.)

Every value of every type, tuple and relation types included, must be denotable by means of some literal (of the applicable type, of course). 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 as literals in turn (implying in particular that a selector invocation with no arguments at all, like the INTEGER selector invocation 4, is a literal by definition). Note finally that there's a logical difference between a literal as such and a constant—a constant is a value, while a literal is a symbol that denotes such a value. (By the same token. there's a logical difference between a literal and a value—as just stated, a value is a constant, such as the constant 3, while a literal is a symbol that denotes such a constant.)

> *Aside:* Some languages also support so called "named constants." A named constant denotes a value—the constant in question—that can be referenced by means of a name that's not just a simple literal representation of that constant. In other words, a named constant resembles a named variable, in that it can be thought of as an abstraction of a storage location that contains a value; however, it differs from a variable in two obvious ways. First, it can never serve as the target for an assignment operation. Second, every reference to the pertinent name always denotes the same value. *End of aside*.

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.

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). Incidentally, the epigraph to the chapter is highly pertinent to the present exercise. Here it is again: "A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up." In other words, types are a good idea because they *raise the level of abstraction* (without a proper type system, everything would be nothing but tedious—and error prone—bit twiddling). And here's another nice quote (this one's from Andrew Wright: "On Sapphire and Type-Safe Languages," *CACM 46*, No. 4, April 2003): "[Types make] program development and debugging easier by making program behavior more understandable."

2.5 A *parameter* is a formal operand in terms of which some operator is defined. An *argument* is an actual operand that's 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 represents 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 represents an invocation of the arithmetic operator "+". (In fact, of course, *every* expression represents some operator invocation. Even a simple variable reference—*V*, say—can be be regarded as representing an invocation of a certain operator: namely, the operator that returns the current value of the specified variable *V*.)

A *database* is a repository for data. (*Note:* In the relational world, we might say, a little more specifically, that a database is a container for relvars. But much more precise definitions are possible; one such can be found in Chapter 5 of this book. See also Appendix A.) A *DBMS* is a software system for managing databases; it provides data storage, 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 "target" key in some other relvar (or possibly in the same relvar). A *pointer* is a value (an *address,* essentially) for which certain special operators—notably certain referencing and dereferencing operators—can (and in fact must) be defined.[26] *Note:* Brief definitions of the referencing and dereferencing operators were 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, RELATION, or (in SQL) CHAR; specific array, relation, and (in SQL) character string 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—a relation type, of course—but it isn't itself a type. A *type* is a named, finite set of values: viz., all possible values of some particular kind.

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 values and variables of type *T*, not the operators defined for the representation that applies to values and variables of type *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 values of type ENO. See the answer to Exercise 2.4 above for further discussion.

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

---

[26] 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 G).

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 user who actually defines 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* 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 user who designs and implements 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 obviously take arguments of system defined types only.

2.6 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, and operators (etc.) 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.

2.7 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 or nothing to do with the relational model as such).

2.8 Because it muddles type and representation.

2.9 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. Points arising:

- Types generated by the TUPLE and RELATION type generators are nonscalar, but there's no reason in principle why generated types have to be nonscalar. SQL in particular supports several scalar type generators (CHAR, NUMERIC, REF, and many others).

- Type generators are known by many different names in the literature, including *type constructors* (the SQL term), *parameterized types*, *polymorphic types*, *type templates*, and *generic types*.

2.10 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 (and as was in fact mentioned in Chapter 1), 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 the years. 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 such matters can be found in the paper "What First Normal Form Really Means" (see Appendix G).

2.11 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 of type *T* can appear.

2.12
```
OPERATOR CUBE ( I INTEGER ) RETURNS INTEGER ;
   RETURN I * I * I ;
END OPERATOR ;
```

2.13
```
OPERATOR AREA_OF_R ( H LENGTH , W LENGTH ) RETURNS AREA ;
   RETURN H * W ;
END OPERATOR ;
```

I'm assuming here, not unreasonably, that (a) it's legal to multiply ("*") a value of type LENGTH by another such value, and (b) the result of such a multiplication is a value of type AREA (another user defined type).

2.14 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 Fig. 1.3 in Chapter 1—is a value of this type.

2.15 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 RATIONAL , CITY CHAR }
  KEY { PNO } ;

VAR SP BASE RELATION
  { SNO CHAR , PNO CHAR , QTY INTEGER }
  KEY { SNO , PNO }
  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 (and certain other items too, beyond the scope of the present discussion), 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—or table type, rather—at all. See Chapter 3 for further discussion.)

- The left to right order in which columns are listed matters in SQL. See Chapter 3 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 Chapters 6 and 12 for further discussion.

One further point: As you can see, I've defined attribute QTY to be of type INTEGER. However, my reason for doing so is partly historical—every DBMS I know supports type INTEGER, while few DBMSs if any support the type that would really be more appropriate in the case at hand (viz., NONNEGATIVE_INTEGER, with the obvious semantics). Of course, we could make NONNEGATIVE_INTEGER a user defined type, but as I've said I don't want to get into too much detail regarding user defined types in this book.

2.16 I assume throughout the following answers a.-g. that a given type *T* always has a selector with the same name. See Chapter 8 for further discussion.

a. Not valid; LOCATION = CITY('London').

b. Valid; BOOLEAN.

c. Presumably valid; MONEY. I'm assuming that multiplying a money value by an integer 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"—see Chapter 8—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.

2.17 Such an operation logically means replacing one type by another, not "updating a type" (types aren't variables and hence can't be updated, by definition). 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). Thus, all the "define type" operation (the TYPE statement, in the case of **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 very same name to refer to a different set of values. Of course, there's nothing to preclude support for some kind of pragmatic "alter type" shorthand to simplify matters—and SQL does support such an operator, in fact—but invoking such a shorthand shouldn't be thought of as "updating the type."

2.18 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 the present book. Refer to the book *Databases, Types, and the Relational Model: The Third Manifesto,* by Hugh Darwen and myself (see Appendix G), if you want to know more.

2.19 Let *T* be an SQL type for which "=" isn't defined and let *C* be a column of type *T*. Then *C* can't be part of a key or 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 (so *T* is necessarily user defined itself), and let *C* be a column of type *T*. Then the effects of making *C* part of a key or foreign key or applying DISTINCT or GROUP BY (etc., etc.) to it will be user defined as well. *Note:* Presumably for this very reason, the standard doesn't actually allow such a column *C* to be used in all of the contexts just mentioned—and possibly not in any of them (?). The specifics of exactly what's allowed are baroque in the extreme, however; so if you want to know more, I'm afraid I'm going to have to refer you to the standard itself (see Appendix G).

2.20 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. (Note too that even though the comparison X = Y gives TRUE, the comparison X||X = Y||Y doesn't!) 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 keys, foreign keys, and certain implementation constructs, such as indexes).

2.21 Because (a) they're logically unnecessary, (b) they're error prone, (c) end users can't readily use them, (d) they're clumsy—in particular, they have a direction to them, which other 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 in a footnote to the answer to Exercise 2.5, "Inclusion Dependencies and Foreign Keys," for further discussion.

2.22 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. E.g., let X be a variable of type CHAR(3), let Y be the character string 'AB' (no trailing space), and let NO PAD apply to the pertinent collation. Then assigning Y to X will actually set X to the string 'AB ' (one trailing space), and after that assignment the comparison X = Y will give FALSE. Again I leave the implications for you to think about.

2.23 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.

2.24 An expression represents an operator invocation, and it denotes a value; it can be thought of as a rule for computing or determining the value in question. (Incidentally, the arguments to that operator invocation are themselves specified as expressions in turn—though the expressions in question might just be simple literals or simple variable references.) By contrast, 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.

2.25 An RVA is an attribute whose type is some relation type, and whose values are therefore relations of that type (see Chapter 7 for further discussion). 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 ...) of values of type *T*. *Note:* Type *T* here is often a tuple type (or something approximating a tuple type). In a system that allows repeating groups, for example, a file 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).

2.26 "Subquery" is an SQL term meaning, loosely, a SELECT expression enclosed in parentheses. Later chapters will elaborate (especially Chapter 12).

2.27 Regarding SQL row and table types, see Chapter 3. As for type BOOLEAN, yes, "=" does apply; TRUE is equal to TRUE and FALSE is equal to FALSE. In SQL, what's more, "<" applies as well!—FALSE is considered to be less than TRUE (i.e., the comparison "FALSE < TRUE" returns TRUE, in SQL).

Chapter 3

Tuples and Relations, Rows and Tables

[I] have reduced several great confused Volumes into a few perspicuous Tables.
—John Graunt (1662)

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 logical 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 showing many 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:** A *heading H* is a set of *n attributes* ($n \geq 0$),[1] each consisting of an *attribute name Ai* and a corresponding *type name Ti*, such that the attribute names *Ai* are all distinct. The value *n* is the *degree* of *H*; a heading of degree one is *unary*, a heading of degree two is *binary*, a heading of degree three is *ternary*, ..., and more generally a heading of degree *n* is *n-ary*. Let each attribute *Ai* ($i = 1, 2, ..., n$) be associated with an *attribute value vi* of type *Ti*, and let each of the *n* attribute : value pairs that results be called a *component*. The set—call it *t*—of all *n* components so defined is a *tuple value* (or just a *tuple* for short) over the attributes of *H*. *H* is the *tuple heading* (or just heading for short) for *t*, and the degree and attributes of *H* are, respectively, the degree and attributes of *t*.

Thus, for example, with reference to either of the pictures above of the tuple for supplier S1, we have:

- *Attribute names:* SNO, SNAME, STATUS, CITY.
- *Corresponding type names*: CHAR, CHAR, INTEGER, CHAR.
- *Attributes:* SNO:CHAR, SNAME:CHAR, STATUS:INTEGER, CITY:CHAR.
- *Corresponding attribute values:* 'S1', 'Smith', 20, '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 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 logical difference. (Recall from Chapter 2 that the

[1] In previous editions of this book, headings were denoted {*H*} instead of *H*.

expression SNO('S1') is a selector invocation—in fact, a literal—of type SNO.) *End of aside.*

- *Heading:* {SNO:CHAR, SNAME:CHAR, STATUS:INTEGER, CITY:CHAR}.
- *Degree:* 4.

By the way, it's sometimes convenient to represent headings on paper (like tuples) by means of a picture, as in this example:

SNO : CHAR	SNAME : CHAR	STATUS : INTEGER	CITY : CHAR

Of course, this picture represents a set, and so the order in which the attributes are shown 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 same heading? *Answer:* 4! (factorial 4) = 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 }
```

The order in which the attributes are specified is arbitrary, of course. Note, however, that **Tutorial D** uses spaces, not colons, to separate attribute names from their corresponding type names.

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' }
```

The order in which the components are specified is again arbitrary. Note, however, that in **Tutorial D** each component is specified by means of the pertinent attribute name by itself (i.e., without the corresponding type name), separated by spaces from an expression denoting the pertinent attribute value; there's no need to specify the attribute type explicitly, because it's necessarily the same as that of the specified expression.

Here's another example of a tuple selector invocation (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 'Johns' , STATUS TX , CITY CX }
```

I'm assuming here that SX, TX, and CX 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 (but without the colon separators), the whole commalist being 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 also (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 this: *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 as we saw in Chapter 1 nulls aren't values—despite the fact that SQL does often, though not invariably, refer 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. Observe that this recommendation isn't just a matter of pedantry; rather, it's a matter of thinking straight. SQL itself manages to make numerous mistakes in its handling of nulls, and some of those mistakes can be traced directly to the fact that SQL does sometimes, though not always, think of null as a value.[2]

Now, if no tuple ever contains any nulls, then no relation does so 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—or pair of consequences, rather—is: *Every subset of a tuple is a tuple and every subset of a heading is a heading*. (I did mention these points in Chapter 1, but now I want to elaborate on them.) 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):

---

[2] Indeed, this ambivalence is reflected in the standard's very definition of the concept, which reads as follows: "**null value:** A special value that is used to indicate the absence of any data value." In other words: Null is a value that means there isn't a value.

SNO : CHAR	CITY : CHAR
S1	London

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

SNO : CHAR
S1

This tuple is of degree one, and its type is TUPLE {SNO CHAR}.

Now, as I'm sure you know, the *empty* set—i.e., the set that contains no elements—is a subset of every set. It follows that the empty heading is a valid heading!—and hence that a tuple with an empty set of components is a valid tuple (though it's a little hard to draw pictures of such a tuple on paper, and I'm not even going to try). A tuple with an empty heading has type TUPLE { }; indeed, we sometimes refer to it explicitly as a *0-tuple,* in order to emphasize the fact that it has no components and is of degree zero. We also sometimes call it an *empty tuple*. Now, you might be thinking 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.

Let's get back to the original tuple for supplier S1 (i.e., the one of degree four) for a moment. Suppose we're given that tuple and we want to access the actual value of some attribute, say the SNO attribute, from that tuple. Then we have to *extract* that value, somehow, from the tuple that contains it. **Tutorial D** uses syntax of the form SNO FROM *t* for this purpose (where *t* is any expression that denotes a tuple with an attribute called SNO). SQL uses dot qualification: *t*.SNO.

Note: It follows from the foregoing paragraph that a value *v* and a tuple *t* that contains just that value *v* aren't the same thing; in particular, they're of different types. This logical difference is analogous to that described in Chapter 2, between a tuple *t* and a relation *r* that contains just that tuple *t*; these aren't the same thing either (they too are of different types).

Now I'd like to turn to the notion of *tuple equality*. (Again I mentioned this notion in Chapter 1, but now I want to elaborate on it.) Recall first from Chapter 2 that the "=" comparison operator is—in fact, must be—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 if not all of the operators of the relational algebra are defined in terms of it.) Here then is a precise definition:

Definition: Tuples *t* and *t′* 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 *v* of *Ai* in *t* is equal to the value *v′* of *Ai* in *t′*.

Also (to repeat from Chapter 1, this might seem obvious, but it needs to be said), two tuples are *duplicates* of each other if and only if they're equal. Thus, e.g., the tuple for supplier S1 in the suppliers relation of Fig. 1.3 is equal to, and is therefore a duplicate of, itself—and it *isn't* equal to, or a duplicate of, anything else (any other tuple in particular).

By the way, it's an immediate consequence of the foregoing 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. Note, moreover, that we can validly say that the 0-tuple is a subset, or "subtuple," of every tuple (just as we can say the empty set is a subset of every set).

So the comparison operator "=", and therefore the comparison operator "≠" also, do both necessarily apply to tuples. However, the operators "<" and ">" do *not* apply. The reason is that tuples are fundamentally sets (sets of components), and such operators make no sense for sets.

In closing this section, let me draw your attention to Exercise 3.16 at the end of the chapter, which I strongly recommend you devote some thought to. Later chapters in the book will appeal to several of the matters raised by that exercise.

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 **Tutorial D**'s tuple types, TUPLE type generator, and tuple selectors, respectively. (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 in SQL—represent two different SQL rows. *Note:* The keyword ROW in an SQL row value constructor invocation is optional; in practice, it's almost always omitted.

Thanks to that left to right ordering, row components ("fields") in SQL can be, and indeed 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 20; logically speaking, however, we can't say that component is "the STATUS component," we can only say it's the *third* 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 (i.e., there's no "0-row").

As discussed in Chapter 2—recall the example involving the SQL row variable SRV—SQL also supports a row assignment operation.[3] 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' )
```

is logically equivalent to:

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

Note carefully in the expanded form of this example that the two individual comparisons in the WHERE clause are connected by OR, not AND.

---

[3] Strictly speaking, I shouldn't be talking about assignments of any kind in this chapter, because assignment has to do with variables and this chapter is concerned with values, not variables. But it's convenient to include at least this brief mention of SQL row assignment 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 expressions denoting comparands in the examples so far have been, specifically, row value constructor invocations. But now I need to explain that SQL has a syntax rule to the effect that if such an invocation consists of a single scalar expression enclosed in parentheses, then the parentheses can optionally be dropped, as here:

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

The "row comparison" in the WHERE clause in this example is thus 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 direct counterpart in the relational model, and in any case they're seriously error prone. (It's relevant to note 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 iterations 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:** Given a heading *H*, a *body B* conforming to *H* is a set of *m* tuples ($m \geq 0$), each with heading *H*. The value *m* is the *cardinality* of *B*. The pair (*H*,*B*)—call it *r*—is a *relation value* (or just a *relation* for short) over the attributes of *H*. *H* is the *relation heading* (or just heading for short) for *r*, and the degree and attributes of *H* and the cardinality of *B* are, respectively, the degree, attributes, and 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, interrelating a collection of *n* values (one such value for each tuple attribute); 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 set of tuples is 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 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, of course.) Also, every relation value is denoted 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[4] of the keyword RELATION, followed by a commalist enclosed in braces of tuple expressions (and those tuple expressions must all be of the same tuple type).

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 of them to the next chapter. In the next few sections, I'll address a series of possibly less weighty issues (?) arising from the definitions.

RELATIONS AND THEIR BODIES

The first point I want to discuss is this: *Every subset of a body is a body*—or, loosely, every subset of a relation is a relation. (Once again I mentioned this fact in Chapter 1, but now I want to say a little more about it.) In particular, 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 headings in informal contexts; throughout the rest of the book, in fact, I'll feel free to regard headings as either including or excluding the attribute type names—whichever best suits my purpose at the time):

SNO	PNO	QTY

[4] But see Exercise 3.15.

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 here:

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, it might be the expression RELATION {TUPLE {SNO 'S1', PNO 'P1', QTY 300}}, which is in fact a relation selector invocation (actually it's a literal). 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 (it's the row assignment example from the section "Row and Table Types in SQL" in Chapter 2):

```
SET SRV = ( 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 ∈ r
```

This expression returns TRUE if *t* appears in *r* and FALSE otherwise. The symbol "∈" (a stylized Greek epsilon) denotes the *set membership* operator; the expression $t \in r$ can be read as "*t* [is] in *r*" or "*t* appears 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).[5] Here's an example ("Get suppliers who supply at least one part"):

[5] Why exactly is the definite article correct here ("the" row)?

```
SELECT SNO , SNAME , STATUS , CITY
FROM   S
WHERE  SNO IN        /* "SNO" coerced to "ROW(SNO)" */
     ( SELECT SNO
       FROM   SP )
```

As I'm sure you know, SQL also supports NOT IN. The **Tutorial D** analog is "∉"; in other words, the **Tutorial D** expression "$t \notin r$" means tuple *t* isn't in relation *r*.

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.

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 (the four dimensions corresponding, of course, to the four attributes of that relation), and the relation overall can thus be said to be four-dimensional. Thus, relations in general are *n*-dimensional, not two-dimensional.[6] 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 *r1* and *r2* of the same relation type, we must at least be able to test whether they're equal. Here's an example, expressed in **Tutorial D** as usual, of an equality comparison on relations:

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

[6] 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.

The left comparand here is the projection of suppliers on {CITY},[7] 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: "The set of supplier cities is equal to the set of part cities" (and it evaluates to either TRUE or FALSE, of course).

Other comparisons might be useful, too. For example, we might want to test whether *r1 includes r2* (meaning every tuple in *r2* is also in *r1*), or whether *r1 properly* includes *r2* (meaning every tuple in *r2* is also in *r1* but *r1* contains at least one tuple that isn't in *r2*). Here's an example involving proper inclusion:

```
S { SNO } ⊃ SP { SNO }
```

The symbol "⊃" here means "properly includes" (or, equivalently, "is a proper superset of"). The meaning of this expression (considerably paraphrased) is: "At least one supplier supplies no parts at all" (which again necessarily evaluates to either TRUE or FALSE).

Other useful relational comparison operators include "⊇" ("includes," "is a superset of"), "⊆" ("is included in," "is a subset of"), and "⊂" ("is properly included in," "is a proper subset of"). *Note:* Of these various operators, the "⊆" operator in particular is usually referred to, a trifle arbitrarily, as *the* relational inclusion operator.

Finally, one extremely common requirement is to be able to perform an "=" comparison between some given relation *r* and an empty relation of the same type—in other words, a test to see whether *r* is empty. So it's convenient to define a shorthand:

```
IS_EMPTY ( r )
```

This expression is defined to return TRUE if relation *r* is empty and FALSE otherwise.[8] I'll be relying on it heavily in chapters to come (especially Chapter 8). The inverse operator can be useful too:

```
IS_NOT_EMPTY ( r )
```

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

---

[7] The **Tutorial D** expression *r*{*A*,*B*,…,*C*} denotes the projection of relation *r* on attributes *A*, *B*, …, *C*. See Chapter 6 for further discussion.

[8] In other words, the expression IS_EMPTY(*r*) is logically equivalent to both of the following: (a) *r* = *r* WHERE FALSE; (b) *r*{} = TABLE_DUM. *Note:* Regarding the second of these, see the section immediately following.

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 that tuple has an empty heading. For exactly the same reason, 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 always 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* can't 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 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 what makes them 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 relations and their meaning in much more detail 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 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 got them into a lot of trouble). 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.

> *Aside:* Perhaps I should say a little more about those pet names TABLE_DUM and TABLE_DEE. 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 and What Alice Found There* (1871). 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 them (the names, that is) for so long now in the relational world that I don't think we're going to change them. *End of aside*.

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 and CREATE VIEW create). 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 just a collection of rows, where (a) the rows are of a certain row type and (b) the collection is (in general) a bag, not necessarily a set. It follows that SQL doesn't really have anything analogous to the RELATION type generator, either—though as we know from Chapter 2 it does support other type generators, including ROW, ARRAY, and MULTISET. 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 sometimes have to be, 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' )
```

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 Fig. 1.3), we must:

1. Ensure that if the *i*th ordinal position, within any of the rows specified by the VALUES expression, corresponds to attribute *A* of the intended relational counterpart (viz., the suppliers relation, in the example), then the *i*th ordinal position in all of those rows corresponds to that same attribute *A*.

2. Ensure that all of the values in the *i*th ordinal position are values of the type appropriate for that attribute *A*.

3. Ensure that the same row isn't specified 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 Chapter 5. As for table comparisons, SQL has no direct support—not even for equality!—but workarounds are available. For example, here's an SQL counterpart to the **Tutorial D** 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 **Tutorial D** 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 )
```

Aside: I said above that SQL has no direct support for table equality comparisons, and that's true. As a consequence, the following putative SQL analog of the **Tutorial D** comparison S{CITY} = P{CITY}—

```
( SELECT DISTINCT CITY FROM S ) =
( SELECT DISTINCT CITY FROM P )
```

—is illegal. But the odd thing is, SQL does have direct support for equality comparisons on *bags*, including as a special case bags of rows in particular.[9] Moreover, it also has an

[9] Here's the pertinent quote from the standard: "Two [bags] *A* and *B* are distinct if there exists a value *V* in the element type of *A* and *B*, including the null value [*sic*], such that the number of elements in *A* that are not distinct from *V* does not equal the number of elements in *B* that are not distinct from *V*." I hope that's perfectly clear! Note that the extract quoted does indeed define what it means for two bags to be equal, because—simplifying considerably—if *A* and *B* aren't distinct in SQL terms, then they must be equal. *Note:* SQL also has direct support for equality testing on arrays.

operator for converting a table to a bag of rows.[10] So we can do the desired equality comparison by converting the tables to bags of rows and then comparing those bags. So far so good ... Believe it or not, however, the operator that converts a table to a bag of rows is called *TABLE* (!). Thus, the desired comparison can legitimately be formulated in SQL as follows:

```
TABLE ( SELECT DISTINCT CITY FROM S ) =
TABLE ( SELECT DISTINCT CITY FROM P )
```

But this trick only works for equality comparisons—SQL has no direct support for "⊃" etc., certainly not for tables, and not for bags of rows either.[11] *End of aside*.

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 attribute names are prohibited). In SQL, analogous 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.[12] **Strong recommendation:** Use AS specifications whenever necessary to give proper column names to columns that otherwise (a) wouldn't have a name at all or (b) would have a name that wasn't unique. Here are some examples:

1. ```
   SELECT DISTINCT SNAME , 'Supplier' AS TAG
   FROM   S
   ```

2. ```
   SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
   FROM   S
   ```

3. ```
   SELECT MAX ( WEIGHT ) AS MBW
   FROM   P
   WHERE  COLOR = 'Blue'
   ```

[10] Note carefully that (as mentioned in Chapter 2) a bag of rows in SQL isn't the same thing as an SQL table, because it can't be operated upon by means of SQL's regular table operators.

[11] What's more, the standard doesn't guarantee that the single column, in each of those two bags of rows resulting from the two TABLE invocations in the example, has any prescribed column name (see footnote 12, following). In particular, it doesn't guarantee that the column name in question is CITY. Of course, this fact is probably insignificant in the present context, but it could easily be very significant indeed in other contexts.

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

```
4.  CREATE VIEW SDS
      AS ( SELECT DISTINCT SNAME , 2 * STATUS AS DOUBLE_STATUS
           FROM   S ) ;

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

6.  SELECT temp.*
    FROM ( SELECT * 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 safely 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'
```

Perhaps more important, note that the recommendation unfortunately can't be followed at all in the case of tables specified by means of VALUES expressions. However, workarounds are available. For example, the following is legal:

```
SELECT temp.SNO , temp.SNAME , temp.STATUS , temp.CITY
FROM ( VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
              ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
              ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
              ( 'S4' , 'Clark' , 20 , 'London' ) ,
              ( 'S5' , 'Adams' , 30 , 'Athens' ) )
       AS temp ( SNO , SNAME , STATUS , CITY )
```

Explanation: I've enclosed the VALUES expression in parentheses (thereby making it a subquery), attached an AS specification, and specified column names as well as a "correlation name" in that AS specification (see Chapter 12; see also Example 6 above).

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, in SQL, 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 our running

example, 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 an SQL table EMP with (among other things) columns representing "employee number" and "manager number," respectively, where a manager number is itself another employee number. Obviously, 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 the following join example (note the specification "ENO AS MNO" in the third line):

```
( 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,              */
/* and the AS specifications at the end of lines 1 and 3 are there */
/* because they're required by the SQL standard (see Chapter 12)   */
```

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—doubtless a common state of affairs in practice). A strategy you might want to consider in such circumstances is the following:

- For each table *T* in the database, define a view *V* that's identical to table *T* except possibly for some column renaming.
- Make sure all views so defined abide by the column naming discipline described above.
- Operate in terms of those views instead of the underlying tables.

Unfortunately, it's impossible to ignore the fact 100 percent that columns do have an ordinal position in SQL. (Of course, it's precisely because of that 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 (but probably aren't limited to):

- SELECT * (see Chapter 12)
- The FROM clause, if more than one table is specified (see Chapter 6)

- Explicit JOIN operations (see Chapter 6)
- UNION, INTERSECT, and EXCEPT operations, if CORRESPONDING isn't specified (see Chapter 6)
- The column name commalist, if specified, following the definition of a range variable (see Chapter 12)
- The column name commalist, if specified, in CREATE VIEW (see Chapter 9)
- INSERT, if no column name commalist is specified (see Chapter 5)
- VALUES expressions as described in the present chapter
- Row assignments and comparisons, also as described in the present chapter
- ALL and ANY comparisons, if the comparands are of degree greater than one (see Chapter 11)

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, and tuple and relation 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. I also 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

3.1 Define as precisely as you can the terms *attribute*, *body*, *cardinality*, *degree*, *heading*, *relation*, *relation type*, and *tuple*.

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

3.3 Write **Tutorial D** tuple selector invocations for a typical tuple from (a) the parts relvar, (b) the shipments relvar. Also show SQL counterparts to those selector invocations.

3.4 Write a typical **Tutorial D** relation selector invocation. Also show an SQL counterpart to that selector invocation.

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.

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 (TVA), (b) a tuple with a relation valued attribute (RVA).

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.

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

3.9 As we saw in the body of the chapter, 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?

3.10 What if any is the logical difference—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 ) )
```

3.11 What exactly does the following SQL expression mean?

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

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

3.13 List as many situations as you can think of in which SQL regards left to right column ordering as significant.

3.14 Give an SQL analog for the **Tutorial D** expression IS_NOT_EMPTY(*r*).

3.15 I said in the body of the chapter that a relation selector invocation in **Tutorial D** consists of the keyword RELATION, followed by a commalist enclosed in braces of tuple expressions (and those tuple expressions must all be of the same tuple type)—and I implied, though I didn't actually say as much, that the type of the relation denoted by the overall expression was RELATION *H*, where TUPLE *H* was the common type of all of the specified tuple expressions. But what if the set of specified tuple expressions is empty?—in other words, what if the relation being specified is empty? How can its type be determined?

Following on from the foregoing, how can we specify an empty table in SQL?

3.16 A tuple is a set (a set of components); so do you think it might make sense to define versions of the usual set operators (union, intersection, etc.) that apply to tuples?

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

3.18 The column naming discipline referred to in the previous exercise relies on the use of AS specifications. But such specifications can appear in SQL in many different contexts; moreover, the syntax sometimes takes the form "*X* AS <*something*>" and sometimes "<*something*> AS *X*" (if you see what I mean); and the keyword is sometimes optional and sometimes mandatory.[13] List all of the contexts in which AS can appear, showing which are of the form "*X* AS ..." and which of the form "... AS *X*", and in which cases the keyword is optional.

ANSWERS

3.1 See the body of the chapter.

3.2 Two values of any kind are equal if and only if they're the very same value (meaning they must be of the same type, a fortiori). In particular, therefore, (a) two tuples *t* and *t'* are equal if and only if they have the same attributes *A1*, *A2*, ..., *An* and, for all *i* (*i* = 1, 2, ..., *n*), the value *v* of *Ai* in *t* is equal to the value *v'* of *Ai* in *t'*; (b) two relations *r* and *r'* are equal if and only if they

[13] 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 specification" if there isn't any AS.

have the same heading and the same body (i.e., their headings are equal and their bodies are equal).

3.3 **Tutorial D** tuple selector invocations (actually literals):

```
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, rather, to use the official SQL term) and the reliance on left to right ordering in these SQL expressions. *Note:* The keyword ROW could be omitted in both cases without changing the semantics.

3.4 The following selector invocation (actually a literal) 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, involving two "row value constructor" invocations):

```
VALUES ROW ( 'S1' , 'P1' , 300 ) ,
       ROW ( 'S1' , 'P2' , 200 )
```

Again the keyword ROW could be omitted in both cases without changing the semantics. By the way, the fact that there are no parentheses enclosing the commalist of 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 the answer to Exercise 3.10 below.

3.5 The list that follows is based on one in my book *An Introduction to Database Systems* (see Appendix G).

- 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 (unlike attributes) 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 in SQL always have 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 in SQL are allowed to include nulls, which relations most certainly aren't.

- Tables (in the sense of tabular pictures) are "flat" or two-dimensional, but relations are *n*-dimensional.

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 this: 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's a **Tutorial D** expression denoting a tuple with a tuple valued attribute (TVA) called ADDR:

```
TUPLE { NAME 'Superman' ,
        ADDR TUPLE { STREET '1600 Pennsylvania Ave. ' ,
                     CITY 'Washington' , STATE 'DC' , ZIP '20500' } }
```

And here's a **Tutorial D** expression denoting a tuple with a relation valued attribute (RVA) called PNO_REL:

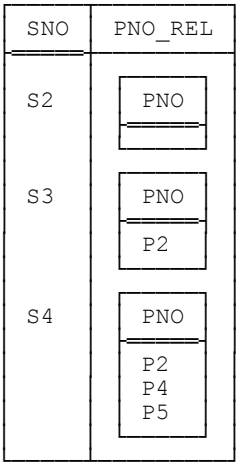
```
TUPLE { SNO 'S2' ,
        PNO_REL RELATION { TUPLE { PNO 'P1' } ,
                           TUPLE { PNO 'P2' } } }
```

3.7 For a relation with one RVA, see relation R4 in Fig. 2.2 in Chapter 2; for an equivalent relation with no RVA, see relation R1 in Fig. 2.1 in Chapter 2. As for one with two RVAs, consider the table on the left below. The intended meaning is: *Course CNO can be taught by every teacher TNO in TEACHER (and no other teachers) and uses every textbook XNO in TEXT (and no other textbooks).* The table on the right represents a relation without RVAs that conveys the same information.

CNO	TEACHER	TEXT
C1	TNO: T2, T4, T5	XNO: X1, X2
C2	TNO: T4	XNO: X2, X4, X5

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 Fig. 2.2 in Chapter 2 by just replacing the PNO_REL value for (say) supplier S2 by an empty relation:



Subsidiary exercise: Why exactly is there no relation without an RVA that represents the same information as the relation just shown?

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, refer to the discussion of the SIBLING example in Chapter 7.)

Note: 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.[14] With reference to relations R4 in Fig. 2.2 in Chapter 2 and R1 in Fig. 2.1 in Chapter 2, for example, we have the following (as will in fact be noted again in Chapter 7):

```
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 in particular.

---

[14] 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). What's more, this notion can readily be extended to sets of relations, thus: Let *s1* and *s2* be sets of relations. Then *s1* and *s2* represent the same information if and only if, for any query *q1* that can be addressed to the relations in *s1*, there's a corresponding query *q2* that can be addressed to the relations in *s2* that produces the same result (and vice versa). For further discussion, see Exercise 9.13 in Chapter 9.

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.) As for why this is so, your guess is as good as mine.

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.

3.10 The first denotes an SQL table of four rows (three distinct ones, plus a duplicate of one of those three). 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 is named in either case.

3.11 The given expression is semantically equivalent to this one:

```
SELECT SNO
FROM   S
WHERE  STATUS > 20
OR    ( STATUS = 20 AND SNO > 'S4' )
OR     STATUS IS NULL
OR     SNO IS NULL
```

3.12 See the body of the chapter.

3.13 See the body of the chapter.

3.14 EXISTS (*t*), where *t* is the SQL analog of the relational expression *r*. *Note:* Another possibility is (SELECT COUNT(*) FROM (*t*)) > 0; however, this possibility is slightly deprecated, for reasons to be explained in Chapter 10.

3.15 The complete syntax for a relation selector invocation in **Tutorial D** is as follows:

```
RELATION [ <heading> ] { <tuple exp commalist> }
```

And the syntax for *<heading>* is as explained in the body of the chapter. Moreover, there's a syntax rule to the effect that a *<heading>* must be specified if the *<tuple exp commalist>* is empty (it can be omitted otherwise). By way of example, therefore, the empty suppliers relation can be specified as follows:

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

As an aside, I note that TABLE_DEE and TABLE_DUM can be thought of as shorthand for the relation selector invocations RELATION { } { TUPLE { } } and RELATION { } { }, respectively.

As for SQL: The SQL analog of a relation selector invocation is a VALUES expression. The syntax is:

```
VALUES <row exp commalist>
```

As you can see, there's nothing here analogous to the optional *<heading>* component of a **Tutorial D** selector invocation. As a consequence, the *<row exp commalist>* mustn't be empty, and SQL has no direct way of specifying an empty table. Thus, workarounds are needed. For example, the empty suppliers table might be specified as follows:

```
( SELECT * FROM S WHERE FALSE )
```

3.16 Yes! However, we would of course want such operators always to produce a valid tuple as a result (i.e., we would want closure for such operations, just as we have closure for relational operations). For tuple union, for example, we would want the input tuples to be such that attributes with the same name have the same value (and are therefore of the same type, a fortiori). By way of example, let *t1* and *t2* be a supplier tuple and a shipment tuple, respectively, and let *t1* and *t2* have the same SNO value. Then the union of *t1* and *t2*, *t1* UNION *t2*, is a tuple of type TUPLE {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR, PNO CHAR, QTY INTEGER}, with components as in *t1* or *t2* or both (as applicable). E.g., if *t1* is (S1,Smith, 20,London) and *t2* is (S1,P1,300)—to adopt an obvious shorthand notation for tuples—then their union is the tuple (S1,Smith,20,London,P1,300). *Note:* This operation might reasonably be called tuple join instead of tuple union.

Of course, it's not just the usual set operators that might reasonably be adapted to tuples specifically—the same goes for certain of the well known relational operators, too. A particularly important example is provided by the tuple projection operator, which is a straightforward adaptation of the relational projection operator. For example, let *t* be a supplier tuple; then the projection *t*{SNO,CITY} of *t* on attributes {SNO,CITY} is that subtuple of *t* that contains just the SNO and CITY components from *t*. Likewise, *t*{CITY} is that subtuple of *t* that contains just the CITY component from *t*, and *t*{ } is that subtuple of *t* that contains no components at all (in other words, it's the 0-tuple). In fact, it's worth noting explicitly that *every* tuple has a projection on the empty set of attributes whose value is, precisely, the 0-tuple.

3.17 See the body of the chapter.

3.18 AS is used in SELECT clauses (to introduce column names); CREATE VIEW (ditto); FROM clauses (to introduce range variable names—by contrast, the syntax used to introduce *column* names in this context doesn't use AS); WITH specifications; and many 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 specification took the form "<*something*> AS *name*"; and (c) in which cases it took the form "*name* AS <*something*>": *No answer provided.*

Chapter 4

No Duplicates, No Nulls

I haven't even mentioned yet the way the silly notions
Discussed so far interreact and lead us into oceans
Of complication and despond and general distress.
Are two nulls equal (duplicates)? I fear, both NO and YES.

—Anon.:
Where Bugs Go

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 this chapter, therefore, I'll use the terminology of SQL rather than that of the relational model (for the most part, at any rate).

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.[1] 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] One reviewer felt strongly that an even more powerful practical argument (in fact, the most practical argument 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 in any case duplicates are much more than just a design issue. Thus, what I'm trying to do here is show the problems duplicates can cause, regardless of whether 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 G).

1. I assume you know that relational DBMSs include a component called the *optimizer*,[2] 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* performs better than *exp1* (at least, we hope so). *Note:* Be aware, however, that the term *query rewrite* is also used in certain commercial products with a different, typically more limited 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 if duplicate rows are allowed. By way of example, consider the (nonrelational) database shown in Fig. 4.1. Note right away that the tables in that database have no keys (and hence no primary keys a fortiori, which is why there's no double underlining in the figure). And by the way: If you're thinking the database is totally unrealistic—and especially if you're thinking that because of that fact you're not going to be convinced by the arguments that follow—I politely request that you suspend judgment until you've seen the further discussion of this example at the beginning of the next section, "Duplicates: Further Issues."

P

PNO	PNAME
P1	Screw
P1	Screw
P1	Screw
P2	Screw

SP

SNO	PNO
S1	P1
S1	P1
S1	P2

Fig. 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.[3]

---

[2] Here's as good a place as any to stress the point that—contrary to common practice in the industry, perhaps— my use of the unqualified term "optimization" (and related terms) in this book always refers to something the DBMS is responsible for, not something the user has to do. In other words, I'm *not* talking about what's sometimes called "manual optimization."

[3] 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. Given that duplicates do have some meaning, therefore, 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?

> *Aside:* In fact the foregoing paragraph touches on another point: namely, that duplicates violate one of the original objectives of the relational model. The objective in question is *explicitness*; that is, the meaning of the data in the database should be as explicit and obvious as possible (since databases are supposed to be suitable for sharing among a wide variety of disparate users and applications). As we've just seen, however, the presence of duplicates strongly suggests that part of the meaning of that data is not explicit but hidden. In fact, duplicates can be regarded as violating one of the most fundamental relational principles of all: viz., *The Information Principle* (to be discussed in Appendix A). *End of aside*.

Now consider the following query on the database of Fig. 4.1: "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 result 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 * 3.

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

Result: P1 * 1, 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.

Aside: 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. *End of aside.*

The first point to notice, then, 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 12 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 combine 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 that (as noted in Chapter 1) the relational model was expressly intended 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; however, 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.

DUPLICATES: FURTHER ISSUES

There's much, much more that could be said regarding duplicates and what's wrong with them, but in this section I'll limit myself to just three further points. The first has to do with the fact that in practice (as I mentioned earlier) base tables, at least, almost never do contain duplicate rows, and hence that the example in the previous section might reasonably be regarded as unrealistic. Well, all right; 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. By way of illustration, let's see what happens to that example from the previous section if we revise the database to make the base tables duplicate free, as in Fig. 4.2 (thanks to a reader of the previous edition, Ed Hynes, for drawing this example to my attention):

P

PNO	PNAME
P1	Screw
P1	Nut
P1	Bolt
P2	Screw

SP

SNO	PNO
S1	P1
S2	P1
S1	P2

Fig. 4.2: A relational database, without duplicates

Now, in the previous section I showed twelve different formulations of the query "Get part numbers for parts that either are screws or are supplied by supplier S1, or both" against the database of Fig. 4.1. Well, here are the results produced by those same twelve formulations against the revised version of the database in Fig. 4.2 (*Exercise:* Check these!):

1. P1 * 3, P2 * 1
2. P1 * 2, P2 * 1
3. P1 * 5, P2 * 3
4. P1 * 6, P2 * 2
5. P1 * 2, P2 * 2
6. P1 * 2, P2 * 2
7. P1 * 2, P2 * 2
8. P1 * 1, P2 * 1
9. P1 * 1, P2 * 1
10. P1 * 3, P2 * 1
11. P1 * 5, P2 * 3
12. P1 * 1, P2 * 1

As you can see, the twelve formulations still produce several different results (results, that is, that differ with respect to their degree of duplication). As I claimed above, therefore, it's clear that even if the input tables themselves don't contain any duplicates, different formulations of the same query can produce results with different degrees of duplication, and optimization is thus still inhibited. So the message is: Making sure that base tables never contain any duplicate rows is *necessary but not sufficient* to avoid duplicate rows entirely.

At the risk of beating a dead horse, I'd like to pursue this point just a moment longer and consider a much simpler example (I didn't lead with this example because it's almost too simple, a fact that can make it easy to miss the real significance of what's going on). Here are two possible formulations of the query "Get supplier numbers for suppliers who supply at least one part" on our usual suppliers-and-parts database (and note that this time the input tables most definitely don't contain any duplicates):

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

```
SELECT SNO
FROM   S NATURAL JOIN SP
```

At least one of these expressions—which?—will produce a result with duplicates, in general. (*Exercise:* Given our usual sample data values, what results do the two expressions produce?)

So what do we conclude from examples like the ones above and the one discussed in the previous section? Well, what I'd *like* to conclude is that you should abide by the following suggestions (and if you do, you can then just forget about the duplicates problem entirely):

- First, never allow duplicates in base tables (by always specifying at least one key—see Chapter 5).
- Second, ensure that query results never contain duplicates (for example, by always specifying DISTINCT in your SQL queries).

Unfortunately, however, life is never quite as simple as we might like, and the second of these suggestions, at least, needs more discussion and explanation. But let me leave it at that for now; I'll come back and revisit it in the next section ("Avoiding Duplicates in SQL").

I turn now to my second point. The fact is, 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 discussed 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 final 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, suppose the person responsible for data entry unintentionally enters the very same row twice—e.g., by inadvertently hitting the return key twice (easily done, by the way). Then there's no straightforward way to delete the "second" row without deleting the "first" as well. 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 we've seen, 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[4]

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 (and as previously noted), 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 (*Database in Depth,* O'Reilly Media Inc., 2005):

> 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.[5] 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 ...

---

[4] See the section "Summarization" in Chapter 7 for an explanation of why I generally set the SQL phrase "set function" in quotation marks.

[5] The implication is that SELECT DISTINCT might take longer to execute than SELECT ALL, even if that 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). This latter issue is explored in depth in a paper by Hugh Darwen, "The Role of Functional Dependence in Query Decomposition" (see Appendix G). *Note:* Let me add as a point of history that the explicit ALL in SELECT ALL wasn't added to SQL until long after the language was first defined. So the original default wasn't really SELECT ALL but just SELECT, making it that much easier to fall into the various traps that duplicates can cause—as well as suggesting, albeit subtly and psychologically, that by specifying something extra (DISTINCT), you were explicitly asking the system to do something extra, thereby inevitably getting worse performance.

> 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 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). *Note:* That same state of affairs is also the source of the term *three-valued logic* (3VL). That is, the notion of nulls, as understood in SQL, inevitably leads to 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—also known as *connectives*—NOT, AND, and OR (using T, U, and F to stand for TRUE, UNKNOWN, and FALSE, respectively):

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—can 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 Fig. 4.3, in which "the CITY is null" for part P1. Note carefully that the shading 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	

Fig. 4.3: A nonrelational database, with a null

Consider now the following (admittedly rather contrived) query on the database of Fig. 4.3: "Get (SNO,PNO) pairs where either the supplier and part cities are different or the part city isn't Paris, or both." Here's the obvious SQL formulation of this query:

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

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

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

(I've added some parentheses for clarity.) For the only data we have, this expression evaluates to UNKNOWN OR UNKNOWN, which reduces to just UNKNOWN. Now, queries in SQL retrieve data for which the expression in the WHERE clause evaluates to TRUE, not to FALSE and not to UNKNOWN;[6] in the example, therefore, nothing is retrieved at all.

But part P1 does have some corresponding city in the real world;[7] in other words, "the null CITY" for part P1 does stand for some real value, say *c*. Now, either *c* 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 *c* isn't Paris, then the expression becomes (again, for the only data we have)

```
( 'London' <> c ) OR ( c <> '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 therefore return the pair (S1,P1), *regardless of what real world 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!

By way of another example, consider the following query on that same table P from Fig. 4.3 (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 Fig. 4.3). SQL, however, will return no part numbers at all.

---

[6] A more accurate statement is: If the boolean expression in a WHERE clause evaluates to UNKNOWN, that UNKNOWN gets coerced to FALSE. Incidentally, it's interesting to note that, by contrast, if the boolean expression in a CHECK clause—i.e., in a CREATE ASSERTION statement (see Chapter 8) —evaluates to UNKNOWN, that UNKNOWN gets coerced not to FALSE but to TRUE! This state of affairs (this inconsistency, rather) might reasonably be regarded as yet another nail in the nulls coffin. See the answer to Exercise 8.21g in Chapter 8 for further discussion.

[7] I'm relying here on the fact that (as noted earlier) the intended interpretation of null is *value unknown*, from which it follows that the fact that "the CITY is null" for part P1 means part P1 does have some city, but we don't know what it is. (In fact, if part P1 had no city at all—i.e., if the property of having a city didn't apply to part P1—then that part shouldn't have been mentioned in the table in the first place. See the discussion of *relvar predicates* in Chapter 5.)

To sum up: If you have any nulls in your database, you're getting wrong answers to certain of your queries. What's more, you have no way of knowing, of course, 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, in SQL. (This is just one of the numerous flaws in SQL's 3VL support; there are many, many others, but most of them are beyond the scope of this book.) To elaborate briefly: As with 2VL, the SQL type BOOLEAN contains just two values, TRUE and FALSE; "the third truth value" is represented, quite incorrectly, by null! Here are some consequences of this fact:

- Assigning UNKNOWN to a variable B of type BOOLEAN actually sets B to null.
- After such an assignment, the comparison B = 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 B has the value UNKNOWN). Note, incidentally, that this state of affairs constitutes a clear violation of *The Assignment Principle* (see Exercise 2.22 in Chapter 2, also Chapter 5).
- In fact, the comparison B = UNKNOWN *always* gives null (meaning UNKNOWN), regardless of the value of B, because it's logically equivalent to the comparison "B = NULL" (not meant to be valid SQL syntax).

To understand the seriousness of such flaws, you might care to meditate on the analogy of a numeric type using null instead of zero to represent zero. *End of aside.*

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 brief look at 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 (like duplicates) violate one of the most fundamental relational principles of all—viz., *The Information Principle*. Once again, see Appendix A for further discussion of that principle.

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, either 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" SUM, MAX, MIN (etc.) all return null if their argument is empty (except for COUNT and COUNT(*), which correctly return zero in such a situation).

- 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. But it would take us much too far afield to get into the detailed implications of *that* state of affairs here.

- Outer joins and "union joins" are expressly designed to produce nulls in their result.[8]

- If the ELSE option is omitted from a CASE expression, an ELSE option of the form ELSE NULL is assumed.

- The expression NULLIF(x,y) returns null if $x = y$ evaluates to TRUE.

- The "referential triggered actions" ON DELETE SET NULL and ON UPDATE SET NULL can both generate nulls (obviously enough).

---

[8] SQL's UNION JOIN operator, which was a flawed attempt to support an already flawed operator called outer union, was introduced in SQL:1992 and dropped again in SQL:2003.

Strong recommendations:

- Do specify NOT NULL, at least 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 or logical equivalent).
- Don't use the keyword UNKNOWN in any context whatsoever.
- Don't omit the ELSE option from a CASE expression unless you're certain it would never have been 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 [NOT] DISTINCT FROM. (If *a* and *b* are both nonnull, then *a* IS NOT DISTINCT FROM *b* reduces to *a* = *b*, and *a* IS DISTINCT FROM *b* reduces to *a* <> *b*.)
- 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 logical equivalences fail to hold only if nulls are present (the symbol "≡" means *is equivalent to*):

```
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 "evaluate to null" without it. (Apologies for the quotation marks here, but the fact is that the phrase "evaluates to null" is a solecism.)

Note: In case you're not familiar with COALESCE, let me elaborate briefly on the last of these recommendations. Essentially, COALESCE is an operator that lets you 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 *a*, *b*, ..., *c* be scalar expressions. Then the expression COALESCE (*a*,*b*,...,*c*) returns null if its arguments are all null, or the value of its first nonnull argument otherwise. Of course, to use it sensibly, you do need to make sure at least one of *a*, *b*, ..., *c* is nonnull! Here's a fairly realistic example:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL SP.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, like supplier S5 in our usual running example—then the COALESCE invocation will replace that null by a zero. (Incidentally, this example also illustrates a situation in which use of ALL instead of DISTINCT isn't just acceptable but is logically required, though it might be implicit. See Chapter 7.) Given our usual sample data, the query thus 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 two 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 row for supplier S5 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 explicit SQL outer join operator in conjunction with COALESCE, as here:

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

Note: I said "there's no reason" why the padding shouldn't be done with proper values, as in the foregoing example, but that phraseology is really a little too glib. The example worked—sort of—because attribute PNO is of type CHAR, and so we could pad with a character string value ('nil' in the example). But what if it had been of some numeric type, say INTEGER? Or, worse, some user defined type? And even in the simple character string case, an argument could be made that the result misrepresents the semantics of the situation (does 'nil' truly represent a part number?). The truth is, padding with a real value instead of null just tends to hide the fact that outer join, no matter what code is used to achieve it, is simply not a respectable operation.[9] Much better to avoid it altogether.

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:

[9] It's not respectable because the result has no proper predicate. Again, see the discussion of such matters in Chapter 5, also (especially) Appendix C.

> All of those examples you give where nulls lead to wrong answers 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 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 software glitches of other kinds, even in the absence of nulls; in my opinion, nulls can only serve to make such failures much 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 those 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 that those "certain cases" might include practical, real world situations; rather, it's up to those who want to defend nulls to prove that they don't. And I venture to suggest that in practice 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 a little too complex, in general, to be discussed in detail here. The SQL mechanism of (nonnull) default values can be used in simple cases; but for a more comprehensive approach to the problem—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 refer you to Appendix C.

EXERCISES

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

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 with optimization, among other things) that states that (*r* WHERE *bx*) UNION (*r* WHERE *by*) ≡ *r* WHERE *bx* OR *by*. If *r* isn't a relation but an SQL table with duplicates, does this law still apply?

4.3 Let *a*, *b*, and *c* be sets. Then *the distributive law of intersection over union* (also used in relational systems to help with optimization among other things) 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?

4.4 Part of the explanation of the FROM clause (in a SELECT – FROM – WHERE expression) in the 1992 version of the standard read as follows:

> [The] result of the <from clause> is the ... cartesian product of the tables identified by [the specifications 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!—notwithstanding the fact that the standard did go on to say that "The cardinality of *CP* is the product of the cardinalities of the identified tables." For example, let tables T1 and T2 be as shown here:

T1

C1
0
0

T2

C2
1
2

Observe now that all of the following fit the foregoing definition for "the" cartesian product *CP* of T1 and T2 (that is, any of them could be "the" multiset referred to):

CP1

C1	C2
0	1
0	1
0	1
0	2

CP2

C1	C2
0	1
0	1
0	2
0	2

CP3

C1	C2
0	1
0	2
0	2
0	2

Can you fix up the wording of the standard appropriately?

4.5 Consider the following SQL 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 (permit duplicates, that is). 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?

4.6 Please write out one googol times: There's no such thing as a duplicate. *Note:* A *googol* is one followed by 100 zeros (i.e., 10 to the hundredth power). A *googolplex* is one followed by a googol zeros (i.e., 10 to the "googolth" power).

4.7 Do you think nulls occur naturally in the real world?

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?)

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 Exercise 4.10 below). Another useful monadic connective is MAYBE,[10] with truth table as follows:

p	MAYBE *p*
T	F
U	T
F	F

Does SQL support this connective?

4.10 Following on from the previous exercise, how many distinct connectives are there altogether in 2VL? What about 3VL? What do you conclude from your answers to these questions?

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 without it would be like an arithmetic without support for certain operations, say "+". Is classical 2VL truth functionally complete? What about SQL's 3VL?

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?

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?

---

[10] Useful, that is, if we buy into the notion that 3VL as such is useful, which of course I don't.

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?

4.15 The following is a legitimate SQL row value constructor invocation: ROW (1,NULL). Is the row it denotes null or nonnull?

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?

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?

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 this 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? What are the implications?

4.19 Suppose the suppliers-and-parts database permits nulls; in particular, suppose columns SP.SNO and SP.PNO permit nulls.[11] 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? Is the following formulation equivalent?

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

4.20 Let *k1* and *k2* be values of the same type. In SQL, then, what exactly does each of the following mean?

---

[11] If {SNO,PNO} is the primary key for shipments, then columns SP.SNO and SP.PNO couldn't permit nulls without violating the entity integrity rule. So in case such a possibility bothers you (it doesn't bother me, because I don't believe in that rule anyway), let me change the example slightly; let me introduce another column, SHIPNO (shipment number), into the shipments table, and let me make {SHIPNO} the primary key. Then {SNO,PNO} will still be a key, but it won't be the *primary* key, and the entity integrity rule therefore won't apply. (Incidentally, the very fact that the entity integrity rule is supposed to apply only to primary keys and not to keys in general seems to me to be another reason to regard that rule with suspicion. Not to mention the fact that it's also supposed to apply only to base tables and not to tables in general, which I think makes it even more suspect still.)

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.

4.21 In the body of the chapter, I said UNION ALL can generate duplicates. But what about INTERSECT ALL and EXCEPT ALL?

4.22 Are the recommendations "Always specify DISTINCT" and "Never specify ALL" duplicates of each other?

4.23 If TABLE_DEE corresponds to TRUE (or *yes*) and TABLE_DUM to FALSE (or *no*), then what corresponds to UNKNOWN (or *maybe*)?

4.24 The following quotes are taken from the SQL standard:[12]

- "The data type boolean comprises the distinct truth values True and False. Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value Unknown as the null value. This [standard] does not make a distinction between the null value of the boolean data type and the truth value Unknown ... [They] may be used interchangeably to mean exactly the same thing."

- "All boolean values and SQL truth values are comparable ... The value True is greater than the value False, and any comparison involving the null value or an Unknown truth value will return an Unknown result."

Do you have any comments on these quotes? In particular, which of the following (if any) do you think are legal SQL expressions? And what do they return, if they're legal?

a. `TRUE OR FALSE`

b. `TRUE OR UNKNOWN`

c. `TRUE OR NULL`

d. `TRUE > FALSE`

e. `TRUE > UNKNOWN`

f. `TRUE > NULL`

[12] Note that the standard uses True, Unknown, and False in prose discussions but TRUE, UNKNOWN, and FALSE in its SQL grammar.

4.25 In his book *Using the New DB2* (Morgan Kaufmann, 1996), in a section titled "A Brief History of SQL," Don Chamberlin—who is widely acknowledged to be "the father of SQL"—has the following to say (I'm quoting the text more or less verbatim, except that I've added some italics):

> During the early development of SQL ... some decisions were made that were ultimately to generate a great deal [of] controversy ... Chief among these were the decisions to support null values [*sic*] and to permit duplicate rows ... I will [briefly examine] the reasons for these decisions ... My purpose here is historical rather than persuasive ... *I recognize that nulls and duplicates are religious topics*, and I do not expect anyone to have a conversion experience after reading this chapter.

Do you agree with Chamberlin that nulls and duplicates are "religious topics"?

ANSWERS

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![13] Now, it might be true for certain purposes that the two entities can be *interchanged*, but that fact isn't sufficient to make them indiscernible. (Indeed, there's a logical difference between interchangeability and indiscernibility, 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 G).

4.2 Before we can answer this question, we need to pin down exactly what WHERE and UNION mean in the presence of duplicates. The paper "The Theory of Bags: An Investigative Tutorial" (see Appendix G) goes into details on such matters; 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 *T* 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:

[13] Note that one way of distinguishing them might simply be by position—*a* is *here* and *b* is *over there*.

```
SELECT C
FROM   T
WHERE  TRUE
OR     TRUE
```

Result: *v* * 2.

```
SELECT C
FROM   T
WHERE  TRUE
UNION  DISTINCT
SELECT C
FROM   T
WHERE  TRUE
```

Result: *v* * 1.

```
SELECT C
FROM   T
WHERE  TRUE
UNION  ALL
SELECT C
FROM   T
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?

4.3 Remarks similar to those in the answer to the previous exercise apply here also. Again I'll skip the details; I'll just say for the record that, first, the answer depends, of course, on what definitions we adopt for UNION and INTERSECT for bags as opposed to sets; second, with the SQL definitions, the law *doesn't* apply. I'll leave development of a counterexample to you.

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.

Note: As I'm sure you observed, the quoted text was taken from the 1992 version of the standard. Later versions use different wording, and they do manage to fix the problem—but

they do so, in effect, by adopting my solution of attaching a unique identifier to each row. I rest my case.

4.5 I don't think this problem can be fixed.

4.6 *No answer provided!*

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, concerning some specific employee Joe, "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.

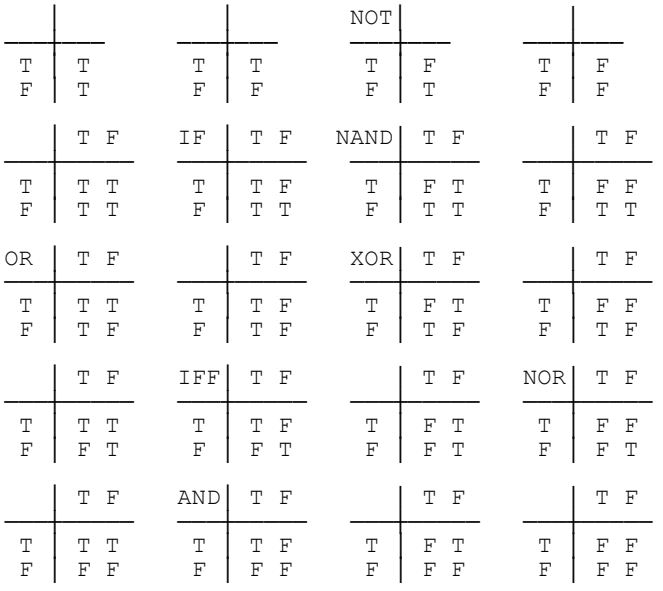
Suppose you had to represent the fact that you don't know Joe's salary in some box on some paper form. Would you enter a null, as such, into that form? I don't think so! Rather, you would leave the box blank, or put a question mark, or write "unknown," or something along those lines. And that blank, 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."

4.8 True (though not in SQL!). Null is a marker that represents the absence of information, while UNKNOWN is a value, just as TRUE and FALSE are values. So there's a logical difference between the two, and to confuse them as SQL does is a logical mistake. (I'd like to say it's a big logical mistake, but all logical mistakes are big mistakes by definition.)

4.9 Yes, it does; SQL's analog of "MAYBE *p*" is "*p* IS UNKNOWN".

4.10 In 2VL there are exactly 4 monadic connectives and exactly 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):[14]

[14] Note that the dyadic tables are shown here in a style slightly different from that used in the body of the chapter. Both styles are acceptable, but (as will be noted again in Chapter 10) sometimes one style is more convenient, sometimes the other is.



In 3VL, by contrast, 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 in particular those who think nulls are a good thing, realize, or at least are prepared to admit to).

4.11 Classical 2VL supports (among other things) NOT, AND, and OR and is thus truth functionally complete, because all possible 2VL connectives can be expressed in terms of NOT and either AND or OR (see the answer to Exercise 10.4 in Chapter 10 for further explanation). And it turns out that SQL's 3VL—under an extremely charitable interpretation of that term!—is truth functionally complete as well. The paper "Is SQL's Three-Valued Logic Truth Functionally Complete?" (see Appendix G) discusses this issue in detail.

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; note too the potential

for serious mistakes (on the part of both users and the system, I might add—and there's some history here). 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 necessarily the case.

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.

4.14 In 3VL (at least as realized in SQL), *r* JOIN *r* isn't necessarily equal to *r,* and INTERSECT isn't a special case of JOIN. Why so? Because in SQL, believe it or not, two nulls don't "compare equal" for join but do "compare equal" for intersection. (I take this state of affairs to be just another 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.

4.15 Here are the rules: Let *x* be an SQL row. Suppose for definiteness and 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. For the given row, both of these expressions evaluate to FALSE, and it follows that the row in question is neither null nor nonnull ... What do you conclude from this state of affairs?

By the way: At least one reviewer commented at this point 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 does to scalars. And as this exercise demonstrates, SQL tries to support this position—at least for rows—but fails. (Of course, it ought logically to support it for tables, too, but in that case it doesn't even try. I mean, there's no such thing as a "null table" in SQL.)

4.16 No. Here are the truth tables:

NOT	
T	F
U	U
F	T

IS NOT TRUE	
T	F
U	T
F	T

4.17 No. For definiteness, consider the case in which *x* is an SQL row. Suppose (as in the answer to Exercise 4.15 above) 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?

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 again there's some history here).

4.19 The query means "Get suppliers who are known not to supply part P2" (note that phrase "known not," and note also 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 SP row for part number P2 in table SP has a null supplier number).

4.20 No two of a., b., c. 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 (!). Here for the record are the rules in question:

- 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 (see Chapter 11 for further explanation).

- In SQL, *k1* and *k2* are duplicates if and only if either (a) they're both 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.

4.21 The output from INTERSECT ALL and EXCEPT ALL can indeed contain duplicates, but only if duplicates are present in the input; unlike UNION ALL, therefore, these two operators never "generate" duplicates.

4.22 Yes! (We don't want duplicates in the database, but that doesn't mean we never want duplicates anywhere else. As I said in the body of the chapter, there's a logical difference between logic and rhetoric.) By the way, here's another nice illustration of essentially the same point that I came across only recently: *Good food is not cheap. Cheap food is not good*.

4.23 A very good question.

4.24 Well, I don't know about you, but I have quite a few comments myself!

- First of all, the phrase "the null value" would be better reduced to just "null" throughout.

- Second, observe that (as noted in Chapter 4) although SQL supports three-valued logic, its BOOLEAN data type has just two values, TRUE and FALSE; "the third truth value" is represented not by a value at all but by null. This state of affairs explains (?) the distinction drawn in the second quote between "boolean values" and "SQL truth values" —as far as SQL is concerned, there are three truth values (TRUE, FALSE, and UNKNOWN) but only two boolean values (TRUE and FALSE).

- Next: "This [standard] does not make a distinction between the null value of the boolean data type and the truth value Unknown ... [They] may be used interchangeably to mean exactly the same thing." But, of course, null doesn't *always* mean "the third truth value," so null and "the truth value Unknown" certainly can't be used "interchangeably" as claimed. In fact, the keyword NULL usually can't be used in place of the keyword UNKNOWN even when UNKNOWN is the sense intended (see c. and f. in the answer to the last part of the exercise below).

- The phrase "the null value of the boolean data type" is also rather strange, since there's just a single null and that null, since it isn't a value, actually has no type at all.

- "Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value Unknown ...": NOT NULL doesn't apply to data types, it applies to *uses* of data types (typically as part of a column definition).

- Formal systems (like SQL) in which the truth values are ordered usually define that ordering to be total. In particular, for three-valued logic, the ordering would typically be such that the comparisons TRUE > UNKNOWN and UNKNOWN > FALSE both return TRUE. SQL, however, defines any comparison involving UNKNOWN (even UNKNOWN = UNKNOWN) to return UNKNOWN.

- Following on from the previous point: TRUE > UNKNOWN and UNKNOWN > FALSE (etc.) are apparently legal SQL expressions—but they're not, according to the standard, legal "boolean value expressions" (despite the fact that they do return a boolean value ... or perhaps I should say, despite the fact that they return "an SQL truth value").

Finally, the six SQL expressions (or would-be expressions):

a. Legal; returns TRUE.

b. Legal; returns null (UNKNOWN).

c. Illegal.

d. Legal; returns TRUE.

e. Legal; returns null (UNKNOWN).

f. Illegal.

4.25 *No answer provided.*

Chapter 5

Base Relvars, Base Tables

Said a young mathematician named Gene
"I always say what I mean—
Or mean what I say—
It's the same, anyway—
Or—at least—well, you know what I mean."

—Anon.:
Where Bugs Go

By now you should be very 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—as you know, it uses the same term, *table*, to mean sometimes a table value and sometimes a table variable. For example, the keyword TABLE in CREATE TABLE clearly refers to a table variable; but when we say, e.g., that 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 type (more precisely, 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 (see Exercise 1.5 in Chapter 1).

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

The topics I'll be covering in the present chapter form somewhat of a mixed bag, but generally speaking they fall into the following broad categories:

- Updating (i.e., relational assignment)
- Candidate and foreign keys
- Predicates

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 and indeed throughout the rest of the book):

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

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

VAR SP BASE RELATION
  { SNO    CHAR ,
    PNO    CHAR ,
    QTY    INTEGER }
  KEY { SNO , PNO }
  FOREIGN KEY { SNO }
          REFERENCES S
  FOREIGN KEY { PNO }
          REFERENCES P ;
```

```
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 ) ) ;
```

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:

a. All that such talk really means is just that the set of tuples we're updating happens to have cardinality one.

b. 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 "tuple level 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[1] (and please note that 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

[1] For another argument against updating through cursors, see Exercise 4.5 in Chapter 4.

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); I mean, we say things like "update the city for supplier S1 from London to New York"; but it's like that business of user friendly terminology I discussed in Chapter 1—it's OK to talk this way only if all parties involved understand that such talk is only an approximation to the truth, and indeed that it tends to obscure the essence of what's really going on.

Triggered Actions

The fact that updating is set level implies among other things 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—mustn't be done until all of the explicitly requested updating has been done. In other words, a set level update must *not* be treated as a sequence of individual tuple level updates (or row level updates, in SQL). SQL, however, unfortunately does treat set level updates as a sequence of row level ones, at least 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 = 'S1' :
     { CITY := 'New York' } ;
```

```
UPDATE S
SET    CITY = 'New York'
WHERE  SNO = 'S1' ;
```

Constraint Checking

The fact that updating is set level has another implication too: namely, that integrity constraint checking also mustn't be done until all of the updating (including triggered actions, if any) has been done. (The UPDATE discussed earlier, involving a change to the city for suppliers S1 and S4, illustrates this point very clearly. See Chapter 8 for further discussion.) Again, therefore, 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 regard (see the subsection immediately preceding). In any case, even if the standard does conform, that's not to say all commercial products do;[2] thus, you should still be on your lookout for violations in this connection.

---

[2] There's at least one product that doesn't (at least, not 100 percent), because it does what it calls *inflight checking*. See Chapter 8 for further discussion.

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 at all (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 such descriptions are (as I said before) strictly incorrect, and 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 a relvar reference (where a relvar reference is basically just the pertinent relvar name). Here's a **Tutorial D** example:

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

Now, it's easy to see that this particular assignment is logically equivalent to the following DELETE statement:

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

More generally, the **Tutorial D** DELETE statement

```
DELETE R WHERE bx ;
```

(where *R* is a relvar reference and *bx* is a boolean expression) is shorthand for, and hence logically equivalent to, the following relational assignment:

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

Alternatively, we might say it's shorthand for this one (either way, it comes to the same thing):

```
R := R MINUS ( R WHERE bx ) ;
```

Turning to INSERT, the **Tutorial D** INSERT statement

```
INSERT R rx ;
```

(where *R* is again a relvar reference and *rx* is a relational expression—typically but not necessarily a relation selector invocation) is shorthand for:

```
R := R UNION rx ;
```

For example, the INSERT statement

```
INSERT SP RELATION { TUPLE { SNO 'S5' , PNO 'P6' , QTY 700 } } ;
```

effectively inserts a single tuple into the shipments relvar SP.

Finally, the **Tutorial D** UPDATE statement also corresponds to a certain relational assignment. However, the details are a little more complicated in this case than they are for INSERT and DELETE, and for that reason I'll defer them to Chapter 7 (specifically, to the discussion of "what if" queries in that chapter).

D_INSERT and I_DELETE

I've said the INSERT statement

```
INSERT R rx ;
```

is shorthand for:

```
R := R UNION rx ;
```

Observe now, however, that this definition implies that an attempt to insert "a tuple that already exists" (i.e., an INSERT in which the relations denoted by *R* and *rx* aren't disjoint) will succeed. (It won't insert any duplicate tuples, of course—it just won't have any effect, at least as far as the tuples in question are concerned.) For that reason, **Tutorial D** additionally supports an operator called D_INSERT ("disjoint INSERT"), with syntax as follows:

```
D_INSERT R rx ;
```

This statement is shorthand for:

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

D_UNION here stands for *disjoint union*. Disjoint union is just like regular union, except that its operand relations are required to have no tuples in common (see Chapter 6). It follows that an attempt to use D_INSERT to insert a tuple that already exists—more generally, an attempt to use D_INSERT when the relations denoted by *R* and *rx* aren't disjoint—will fail.

What about DELETE? Well, observe first that the DELETE syntax shown above—

```
DELETE R WHERE bx ;
```

—is actually just a special case (though it's far and away the commonest case in practice). The more general form parallels the syntax of INSERT:

```
DELETE R rx ;
```

Here *R* is a relvar reference and *rx* is a relational expression (perhaps just a relation selector invocation).[3] This more general form of DELETE is defined to be shorthand for:

```
R := R MINUS rx ;
```

For example, the DELETE statement

```
DELETE SP RELATION { TUPLE { SNO 'S1' , PNO 'P1' , QTY 300 } } ;
```

effectively deletes a single tuple from the shipments relvar SP.

It should be clear, however, that the foregoing definition implies that an attempt to delete "a tuple that doesn't exist" (i.e., a DELETE in which the relation denoted by *rx* isn't wholly included in the relation denoted by *R*) will succeed. For that reason, **Tutorial D** additionally supports an operator called I_DELETE ("included DELETE"), with syntax as follows:

```
I_DELETE R rx ;
```

This statement is shorthand for:

```
R := R I_MINUS rx ;
```

I_MINUS here stands for *included minus*; the expression *r1* I_MINUS *r2* is defined to be the same as *r1* MINUS *r2* (see Chapter 6), except that every tuple appearing in *r2* must also appear in *r1*—in other words, *r2* must be included in *r1*. It follows that an attempt to use I_DELETE to delete a tuple that doesn't exist—more generally, an attempt to use D_INSERT when the relation denoted by *rx* isn't wholly included in the relation denoted by *R*—will fail.

Note: Now that I've introduced D_INSERT and I_DELETE, please understand that discussions elsewhere in this book that refer to INSERT and DELETE operations in **Tutorial D** should be taken for simplicity as applying to D_INSERT and I_DELETE operations as well, wherever the sense demands it.

Table Assignment in SQL

SQL has nothing directly comparable to **Tutorial D**'s D_INSERT and I_DELETE. Apart from this difference, however, SQL's support for INSERT, DELETE, and UPDATE operations

---

[3] The common special case DELETE *R* WHERE *bx* can be thought of as shorthand for DELETE *R* (*R* WHERE *bx*).

resembles that of **Tutorial D** fairly closely and there's little more to be said, except for a few points regarding INSERT specifically:

- First, the source for an SQL INSERT operation is specified by means of a table expression (typically but not necessarily a VALUES expression—see Chapter 3). Contrary to popular opinion, therefore, INSERT in SQL really does insert a table, not a row, though that table (the *source table*) might and often will contain just one row, or even no rows at all.

- Second, INSERT in SQL is defined in terms of neither UNION nor D_UNION, but rather in terms of SQL's "UNION ALL" operator (see Chapter 6). As a consequence, an attempt to insert a row that already exists will fail if the target table is subject to a key constraint but will succeed (and will insert a duplicate row) otherwise.

- Third, INSERT in SQL supports an option according to which the target table reference can be followed by a parenthesized column name commalist, identifying the columns into which values are to be inserted; the *i*th target column corresponds to the *i*th column of the source table. Omitting this option is equivalent to specifying all of the columns of the target table, in the left to right order in which they appear within that table.
  Recommendation: Never omit 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.[4]

  Here's another example (incidentally, this one makes it very clear that INSERT really does insert a table and not a row):

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

As for relational assignment: Unfortunately SQL doesn't have a direct counterpart to this operator. The closest it can get to the generic assignment

```
R := rx ;
```

[4] Even though this tactic—i.e., specifying the option—does fix the problem at hand, observe that it does still involve left to right column ordering and thus is still somewhat nonrelational in spirit. As Hugh Darwen once remarked to me (in a private communication): "The syntax of a language should in all places be *in the spirit of* that language. Then it's easier to learn, because people get to know what to expect. A proper relational language attaches no significance to column ordering. Not *anywhere*."

is this pair of statements, executed in sequence:

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

(*T* and *tx* here are the SQL analogs of *R* and *rx*, respectively.) Note in particular that (as pointed out in the answer to Exercise 1.16 in Chapter 1) this sequence of statements could fail where its relational counterpart, the relational assignment, would succeed—for example, if table *T* is subject to the constraint that it must never be empty.

The Assignment Principle

I'd like to 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 (see Exercise 2.22 in Chapter 2). *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, as I'm sure you realize, it's more or less the definition of the assignment operation. I'll have more to say about it in Chapter 8 (at least by implication) when I discuss an extended form of the assignment operation known as *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, or of, *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—let's call it SC—{SNO,CITY}, 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 SC value). But it doesn't have the irreducibility property, because we could discard the CITY attribute and what's left, the singleton set {SNO}, would still have the uniqueness property. So we don't regard SC as a key, because it's "too big" (i.e., it's reducible). 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 SC was a key for relvar S. Then the DBMS wouldn't enforce the constraint that supplier numbers are "globally" unique; instead, it would enforce only the weaker constraint that supplier numbers are "locally" unique, in the sense that they're unique within the pertinent city. So this is one reason—not the only one—why we require keys not to contain 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 is reducible. (By the way, you might think this recommendation rather obvious, but I've certainly seen it violated in practice; in fact, I've even seen such violations explicitly recommended, by writers who really ought to know better.)

Now, all of the relvars we've seen in this book so far have had just one key. Here by contrast are several self-explanatory examples (in **Tutorial D** only, for brevity) of relvars with two or more keys. Note the overlapping nature of those keys in the second and third examples. *Note:* I assume the availability of certain user defined types in these definitions.

```
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 persons 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 a tiny syntactic sleight of hand here. A key is a set of attributes, and an attribute is 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 various points 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.[5] (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 sometimes said to be alternate keys). But whether some key is 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. Certainly it isn't inviolable.

> *Aside:* **Tutorial D** in fact has no syntax for distinguishing between primary and alternate keys, supporting as it does just simple KEY specifications. SQL, by contrast, supports explicit PRIMARY KEY specifications in addition to the UNIQUE specifications I've been showing (in CREATE TABLE statements) prior to this point. A given base table can have any number of UNIQUE specifications but at most one PRIMARY KEY specification. However, a PRIMARY KEY specification is essentially equivalent to a UNIQUE specification, except that it implicitly and additionally causes a NOT NULL constraint to be defined for every column in the key in question. In my examples I'll continue to use UNIQUE specifications exclusively, just to be definite. *End of aside*.

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—i.e., the entire heading—of *R* certainly has the uniqueness property. Thus, either that combination also has the irreducibility property, or there's some proper subset of that combination that does. Either way, there's always 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:** In SQL, for base tables at any rate, use UNIQUE (and/or PRIMARY KEY) 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 the tuple for supplier S1 is:

```
TUPLE { SNO 'S1' }
```

---

[5] 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" the key in question—though such a manner of speaking is likely to cause confusion, and I wouldn't recommend it.

(a subtuple of the tuple for that supplier—recall 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. And so now it should be clear just how keys, like so many other things in the relational model, rely crucially on the concept of *tuple equality*. To spell the point out: In order to enforce some key uniqueness constraint, we need to be able to tell whether two key values are equal, and that's precisely a matter of testing two tuples for equality—even when, as in the case of relvar S, the tuples in question are of degree one and thus "look like" simple scalar values.

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 for *R* that isn't a key for *R* is called a *proper* superkey for *R*). 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 dependency*.[6] I don't want to get into a lot of detail regarding that concept here—I'll come back to it in Chapter 8—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 *X* be any subset of the heading of *R*. Then the functional dependency (FD)

```
SK → X
```

holds in *R*, necessarily. To elaborate briefly: In general, the functional dependency $SK \rightarrow X$ means that whenever two tuples of *R* have the same value for *SK*, they also have the same value for *X*. 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 *X*. In other words, loosely: We always have functional dependency arrows "out of superkeys" (and therefore out of keys in particular) to everything else in the relvar.

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 supposed to correspond to values of some key—the *target key*—in some other relvar (or possibly in 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 key {SNO} in S and values of the 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.")

---

[6] Also known as functional *dependence*. The terms *dependence* and *dependency* are used interchangeably in the literature (and in this book), in contexts such as the one at hand.

Here now is a more precise definition:

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 (i.e., are of the same type). 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*); *K* (not *K′*) is the corresponding *referenced key* (or *target key*); the associated constraint is a *referential constraint*; and *R2* and *R1* are the *referencing relvar* and the corresponding *referenced relvar* (or *target relvar*), respectively, for that constraint.

As an aside, I note that the relational model as originally formulated required foreign keys to correspond not just to some key, but very specifically to the primary key, of the referenced relvar. Since we don't insist on primary keys, however, we certainly can't insist that foreign keys correspond to 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 key—which we can regard, harmlessly, as the primary key, if we want to—in S and P, respectively. Here by way of contrast is a more complicated example:

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

```
CREATE TABLE EMP
  ( ENO VARCHAR(6) NOT NULL ,
    MNO VARCHAR(6) NOT NULL ,
     ..... ,
    UNIQUE ( ENO ) ,
    FOREIGN KEY ( MNO )
            REFERENCES EMP ( ENO ) ) ;
```

As you can see, there's a significant difference between the **Tutorial D** and SQL FOREIGN KEY specifications in this example. I'll explain the **Tutorial D** one first.

- Within a given tuple, 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. 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 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 is specified not just as EMP but rather as EMP{ENO} RENAME {ENO AS MNO}. *Note:* The RENAME operator is described in detail in the next chapter; for now, I'll just assume it's self-explanatory.

- Turning now to SQL: In SQL the key *K* in the referenced table *T1* and the corresponding foreign key *FK* in the referencing table *T2* are sequences, not sets, of columns. (In other words, key and foreign key values in SQL are rows, not tuples, and left to right column ordering is significant once again.) Let those columns, in sequence as defined within the FOREIGN KEY specification in the definition of table *T2*, be *B1*, *B2*, ..., *Bn* (for *FK*) and *A1*, *A2*, ..., *An* (for *K*), thus:[7]

```
FOREIGN KEY ( B1 , B2 , ..., Bn )
            REFERENCES T1 ( A1 , A2 , ..., An )
```

Then columns *Bi* and *Ai* ($1 \leq i \leq n$) must be of the same type—no coercions here—but they don't have to have the same name. That's why, in the example, the SQL specification

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

is sufficient 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 corresponding to some key of *T* itself (as in the EMP example)
- When some table *T2* has two distinct foreign keys, both corresponding to 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 5.16 and the answer to that exercise at the end of the chapter for further discussion.

---

[7] Columns *A1*, *A2*, ..., *An* must be the columns named in some UNIQUE or PRIMARY KEY specification in the definition of table *T1*, but they don't have to appear in that UNIQUE or PRIMARY KEY specification in the same sequence as they do in the FOREIGN KEY specification for table *T2*. Moreover, they, and the parentheses surrounding them, can be omitted entirely from this latter specification—but if so, then (a) they must appear in a PRIMARY KEY specification, not a UNIQUE specification, for table *T1*, and (b) they must appear in that specification in the appropriate sequence.

Referential Actions

As you probably know, SQL supports not just foreign keys as such but also certain associated *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 a FOREIGN KEY specification that looks like this:

```
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 certainly the foundation of the database field, but it's *only* the foundation. In other words, 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:

- *Type theory:* Type theory provides the most obvious example of such an "additional feature." We saw in Chapter 2 that "types are orthogonal to tables," but we also saw that full and proper type support in relational systems—including support for user defined types, and perhaps even support for type inheritance—is highly desirable, to say the least. (In my own opinion, in fact, a system without such support scarcely deserves the label "relational." See Appendix A for further discussion.)

- *Triggered procedures:* Strictly speaking, a triggered procedure is an action (the *triggered action*) to be carried out 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 triggered action is DELETE (actually the "procedure" in this particular case is specified declaratively), and the triggering event is ON DELETE.[8] 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 quite 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*.

---

[8] 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."

- *Recovery and concurrency:* By way of a third example, the relational model has almost nothing to say about recovery and concurrency controls, but this omission obviously doesn't mean that relational systems shouldn't provide such controls. (Actually it could be argued that the relational model does say something about such matters implicitly, because it 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 point that they're not 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 keys as well, but in the case of keys 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.

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 something like this:

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

This predicate is the *interpretation*, or *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 of course 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 something that's unequivocally either true or false. 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 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 at some given time a certain tuple plausibly could appear in some relvar but 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 Appendixes A and C for more on this important notion).

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're probably 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 "comparison predicates," "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 gives 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 different and rather 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. 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. Note too that when I say 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 design goals are different at that level. As I explained in Chapter 1, 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 commercial 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. (As a matter of fact, at least one of the products I have in mind here already has failed.) The products in question 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 of such products are beyond the scope of this book.

Here now is 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.[9] 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 theorems!

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?

[9] Or the relational calculus. Either way, it comes to the same thing (see Chapter 10).

- 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 formal logical treatment and can be given logical answers.

Moreover, 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 does so deserve. Certainly most of them are ad hoc to a degree, instead of being firmly founded, as the relational model is, on set theory and predicate logic. I'll expand on these issues in Appendix A.

EXERCISES

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.

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.

5.3 Why are SQL's "positioned update" operations a bad idea?

5.4 In **Tutorial D,** INSERT and D_INSERT are defined in terms of UNION and D_UNION, respectively, and DELETE and I_DELETE are defined in terms of MINUS and I_MINUS, respectively. In SQL, by contrast, INSERT is defined in terms of UNION ALL, and there's nothing analogous to D_INSERT. There's also nothing in SQL analogous to I_DELETE; but what about the regular SQL DELETE operator? How do you think that's defined?

5.5 Let the SQL base table SS have the same columns as table S. 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? *Note:* Thinking about **Tutorial D** analogs of the two statements might help you answer this question.

5.6 (*This is essentially a repeat of Exercise 2.22 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?

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

5.8 Why doesn't it make sense to say a relation has a key?

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

5.10 "Key values are not scalars but tuples." Explain this remark.

5.11 Let relvar *R* be of degree *n*. What's the maximum number of keys *R* can have?

5.12 What's the difference between a key and a superkey? And given that the superkey concept apparently makes sense, do you think it would make sense to define any kind of *subkey* concept?

5.13 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 relvars lead inevitably to a requirement for null support? (*Answer:* No, they don't, but they do serve to show how seductive the nulls idea can be.) What can be done in the example if nulls are prohibited?

5.14 Why doesn't SQL have anything analogous to **Tutorial D**'s renaming option in its foreign key specifications?

5.15 Can you think of a situation in which two relvars *R1* and *R2* might each have a foreign key referencing the other? What are the implications of such a situation?

5.16 The well known *bill of materials* application involves a relvar—PP, say—showing which parts ("major" parts) contain which parts ("minor" parts) as immediate components, and showing also the corresponding quantities (e.g., "part P1 contains part P2 in quantity 4"). Of course, immediate components are themselves parts, and they can have further immediate components of their own. Give appropriate base relvar (**Tutorial D**) and base table (SQL) definitions. What referential actions do you think might make sense in this example?

5.17 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 the product doesn't support but might be useful?

5.18 Define the terms *proposition* and *predicate*. Give examples.

5.19 State the predicates for relvars P and SP from the suppliers-and-parts database.

5.20 What do you understand by the terms *intension* and *extension?*

5.21 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.

5.22 Explain *The Closed World Assumption* in your own terms. Could there be such a thing as *The Open World Assumption?*

5.23 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 pertinent set of attributes is empty. What are the implications? Can you think of any uses for such a key?

5.24 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?

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

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

5.27 In Chapter 1 I said I'd be indicating primary key attributes, in tabular pictures of relations, by double underlining. At that point, however, I hadn't discussed the logical difference between relations and relvars; and in this chapter we've seen that keys in general apply to relvars, not relations. Yet I've shown numerous tabular pictures in previous chapters that represent relations as such (I mean, relations that aren't just a sample value for some relvar), and I've certainly been using the double underlining convention in those pictures. So what can we say about that convention now?

ANSWERS

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 duplicate records; and no nulls; and no repeating groups; and no pointers; and no anonymous fields (and on and on). Partly as a consequence of these facts, it really is much better to think of a relvar like this: The heading represents some predicate (or some *intension*), and the body at any given time represents the *extension* of that predicate at that time.

5.2 Loosely, the specified remark means the UPDATE operation in question "updates 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 the new value 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.

5.3 Because relational operations are fundamentally set level and SQL's "positioned update" operations are necessarily tuple level (or row level, rather), by definition. 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.

5.4 It's defined in terms of EXCEPT ALL. Consider the SQL DELETE statement:

```
DELETE FROM T WHERE bx ;
```

Let *temp* denote the result of the expression SELECT * FROM *T* WHERE *bx*. Note that if row *r* appears exactly *n* times in *temp* ($n \geq 0$), it also appears exactly *n* times in *T*. Then the effect of the specified DELETE statement is to assign the result of the expression

```
SELECT * FROM T EXCEPT ALL SELECT * FROM temp
```

to table *T*. (Note that EXCEPT DISTINCT would have the additional effect of eliminating duplicates from *T* that don't appear in *temp*.)

5.5 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.

As for **Tutorial D** analogs of the two SQL statements: Well, note first of all that in order for such analogs even to exist, it's necessary to assume that table SS doesn't permit duplicates, because "relvars that permit duplicates" aren't supported in **Tutorial D** (in fact, they're a contradiction in terms). Under this assumption, however, a **Tutorial D** analog of the first statement is reasonably straightforward:

```
INSERT SS ( S WHERE SNO = 'S6' ) ;
```

As for the second statement, the closest we can get in **Tutorial D** is:

```
INSERT SS RELATION { TUPLE FROM ( S WHERE SNO = 'S6' ) } ;
```

Recall from Chapter 3 that the expression TUPLE FROM *rx* extracts the single tuple from the relation denoted by the relational expression *rx* (note that that relation must have cardinality one). So if relvar S does contain a (necessarily unique) tuple for supplier S6, the foregoing INSERT will behave more or less as its SQL counterpart. But if relvar S doesn't contain such a tuple, then the INSERT will fail (more precisely, the TUPLE FROM invocation will fail), whereas the SQL analog will as already noted insert a row of all nulls. *Subsidiary exercise:* Which behavior do you think is more reasonable (or more useful)?—**Tutorial D**'s or SQL's?

5.6 *The Assignment Principle* states that after assignment of the value *v* to the variable *V*, the comparison *V* = *v* must evaluate to 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 possibly certain user defined types as well. *Negative consequences:* Too many to list here.

5.7 As in the body of the chapter, I assume the availability of certain user defined types in the following definitions. For simplicity, I also choose to overlook the fact that some of the column names I've chosen (which?) are in fact reserved words in SQL.

```
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 ,
    TIME  TIME_OF_DAY NOT NULL ,
    GATE  GATE        NOT NULL ,
    PILOT NAME        NOT NULL ,
    UNIQUE ( DAY , TIME , GATE ) ,
    UNIQUE ( DAY , TIME , 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 ) ) ;
```

5.8 Because keys imply constraints; constraints apply to variables, not values; and relations are values, not variables. (That said, it's certainly possible, and sometimes useful, to think of some 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.)

5.9 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 the functional dependency $K \rightarrow X$ holds in relvar *A*. Suppose now that relvar *B* has a foreign key referencing that "reducible key" in *A*. Then the functional dependency $K \rightarrow X$ holds in *B* as well. As a result, the combination of *A* and *B* will involve some redundancy; in fact, the same will be true of *B* considered in isolation. Indeed, *B* might not even be in Boyce/Codd normal form.

> *Aside:* Details of Boyce/Codd normal form and other normal forms higher than 1NF are beyond the scope of this book. However, I'm sure you know something about them anyway, so I'll feel free to mention them from time to time without further apology. For

an indepth tutorial treatment of such topics, see the book *Database Design and Relational Theory: Normal Forms and All That Jazz* (O'Reilly, 2012), referenced in Appendix G. *End of aside*.

5.10 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'.

5.11 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*)!).[10] Relvars TAX_BRACKET and MARRIAGE—see the answer to Exercise 5.7 above—are examples of relvars with the maximum possible number of keys; so is any relvar of degree zero, which necessarily has just one key, necessarily an empty one. (If *n* = 0, the formula becomes 0!/(0!*0!), and 0! is 1.) See also the answers to Exercises 5.23 and 5.25.

5.12 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 *X* is a subset of *K*. (And *X* is a *proper* subkey for *R* if it's a subkey for *R* that's not a key for *R*.) 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*.) By way of illustration, here's a definition of third normal form that makes use of the subkey concept: Relvar *R* is in third normal form, 3NF, if and only if, for every nontrivial functional dependency $X \rightarrow Y$ to which *R* is subject, *X* is a superkey or *Y* is a subkey. (A nontrivial functional dependency is one for which the right side isn't a subset of the left side.)

5.13 First some sample data:

[10] Recall from Chapter 3 that the expression *n*! (which is read as either "*n* factorial" or "factorial *n*" and is often pronounced "*n* bang") is defined as the product *n* * (*n*−1) * ... * 2 * 1.

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 way 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	...

MANAGED_BY

ENO	...
E4	E2
E3	E2
E2	E1

Subsidiary exercise: What are the predicates for relvar MANAGED_BY and the two versions of relvar EMP here? Thinking carefully about this question should serve to reinforce the suggestion that the second design is preferable.

5.14 Because it doesn't need to, on account of the fact that column correspondences are established in SQL (in this context, at least, though not in all contexts) on the basis of ordinal position rather than name. See the discussion in the body of the chapter.

5.15 Note first that such a situation must represent a one to one relationship, by definition. One obvious case arises if we split some relvar "vertically," as in the following example (suppliers):

```
VAR SNT BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER }
  KEY { SNO }
  FOREIGN KEY { SNO } REFERENCES SC ;

VAR SC BASE RELATION
  { SNO CHAR , 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 a mechanism for defining two or more relvars at the same time as well. See the discussion of *multiple assignment* in Chapter 8.

5.16 **Tutorial D** definitions (I assume here that **Tutorial D** supports the self-explanatory 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 *p* will cascade to delete parts that are components of *p* but not parts of which *p* is a component.

SQL definitions:

```
CREATE TABLE P ( PNO ... , ... , UNIQUE ( PNO ) ) ;

CREATE TABLE PP ( MAJOR_PNO ... , MINOR_PNO ... , 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 ) ;
```

Regarding the specification ON DELETE RESTRICT here, see the answer to Exercise 5.17 below.

Note: In this example, the two foreign keys in table PP both refer to the same key in table P. Now, in the body of the chapter, I said that in such a case "you might want to ensure ... that one of the foreign keys has the same column names as [the target key], even though the other one doesn't (and can't)." As you can see, however, I haven't adopted my own suggestion in the case at hand; instead, I've opted for a more symmetric design, in which each of the foreign key columns has a name consisting of the corresponding target column name prefixed with a kind of *role* name (MAJOR_ and MINOR_, respectively).

5.17 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 do you think the difference is (if any) between NO ACTION and RESTRICT? Does it make sense? Is it useful?

5.18 Loosely, a predicate is a truth valued function, and a proposition is a predicate with an empty set of 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.

5.19 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.*

5.20 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. In other words, the intension corresponds to the heading and the extension to the body.

5.21 *No answer provided.*

5.22 *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. (Loosely speaking, in other words, *The Closed World Assumption* says tuple *t* appears in relvar *R* **if and only if** *t* satisfies the predicate for *R*, while *The Open World Assumption* says tuple *t* appears in relvar *R* **only if** *t* satisfies the predicate for *R*.)

What are the implications of the foregoing? Well, first let's agree to abbreviate *Closed World Assumption* and *Open World Assumption* to CWA and OWA, respectively. Now consider the query "Is supplier S6 under contract?" Of course, the system has no understanding of what it means for a "supplier" to be "under contract," and so we have to formulate the query a little more precisely, thus: "Does there exist a tuple for supplier S6 in relvar S?" Given our usual sample data values, the answer is *no*, and under the CWA that *no* is interpreted as meaning supplier S6 isn't under contract. Under the OWA, however, that same *no* is interpreted as meaning it's unknown whether supplier S6 is under contract. So far, so good (perhaps). But from this point on, things start getting murky quite fast ... As I've shown elsewhere—see my paper "The Closed World Assumption," mentioned in Appendix G—the fact that a certain tuple is missing from a certain relation sometimes has to be taken to mean that the corresponding proposition is false, not unknown, even under the OWA. So the OWA clearly raises problems of interpretation. Moreover, the very idea that a proposition might evaluate to unknown instead of true or false seems inevitably to lead to a need for three-valued logic and All That That Entails (see Chapter 4). Thus, my possibly somewhat conservative conclusion is that we should stay with the CWA, at least for the foreseeable future.

5.23 To say relvar *R* has an empty key is to say *R* can never contain more than one tuple. Why? Because every tuple has the same value for the empty set of attributes—namely, the empty

tuple (see the answer to Exercise 3.16 in Chapter 3); thus, if *R* had an empty key, and if *R* were to contain two or more tuples, we would have a key uniqueness violation on our hands. And, yes, 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.

5.24 See the answer to Exercise 5.18 above.

5.25 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*;[11] 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 when the value of Z is TABLE_DEE and FALSE when 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 Exercise 5.23 above).

5.26 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 the current value of relvar 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 don't necessarily mean a relation that's the value of some relvar.

5.27 There are two cases to consider: (a) The relation depicted is a sample value for some relvar *R*; (b) the relation depicted is a sample value for some relational expression *rx*, where *rx* is something other than a simple relvar reference (recall that a relvar reference is basically just

---

[11] 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.

the pertinent relvar name). In the first case, double underlining simply indicates that a primary key *PK* has been declared for *R* and the pertinent attribute is part of *PK*. In the second case, you can think of *rx* as the defining expression for some temporary relvar *R* (think of it as a view defining expression and *R* as the corresponding view, if you like); then double underlining indicates that a primary key *PK* could in principle be declared for *R* and the pertinent attribute is part of *PK*.

Note: See the answer to Exercise 7.23 in Chapter 7 for further discussion of this issue.

Chapter 6

SQL and Relational Algebra I:

The Original Operators

Join the union!

—Susan B. Anthony (1869)

This is the first of two chapters on the operators of the relational algebra; it discusses the original operators (i.e., the ones briefly described in Chapter 1) in some 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

I'll 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.[1] Second, recall too that the fact that the output is the same kind of thing as the input(s)—they're all relations—constitutes 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 there called "the original operators" (restrict, project, product, union, intersect, difference, and join); however, 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:

- The operators of the algebra are *generic*, meaning they apply (in effect) to *all possible relations*. For example, we don't need one specific join operator to join departments and employees and another, different, join operator to join suppliers and shipments. (Incidentally, do you think an analogous remark applies to object systems?).

[1] Actually, we'll be meeting some operators later in this chapter—*n*-adic join, q.v., is a case in point—that are allowed to take no relations at all as input, though they do still produce a relation as output.

- The operators are all *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.

- Of course, 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** (just so long as the relvars denoted by those names 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[2]—but in principle we could equally well denote the very same operand by means of an appropriate relation literal instead.[3]

  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 numeric expression, but at time *t* it means exactly the same as 3 + 2, no more and no less.

- 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 relational *algebra* operators as such—though, regrettably, you'll often come across statements to the contrary in database textbooks and elsewhere.

I also need to say something here about **Tutorial D** specifically, because it's in its support for the algebra in particular that the design of**Tutorial D** differs most significantly from that of SQL. The overriding point is this: In operations like UNION or JOIN that need some kind of correspondence to be established between attributes of their operands, **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 only attribute that P and S have in common (i.e., the only *common attribute*).

To repeat, **Tutorial D** establishes the correspondence between operand attributes, when such a correspondence is required, by insisting that the attributes in question in fact be one and the same. And it applies this same rule uniformly and consistently across the board, in all pertinent contexts. By contrast, SQL uses different rules in different contexts. Sometimes it uses

---

[2] This is true in the algebra but not necessarily in SQL. 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.

[3] Again, this is something that's true in the algebra but not necessarily in SQL. See the BNF grammar for SQL table expressions in Chapter 12.

ordinal position (we've already seen an example of this case in connection with foreign keys, as discussed in the previous chapter). Sometimes it uses explicit specifications (and such explicit specifications take different forms in different contexts). Sometimes it requires the attributes in question (or columns, rather) to have the same name—and then the correspondence is sometimes established explicitly, sometimes implicitly. And regardless of whether it requires the columns in question to have the same name, sometimes it requires those columns to be of the same type, and sometimes it doesn't. In order to illustrate a few of these possibilities, let's consider the P JOIN S example again. Here's one possible formulation of that join in SQL:

```
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
```

In this formulation, the required column correspondence is specified explicitly in the WHERE clause. As you probably know, however, examples like this one can in fact be formulated in several different ways in SQL.[4] Here are three more SQL formulations of the P JOIN S example (I've numbered them for purposes of subsequent reference; as you can see, formulations 2 and 3 are a little closer to the spirit of **Tutorial D**):[5]

1.
```
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
```

2.
```
SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , CITY ,
       S.SNO , S.SNAME , S.STATUS
FROM   P JOIN S
USING  ( CITY )
```

3.
```
SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , CITY ,
       S.SNO , S.SNAME , S.STATUS
FROM   P NATURAL JOIN S
```

Observe now that:

- In formulation 1, the column correspondence is again specified explicitly, but this time by means of an ON clause instead of a WHERE clause.

---

[4] The formulation just shown was the only one supported in SQL as originally defined. The other possibilities were added in SQL:1992.

[5] Here's a test of your SQL knowledge: For which of these three formulations do the corresponding columns (just the CITY columns, in the example) have to be of the same type?

- In formulation 2, the correspondence is based on common column names, but it's still specified explicitly by means of the USING clause.

- In formulation 3, the correspondence is again based on common column names, but this time it's implicit.

Here are some further points of difference between SQL and **Tutorial D**, most of them also arising from the difference in the languages' respective approaches to establishing attribute (or column) correspondence for the purposes of operators like join:

- SQL permits, and sometimes requires, dot qualified names. **Tutorial D** doesn't. (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 for other reasons 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 in the previous bullet item. (I'll be discussing SQL's correlation names in detail in Chapter 12.)

- As well as supporting (either explicitly or implicitly) 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 consequence of this difference is that SQL is a highly redundant language, in that it typically provides numerous different ways of formulating the same query, a fact that can have serious negative implications for both the user and the optimizer. (I once wrote a paper on this topic called "Fifty Ways to Quote Your Query"—see Appendix G—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 query formulations 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.[6] 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 said near the beginning of the previous section, closure is what makes it possible to write nested relational expressions, 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.

Now, when I say the result of every algebraic operation is another relation, I hope it's clear that I'm talking from a conceptual point of view; I don't mean the system always has to materialize those results in their 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 explicit dot qualifications 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 any given tuple of the join is formed, the system can test that tuple right away against the condition PNAME > SNAME (P.PNAME > S.SNAME in the SQL version) to see if it belongs in the final output, discarding it if not.[7] 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. (As an aside, I remark that the process by which tuples of an intermediate result are produced and passed on to another operation one at a time instead of en bloc is sometimes referred to as *pipelining*.)

The foregoing example raises another 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 that result has any such attributes? What *is* the heading of that result? More

---

[6] With one slight exception (?): Some writers regard relational inclusion ("⊆") as a relational operation—more specifically, as part of the relational algebra—even though it produces a result that's a truth value, not a relation. The point isn't very important, however; certainly it's not worth fighting over here.

[7] I assume for the sake of the example that the comparison PNAME > SNAME is a sensible one—though if it is, then attributes PNAME and SNAME must presumably represent "the same kind of information," and in accordance with my own recommendations in Chapter 3, therefore, I really ought to have given them the same name.

generally, what's the heading for the result of *any* algebraic operation? Clearly, what we need is a set of rules—to be specific, a set of *relation type inference rules*—such that if we know the headings (and therefore the types) of the input relations for an operation, we can infer the heading (and therefore the type) of the output relation from that operation. And the relational model does include such a set of rules. In the case at hand, for example, those rules say the output from P JOIN S is of this type:

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

In fact, for join, the heading of the output is simply 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, in that output. 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 (for example):

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

Note that type inference rules of some kind are definitely needed in order to support the closure property fully. After all, closure says every result is a relation, and relations have a heading as well as a body, so every result must have a proper relational heading as well as a proper relational body.

Now, the RENAME operator mentioned in the previous section is needed in large part because of the foregoing type inference rules; among other things, 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 relation *r* have an attribute called *A* and no attribute called *B*. Then (and only then) the expression *r* RENAME {*A* AS *B*} denotes an (attribute) *renaming* on *r*, and it returns the relation with heading identical to that of *r* except that attribute *A* in that heading is renamed *B*, and body identical to that of *r* (except that references to *A* in that body—more precisely, in tuples 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 (it's identical to our usual suppliers relation, except that the city attribute is called SCITY):

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 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, as we saw in Chapter 3, it doesn't really have a notion of "relation type" (or table type, rather) anyway. Second, it can produce results with columns that effectively have no name at all (for example, consider SELECT PNO, 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 columns that otherwise (a) wouldn't have a name at all or (b) would have a name that wasn't unique. 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. Here it is again:

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

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

As you can see, the counterpart in the SQL version to **Tutorial D**'s PNAME > SNAME is P.PNAME > S.SNAME (note the "P." and "S." qualifiers)—which is curious when you come to think about it, because that expression is supposed to apply to the result of the FROM clause (see the section "Evaluating SQL Expressions," later in this chapter), and tables P and S certainly aren't part of that result! Indeed, it's quite difficult to explain how references to the names P and S 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 indeed 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 (and only then) (a) *bx* is a *restriction condition* on *r*, (b) the expression *r* WHERE *bx* denotes the *restriction* of *r* according to *bx*, and (c) it returns the relation with heading the same as that of *r* and 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 expression of the form *r* WHERE *bx* where *bx* is a boolean 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*, of course, 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 operator (also with selectors as described in Chapter 2). SQL's SELECT operator—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 explicitly asked to do so). *Note:* UNGROUP and EXTEND are described in the next chapter.

PROJECTION

Definition: Let relation *r* have attributes called *A1*, *A2*, ..., *An* (and possibly others). Then (and only then) the expression *r*{*A1*,*A2*,...,*An*} denotes the *projection* of *r* on {*A1*, *A2*, ..., *An*}, and it returns the relation with heading consisting of attributes *A1*, *A2*, ..., *An* and body consisting of all tuples *t* such that there exists a tuple in *r* that has the same value for attributes *A1*, *A2*, ..., *An* as *t* does.

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.[8] The result heading has attributes (or columns) COLOR and CITY—in that left to right order, in SQL.

Let *r* be a relation. Then:

- The projection of *r* on all of its attributes just returns *r*. Such a projection is known as an *identity* projection. For example, the expression SP{SNO,QTY,PNO} denotes the identity projection of the relation that's the current value of relvar SP.

- The projection *r*{ }—in other words, the projection of *r* on no attributes at all—returns TABLE_DEE if *r* is nonempty, TABLE_DUM otherwise. Such a projection is sometimes called a *nullary* projection; however, the term *nullary* is best avoided because of the potential confusion with SQL-style nulls.

  Note: Just to remind you, TABLE_DEE is the unique relation with no attributes and just one tuple—the 0-tuple, of course—and TABLE_DUM is the unique relation with no attributes and no tuples at all. The fact that projecting *r* on no attributes always yields one of these two relations is a direct consequence of the fact that every tuple has the same value for the empty set of attributes (namely, the 0-tuple). See the answer to Exercise 3.16 in Chapter 3 if you need to refresh your memory regarding this point.

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 }
```

[8] I remark in passing out that the phrase "duplicate elimination," which is used almost universally (not just in SQL contexts), would more accurately be *duplication* elimination.

mean exactly the same thing. This feature can save a lot of writing (think of projecting a relation of degree 100 on 99 of its attributes).[9] Analogous remarks apply, wherever they make sense, to all of the operators in **Tutorial D**.

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

```
P JOIN S { CITY }
```

to mean

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

and not

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

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

JOIN

Before I get to the join operator as such, it's helpful 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 (meaning they are in fact the very same attribute)—equivalently, if and only if the set theory union of the headings of *r1* and *r2* is itself a legal heading. Note that this concept applies not just 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 (note how the definition appeals to the fact that tuples are sets and hence can be operated upon by set theory operators such as union):

> **Definition:** Let relations *r1* and *r2* be joinable. Then (and only then) the expression *r1* JOIN *r2* denotes the *natural join* (or just the *join* for short) of *r1* and *r2*, and it returns the relation with heading the set theory union of the headings of *r1* and *r2* and 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 dot qualifications from the SQL version where they aren't needed:

---

[9] A relvar (as opposed to a relation) of such a high degree is perhaps unlikely, since it would almost certainly be in violation of the principles of normalization. But such violations aren't exactly unknown in practice.

`P JOIN S`	`SELECT PNO , PNAME , COLOR , WEIGHT ,`
	`  P.CITY /* or S.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 a 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, rather—CITY, PNO, PNAME, COLOR, WEIGHT, SNO, SNAME, and STATUS (in that order in SQL, though not of course in the **Tutorial D** analog). *Note:* I'll have more to say on this SQL column ordering issue in the subsection "Explicit JOINs in SQL," later in this section.

There are several further 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 (i.e., have the same heading). For example, the following expressions are logically equivalent:

```
P { CITY } INTERSECT S { CITY }

P { CITY } JOIN S { CITY }
```

However, I'll have more to say about INTERSECT as such 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) as noted earlier, every possible 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 example, the following expressions are logically equivalent:

```
P { ALL BUT CITY } TIMES S { ALL BUT CITY }

P { ALL BUT CITY } JOIN S { ALL BUT CITY }
```

For completeness, however, I'll give the definition anyway:

> **Definition:** Let relations *r1* and *r2* have no attribute names in common. Then (and only then) the expression *r1* TIMES *r2* denotes the *cartesian product* (or just the *product* for short) of *r1* and *r2*, and it returns the relation with heading the set theory union of the

headings of *r1* and *r2* and 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 } )
  TIMES  /* or JOIN */
( S RENAME { CITY AS SCITY } )
```

```
SELECT PNO , PNAME , COLOR ,
       WEIGHT , P.CITY AS PCITY ,
       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 (in fact I've renamed them both, purely for reasons of symmetry). 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 usually thought of as a dyadic operator specifically; 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*.[10] 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 *r*, 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 once again 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 an empty set of numbers is 0*.[11] (To see this claim is reasonable, consider a piece of code that computes the sum of *n* numbers by initializing the sum to 0 and then iterating over those *n* numbers. What happens if $n = 0$?)

---

[10] Relations *r1*, *r2*, ..., *rn* must be joinable, of course (see Exercise 6.16). *Note:* For psychological reasons, **Tutorial D** also supports *n*-adic versions of INTERSECT and TIMES, but I'll skip the details here.

[11] As noted in Chapter 4, the SQL "set function" SUM yields null, not zero, if it's invoked on an empty set of numbers. But this is just a logical mistake on the part of SQL—it has no bearing on the present discussion.

- In like fashion, 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 an empty set of numbers is 1.

- In the relational algebra, *TABLE_DEE is the identity with respect to JOIN*; that is, for all relations *r*, the expressions *r* JOIN TABLE_DEE and TABLE_DEE JOIN *r* are both identically equal to *r* (see the paragraph immediately following). As a consequence, the join of an empty set of 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, denoted by table expressions *tx1* and *tx2*, say; *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. *Note:* Actually, Case 1 is logically identical to a Case 3 expression[12]—see below—in which the specified columns *C1*, *C2*, ..., *Cn* are *all* of the common columns (i.e., all of the columns that appear in both *t1* and *t2*), in the order in which they appear in *t1*.

[12] Except that the set of common columns can be empty in Case 1 but not in Case 3.

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* appear, in the order in which they appear in *t1*; then the other columns of *t2* appear, 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.[13]

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[14]—then make sure columns with the same name are of the same type, and make sure you do some appropriate renaming as well. For example:

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

   It's not really clear why you'd ever want to use such a formulation, however, given that it's logically equivalent to the following slightly less cumbersome one:

---

[13] Perhaps I should inject a small note of caution here. In practice, it's very common for SQL tables to have some kind of "comments" column; thus, there's a risk that NATURAL JOIN might produce unexpected results, unless some appropriate naming discipline is followed (or some appropriate renaming is done) in connection with such columns.

[14] Greater-than join is a special case of what's called θ-join, which I'll be discussing later in this chapter.

```
SELECT SNO , SNAME , STATUS , S.CITY AS SCITY ,
       PNO , PNAME , COLOR , WEIGHT , P.CITY AS PCITY
FROM   S , P
WHERE  S.CITY > P.CITY
```

3. In Case 3 (JOIN USING), make sure columns with the same name are of the same type.

4. In Case 4 (CROSS JOIN), make sure there aren't any common column names.

Recall finally that, as noted in Chapter 1, an explicit JOIN invocation isn't allowed in SQL as a "stand alone" table expression (i.e., one at the outermost level of nesting). Nor is it allowed as the table expression in parentheses that constitutes a subquery (see Chapter 12).

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 *T*. Then (and only then) the expression *r1* UNION *r2* denotes the *union* of *r1* and *r2*, and it returns the relation of type *T* with body the set of all tuples *t* such that *t* appears in at least one of *r1* and *r2*.

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

```
P { STATUS , CITY } UNION
S { CITY , STATUS }
```

```
SELECT STATUS , CITY
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 in order 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. As for the CORRESPONDING specification in the SQL formulation, that specification allows us to ignore the possibility that those columns might appear at different ordinal positions within the operand tables. **Recommendations:**

- Make sure every column of the first operand table has the same name and type as some column of the second operand table and vice versa.

  Aside: Here's another SQL question for you: Does SQL in fact allow corresponding columns in those operand tables to be of different types? *Answer:* Yes, it does. The standard's own definition of the result of *A* UNION *B* (not meant to be valid SQL syntax) runs something like this: Let *r* be a row that's a duplicate of both some row in *A* and some row in *B*. Then the result contains (a) exactly one duplicate of every such row *r* and (b) no row that's not a duplicate of some such row *r*. The reason for this somewhat convoluted definition is that two rows can be duplicates in SQL without being identical, owing to the fact that (as we saw in Chapter 2, section "Type Checking and Coercion in SQL") two scalar values in turn can "compare equal" without being identical.

  Note, incidentally, that the foregoing definition is still not complete, in that it fails to specify exactly which particular "duplicate of row *r*" actually appears in the result. *End of aside*.

- Always specify CORRESPONDING if possible.[15] If it isn't—in particular, if the SQL product you're using doesn't support it—then 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 left to right 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).[16]

  Note: This recommendation is perhaps a little debatable. At least the BY option might sometimes save keystrokes (though not always—see the example below). But it's misleading, because it means the union operands aren't the specified tables as such but certain projections of those tables; it's also unnecessary, because those projections could always be specified explicitly anyway. For example, the SQL expression

```
SELECT * FROM P
UNION  CORRESPONDING BY ( CITY )
SELECT * FROM S
```

  is logically equivalent to this (shorter!) one:

---

[15] I omitted CORRESPONDING from examples in earlier chapters because at the time it would only have been a distraction.

[16] In the interest of completeness, I note that omitting the BY option is actually equivalent to specifying BY (*C1*,*C2*,...,*Cn*), where *C1*, *C2*, ..., *Cn* are all of the common columns, in the left to right order in which they appear in the first operand table.

```
SELECT CITY FROM P
UNION
SELECT CITY FROM S
```

- 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.

Tutorial D also supports *n*-adic forms of both UNION and D_UNION. The syntax consists—with one small exception, explained below—of the operator name (i.e., UNION or D_UNION), followed by a commalist in braces of relational expressions *r1*, *r2*, ..., *rn*. The relations denoted by *r1*, *r2*, ..., *rn* must all be of the same type. For example, the foregoing D_UNION example could alternatively be expressed as follows:

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

Note: The union or disjoint union of a single relation *r* is just *r*. The union or disjoint union of no relations at all is the empty relation of the pertinent type—but that type needs to be specified explicitly, since there aren't any relational expressions from which the type can be inferred. Thus, for example, the expression

```
UNION { SNO CHAR , STATUS INTEGER } { }
```

denotes the empty relation of type RELATION {SNO CHAR, STATUS INTEGER}. Compare the answer to Exercise 3.15 in Chapter 3.

Intersection

Definition: Let relations *r1* and *r2* be of the same type *T*. Then (and only then) the expression *r1* INTERSECT *r2* denotes the *intersection* of *r1* and *r2*, and it returns the relation of type *T* with body the set of all tuples *t* such that *t* appears in each of *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 *T*. Then (and only then) the expression *r1* MINUS *r2* denotes the *difference* between *r1* and *r2* (in that order), and it returns the relation of type *T* with body the set of all tuples *t* such that *t* appears in *r1* and not in *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. Note, however, that minus is strictly dyadic—**Tutorial D** doesn't support any kind of "*n*-adic minus" operation (see Exercise 6.17 at the end of the chapter). But it does support "included minus" (I_MINUS), which is a version of minus that requires the second operand to be included in the first (i.e., the second operand mustn't contain any tuples that aren't also contained in the first operand). For example:

```
S { CITY } I_MINUS P { CITY }
```

Given our usual sample data, this expression will produce a run time error, because there's at least one part city that isn't also a supplier city. SQL has no direct counterpart to I_MINUS.

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 set {restrict, project, join, union, difference}; another can be obtained by replacing join in this set by product. *Note:* You might be surprised to see no mention here of rename. 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 discussion 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 ONE 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 (it would have been sufficient to do just one attribute renaming; once again I did two for symmetry). The purpose of the condition SA < SB is twofold:[17]

- 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 might otherwise be rather complicated expressions:

```
WITH ( t1 := ( S RENAME { SNO AS SA } ) { SA , CITY } ,
       t2 := ( S RENAME { SNO AS SB } ) { SB , CITY } ,
       t3 := t1 JOIN t2 ,
       t4 := t3 WHERE SA < SB ) :
t4 { SA, SB }
```

As the example suggests, a WITH specification in **Tutorial D** can appear as a prefix to a relational expression. Such a specification consists of the keyword WITH followed by a

[17] Note, incidentally, that the condition SA < SB wouldn't be legal if supplier numbers were of some user defined type (SNO, say) and the operator "<" hadn't been defined in connection with that type.

parenthesized commalist of assignments of the form *name* := *expression*, that parenthesized commalist then being followed in its entirety by a colon. For each of those "*name* := *expression*" assignments, the expression on the right side is evaluated and the result effectively assigned to the temporary variable whose name appears on the left side (in the example, I've used the names *t1*, *t2*, *t3*, *t4*—"*t*" for temporary). Also, the assignments are executed in sequence as written; as a consequence, any given assignment in the commalist is allowed to refer to names introduced in assignments earlier in that same commalist. Those introduced names can also be referenced in the relational expression that appears following the colon.

Tutorial D allows WITH specifications to appear on statements as well as on expressions. For example:

```
WITH ( temp := RELATION { TUPLE { SNO 'S5' , PNO 'P6' , QTY 250 } } ) :
SP := SP UNION temp ;
```

SQL too supports a WITH construct, with these differences:

- WITH in **Tutorial D** can be used at any level of nesting. By contrast, WITH in SQL can be used only at the outermost level.[18]
- WITH in **Tutorial D** can be used in connection with expressions of any kind.[19] By contrast, WITH in SQL can be used only in connection with table expressions specifically.
- SQL uses the keyword AS in place of **Tutorial D**'s assignment symbol (":=").
- SQL doesn't use the enclosing parentheses or colon separator.
- As already noted, **Tutorial D** allows WITH specifications on statements as well as expressions. SQL doesn't.

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

Here's an SQL version of the example:

---

[18] This particular limitation was added in SQL:2011; it didn't apply to SQL:1999, which is where WITH specifications were first introduced, nor to SQL:2003.

[19] Except that the expression in question mustn't be such that it relies on context for its evaluation; in other words, it must be what's called a closed expression. For example, "S WHERE STATUS = 20" is closed, but "STATUS = 20" isn't (it's open instead). Of course, a similar rule applies in SQL also.

```
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
```

Aside: It's worth noting, however, that SQL's WITH construct is not nearly as useful in practice as its **Tutorial D** counterpart, because neither the syntactic nor the semantic structure of SQL lends itself readily to the idea of breaking large expressions down into smaller ones. For example, as noted earlier in this chapter, the relational functionality of UNGROUP, EXTEND, RENAME, and projection is all bundled into just one clause in SQL (viz., the SELECT clause portion of an SQL SELECT expression). *End of aside*.

In closing this section, I should make it clear that WITH isn't really an operator of the relational algebra as such—it's just a syntactic device to help with the formulation of complicated expressions (especially ones involving common subexpressions). I'll be making extensive use of it in the pages ahead.

WHAT DO RELATIONAL EXPRESSIONS MEAN?

Recall now from Chapter 5 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 Chapter 5, 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 s , SNAME n , STATUS t }
```

such that a tuple of the form

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

currently appears in relvar S for some CITY value *c* (where *c* is a value of type CHAR, of course). In other words, the result represents the current extension of a predicate that looks like this:

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

This predicate can be understood as representing the meaning of the relational expression 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 discussion of bound variables and quantifiers).[20] *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 somewhere* [in other words, in some city, but we don't know which].

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 always be determined from the predicates for the relvars involved in that expression, together with the semantics of the 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 "≠"):

---

[20] 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 (see Appendix G).

```
( ( S RENAME { CITY AS SCITY } )
  TIMES
  ( P RENAME { CITY AS PCITY } ) )
WHERE SCITY ≠ PCITY
```

```
SELECT SNO , SNAME , STATUS ,
       S.CITY AS SCITY , PNO ,
       PNAME , COLOR , WEIGHT ,
       P.CITY AS PCITY
FROM   S , P
WHERE  S.CITY <> P.CITY
```

Now let's focus on the SQL formulation specifically. You can think of the expression constituting that formulation as being evaluated in three steps, as follows:

1. The FROM clause is evaluated, to yield the product of tables S and P. *Note:* If we were doing this relationally, we would have to rename at least one of 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 agree to overlook this detail.

2. Next, the WHERE clause is evaluated, to yield 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* rows in which the two city values are equal—in which case we would now have formed what's called the *equijoin* of suppliers and parts on cities. In other words, an equijoin is a θ-join for which θ is "=". *Exercise:* What's the difference between an equijoin and a natural join?

3. Finally, the SELECT clause is evaluated, to yield a "projection" of that restriction on the columns specified in the SELECT clause—"projection" in quotes, because it won't actually eliminate duplicates as true projection does, unless DISTINCT is specified. (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 for now I want to overlook 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 denotes 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; au contraire, it can use any algorithm it likes, just so long as whatever algorithm it does use is guaranteed to give the same result as the conceptual one. And there are often good reasons—usually performance reasons—for using a different algorithm, thereby (for example) 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. (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.[21])

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 1,000 suppliers and 1,000,000 shipments, of which 500 are for part P2. If the expression were simply evaluated by brute force, as it were (i.e., without any optimization at all), the sequence of events would be:

1. *Join S and SP:* This step involves reading the 1,000 supplier tuples; reading the 1,000,000 shipment tuples 1,000 times each, once for each of the 1,000 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 here for simplicity that tuples are physically stored as such on the disk, 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, in Step 1 I was assuming for the sake of the example—realistically or otherwise—that the 1,000,000 intermediate result tuples required too much space to be kept in main memory.)

---

[21] Cost based optimizing is beyond the scope of this book because it has to do with how the data is physically stored, which isn't a relational issue by definition. But I should at least note that such optimizing is possible in the first place only because (as we saw in Chapter 1) the relational model insists on there being a sharp and rigid distinction between the logical and physical levels of the system, which has the effect among other things of keeping access strategies out of applications.

3. *Project the result of Step 2:* This step involves no tuple reads or writes (i.e., to or from the disk) at all, so we can ignore it.

By contrast, 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 1,000 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 1,002,001,000 tuple reads and writes, whereas the second involves only 1,001,000; thus, it's clear the second procedure is likely to be over 1,000 times faster than the first. It's also clear 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, loosely, that a join followed by a restriction can always be transformed into a restriction followed by a join (this 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 intersection can be transformed into one involving join instead), but there's much more to the issue than that, as I hope is obvious from the example.

Now, there are many possible transformation laws, and this isn't the place for an exhaustive discussion. But I would at least like to highlight a few important cases and make a few 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, restriction distributes over union, intersection, and difference. It also distributes over join, provided the restriction condition consists at its most complex of the AND of two separate restriction 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, typically, (a) to reduce the number of tuples to be scanned in the next operation in sequence and (b) to reduce the number of tuples in the output from that operation as well.

Here are some other specific cases of the distributive law, this time involving projection. First, projection distributes over union, though not over intersection or 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 for "doing 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. Similarly, in relational algebra, union, intersection, and join are commutative, but difference isn't. So, for example, if a query involves a join of two 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—i.e., whichever of *r1* and *r2* has the lower cardinality—as the outer one in computing the join.[22]

[22] Strictly speaking, the SQL analogs of these operators *aren't* 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.

- 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—e.g., $(a + b) + c$ is equal to $a + (b + c)$—but subtraction and division aren't. Similarly, in relational algebra, union, intersection, and join are 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.[23] 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 physical 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. Perhaps I should add, however, that while many such transformations are available for sets, fewer are available for bags (as indeed we saw in some of the exercises in Chapter 4); and fewer still are available if column ordinal position has to be taken into account; and fewer still are available if nulls and 3VL have to be taken into account as well. What do you conclude?

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 proper 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 use SQL's ALTER TABLE (or something analogous to that operator) to "add a new attribute" to relvar *R2*, say, that has the same name as one already existing in relvar *R1?*

Well, first let me clarify 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); equivalently, they require attributes of different types to have different names. 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.[24] Note that this requirement

---

[23] We could even, if the query involves (say) four relations *r1*, *r2*, *r3*, and *r4*, join (e.g.) *r1* and *r3* first, join *r4* and *r2* second, and then join the results of those two joins.

[24] 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**.

(that attributes of different types have different names) imposes no serious functional limitations, thanks to the availability of the attribute 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 indeed from the host language's point of view as well. In other words, SQL 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 pertinent quote:[25] "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 was prevalent 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," logical data independence is preserved by changing the *mapping* between the two definitions.

Here's how the mechanism I've just described might look in SQL. First let me 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 ) ) ;
```

[25] It's 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).

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, of course—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:

```
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 "≝" 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, logical 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 (but then so too is the simpler expression SELECT * FROM S, come to that). 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 do change subsequently, all you'll have to do is change the view definitions accordingly.

EXERCISES

6.1 What if anything is wrong with the following SQL expressions or would-be 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`

h. `SELECT * FROM S JOIN SP ON S.SNO = SP.SNO`

i. `SELECT * FROM ( S NATURAL JOIN P ) AS temp`

j. `SELECT * FROM S CROSS JOIN SP CROSS JOIN P`

6.2 Closure is important in the relational algebra 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?

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?

6.4 Why do you think the project operator is so called?

6.5 For each of the following **Tutorial D** expressions on the suppliers-and-parts database, give both (a) an SQL analog and (b) an informal interpretation of the expression (i.e., a corresponding predicate) in natural language. Also show the result of evaluating the expressions, given our usual sample values for relvars S, P, and SP.

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 } ) { SNO , PNO }`

e.
```
JOIN { ( S RENAME { CITY AS SC } ) { SC } ,
       ( P RENAME { CITY AS PC } ) { PC } }
```

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

6.7 Which if any of the relational algebra operators described in this chapter have a definition that doesn't rely on tuple equality?

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?

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.

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.

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 algebra operators, if any, are idempotent?

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?

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 boolean expression $x > y$ AND $y > 3$ AND $x > 3$. (The

equivalence is based on the fact that the comparison operator ">" is *transitive*—i.e., $x > y$ and $y > z$ together imply $x > z$.) 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 product that actually performs this kind of optimization?

6.14 Consider the following **Tutorial D** expression:

```
WITH ( pp := P WHERE COLOR = 'Purple' ,
       tx := SP RENAME { SNO AS X } ) :
S WHERE ( tx 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 intuitive understanding of what the expression means? Justify your answer.

6.15 SQL has no direct counterpart to either D_UNION or I_MINUS. How best might the D_UNION and I_MINUS examples from the body of the chapter—i.e., S{CITY} D_UNION P{CITY} and S{CITY} I_MINUS P{CITY}—be simulated in SQL?

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)?

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 (or I_MINUS) make sense?

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.

6.19 What exactly does the following SQL expression return?

```
SELECT DISTINCT S.*
FROM   S , P
```

Warning: There's a trap here.

ANSWERS

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 P JOIN (S{CITY}) and (P JOIN S){CITY}. *Answer:* The first yields full part details (PNO, PNAME, COLOR, WEIGHT, and CITY) for parts in the same city as at least one supplier, the second yields just the CITY values for those same parts (speaking a trifle loosely in both cases).

- 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*.

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 (even though the SELECT clause explicitly specifies S.SNO, not SP.SNO, and SNO values are unique in table S). e. Compile time error: S NATURAL JOIN P has no column called S.CITY.[26] f. Nothing wrong (though it would be nice if CORRESPONDING were specified). 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.[27] h. The result has duplicate column names (as well as left to right column ordering). i. Compile time (syntax) error: AS specification not allowed (because the expression "(S NATURAL JOIN P)" is neither a table name nor a table subquery—see Chapter 12). j. The result has duplicate column names (as well as left to right column ordering).

---

[26] 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 dot qualified name anyway, as in the case at hand. (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.)

[27] In other words, although a column reference of the form "S.SNO" would be legal in the SELECT clause here—see part e. of the exercise—the expanded form of the expression "S.*" in that same context includes no such reference!

6.2 No! In particular, certain relational divides that—by analogy—you might intuitively expect to fail in fact don't. Here are some examples (which might not make much sense until you've read the relevant section of Chapter 7):

- Let relation *z* be of type RELATION {PNO CHAR} and let its body be empty. Then the expression

  ```
  SP { SNO , PNO } DIVIDEBY z { PNO }
  ```

  reduces to the projection SP{SNO} of SP on SNO.

- Let *z* be either TABLE_DEE or TABLE_DUM. Then the expression

  ```
  r DIVIDEBY z
  ```

  reduces to *r* JOIN *z*. In other words, if *z* is TABLE_DEE, the result is just *r*; if *z* is TABLE_DUM, the result is the empty relation of the same type as *r*.

- 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.

- Finally, *r* DIVIDEBY *r* gives TABLE_DUM if *r* is empty, TABLE_DEE otherwise.

6.3 First of all, observe that S, SP, and P are indeed "3-way joinable," as is required for the expression JOIN {S,SP,P} to be well formed (see the answer to Exercise 6.16 below). The joining attributes are SNO, PNO, and CITY (each of which is a common attribute for exactly two of the relations to be joined, as it happens). The result 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. Given our usual sample values, 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 simplest SQL formulation is just

```
S NATURAL JOIN SP NATURAL JOIN P
```

(though it might be necessary, depending on context, to prefix this expression with "SELECT * FROM"—see Chapter 12).

6.4 In two-dimensional analytic or 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 one-dimensional spaces of the point (*x*,*y*) in two-dimensional space. These notions are readily generalizable to *n* dimensions (recall from Chapter 3 that relations are indeed *n*-dimensional).

6.5 Throughout these answers, I show SQL expressions that aren't necessarily direct transliterations of their **Tutorial D** (i.e., algebraic) counterparts but are, rather, "more natural" formulations of the query in SQL terms.

a. SQL analog:

```
SELECT DISTINCT CITY
FROM   S NATURAL JOIN SP
WHERE  PNO = 'P2'
```

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 preliminary projections (of S on {SNO,CITY} and P on {PNO,CITY}) 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

6.6 Intersection and 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" (or "is equivalent to") and "∈" to stand for "appears in" (as usual in both cases), 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.

6.7 RENAME is the only one—and even that one's debatable! See the answer to Exercise 7.3 in the next chapter.

6.8 The 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 in Chapter 4 for further discussion.

6.9 **Tutorial D** on the left, SQL on the right, as usual (the solutions aren't unique, in general; note too that the **Tutorial D** solutions in particular could often be improved by using operators to be described in Chapter 7):

a.
```
SP
```
```
SELECT * FROM SP
or
TABLE SP /* see Chapter 12 */
```

b.
```
( SP WHERE PNO = 'P1' )
                    { SNO }
```
```
SELECT SNO
FROM   SP
WHERE  PNO = 'P1'
```

c.
```
S WHERE STATUS ≥ 15
    AND STATUS ≤ 25
```
```
SELECT *
FROM   S
WHERE  STATUS BETWEEN 15 AND 25
```

d.
```
( ( S JOIN SP )
  WHERE CITY = 'London' )
                    { PNO }
```
```
SELECT DISTINCT PNO
FROM   SP , S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

e.
```
P { PNO } MINUS ( ( S JOIN P )
  WHERE CITY = 'London' ) { PNO }
```
```
SELECT PNO
FROM   P
EXCEPT CORRESPONDING
SELECT PNO
FROM   SP , S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

f.
```
WITH ( z := SP { SNO , PNO } ) :
( ( z RENAME { PNO AS X } )
  JOIN
  ( z RENAME { PNO AS Y } ) )
{ X , Y }
```
```
SELECT DISTINCT XX.PNO AS X ,
                YY.PNO AS Y
FROM   SP AS XX , SP AS YY
WHERE  XX.SNO = YY.SNO
```

g.
```
( S WHERE STATUS <
  STATUS FROM ( TUPLE FROM
     ( S WHERE SNO = 'S1' ) ) )
                      S { SNO }
```
```
SELECT SNO
FROM   S
WHERE  STATUS <
     ( SELECT STATUS
       FROM   S
       WHERE  SNO = 'S1' )
```

Note: The expression STATUS FROM (TUPLE FROM ...) in the **Tutorial D** solution here extracts the STATUS value from the single tuple in the relation that's the TUPLE FROM argument (that relation must have cardinality one). By contrast, the SQL solution 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 ( tx := S WHERE
               CITY = 'London' ,
       ty := SP RENAME
               { PNO AS Y } ) :
( ( P WHERE
  ( ty WHERE Y = PNO ) ) { SNO }
                  ⊇ tx { SNO } )
                       { PNO }
```
```
SELECT PNO
FROM   P
WHERE  NOT EXISTS (
SELECT * FROM S
WHERE  CITY = 'London'
AND    NOT EXISTS (
SELECT *
FROM   SP
WHERE  SP.SNO = S.SNO
AND    SP.PNO = P.PNO ) )
```

Note the use of a relational comparison in the **Tutorial D** expression here. The SQL version uses EXISTS (see Chapter 10). A more elegant **Tutorial D** solution can be found as the answer to Exercise 7.9e in Chapter 7.

i.
```
( S { SNO } JOIN P { PNO } )
            MINUS SP { SNO , PNO }
```
```
SELECT SNO , PNO
FROM   S , P
EXCEPT CORRESPONDING
SELECT SNO , PNO
FROM   SP
```

```
j.  WITH ( tx := SP WHERE                  |  SELECT SNO
                    SNO = 'S2' ,           |  FROM   S
           ty := SP RENAME                 |  WHERE  NOT EXISTS (
                   { SNO AS Y } ) :        |  SELECT *
    S WHERE ( ty WHERE Y = SNO )           |  FROM   SP AS SPX
            { PNO } ⊇ tx { PNO }           |  WHERE  SNO = 'S2'
                                           |  AND    NOT EXISTS (
                                           |  SELECT *
                                           |  FROM   SP AS SPY
                                           |  WHERE  SPY.SNO = S.SNO
                                           |  AND    SPY.PNO = SPX.PNO ) )
```

A more elegant **Tutorial D** solution can be found as the answer to Exercise 7.9f in Chapter 7.

6.10 It's intuitively obvious that all three statements are true. *No further answer provided.*

6.11 Union isn't idempotent in SQL, because the expression SELECT * FROM *T* UNION CORRESPONDING SELECT * FROM *T* isn't identically equal to SELECT * FROM *T*.[28] That's because if *T* contains any duplicates, they'll be eliminated from the result of the union. (And what happens if *T* contains any nulls? Good question!)

Join and intersection are also idempotent in the relational model but not in SQL, "thanks" again to duplicates and nulls). Note, however, that cartesian product is *not* idempotent, in general; in fact, the expression *r* TIMES *r* fails on a syntax error, except in the very special case where the heading of *r* is empty.

6.12 As explained in the body of the chapter, 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 bar none of its attributes (in other words, it denotes the identity projection of *r*); it returns *r*. The corresponding predicate is identical to that for *r*.

6.13 So far as I know, DB2 and Ingres both perform this kind of optimization (DB2 refers to it as "predicate transitive closure"). Other products might do so too.

---

[28] The CORRESPONDING specification could safely be omitted here—why, exactly?—but it's easier, and shouldn't hurt, always to specify CORRESPONDING, even when it's logically unnecessary.

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 a relation identical to the current value of relvar S (i.e., all five suppliers, loosely speaking). For further explanation—in particular, for justification of the fact that this is indeed the correct answer—see Chapter 11.

6.15 For S{CITY} D_UNION P{CITY}, 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 original, 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:* In case you're wondering about that AS *temp* specification, it's there because it's required—on the subquery in the FROM clause but not on the subquery in the WHERE clause (where in fact it would be illegal!)—for reasons explained in Chapter 7.

Turning now to S{CITY} I_MINUS P{CITY}, a rough equivalent in SQL might look like this:

```
SELECT CITY
FROM ( SELECT CITY FROM S
       EXCEPT CORRESPONDING
       SELECT CITY FROM P ) AS temp
WHERE  NOT EXISTS
     ( SELECT CITY FROM P
       EXCEPT CORRESPONDING
       SELECT CITY FROM S )
```

Again, however, this SQL expression isn't precisely equivalent to the original, however. To be specific, if part cities aren't a subset of supplier cities, then the SQL expression won't fail at run time but will simply return an empty result.

6.16 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 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—sometimes *n*-way joinable, for emphasis—if and only if, for all *i*, *j* ($1 \le i \le n$, $1 \le j \le n$), relations *ri* and *rj* are joinable. *Note:* It's worth pointing out that dyadic

joinability isn't transitive; that is, just because (a) *r1* and *r2* are joinable and (b) *r2* and *r3* are joinable, it doesn't necessarily follow that (c) *r1* and *r3* are joinable. Development of an example to illustrate this point is left as a subsidiary exercise.

6.17 It's possible to define *n*-adic versions of JOIN, UNION, and D_UNION because the operators (a) are all both commutative and associative and (b) all have a corresponding identity value.

SQL does effectively support (analogs of) *n*-adic join and union, though not for $n < 2$. For join, the syntax is:[29]

```
[ SELECT * FROM ] t1 NATURAL JOIN t2
                     NATURAL JOIN t3
                       ...........
                     NATURAL JOIN tn
```

For union, the syntax is:

```
SELECT * FROM t1 UNION CORRESPONDING SELECT * FROM t2
                 UNION CORRESPONDING SELECT * FROM t3
                   ..............................
                 UNION CORRESPONDING SELECT * FROM tn
```

An *n*-adic version of MINUS or I_MINUS makes no sense because MINUS and I_MINUS are neither commutative nor associative, nor do they have a corresponding identity value.

6.18 For a brief justification, see the answer to Exercise 6.12 above. A longer one follows. Consider the projection S{SNO} of (the relation that's current value of) the suppliers relvar S on {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*{ }. As we saw in the answer to Exercise 3.16 in Chapter 3, 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:

---

[29] Actually it's not quite correct to say the SQL expression shown denotes an *n*-adic join, because (a) for *n*-adic join, the relations involved are required to be *n*-way joinable and the sequence in which they're specified is irrelevant, whereas (b) the SQL expression shown is defined to perform the individual dyadic joins in the order specified (first join *t1* and *t2*, then join the result of that join and *t3*, and so on).

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

Observe now that this last predicate is in fact a proposition: It evaluates to TRUE or FALSE, unequivocally. 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.

6.19 The expression returns the current value of table S—*unless* table P is currently empty, in which case it returns an empty result.

Chapter 7

SQL and Relational Algebra II:

Additional Operators

Algebra is the part of advanced mathematics that is not calculus.
—John Derbyshire:
Unknown Quantity:
A Real and Imaginary History of Algebra (2006)

As I've said several times already, a relational algebra operator is an operator that (a) takes one or more relations—or possibly no relations at all, in the case of the *n*-adic versions of operators such as join (see Chapter 6)—as input and (b) 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. The previous chapter described the original operators (join, project, etc.); by contrast, 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 might best be realized in SQL.

Note: By its nature, this chapter is necessarily something of a miscellany. Thus, you might just want to skim it lightly on a first pass, and come back to it later if you need to gain a deeper understanding of one or more of the topics discussed. Perhaps it would help to say up front that, from a practical point of view at least, the most important topics are probably these:

- Semijoin and semidifference (MATCHING and NOT MATCHING)
- EXTEND
- Image relations
- Aggregate operators

But I'll begin with a brief discussion of exclusive union.

EXCLUSIVE UNION

In set theory, union is *inclusive*; that is, given sets *s1* and *s2*, an element appears in their union if and only if it appears in either or both of *s1* or *s2*. Thus, the UNION operator can be seen as the set theory counterpart to logical OR, which is inclusive in a similar sense. But logic additionally defines an exclusive version of OR (XOR), and so we can define an exclusive union operator analogously: The exclusive union (XUNION) of two sets *s1* and *s2* is the set of elements appearing in *s1* or *s2* but not both. And, of course, we can define a relational version of this operator as well:

> **Definition:** Let relations *r1* and *r2* be of the same type *T*. Then (and only then) the expression *r1* XUNION *r2* denotes the *exclusive union* of *r1* and *r2*, and it returns the relation of type *T* with body the set of all tuples *t* such that *t* appears in exactly one of *r1* and *r2*.

For example (assuming as we did in Chapter 6, in the section "Union, Intersection, and Difference," that parts have an extra attribute called STATUS, of type INTEGER):

```
P { STATUS , CITY } XUNION
S { CITY , STATUS }
```

```
SELECT STATUS , CITY
FROM   P
WHERE ( STATUS , CITY ) NOT IN
      ( SELECT STATUS , CITY
        FROM   S )
UNION  CORRESPONDING
SELECT CITY , STATUS
FROM   S
WHERE ( CITY , STATUS ) NOT IN
      ( SELECT CITY , STATUS
        FROM   P )
```

Tutorial D also supports an *n*-adic form of XUNION. However, the details are a little tricky; for that reason, I'll just give a definition here, without further discussion. You can find more details, if you're interested, in the paper "*N*-adic vs. Dyadic Operators: An Investigation" (see Appendix G).

> **Definition:** Let relations *r1*, *r2*, ..., *rn* ($n \geq 0$) all be of the same type *T*. Then (and only then) the expression XUNION {*r1*,*r2*,...,*rn*} denotes the *exclusive union* of *r1*, *r2*, ..., *rn*, and it returns the relation of type *T* with body the set of all tuples *t* such that *t* appears in exactly *m* of *r1*, *r2*, ..., *rn*, where *m* is odd (and possibly different for different tuples *t*).

The exclusive union of a single relation *r* is just *r*. The exclusive union of no relations at all is the empty relation of the pertinent type—but that type needs to be specified explicitly, since there aren't any relational expressions from which the type can be inferred. Thus, for example, the expression

```
XUNION { SNO CHAR , STATUS INTEGER } { }
```

denotes the empty relation of type RELATION {SNO CHAR, STATUS INTEGER}.

Note: In set theory, exclusive union is more usually known as *symmetric difference*. Thus, the keyword XMINUS might be acceptable as an alternative to XUNION.

SEMIJOIN AND SEMIDIFFERENCE

Join is one of the most familiar of all of the relational operators. In practice, however, it turns out that queries that require the join operator at all often really require a variation on that operator called semijoin. (You might not have heard of semijoin before, but in fact it's quite important.) Here's the definition:

> **Definition:** Let relations *r1* and *r2* be joinable,[1] and let *r1* have attributes called *A1*, *A2*, ..., *An* (and no others). Then (and only then) the expression *r1* MATCHING *r2* denotes the *semijoin* of *r1* with *r2* (in that order), and it returns the relation denoted by the expression (*r1* JOIN *r2*) {*A1*,*A2*, ...,*An*}.

In other words, *r1* MATCHING *r2* is the join of *r1* and *r2*, projected back on the attributes of *r1* (and so the heading of the result is the same as that 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 )
```

Note that the expressions *r1* MATCHING *r2* and *r2* MATCHING *r1* aren't equivalent, in general—the first returns some subset of *r1*, the second returns some subset of *r2*. 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 now to semidifference:[2] 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. Here's the definition:

---

[1] Recall from Chapter 6 that two relations are joinable if and only if attributes with the same name are of the same type (i.e., are in fact one and the same attribute, formally speaking).

[2] Also known, a trifle inappropriately (?), as *antijoin*.

Definition: Let relations *r1* and *r2* be joinable. Then (and only then) the expression *r1* NOT MATCHING *r2* denotes the *semidifference* between *r1* and *r2* (in that order), and it returns the relation denoted by the expression *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 )
```

As with MATCHING, the heading of the result is the same as that of *r1*. *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 in this book doesn't have any conventional computational capabilities. Now, SQL does; certainly we can write queries in SQL of the form 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 EXTEND is what we need. By way of example, suppose part weights (in relvar 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 :
     { GMWT := WEIGHT * 454 }
```

```
SELECT P.* , WEIGHT * 454 AS GMWT
FROM   P
```

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

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Important: Relvar P is *not* changed in the database! EXTEND is *not* like SQL's ALTER TABLE; the EXTEND expression is just an expression, and like any expression it simply

denotes a value. In particular, therefore, it can be nested inside other expressions. Here's an example (the query is "Get part number and gram weight for parts with gram weight greater than 7000 grams"):

```
( ( EXTEND P :                          |   SELECT PNO ,
    { GMWT := WEIGHT * 454 } )          |          WEIGHT * 454 AS GMWT
         WHERE GMWT > 7000.0 )          |   FROM   P
                 { PNO , GMWT }         |   WHERE  WEIGHT * 454 > 7000.0
```

As you can see, there's an interesting difference between the **Tutorial D** and SQL versions of this example. To be specific, the (sub)expression WEIGHT * 454 appears once in the **Tutorial D** version but twice in the SQL version. In the SQL version, therefore, we have to hope the implementation will be smart enough to realize that it can evaluate that subexpression just once per tuple (or once per row, rather) instead of twice.

The problem that this example illustrates is that SQL's SELECT − FROM − WHERE template is *too rigid*. What we need to do, as the **Tutorial D** formulation makes clear, is take a restriction of an extension (and then take a projection of that restriction); in SQL terms, in other words, 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 (see the section "Evaluating SQL Table Expressions" in Chapter 6). 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 rename[3]—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 on a syntax error at compile time.

Actually, SQL 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 of the otherwise implicit dot qualifications explicitly, for clarity):

```
SELECT temp.PNO , temp.GMWT
FROM ( SELECT PNO , WEIGHT * 454 AS GMWT
       FROM P ) AS temp
WHERE  temp.GMWT > 7000.0
```

[3] And/or ungroup (see later in this chapter).

But this functionality—namely, the ability to allow nested subqueries in the FROM clause—wasn't part of SQL as originally defined (it was introduced into the standard with SQL:1992). 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 world SQL queries.

Note: I need to say a little more about the FROM clause in the foregoing example. As you can see, it takes the form

```
FROM ( ... ) AS temp
```

Formally speaking, it's the parenthesized expression within this FROM clause that constitutes the nested subquery (see Chapter 12). And—here comes the point—SQL has a syntax rule to the effect that a nested subquery in the FROM clause *must* be accompanied by an explicit AS specification that defines a name for the table denoted by that subquery,[4] even if that name is never explicitly referenced elsewhere in the overall expression. In fact, in the example at hand, we could omit all of the explicit references to the name *temp* (i.e., all of the explicit "*temp*." dot qualifications) if we wanted to, thus:

```
SELECT PNO , GMWT
FROM ( SELECT PNO , WEIGHT * 454 AS GMWT
       FROM P ) AS temp
WHERE  GMWT > 7000.0
```

But that AS *temp* specification is still needed nonetheless.

I'll close this section with a formal definition of the EXTEND operator:

> **Definition:** Let relation *r* not have an attribute called *A*. Then (and only then) the expression EXTEND *r* : {*A* := *exp*} denotes an *extension* of *r*, and it returns the relation with heading the heading of *r* extended with attribute *A* and body the set of all tuples *t* such that *t* is a tuple of *r* extended with a value for *A* that's computed by evaluating the expression *exp* on that tuple of *r*.

Observe that the result has cardinality equal to that of *r* and degree equal to that of *r* plus one. The type of *A* in that result is the type of *exp*.

---

[4] More accurately, it defines a corresponding *range variable*. See Chapter 12 for further explanation.

IMAGE RELATIONS

Image relations are a hugely useful abstraction in general, and it's a little surprising that so few people in the database community seem to be aware of them (though to be fair they haven't been very widely discussed in the literature). However, I need to alert you up front to the fact that some people do seem to find them a little difficult to come to grips with; I don't really know why this should be—the idea is actually quite straightforward—but as I say, I think it's only fair to make you aware of this state of affairs. You have been warned!

First, then, here's an informal definition: An image relation is, loosely, the "image" of some tuple *t1* within some relation *r2* (where, typically, tuple *t1* is a tuple in the current value of some relvar *R1* and relation *r2* is the current value of some relvar *R2*, usually but not necessarily distinct from *R1*). By way of example, consider the suppliers-and-parts database, with its usual sample values. Let tuple *t1* be the tuple for supplier S4 in the current value of relvar S, and let relation *r2* be the current value of relvar SP. Then the following is the image of that supplier tuple *t1* within that shipments relation *r2*:

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 }
```

In other words, restrict the current value of SP to just the tuples for S4, and then project away attribute SNO (since its value is the same, viz., S4, in every tuple in that restriction).

Here now is a formal definition of image relations in general:

> **Definition:** Let relations *r1* and *r2* be joinable; let *t1* be a tuple of *r1*; let *t2* be a tuple of *r2* that has the same values as tuple *t1* for those attributes that are common to *t1* and *t2*; 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 }
```

Explanation:

- First of all, the roles of *r1* and *r2* from the formal 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, of course, and similarly for "the shipments relation").

- Next, observe that the boolean expression in the WHERE clause here involves an equality comparison between two relations (actually two projections). We can imagine that boolean expression 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*.[5] 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 and well defined. WHERE clauses are one such context, as the foregoing example indicates, and we'll see another in the section "Image Relations Revisited," later in this chapter.

Aside: SQL has no direct support for image relations as such. Here for interest is an SQL formulation of the query "Get suppliers who supply all parts" (I show it for your consideration, but I'm not going to discuss it in detail here, except to note that it can obviously (?) be improved in a variety of ways):

---

[5] As noted elsewhere in this book, in mathematics the expression "*n*!" (*n* factorial) is often pronounced "*n* bang"; hence my choice of pronunciation for the symbol "!!". *Note:* If you prefer a syntax that's based on keywords, you could consider replacing "!!*r*" by "IMAGE IN *r*".

```
SELECT *
FROM   S
WHERE  NOT EXISTS
     ( SELECT PNO
       FROM   SP
       WHERE  SP.SNO = S.SNO
       EXCEPT CORRESPONDING
       SELECT PNO
       FROM   P )
AND    NOT EXISTS
     ( SELECT PNO
       FROM   P
       EXCEPT CORRESPONDING
       SELECT PNO
       FROM   SP
       WHERE  SP.SNO = S.SNO )
```

End of aside.

To get back to image relations as such, it's worth noting that the "!!" operator can be defined in terms of MATCHING. For example, the expression 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 in that expression 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.

It follows that, in the context under consideration, the overall expression "SP MATCHING ..." is logically equivalent to the image relation reference "‼SP" as claimed.

By way of another example, suppose we're given a revised version of the suppliers-and-parts database—one that's simultaneously both extended and simplified, compared to our 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 ‼SP ⊇ ‼PJ
```

Exercise: Give an SQL analog of this expression.

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
  ( ‼( S WHERE
         CITY = 'London' ) ) ;
```

```
DELETE FROM SP
WHERE  SNO IN
     ( SELECT SNO FROM S
       WHERE CITY = 'London' ) ;
```

For a given shipment, the relation denoted by the specified image relation reference—i.e., ‼(S WHERE ...)—is either empty, if the corresponding supplier isn't in London, or contains exactly one tuple otherwise. *Note:* I'm not claiming that image relations are a big help with this particular example; I just wanted to show an image relation reference of the form ‼*rx* where the specified relational expression *rx* was something more complicated than just a simple relvar reference (i.e., not just a simple relvar name). Here for the record is another formulation of the example in **Tutorial D** that avoids the image relation reference (actually it's very similar to the SQL formulation):

```
DELETE SP WHERE SNO ∈ ( S WHERE CITY = 'London') { SNO }
```

DIVIDE

I include a discussion of divide in this chapter mainly just to show why I think—contrary to conventional wisdom, perhaps—it isn't very important; in fact, I think it should be dropped. You can skip this section if you like.

I have at least three reasons 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.[6] Instead, I'll limit my attention here to the original and simplest one. Here's a definition:

> **Definition:** Let relations *r1* and *r2* be such that the heading of *r2* is some subset of the heading of *r1*, and let *A1*, *A2*, ..., *An* be the attributes of *r1* that aren't also attributes of *r2*. Then (and only then) the expression *r1* DIVIDEBY *r2* denotes the *division* of *r1* by *r2*,[7] and it's logically equivalent to the following:
>
> ```
> WITH ( temp := r1 { A1 , A2 , ... , An } ) :
>        temp NOT MATCHING ( ( temp JOIN r2 ) NOT MATCHING r1 )
> ```
>
> For example, the expression
>
> ```
> SP { SNO , PNO } DIVIDEBY P { PNO }
> ```

(given our usual sample data values) yields:

SNO
S1

Note: I recommend that you take a few moments right now to check this result. (*Pause*.)

Now, if you do indeed check the result as recommended, you'll see why the DIVIDEBY expression might loosely be characterized as representing 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

---

[6] The seven different operators are discussed in excruciating detail in the paper "A Brief History of the Relational Divide Operator" (see Appendix G).

[7] **Tutorial D** doesn't directly support the original divide operator, and *r1* DIVIDEBY *r2* is thus not valid **Tutorial D** syntax.

numbers) for the suppliers in question, in which case the division will need to be followed by a join, thus:

```
( SP { SNO , PNO } DIVIDEBY P { PNO } ) JOIN S
```

But we already know how to formulate this latter 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, so it seems 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 the operator 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*. (At least, we do so just as long as *r2* isn't empty. What happens if it is?) In other words, product and divide are inverses of each other, in a sense.

Now, I've said the expression

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

can loosely be characterized as a formulation of the query "Get supplier numbers for suppliers who supply all parts"; indeed, 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 query "Get supplier numbers for suppliers who *supply at least one part and in fact* supply all parts."[8] 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 explicitly, intended 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.[9] But I do need to say something about them before I can get back to the main theme of the chapter.

---

[8] 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 any analogous DIVIDEBY expression. And if you're still wondering, then see the further discussion of this example in Chapter 11.

[9] But nonscalar aggregate operators can be defined too, as we'll see in the section "GROUP, UNGROUP, and Relation Valued Attributes."

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. The examples on the left below illustrate the the use of the COUNT aggregate operator specifically in **Tutorial D**:

```
X := COUNT ( S ) ;                  |  SELECT COUNT ( * ) AS X
                                    |  FROM   S

Y := COUNT ( S { STATUS } ) ;       |  SELECT COUNT ( DISTINCT STATUS ) AS Y
                                    |  FROM   S
```

Note that I carefully don't say the examples on the right "illustrate the use of the COUNT aggregate operator in SQL." That's because I'm going to argue that SQL doesn't really support aggregate operators at all!—at least, not properly. But first things first ... Until further notice, let me focus on **Tutorial D**, and in particular on the COUNT examples just shown. Given our usual sample values:

- The first example assigns the value 5 (the number of tuples in the current value of relvar S) to the variable X.

- The second example 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. 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.

Aside: The aggregate operators AND, OR, and XOR apply to aggregates of boolean values specifically. AND in particular can be useful in connection with certain integrity constraints (see Chapter 8 for further discussion). As for SQL, SQL's counterparts to AND and OR are called EVERY and SOME, respectively (there's no counterpart to XOR). SOME can alternatively be spelled ANY; likewise, in ALL or ANY comparisons (see Chapter 12), ANY can alternatively be spelled SOME. Oddly enough, however, the SQL "set function"

EVERY can't alternatively be spelled ALL, and in ALL or ANY comparisons ALL can't alternatively be spelled EVERY. *End of aside*.

Here are some **Tutorial D** examples:

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, if the expression <*exp*> is more complicated than a simple <*attribute ref*>, then the invocation

   ```
   agg ( rx , exp )
   ```

   is essentially shorthand for the following:

   ```
   agg ( EXTEND rx : { a := exp } , a )
   ```

I turn now 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 have been surprised to hear me claim earlier that SQL doesn't really support aggregate operators at all—especially since most people would surely consider SELECT expressions like those on the right above to be, precisely, SQL aggregate operator invocations.[10] But they aren't. Let me explain. As we know, the counts are 5 and 3, respectively. But those

[10] It might be claimed, somewhat more reasonably, that *the COUNT invocations within* those expressions are SQL aggregate operator invocations. But the whole point about such invocations is that they can't appear as "stand alone" expressions in SQL; rather, they can only appear as part of some table expression, because they rely on that expression to identify the table over which the aggregation is to be done. For example, a statement like "SET X = COUNT(*);" would be meaningless in SQL, since it fails to identify the table whose rows are to be counted.

SELECT expressions don't yield those counts as such, as true aggregate operator invocations would; rather, they yield 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

```
/* the lack of double underlining in these */
/* tables is not a mistake -- see Exercise */
/* 7.23 at the end of the chapter          */
```

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 yet, of course; for present purposes, however, you can regard it loosely as what's represented in SQL by a SELECT expression with a GROUP BY clause. Now, the foregoing SELECT 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 ( )
```

Aside: In case these expressions look strange to you, I should explain that, first, SQL does allow GROUP BY clauses with the operand commalist enclosed in parentheses; second, that commalist can be empty, just so long as those parentheses are specified; finally, specifying such an empty commalist is equivalent to omitting the GROUP BY clause entirely, because:

a. Such a GROUP BY effectively means "group by no columns."

b. Every row has the same value for no columns: namely, the 0-row (despite the fact that SQL doesn't actually support the 0-row!).

c. Every row in the table is thus part of the same group; in other words, the entire table is treated as a single group,[11] and that's effectively what happens when the GROUP BY clause is omitted entirely.

End of aside.

[11] But see further remarks on this topic in the next aside, on pages 232-233.

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 SQL-style "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 a detailed 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 "straightforward" 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* ($nG \leq nT$), 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., the grouped 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 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 happens to be 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 commalist 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 repeat from Chapter 1—as Wittgenstein said, *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. *End of aside.*

Empty Arguments

The foregoing aside raises yet another issue. To be specific, let *agg* be an aggregate operator; then what should happen if *agg* is invoked on an empty argument? For example, given our usual sample data values, what value should the following statement assign to X?

```
X := SUM ( SP WHERE SNO = 'S5' , QTY ) ;
```

The answer, of course, is zero; as explained in Chapter 6 under the discussion of *n*-adic join, zero is the identity value with respect to addition, and the sum of no numbers is therefore

zero. More generally, let *agg* be an aggregate operator (other than COUNT), and let *av* be the aggregate value over which some given invocation of *agg* is to be evaluated. If *av* is of cardinality one, the result of the invocation in question is the single value contained in *av*. If *av* is of cardinality zero (i.e., if *av* is empty), and if all of the following are true—

a. The invocation in question is essentially just shorthand for repeated invocation of some dyadic operator *op*;

b. An identity value *iv* exists for *op*;

c. The semantics of *agg* don't demand the result of an invocation to be a value actually appearing in *av*;

—then

d. The result of the invocation in question is that identity value *iv*.

Thus, for the aggregate operators discussed in this section, identity values (and hence the result returned if the argument is empty) are as follows:[12]

- AND: TRUE.
- OR and XOR: FALSE.
- COUNT and SUM: Zero. *Note:* The type of the result in these cases is INTEGER (for COUNT) and the type of the specified argument expression (for SUM). By way of example, if relvar P is currently empty, COUNT (P) returns 0 and SUM (P,WEIGHT) returns 0.0.
- AVG: Since asking for the average of an empty set is effectively asking for zero to be divided by zero, the only reasonable response is to raise an exception (and careful coding might sometimes be called for, therefore).
- MAX and MIN: By definition, asking for the maximum or minimum of some set of values is asking for some specific value from within that set. If the set in question happens to be empty, therefore, the only reasonable response is, again, to raise an exception (and careful coding might again sometimes be called for, therefore).

[12] By contrast, as noted in Chapter 4, the SQL analogs of these operators all return null if their argument is empty (except for COUNT and COUNT(*), which do correctly return zero).

To return to MAX and MIN for a moment: Actually there's an argument that says the MAX and MIN of an empty aggregate shouldn't be undefined after all. For definiteness, consider MAX specifically. Define a dyadic operator MAX2 that returns the larger of its two arguments (more precisely, define MAX2{*x1*,*x2*} to return *x1* if $x1 \geq x2$ and *x2* otherwise). Then (a) any given MAX invocation is essentially just shorthand for repeated invocation of MAX2, and (b) MAX2 clearly has an identity value, viz., "negative infinity" (meaning the minimum value of the pertinent type); so we might reasonably define MAX to return that identity value if its aggregate argument is empty. Likewise, we might reasonably define MIN to return "positive infinity" (the maximum value of the pertinent type) if its aggregate argument is empty. Perhaps the best approach in practice would be to provide both versions of MAX—they are, after all, different operators—and let the user decide. We might even provide a third version, one that takes an additional argument *x*, where *x* is supplied by the user and is the value to be returned if the aggregate argument is empty.

IMAGE RELATIONS REVISITED

In this section, I just want to present a series of examples that show the usefulness of image relations in connection with aggregate operators as discussed in the previous section.

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
```

Incidentally, 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 minimum shipment quantity is less than half the maximum shipment quantity (taken over all shipments for the supplier in question in both cases).

```
S WHERE MIN ( ‼SP , QTY ) < 0.5 * MAX ( ‼SP , QTY )
```

Here I'm assuming that MIN and MAX have been defined to return "positive infinity" and "negative infinity," respectively, if their aggregate argument is empty (see the discussion of such matters at the very end of the previous section). Note that under that assumption (and given our usual sample values), the result contains no tuple for supplier S5.

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 very contrived, but it illustrates the point that we might occasionally need to do 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 the RENAME invocation in this example—

```
SP RENAME { SNO AS SN , PNO AS PN }
```

—illustrates the "multiple" form of the RENAME operator. The individual renamings in such a RENAME invocation are effectively executed in parallel.[13] Similar "multiple" forms 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 } ;
```

[13] Because of this fact, RENAME can be used to switch the names of attributes, like this: *R* RENAME {*A* AS *B*, *B* AS *A*}.

SUMMARIZATION

It's convenient to begin this section with an example (which I'll label SX1—"SUMMARIZE Example 1"—for purposes of subsequent reference):

```
SUMMARIZE SP PER ( S { SNO } ) : { PCT := COUNT ( PNO ) }
```

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

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

Explanation: In this example, the relation to be summarized—"the SUMMARIZE relation"—is the current value of relvar SP, and the relation controlling or driving the summmarization—"the PER relation"—is the current value of the expression S{SNO} (in other words, it's the projection on {SNO} of the current value of relvar S). The result consists of five tuples, one for each tuple in the PER relation. In turn, each result tuple consists of a PER tuple extended with a certain count, that count being a count of the number of PNO values in the SUMMARIZE relation that are paired with the SNO value in that particular PER tuple.

Here now is a definition:

> **Definition:** Let relations *r1* and *r2* be such that the heading of *r2* is some subset of that of *r1*. Let *r2* have attributes called *A1*, *A2*, ..., *An* and no others (in particular, no attribute called *B*). Then (and only then) the expression SUMMARIZE *r1* PER (*r2*) : {*B* := *exp*} denotes a *summarization* of *r1* according to *r2*, and it returns the relation with (a) heading consisting of attributes *A1*, *A2*, ..., *An*, and *B* and (b) body consisting of all tuples *t* such that *t* is a tuple of *r2*, extended with a value *b* for attribute *B*. That value *b* is computed by evaluating the expression *exp* over all tuples of *r1* that have the same value for attributes *A1*, *A2*, ..., *An* as *t* does.

Points arising from this definition:

- With reference to Example SX1, *r1* is the current value of SP and *r2* is the current value of S{SNO}. Observe in particular, therefore, that the heading of *r2* is indeed a subset of the heading of *r1*, as required.

- The result has cardinality equal to that of *r2* and degree equal to that of *r2* plus one. The type of attribute *B* in that result is the type of *exp*. (In terms of Example SX1, the result has cardinality five and degree two, and "attribute *B*"—i.e., attribute PCT—is of type INTEGER.)

- The expression *exp* will typically be what's called an *open* expression. An open expression is one that can't appear in all possible contexts in which expressions in general can appear, but only in certain specific contexts: namely, those that suffice to give it meaning. *Note:* Actually this notion should be familiar to you; we've encountered it before, in a footnote in the previous chapter. To repeat the example from that footnote, the expression STATUS = 20 is open; by contrast, the expression S WHERE STATUS = 20 is closed. (As a matter of fact we've encountered another example of open expressions in this chapter too: viz., image relation references.)

- More specifically, the expression *exp* can, and in practice usually will, include at least one *summary*. COUNT (PNO) is an example. Note carefully that this "summary" is *not* an invocation of the COUNT aggregate operator; to be specific, the COUNT aggregate operator takes a relation as its argument, while the argument in the expression COUNT (PNO) is not a relation but an attribute. In fact, the expression COUNT (PNO) is really rather special—it has no meaning outside the context of an appropriate SUMMARIZE, and it can't be used outside that context. Note, therefore, that my earlier criticisms of COUNT and the rest in SQL (to the effect that they can't appear "stand alone") apply with just as much force to **Tutorial D**'s "summaries." 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 Revisited," later.

- So there's a logical difference between aggregate operators and summaries. However, at least it's true that every aggregate operator does have a summary counterpart (and vice versa). It's also true that each aggregate operator has the same name as its summary counterpart. We'll see some more examples of such summaries later.

I've said that the heading of the PER relation *r2* is required to be the same as that of some projection of the SUMMARIZE relation *r1*. However, in the special case where *r2* doesn't merely have the same heading as some projection of *r1* but actually is such a projection, **Tutorial D** provides us with a tiny shorthand. To be specific, it allows the PER specification to be replaced by a BY specification, as here ("Example SX2"):

```
SUMMARIZE SP BY { SNO } : { PCT := COUNT ( PNO ) }
```

Here's the result:

SNO	PCT
S1	6
S2	2
S3	1
S4	3

As you can see, this result differs from the result in Example SX1 in that it contains no tuple for supplier S5. That's because BY {SNO} here is defined to be shorthand for PER (SP{SNO})—SP, because it's the current value of SP that we want to summarize—and SP doesn't currently contain a tuple for supplier S5.

Now, Example SX2 can be expressed in SQL more or less directly, as follows:

```
SELECT SNO , COUNT ( ALL PNO ) AS PCT
FROM   SP
GROUP  BY SNO
```

Recall now from the section "Aggregate Operators" that, *very* loosely speaking, aggregation—to the extent it's supported at all, that is—is represented in SQL by a SELECT expression without an explicit GROUP BY. By contrast, as the foregoing example suggests (and as previously noted in that section "Aggregate Operators," in fact), summarization is represented in SQL by a SELECT expression *with* an explicit GROUP BY clause (usually, at any rate, but see further discussion later). Points arising:

- You can think of such an expression as being evaluated as follows. First, the table specified by the FROM clause is partitioned into set of disjoint "groups"—actually tables—as specified by the grouping column(s) in the GROUP BY clause; second, result rows are obtained, one for each group, by computing the specified summary (or summaries, plural) for the pertinent group and appending other items as specified by the SELECT item commalist. *Note:* The SQL analog of the term *summary* is "set function"; that 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. For these reasons, in fact, I generally set the term in quotation marks.

- 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 such row contains a different unique 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. (The alternative is DISTINCT, of course. In the case at hand, however, it makes no difference whether ALL or DISTINCT is specified, because the combination of supplier number and part number is a key for SP.)

- The "set function" COUNT(*) is special—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 affecting the result.)

Now let's get back to Example SX1. Here's a possible SQL formulation of that example:

```
SELECT S.SNO , ( SELECT COUNT ( PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

The important point about this example is that the result now does contain a row for supplier S5, because (thanks to the FROM clause, which takes the form FROM S) that result contains one row for each supplier number in table S, not table SP. 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 another trap for the unwary 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 same text were to appear in a FROM clause, the "AS *name*" specification would be understood as defining a name for the *table* denoted by that subquery.[14] In the SELECT clause, however, that very same "AS *name*" specification is understood as defining a name for the pertinent *column* of the overall result. It follows that the following SQL expression is *not* logically equivalent to the one shown above:

```
SELECT S.SNO , ( SELECT COUNT ( 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) the standard doesn't guarantee that that column in the overall result has any particular column name; in particular, it *doesn't* guarantee that it's called PCT. *End of aside.*

To revert to the main thread of the discussion: As a matter of fact, Example SX2 could also be expressed in SQL without using GROUP BY, as follows:

---

[14] More accurately, it would be understood as defining a range variable that ranges over that table (see Chapter 12).

```
SELECT DISTINCT SPX.SNO , ( SELECT COUNT ( SPY.PNO )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS PCT
FROM   SP AS SPX
```

As these examples suggest, SQL's GROUP BY clause is in fact logically redundant (a fact that I'm sure will come as a surprise to some readers); that is, any relational expression that can be represented using GROUP BY can also be represented without it. But there's another point that needs to be made here too. Suppose Example SX1 had requested, not the count of part numbers, but the sum of quantities, for each supplier. In **Tutorial D**:

```
SUMMARIZE SP PER ( S { SNO } ) : { TOTQ := SUM ( QTY ) }
```

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 ( 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 (as mentioned in Chapter 4) if any SQL "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 ( QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

Suppose now that the example had asked for the sum of quantities for each supplier, but only where that sum is greater than 250. In **Tutorial D**, then, we can simply enclose the formulation shown earlier in parentheses and apply the pertinent restriction to it, thus:

```
( SUMMARIZE SP PER ( S { SNO } ) : { TOTQ := SUM ( QTY ) } )
                                                    WHERE TOTQ > 250
```

Result:

SNO	TOTQ
S1	1300
S2	700
S4	900

A "natural" SQL formulation of this query would be as follows (note the HAVING clause):

```
SELECT SNO , SUM ( QTY ) AS TOTQ
FROM   SP
GROUP  BY SNO
HAVING SUM ( QTY ) > 250  /* not TOTQ > 250 !!! */
```

But it could also be formulated like this:

```
SELECT DISTINCT SPX.SNO , ( SELECT SUM ( SPY.QTY )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS TOTQ
FROM   SP AS SPX
WHERE  ( SELECT SUM ( SPY.QTY )
         FROM   SP AS SPY
         WHERE  SPY.SNO = SPX.SNO ) > 250
```

As this example suggests, then, HAVING, like GROUP BY, is also logically redundant—any relational expression that can be represented with it can also be represented without it. So GROUP BY and HAVING could both be dropped from SQL without any loss of relational functionality! And while it might be true that the GROUP BY and HAVING versions of some query are often more succinct,[15] it's also true that they sometimes deliver the wrong answer. For example, consider what would happen in the foregoing example if we had wanted the sum to be less than, instead of greater than, 250. Simply replacing ">" by "<" in the GROUP BY / HAVING formulation does *not* work. (Why not? Does it work in the non GROUP BY / non HAVING formulation?) **Recommendations:** If you do use GROUP BY or HAVING, make sure the table in the FROM clause is the one you really want to drive the operation (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.

---

[15] More tests of your SQL knowledge: In the example under discussion, would it be possible to save keystrokes by using WITH to introduce a name for the common subexpression "SELECT SUM(SPY.QTY) FROM SP AS SPY WHERE SPY.SNO = SPX.SNO"? Also, would it be legal to attach "AS TOTQ" to the appearance of that subexpression within the WHERE clause?

There's one more thing I need to say about GROUP BY and HAVING. Consider the following SQL expression:[16]

```
SELECT SNO , CITY , SUM ( QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO
```

Observe that column CITY is mentioned in the SELECT item commalist here but isn't one of the grouping columns. That mention is legitimate, however, because table S is subject to a certain functional dependency—see Chapter 8—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 dependency. 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 REVISITED

The SUMMARIZE operator has been part of **Tutorial D** since its inception. When image relations were introduced, however, that operator became logically redundant—and while there might be reasons (perhaps pedagogic ones) to retain it, the fact is that most summarizations can be more succinctly expressed by using image relations in combination with the EXTEND operator, as I now proceed to show.[17]

First of all, 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 } ) : { PCT := COUNT ( PNO ) }
```

Here by contrast is an equivalent EXTEND formulation:

```
EXTEND S { SNO } : { PCT := COUNT ( ‼SP ) }
```

---

[16] Note that no COALESCE is needed in this example—but why not?

[17] Not to mention the fact that SUMMARIZE involves a syntactic construct that looks a bit like an aggregate operator invocation but isn't one—which (as pointed out earlier) is a good reason why it might be better to dispense with SUMMARIZE altogether.

(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 this example suggests, EXTEND 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 Revisited." In each case, I show an EXTEND formulation first, followed by an equivalent SUMMARIZE formulation; equivalent SQL formulations are left as an exercise.

Example 6: For each supplier, get supplier details and total shipment quantity, taken over all shipments for the supplier in question.

```
EXTEND S : { TOTQ := SUM ( !!SP , QTY ) }
```

SUMMARIZE analog:

```
S JOIN ( SUMMARIZE SP PER ( S { SNO } ) : { TOTQ := SUM ( QTY ) } )
```

Example 7: For each part supplied, get part details and total, maximum, and minimum shipment quantity, taken over all shipments for the part in question.

```
EXTEND ( P MATCHING SP ) : { TOTQ := SUM ( !!SP , QTY ) ,
                             MAXQ := MAX ( !!SP , QTY ) ,
                             MINQ := MIN ( !!SP , QTY ) }
```

SUMMARIZE analog:

```
P JOIN ( SUMMARIZE SP BY { PNO } : { TOTQ := SUM ( QTY ) ,
                                     MAXQ := MAX ( QTY ) ,
                                     MINQ := MIN ( QTY ) } )
```

Note the use of the multiple forms of EXTEND and SUMMARIZE 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 : { TOTQ  := SUM ( !!SP , QTY ) ,
             GTOTQ := SUM (  SP , QTY ) }
```

Result:

SNO	TOTQ	GTOTQ
S1	1300	3100
S2	700	3100
S3	200	3100
S4	900	3100
S5	0	3100

SUMMARIZE analog:

```
JOIN { S , SUMMARIZE SP PER ( S { SNO } ) : { TOTQ  := SUM ( QTY ) } ,
           SUMMARIZE SP BY { } : { GTOTQ := SUM ( QTY ) } }
```

Example 9: Let city *c* be such that some supplier in *c* supplies some part in *c*. For each such city *c*, get *c* and the maximum and minimum shipment quantities for all shipments for which the supplier and part are both in city *c*.

```
WITH ( temp := JOIN { S , SP , P } ) :
EXTEND temp { CITY } : { MAXQ := MAX ( ‼temp , QTY ) ,
                         MINQ := MIN ( ‼temp , QTY ) }
```

SUMMARIZE analog:

```
WITH ( temp := JOIN { S , SP , P } ) :
SUMMARIZE temp BY { CITY } : { MAXQ := MAX ( QTY ) ,
                               MINQ := MIN ( QTY ) }
```

The point of this rather contrived example is to illustrate the usefulness of WITH, in connection with "SUMMARIZE-type" EXTENDs in particular, in avoiding the need to write out some possibly lengthy subexpression several times. *Note:* This book generally has little to say about performance matters, but I think it's worth pointing out that we would surely expect the system, in examples like this one, to evaluate the pertinent subexpression once instead of several times. In other words, the use of WITH can be one of those nice win-win situations that are good for both the user and the DBMS.

GROUP, UNGROUP, AND RELATION VALUED ATTRIBUTES

Recall from Chapter 2 that relations with relation valued attributes (RVAs for short) are legal. Refer to Fig. 7.1, which shows relations R1 and R4 from Figs. 2.1 and 2.2, respectively, in that chapter; note that R4 has an RVA and R1 doesn't, but the two relations clearly represent the same information.

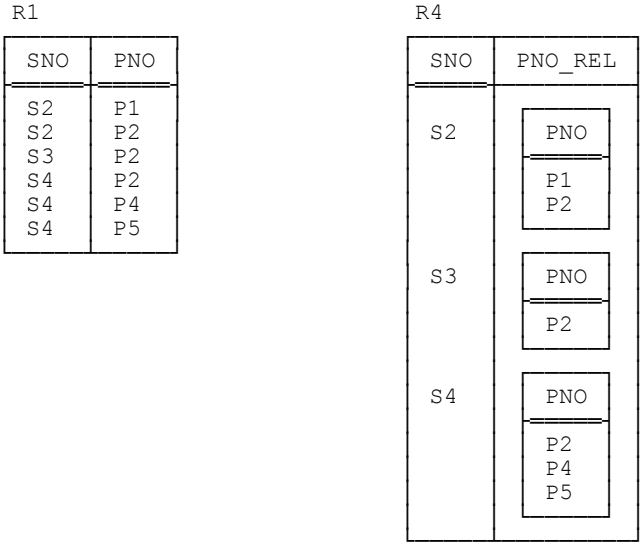


Fig. 7.1: Relations R1 and R4 from Figs. 2.1 and 2.2 in Chapter 2

Note: For the record, the type of relation R4 in the figure is RELATION {SNO CHAR , PNO_REL RELATION {PNO CHAR}}.

Now, we obviously 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—as in fact was previously noted in the answer to Exercise 3.7 in Chapter 3—given the relations shown in Fig. 7.1, the expression

```
R1 GROUP { PNO } AS PNO_REL
```

will produce R4, and the expression

```
R4 UNGROUP PNO_REL
```

will produce R1.

By the way, it's worth noting that the following expression—

```
EXTEND R1 { SNO } : { PNO_REL := ‼R1 }
```

—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.[18]

And by the way again: If R4 includes a 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 G) or the book *Databases, Types, and the Relational Model: The Third Manifesto* (again, see Appendix G), by Hugh Darwen and myself.

The SQL counterparts to GROUP and UNGROUP are quite complex, and I don't propose to go into details here. However, I will at least show approximate SQL analogs (?) of the **Tutorial D** examples above.[19] First GROUP:

```
SELECT DISTINCT X.SNO ,
                CAST ( TABLE ( SELECT Y.PNO
                               FROM   R1 AS Y
                               WHERE  Y.SNO = X.SNO )
                AS ROW ( PNO VARCHAR(6) ) MULTISET ) AS PNO_REL
FROM   R1 AS X
```

Now UNGROUP:

```
SELECT SNO , X.PNO
FROM   R4 , UNNEST ( PNO_REL ) AS X ( PNO )
```

Note: I can't help pointing out a certain irony in SQL's version of the GROUP example. As you can see, the SQL expression in that example involves a subquery in (a CAST invocation within) the SELECT clause. Of course, a subquery denotes a table; in SQL, however, that table is often coerced—in the context of a SELECT clause in particular—to a single row, or more frequently to a single column value from within that single row. In the case at hand, however, we don't want any such coercion; so we have to tell SQL explicitly, by means of the TABLE operator,[20] not to do what it normally would do (by default, as it were) in such a context.

RVAs Make Outer Join Unnecessary

There are several further points worth making in connection with relation valued attributes. First of all, they make outer join unnecessary! Second, it turns out they're sometimes necessary even in base relvars. Third, they're conceptually necessary anyway in order to support relational

---

[18] As a matter of fact, UNGROUP can also be defined in terms of EXTEND, though the details are rather more complicated than they are for GROUP.

[19] I do *not* guarantee that these SQL expressions are completely legal, or adequate even if they're legal—the relevant portions of the SQL standard are extremely difficult to understand. If you seriously want to know more about SQL's TABLE and UNNEST operators, then I recommend Hugh Darwen's book *SQL: A Comparative Survey* (see Appendix G).

[20] We've met this operator before—see the discussion of table equality comparisons in the section "Tables in SQL" in Chapter 3.

comparison operations. And fourth, they make it desirable to support certain additional aggregate operators. I'll elaborate on each of these points in turn.

I'll begin by showing a slightly more complicated example of an RVA. Consider the following **Tutorial D** expression:

```
EXTEND S : { PQ := !!SP }
```

Suppose we evaluate this expression and assign the result to a relvar SPQ. A sample value for SPQ, corresponding to our usual sample values for relvars S and SP, is shown (in outline) in Fig. 7.2 below. Attribute PQ is relation valued.

SNO	SNAME	STATUS	CITY	PQ
S1	Smith	20	London	PNO: P1, P2, .., P6 / QTY: 300, 200, ..., 100
S2	Jones	10	Paris	PNO: P1, P2 / QTY: 300, 400
..	.....	..	.....	....
S5	Adams	30	Athens	PNO / QTY

Fig. 7.2: Relvar SPQ (sample value)

Now consider the following SQL expression:

```
SELECT SNO , SNAME , STATUS , CITY , PNO , QTY
FROM   S NATURAL LEFT OUTER JOIN SP
```

The result of evaluating this expression is shown (again in outline) in Fig. 7.3.

SNO	SNAME	STATUS	CITY	PNO	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
..	.....	..	.....	..	...
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
..	.....	..	.....	..	...
S5	Adams	30	Athens	▒▒	▒▒▒

Fig. 7.3: Left outer join of S and SP (sample value)

Observe now that with our usual sample values, the set of shipments for supplier S5 is empty, and that:

- In Fig. 7.2, that empty set of shipments is represented by an empty set.
- In Fig. 7.3, by contrast, that empty set is represented by nulls (indicated by shading in the figure).

To represent an empty set by an empty set seems like such an obviously good idea! In fact, as I said earlier, *there would be no need for outer join at all* if RVAs were properly supported. Thus, one advantage of RVAs is that they deal more elegantly with the problem that outer join is intended to solve than outer join itself does—and I'm tempted to say that this fact all by itself, even if there were no other advantages, is a big argument in favor of RVAs.

At the risk of laboring the obvious, I'd like to say too that if there aren't any shipments for supplier S5, it means, to repeat, that *the set of shipments for supplier S5 is empty* (and that's exactly what the relation in Fig. 7.2 says). It certainly doesn't mean that supplier S5 supplies some unknown part in some unknown quantity; and yet *unknown* is—and in fact was originally and explicitly intended to be—the way null is usually interpreted. So Fig. 7.3 not only involves nulls (which as we saw in Chapter 4 are bad news anyway, for all kinds of reasons), it actually misrepresents the semantics of the situation.

RVAs in Base Relvars

Let's look at some typical operations involving relvar SPQ (Fig. 7.2). Consider first the following queries:

- Get supplier numbers for suppliers who supply part P2.

```
( ( SPQ UNGROUP PQ ) WHERE PNO = 'P2' ) { SNO }
```

- Get part numbers for parts supplied by supplier S2.

```
( ( SPQ WHERE SNO = 'S2' ) UNGROUP PQ ) { PNO }
```

As you can see, the natural language versions of these two queries are symmetric, but the **Tutorial D** formulations on the RVA design (Fig. 7.2) aren't. By contrast, **Tutorial D** formulations of the same queries against our usual (non RVA) design *are* symmetric, as well as being simpler than their RVA counterparts:

```
( SP WHERE PNO = 'P2' ) { SNO }

( SP WHERE SNO = 'S2' ) { PNO }
```

In fact, the queries on the RVA design effectively involve mapping that design to the non RVA design anyway (that's what the UNGROUPs do).

Similar remarks apply to updates and constraints. For example, suppose we need to update the database to show that supplier S2 supplies part P5 in a quantity of 500. Here are **Tutorial D** formulations on (a) the non RVA design, (b) the RVA design:

```
a.  INSERT SP
           RELATION { TUPLE { SNO 'S2' , PNO 'P5' , QTY 500 } } ;

b.  UPDATE SPQ WHERE SNO = 'S2' :
         { INSERT PQ
                  RELATION { TUPLE { PNO 'P5' , QTY 500 } } } ;
```

Once again, the natural language requirement is stated in a symmetric fashion; its formulation in terms of the non RVA design is symmetric too; but its formulation in terms of the RVA design isn't (in fact, it's quite cumbersome). And, of course, the reason for this state of affairs is that the RVA design itself is asymmetric—in effect, it regards parts as subordinate to suppliers, instead of giving parts and suppliers equal weight, as it were.

Examples like the ones discussed above tend to suggest that RVAs in base relvars are probably a bad idea (certainly relvar SPQ in particular isn't very well designed). But this position might better be seen as a guideline, not an absolute limitation, because in fact there are cases—comparatively rare ones perhaps—where a base relvar with an RVA is exactly the right design. A sample value for such a relvar (SIBLING) is shown in Fig. 7.4. The intended interpretation is that the persons identified within any given PERSONS value are all siblings of one another, and have no other siblings. Thus, Amy and Bob are siblings; Cal, Don, and Eve are siblings; and Fay is an only child. Note that the relvar has just one attribute (an RVA) and three tuples. Note too that the sole key involves an RVA.

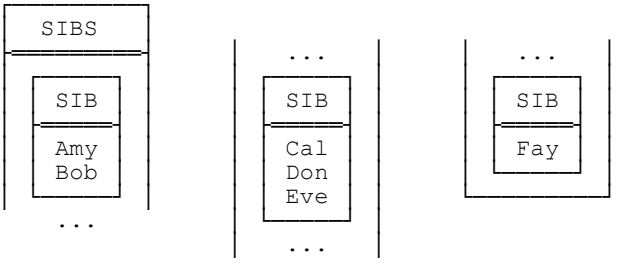


Fig. 7.4: Relvar SIBLING (sample value)

Note: It's important to understand that no non RVA relation exists that represents exactly the same information, no more and no less, as the relation shown in Fig. 7.4 does. In particular, if we ungroup that relation as follows—

```
SIBLING UNGROUP SIBS
```

—we lose the information as to who's a sibling of whom.

RVAs Are Necessary for Relational Comparisons

Consider once again this example from the section on image relations earlier in this chapter:

```
S WHERE ( ‼SP ) { PNO } = P { PNO }
```

("suppliers who supply all parts"). Clearly, the boolean expression in the WHERE clause here involves a relational comparison (actually an equality comparison). Recall now from Chapter 6 that an expression of the form *r* WHERE *bx* denotes a restriction as such only if *bx* is a restriction condition, and *bx* is a restriction condition if and only if every attribute reference in *bx* identifies some attribute of *r* and there aren't any relvar references. In the example, therefore, the boolean expression isn't a genuine restriction condition, because (a) it involves some references to attributes that aren't attributes of S and (b) it also involves some relvar references (viz., to relvars SP and P). In fact, the example overall is really shorthand for something that might look like this:

```
WITH ( t1 := EXTEND S  : { X := ( ‼SP ) { PNO } } ,
       t2 := EXTEND t1 : { Y := P { PNO } } ) :
t2 WHERE X = Y
```

Now the boolean expression in the WHERE clause (in the last line) is indeed a genuine restriction condition. Observe, however, that attributes X and Y are both RVAs. As the example suggests, therefore, RVAs are always involved, at least implicitly, whenever relational comparisons are performed.

Aggregate Operators

Consider relvar SPQ again, with sample value as shown in Fig. 7.2. Attribute PQ is relation valued. And just as it makes sense (and is useful) to define, e.g., numeric aggregate operators such as SUM on numeric attributes, so it makes sense, and is useful, to define relational aggregate operators on relation valued attributes. For example, the following expression returns the union of all of the relations currently appearing as values of attribute PQ in relvar SPQ:

```
UNION ( SPQ , PQ )
```

Or equivalently (but why exactly is this equivalent?):

```
UNION ( SPQ { PQ } )
```

Tutorial D supports the following relation valued aggregate operators: UNION, D_UNION, and INTERSECT. And SQL has analogs of UNION and INTERSECT (though not D_UNION); however, they're called, not UNION and INTERSECT as one might reasonably have expected, but FUSION and INTERSECTION [*sic*], respectively. (It would be very naughty of me to suggest that if union is called FUSION, then intersection ought surely to be called FISSION, so I won't.)

"WHAT IF" QUERIES

"What if" queries are a frequent requirement; they're used to explore the effect of making certain changes to the database without actually having to make (and subsequently unmake, possibly) the changes in question. Here's an example ("What if parts in Paris were in Nice instead and their weight was doubled?"):

```
EXTEND P WHERE CITY = 'Paris' :
             { CITY := 'Nice' , WEIGHT := 2 * WEIGHT }
```

As you can see, this expression makes use of EXTEND once again. This time, however, the target attributes in the assignments in braces aren't "new" attributes, as they normally are for EXTEND; instead, they're attributes already existing in the specified relation. What the expression does is this: It yields 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 weight is double that in tuple *t1* and the city is Nice, not Paris. In other words, the expression overall is shorthand for the following:

```
WITH ( t1 := P WHERE CITY = 'Paris' ,
       t2 := EXTEND t1 : { NC := 'Nice' , NW := 2 * WEIGHT } ,
       t3 := t2 { ALL BUT CITY , WEIGHT } ) :
t3 RENAME { NC AS CITY , NW AS WEIGHT }
```

Aside: Here for purposes of comparison is an SQL analog of the foregoing **Tutorial D** expression:

```
WITH t1 AS ( SELECT P.*
             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
```

End of aside.

Here now for the record is a formal definition for this alternative version of EXTEND:

Definition: Let relation *r* have an attribute called *A*. Then (and only then) the expression EXTEND *r* : {*A* := *exp*} denotes an *extension* of *r*, and it returns the relation with heading the same as that of *r* and body the set of all tuples *t* such that *t* is derived from a tuple of *r* by replacing the value of *A* by a value that's computed by evaluating the expression *exp* on that tuple of *r*.[21]

And now I can take care of some unfinished business from Chapter 5. In that chapter, I said the relational UPDATE operator was shorthand for 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:

```
P := ( P WHERE CITY ≠ 'Paris' )
       UNION
     ( EXTEND ( P WHERE CITY = 'Paris' ) :
                      { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } ) ;
```

[21] Note, therefore, that relation *r* isn't exactly being "extended" in the usual sense, so it might be nice to find a better keyword than EXTEND for the purpose.

Alternatively, recall from Chapter 5 that "updating relvar *R*" really means we're replacing the relation *r1* that's the original value of *R* by another relation *r2*, where *r2* is computed as (*r1* MINUS *s1*) UNION *s2* for certain relations *s1* and *s2*. In the case at hand, using "$\stackrel{\text{def}}{=}$" as in Chapter 6 to denote "is defined as," we have:

```
s1  def=  P WHERE CITY = 'Paris'

s2  def=  EXTEND ( P WHERE CITY = 'Paris' ) :
                    { CITY := 'Nice' , WEIGHT := 2 * WEIGHT } )
```

Thus, the expanded form of the UPDATE becomes:

```
P := ( P MINUS s1 ) UNION s2 ;
```

Note: Actually, we could safely replace MINUS and UNION here by I_MINUS and D_UNION, respectively, and we could safely drop the parentheses. (In both cases, why?)[22]

A NOTE ON RECURSION

Consider the following lightly edited extract from Exercise 5.16 in Chapter 5:

> The well known *bill of materials* application involves a relvar—PP, say—showing which parts contain which parts as immediate components. Of course, immediate components are themselves parts, and they can have further immediate components of their own.

Fig. 7.5 shows (a) a sample relation value for that relvar PP and (b) the relation that's the *transitive closure* of that sample relation value,[23] shown as the corresponding value of a relvar TC. The predicates are as follows:

- PP: *Part PX contains part PY as an immediate component.*
- TC: *Part PX contains part PY as a component at some level, but not necessarily as an immediate component.*

For example, if part P1 contains part P2 as a component and part P2 contains part P4 as a component, then certainly part P1 contains part P4 as a component "at some level."

---

[22] Alternatively, we could replace the entire assignment by the *multiple* assignment "DELETE P *s1*, INSERT P *s2*;" (see Chapter 8).

[23] Nothing to do with the closure property of the relational algebra.

PP

PX	PY
P1	P2
P1	P3
P2	P4
P3	P4
P4	P5
P5	P6

TC

PX	PY
P1	P2
P1	P3
P2	P4
P3	P4
P4	P5
P5	P6
P1	P4
P1	P5
P1	P6
P2	P5
P2	P6
P4	P6
P3	P5
P3	P6

Fig. 7.5: Relvars PP and TC (sample values)

Given a relation value *pp* for relvar PP, the relation value *tc* that's the transitive closure of *pp* can be defined as follows (observe that the definition involves a recursive reference to *tc*):

Definition: The pair (*px*,*py*) appears in *tc* if and only if:

a. It appears in *pp*, or

b. There exists some *pz* such that the pair (*px*,*pz*) appears in *pp* and the pair (*pz*,*py*) appears in *tc*.

In other words, if we think of *pp* as representing a directed graph, with a node for each part and an edge from each node to each corresponding immediate component node, then (*px*,*py*) appears in the transitive closure if and only if there's a path in that graph from node *px* to node *py*.

Aside: In practice relvar PP would probably have a QTY attribute as well, showing how many instances of the immediate component part PY are needed to make one instance of part PX, and we would probably want to compute not just the transitive closure as such, but also the total number of instances of part PY needed to make one instance of part PX: the *gross requirements* problem. I ignore this refinement here for simplicity. *End of aside.*

It's also possible to define the transitive closure by means of an iterative procedure:

```
tc := pp ;
do until tc reaches a "fixpoint" ;
   WITH ( t1 := pp RENAME { PY AS PZ } ,
          t2 := tc RENAME { PX AS PZ } ,
          t3 := ( t1 JOIN t2 ) { PX , PY } ) :
   tc := tc UNION t3 ;
end ;
```

Loosely speaking, this code works by repeatedly forming an intermediate result consisting of the union of (a) the previous intermediate result and (b) a relation computed on the current iteration. The process is repeated until that intermediate result reaches a *fixpoint* (i.e., until it ceases to grow). *Note:* It's easy to see the code is very inefficient!—in effect, each iteration repeats the entire computation of the previous one. In fact, it's little more than a direct implementation of the original (recursive) definition. However, it could clearly be made more efficient if desired. Similar remarks apply to all of the code samples in the present section.

Turning now to **Tutorial D,** we could define a recursive operator (TCLOSE) to compute the transitive closure as follows:[24]

```
OPERATOR TCLOSE ( XY RELATION { X ... , Y ... } )
             RETURNS RELATION { X ... , Y ... } ;
   RETURN ( WITH ( t1 := XY RENAME { Y AS Z } ,
                   t2 := XY RENAME { X AS Z } ,
                   t3 := ( t1 JOIN t2 ) { X , Y } ,
                   t4 := XY UNION t3 ) :
            IF t4 = XY THEN t4     /* unwind recursion     */
               ELSE TCLOSE ( t4 )  /* recursive invocation */
            END IF ) ;
END OPERATOR ;
```

Now, e.g., the expression TCLOSE(*pp*) will return the transitive closure of *pp*. Hence, for example, the expression

```
( TCLOSE ( PP ) WHERE PX = 'P1') { PY }
```

will give the "bill of materials" for part P1, and the expression

```
( TCLOSE ( PP ) WHERE PY = 'P6') { PX }
```

will give the "where used" list for part P6. *Note:* Computing the bill of materials for a given part is sometimes referred to as *part explosion*; likewise, computing the "where used" list for a given part is referred to as *part implosion*.

Now, SQL too supports what it calls "recursive queries." Here's an SQL expression to compute the transitive closure of PP:

---

[24] Actually **Tutorial D** goes beyond the relational algebra as conventionally understood in that it provides TCLOSE as a built in operator. I show it as a user defined operator here just to show how recursive operators might be defined in **Tutorial D**.

```
WITH RECURSIVE TC ( PX , PY ) AS
   ( SELECT PP.PX , PP.PY
     FROM   PP
     UNION  CORRESPONDING
     SELECT PP.PX , TC.PY
     FROM   PP , TC
     WHERE  PP.PY = TC.PX )

SELECT PX , PY
FROM   TC
```

As you can see, this expression too is very close to being a direct transliteration of the original recursive definition.

Note: This book deliberately has very little to say about commercial SQL products. However, I'd like to offer a brief remark here regarding Oracle specifically. As you might know, Oracle has had some recursive query support for many years. By way of example, the query "Explode part P1" can be expressed in Oracle as follows:

```
SELECT  LEVEL , PY
FROM    PP
CONNECT BY PX = PY
START   WITH PX = 'P1'
```

I don't want to explain in detail how this expression is evaluated—but I do want to show the result it produces, given the sample data of Fig. 7.5. Here it is:

LEVEL	PY
1	P2
2	P4
3	P5
4	P6
1	P3
2	P4
3	P5
4	P6

Note carefully that this result *isn't a relation* (and the relational closure property has thereby been violated). First of all, it contains some duplicate rows; for example, the row (2,P4) appears twice. More important, those duplicate rows are *not* duplicate rows as we usually understand them in SQL; that is, they aren't just "saying the same thing twice," as I put it in Chapter 4. To spell the point out, one of those two (2,P4) rows reflects the path in the graph from part P1 to part P4 *via part P2*; the other reflects the path in the graph from part P1 to part P4 *via part P3*. Thus, if we deleted one of those rows, we would lose information.

Aside: Actually the same kind of problem can arise in the SQL standard if the recursive query in question uses UNION ALL instead of UNION DISTINCT—as in practice such queries very typically do. Further details are beyond the scope of this book; however, if you try to code the gross requirements problem in SQL you might see for yourself why it's tempting, at least superficially, to use UNION ALL. *End of aside.*

Note too that in addition to the foregoing violations, *the ordering of the rows* in the Oracle result is significant as well. For example, the reason we know the first (2,P4) row corresponds to the path from P1 to P4 via P2 specifically is because it immediately follows the row corresponding to the path from P1 to its immediate component P2. Thus, if we reordered the rows, again we would lose information.

Cycles

Consider Fig. 7.5 once again. Suppose the relation *pp* shown as a value for relvar PP in that figure additionally contained a tuple representing, say, the pair (P5,P1). Then there would be a cycle in the data (actually two cycles, one involving parts P1-P2-P4-P5-P1 and one involving parts P1-P3-P4-P5-P1). In the case of bill of materials, such cycles should presumably not be allowed to occur, since they make no sense. Sometimes, however, they do make sense; the classic example is a transportation network, in which there are routes from, say, New York (JFK) to London (LHR), London to Paris (CDG), and Paris back to New York again (as well as routes in all of the reverse directions, of course).

Now, the existence of a cycle in the data has no effect on the transitive closure as such. However, it does have the potential to cause an infinite loop in certain kinds of processing. For example, a query to find travel routes from New York to Paris might—if we're not careful—produce results as follows:

```
JFK - LHR - CDG
JFK - LHR - JFK - LHR - CDG
JFK - LHR - JFK - LHR - JFK - LHR - CDG
etc., etc. etc.
```

Of course, it might at least be possible to formulate the query in such a way as to exclude segments in which the destination city is JFK (since we certainly don't want a route that takes us back to where we started). But even this trick will still allow routes such as:

```
JFK - ORD - LHR - ORD - LHR - ORD - LHR - ... - CDG
```

(ORD = Chicago). Moreover, it still won't prevent an infinite loop. Now, we might prevent the infinite loop as such by rejecting routes involving, say, more than four segments; but under such a scheme we could still get, e.g., the route JFK-ORD-LHR-ORD-CDG. Clearly, what we need is a more general mechanism that will allow the query to recognize when a given node in the graph

has previously been visited. And SQL does in fact include a feature, the CYCLE clause, that can be used in recursive queries to achieve such an effect. The specifics are a little complicated, and I don't want to get into details here; suffice it to say that the CYCLE clause provides a means of tagging nodes (i.e., rows) as they're visited, and then stopping the recursion if a tagged node is subsequently encountered again. For more details, I refer you to the standard document itself.

WHAT ABOUT ORDER BY?

The last topic I want to address in this chapter is ORDER BY (just ORDER, in **Tutorial D**). Now, despite the title of this chapter, ORDER BY isn't actually part of the relational algebra; in fact, as I said in Chapter 1, 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). *Note:* 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[25] (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 ORDER ( ASC SNO )
```

```
SELECT DISTINCT S.*
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 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.[26] Second, it's not a function. All of the operators of the relational algebra described in this book—in fact, read-only operators in general, as that term is usually understood—*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`

[25] In particular, therefore, it can't appear in a view definition—despite the fact that at least one well known SQL product allows it to! *Note:* It's sometimes suggested—and, sadly, the SQL standard now explicitly supports this idea—that ORDER BY is needed in connection with what are called *quota queries*, but it isn't (see Exercise 7.14).

[26] I suppose SQL might claim it *is* defined for rows, as opposed to tuples (again, see Chapter 3).

- `S5 , S4 , S1 , S3 , S2`

Note: It would be remiss of me not to mention in passing that although the operators of the relational algebra described in this book are indeed functions, most of them have counterparts in SQL that aren't. 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 second of these); these 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; indeed, it *isn't* defined, at least as far as the standard is concerned. 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 the implications of this state of affairs.

EXERCISES

7.1 For each of the following **Tutorial D** expressions on the suppliers-and-parts database, give both (a) an SQL analog and (b) an informal interpretation of the expression (i.e., a corresponding predicate) in natural language. Also show the result of evaluating the expression, given our usual sample values for relvars S, P, and SP.

a. `S MATCHING ( SP WHERE PNO = 'P2' )`

b. `S NOT MATCHING ( SP WHERE PNO = 'P2' )`

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 : { TAG := 'Supplier' }`

g.
```
EXTEND ( S MATCHING ( SP WHERE PNO = 'P2' ) ) :
                          { TRIPLE_STATUS := 3 * STATUS }
```

h. `EXTEND ( P JOIN SP ) : { SHIPWT := WEIGHT * QTY }`

i. `EXTEND P : { GMWT := WEIGHT * 454 , OZWT := WEIGHT * 16 }`

j. `EXTEND P : { SCT := COUNT ( !!SP ) }`

k. `EXTEND S : { NP := COUNT ( ( SP RENAME { SNO AS X } ) WHERE X = SNO ) }`

l. `SUMMARIZE S BY { CITY } : { SUM_STATUS := SUM ( STATUS ) }`

m.
```
SUMMARIZE ( S WHERE CITY = 'London' ) PER ( TABLE_DEE ) :
                                                { N := COUNT ( SNO ) }
```

Note: **Tutorial D** allows the PER specification to be omitted from a SUMMARIZE, in which case PER (TABLE_DEE)—equivalently, BY { }—is assumed by default. The foregoing SUMMARIZE could therefore be simplified slightly, thus:

```
SUMMARIZE ( S WHERE CITY = 'London' ) : { N := COUNT ( SNO ) }
```

n. `EXTEND SP WHERE SNO = 'S1' : { SNO := 'S7' , QTY := 0.5 * QTY }`

7.2 In what circumstances if any are *r1* MATCHING *r2* and *r2* MATCHING *r1* equivalent?

7.3 Show that RENAME isn't primitive.

7.4 Give an expression involving EXTEND instead of SUMMARIZE that's logically equivalent to the following:

```
SUMMARIZE SP PER ( S { SNO } ) : { NP := COUNT ( PNO ) }
```

7.5 Consider the following **Tutorial D** expressions. Which if any are equivalent to which of the others? Show an SQL analog in each case.

a. `SUMMARIZE r PER ( r { } ) : { CT := SUM ( 1 ) }`

b. `SUMMARIZE r PER ( TABLE_DEE ) : { CT := SUM ( 1 ) }`

c. `SUMMARIZE r BY { } : { CT := SUM ( 1 ) }`

d. `EXTEND TABLE_DEE : { CT := COUNT ( r ) }`

7.6 Consider the relational aggregate operators UNION and INTERSECT. What do you think these operators should return if their argument (a set or bag of relations) happens to be empty?

7.7 Let relation R4 in Fig. 7.1 denote the current value of some relvar. If R4 is as described in Chapter 2, what's the predicate for that relvar?

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
```

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. Get city names for cities in which at least one supplier or part is located, but not both.

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

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

7.10 Let relation *pp* be as defined in the section "A Note on Recursion" and let TCLOSE be the transitive closure operator. What does the expression TCLOSE(TCLOSE(*pp*)) denote?

7.11 Given our usual sample values for the suppliers-and-parts database, what does the following **Tutorial D** expression denote?

```
EXTEND S : { PNO_REL := ( ‼SP ) { PNO } }
```

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' } } } ;
```

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.

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 quota is two. Give

Tutorial D and SQL formulations of this query. Given our usual data values, what exactly do these formulations return?

7.15 Using the **Tutorial D** SUMMARIZE operator, how would you deal with the query "For each supplier, get the supplier number and the sum of *distinct* shipment quantities for shipments by that supplier"?

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 */
```

—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.

7.17 What does the following **Tutorial D** expression mean?

```
S WHERE ( ‼(‼SP) ) { PNO } = P { PNO }
```

7.18 Is there a logical difference between the following two **Tutorial D** expressions? If so, what is it?

```
EXTEND TABLE_DEE : { NSP := COUNT (  SP ) }

EXTEND TABLE_DEE : { NSP := COUNT ( ‼SP ) }
```

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?

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?

7.21 What's the logical difference, if any, between the following SQL expressions?

```
SELECT COUNT ( * ) FROM S

SELECT SUM ( 1 ) FROM S
```

7.22 By definition, ORDER BY (or just ORDER, in **Tutorial D**) can't appear in a relational expression (or table expression, rather, in SQL). So where can it appear?

7.23 Why don't the tables shown on pages 231 and 249 (Fig. 7.3) have any doubly underlined columns?

ANSWERS

Here first are answers to certain exercises that were stated inline in the body of the chapter. In one, we were given relvars as follows—

```
S   { SNO }         /* suppliers                */
SP  { SNO , PNO }   /* supplier supplies part   */
PJ  { PNO , JNO }   /* part is used in project  */
J   { JNO }         /* projects                 */
```

—and we were asked for a SQL formulation of the query "Get all (*sno*,*jno*) pairs such that *sno* appears in S, *jno* appears in J, and supplier *sno* supplies all parts used in project *jno*." A possible formulation is as follows:

```
SELECT SX.SNO , JX.JNO
FROM   S AS SX , J AS JX
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P AS PX
       WHERE  EXISTS
            ( SELECT *
              FROM   PJ AS PJX
              WHERE  PJX.PNO = PX.PNO
              AND    PJX.JNO = JX.JNO )
       AND    NOT EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.PNO = PX.PNO
              AND    SPX.SNO = SX.SNO ) )
```

Note: For a detailed discussion of how to tackle complicated SQL queries like this one, see Chapter 11.

Another inline exercise asked what happens if (a) *r1* and *r2* are relations with no attribute names in common, (b) *r2* is empty, (c) we form the product *r1* TIMES *r2*, and finally (d) we divide that product by *r2*. *Answer:* It should be clear that the product is empty, and hence the final result is empty too (it has the same heading as *r1*, but of course it isn't equal to *r1*, in general). Do note, however, that dividing by an empty relation isn't an error (it's not like dividing by zero in arithmetic). See the answer to Exercise 6.2 in Chapter 2.

In another inline exercise, we were given this expression—

```
SELECT SNO , SUM ( QTY ) AS TOTQ
FROM   SP
GROUP  BY SNO
HAVING SUM ( QTY ) > 250
```

—as an SQL formulation of the query "For each supplier, get the supplier number and corresponding total shipment quantity, but only where that total quantity is greater than 250." Then we were told that (a) if we wanted that total quantity to be less than, instead of greater than, 250, and so (b) we replaced that ">" by "<" in the last line, then (c) the resulting expression would *not* do the trick. But why not? *Answer:* Consider supplier S5, who (given our usual sample data) currently supplies no parts at all. The total quantity for that supplier is zero, and so supplier S5 should be represented in the result. In SQL, however, the total quantity for such a supplier is considered to be null, not zero; the comparison in the HAVING clause will therefore evaluate to UNKNOWN, not TRUE, and so that supplier won't be represented in the result after all. *Note:* Of course, the obvious fix for this problem is to replace SUM (QTY) in the HAVING clause by COALESCE (SUM (QTY), 0). *Subsidiary exercise:* Do we also need to apply that same fix in the SELECT clause? If not, why not?

7.1 Throughout these answers, I show SQL expressions that aren't necessarily direct transliterations of their **Tutorial D** (i.e., algebraic) counterparts but are, rather, "more natural" formulations of the query in SQL terms. The solutions aren't necessarily unique. *Note:* This latter remark applies to many of the code solutions throughout the rest of this book, and I won't bother to make it again.

a. SQL analog:

```
SELECT *
FROM   S
WHERE  SNO 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 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, that's 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, 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 DISTINCT SNO , S.* , 3 * STATUS AS TRIPLE_STATUS
FROM   S NATURAL JOIN SP
WHERE  PNO = 'P2'
```

Predicate: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, supplies part P2, and has TRIPLE_STATUS equal to 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

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.*

PNO	PNAME	COLOR	WEIGHT	CITY	SNO	QTY	SHIPWT
P1	Nut	Red	12.0	London	S1	300	3600.0
P1	Nut	Red	12.0	London	S2	300	3600.0
P2	Bolt	Green	17.0	Paris	S1	200	3400.0
P2	Bolt	Green	17.0	Paris	S2	400	6800.0
P2	Bolt	Green	17.0	Paris	S3	200	3400.0
P2	Bolt	Green	17.0	Paris	S4	200	3400.0
P3	Screw	Blue	17.0	Oslo	S1	400	6800.0
P4	Screw	Red	14.0	London	S1	200	2800.0
P4	Screw	Red	14.0	London	S4	300	4200.0
P5	Cam	Blue	12.0	Paris	S1	100	1200.0
P5	Cam	Blue	12.0	Paris	S4	400	4800.0
P6	Cog	Red	19.0	London	S1	100	1900.0

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 , SUM ( STATUS ) AS SUM_STATUS
FROM   S
GROUP  BY CITY
```

Predicate: *The sum of status values for suppliers in city CITY is SUM_STATUS.*

CITY	SUM_STATUS
London	40
Paris	40
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 (see the answer to Exercise 7.23).

n. SQL analog:

```
SELECT 'S7' AS SNO , PNO , QTY * 0.5 AS QTY
FROM   SP
WHERE  SNO = 'S1'
```

Predicate: *SNO is S7 and supplier S1 supplies part PNO in quantity twice QTY.*

SNO	PNO	QTY
S7	P1	150
S7	P2	100
S7	P3	200
S7	P4	100
S7	P5	50
S7	P6	50

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 *r1* JOIN *r2* (and this latter expression reduces in turn to *r1* INTERSECT*r2*).

7.3 RENAME isn't primitive because (for example) the expressions

```
S RENAME { CITY AS SCITY }
```

and

```
( EXTEND S : { SCITY := CITY } ) { ALL BUT CITY }
```

are equivalent. *Note:* Possible appearances to the contrary notwithstanding, EXTEND isn't primitive either—it can be defined in terms of join (at least in principle), as is shown in the book *Databases, Types, and the Relational Model: The Third Manifesto,* by Hugh Darwen and myself (see Appendix G).

7.4 `EXTEND S { SNO } : { NP := COUNT ( ‼SP ) }`

7.5 You can determine which of the expressions are equivalent to which by inspecting the following results of evaluating them. Note that the "summary" SUM(1), evaluated over *n* tuples, is equal to *n*. (Even if *n* is zero! SQL, of course, would say the result is null in such a case.)

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, as are b. and d., but a. and b. aren't. SQL analogs:

a.
```
SELECT COUNT ( * ) AS CT
FROM   r
EXCEPT CORRESPONDING
SELECT 0 AS CT
FROM   r
```

b.
```
SELECT COUNT ( * ) AS CT
FROM   r
```

c. Same as a.

d. Same as b.

7.6 They return, respectively, the empty relation and the universal relation (of the applicable type in each case). *Note:* The universal relation of type RELATION *H* is the relation of that type that contains all possible tuples of type TUPLE *H*. The implementation might reasonably want to outlaw invocations of INTERSECT on an empty argument (at least if those invocations really need the result to be materialized).

Just to remind you, SQL's analogs of the UNION and INTERSECT aggregate operators are called FUSION and INTERSECTION, respectively. If their arguments are empty, they both return null. Otherwise, they return a result as follows (these are direct quotes from the standard; *T* is the table over which the aggregation is being done). First FUSION:

> [The] result is the multiset *M* such that for each value *V* in the element type, including the null value [*sic*], the number of elements of *M* that are identical to *V* is the sum of the number of identical copies of *V* in the multisets that are the values of the column in each row of *T*.

(The "element type" is the type of the elements of the multisets in the argument column.) Now INTERSECTION:

> [The] result is a multiset *M* such that for each value *V* in the element type, including the null value, the number of duplicates of *V* in *M* is the minimum of the number of identical copies of *V* in the multisets that are the values of the column in each row of *T*.

Note the asymmetry, incidentally: In SQL, INTERSECTION (and INTERSECT) are defined in terms of MIN, but FUSION (and UNION) are defined in terms not of MAX but of SUM (?).

7.7 The predicate can be stated in many different ways, of course. 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?).

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 thus the original GROUP expression is logically equivalent to the following:

```
EXTEND SP : { X := TABLE_DEE }
```

The expression *r* UNGROUP X yields the original SP relation again.

7.9 **Tutorial D** on the left, SQL on the right as usual:

a.
```
N := COUNT ( SP
       WHERE SNO = 'S1' ) ;
```

```
SET N = ( SELECT COUNT ( * )
            FROM   SP
            WHERE  SNO = 'S1' ) ;
```

b.
```
( S WHERE CITY =
  MIN ( S , CITY ) ) { SNO }
```

```
SELECT SNO
FROM   S
WHERE  CITY =
     ( SELECT MIN ( CITY )
       FROM   S )
```

c.
```
S { CITY }
WHERE COUNT ( ‼S ) > 1
```

```
SELECT DISTINCT CITY
FROM   S AS SX
WHERE  ( SELECT COUNT ( * )
           FROM   S AS SY
           WHERE  SY.CITY = SX.CITY ) > 1
```

d.
```
S { CITY } XUNION
  XUNION P { CITY }
```

```
SELECT CITY
FROM   S
WHERE  CITY NOT IN ( SELECT CITY
                     FROM   P )
UNION  CORRESPONDING
SELECT CITY
FROM   P
WHERE  CITY NOT IN ( SELECT CITY
                     FROM   S )
```

e.
```
( P WHERE ( ‼SP ) { SNO } ⊇
( S WHERE CITY = 'London' ) { SNO } )
                             { PNO }
```

```
SELECT PNO
FROM   P
WHERE  NOT EXISTS (
SELECT * FROM S
WHERE  CITY = 'London'
AND    NOT EXISTS (
SELECT *
FROM   SP
WHERE  SP.SNO = S.SNO
AND    SP.PNO = P.PNO ) )
```

f.
```
S WHERE ( ‼SP ) { PNO } ⊇
  ( SP WHERE SNO = 'S2' ) { PNO }
```

```
SELECT SNO
FROM   S
WHERE  NOT EXISTS (
SELECT *
FROM   SP AS SPX
WHERE  SNO = 'S2'
AND    NOT EXISTS (
SELECT *
FROM   SP AS SPY
WHERE  SPY.SNO = S.SNO
AND    SPY.PNO = SPX.PNO ) )
```

7.10 It's the same as TCLOSE (*pp*). In other words, transitive closure is idempotent. *Note:* I'm extending the definition of idempotence somewhat here. In Chapter 6, I said (in effect) that a *dyadic* operator *Op* is idempotent if and only if *Op*(*x*,*x*) = *x* for all *x*; now I'm saying (in effect) that a *monadic* operator *Op* is idempotent if and only if *Op*(*Op*(*x*)) = *Op*(*x*) for all *x*. (Actually, mathematics textbooks typically define idempotence as a concept that applies to just one of these two cases; but some define it for the monadic case only and others for the dyadic case only. However, both cases clearly make sense.)

7.11 It denotes a relation of type

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR ,
                                          PNO_REL RELATION { PNO CHAR } }
```

that looks 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

The expression is logically equivalent to the following:

```
( S JOIN SP { SNO , PNO } ) GROUP { PNO } AS PNO_REL
```

Attribute PNO_REL is an RVA. 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 differs from our usual suppliers relation only in that it'll have no tuple for supplier S5.

7.12 The first is straightforward: It inserts a new tuple, with supplier number S6, name 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, extracted from Appendix D, the **Tutorial D** grammar for *<relation assign>* (the names of the syntactic categories are meant to be self-explanatory):

```
<relation assign>
    ::=   <relvar name> := <relation exp>
        | <insert> | <d_insert> | <delete> | <i_delete> | <update>
```

```
<insert>
    ::=   INSERT <relvar name> <relation exp>

<d_insert>
    ::=   D_INSERT <relvar name> <relation exp>

<delete>
    ::=   DELETE <relvar name> <relation exp>
        | DELETE <relvar name> [ WHERE <boolean exp> ]

<i_delete>
    ::=   I_DELETE <relvar name> <relation exp>

<update>
    ::=   UPDATE <relvar name> [ WHERE <boolean exp> ] :
                               { <attribute assign commalist> }
```

And an *<attribute assign>*, if the attribute in question is relation valued, is basically just a *<relation assign>* (except that the pertinent *<attribute name>* appears in place of the target *<relvar name>* in that *<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 part P5.

7.13 Query a. is easy:

```
WITH ( tx := ( SSP RENAME { SNO AS XNO } ) { XNO , PNO_REL } ,
       ty := ( SSP RENAME { SNO AS YNO } ) { YNO , PNO_REL } ) :
( tx JOIN ty ) { XNO , YNO }
```

Note that the join here is being done on an RVA (and so is 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:[27]

```
WITH ( pps := ( SSP UNGROUP PNO_REL ) GROUP { SNO } AS SNO_REL ,
       tx  := ( pps RENAME { PNO AS XNO } ) { XNO , SNO_REL } ,
       ty  := ( pps RENAME { PNO AS YNO } ) { YNO , SNO_REL } ) :
( tx JOIN ty ) { XNO , YNO }
```

[27] The 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 "hardwired 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.

7.14
```
WITH ( t1 := P RENAME { WEIGHT AS WT } ,
       t2 := EXTEND P : { N_HEAVIER :=
                          COUNT ( t1 WHERE WT > WEIGHT ) } ) :
( t2 WHERE N_HEAVIER < 2 ) { ALL BUT N_HEAVIER }
```

```
SELECT *
FROM   P AS PX
WHERE ( SELECT COUNT ( * )
        FROM   P AS PY
        WHERE  PX.WEIGHT < PY.WEIGHT ) < 2
```

Both formulations return parts P2, P3, and P6 (i.e., a result of cardinality three, even though the specified quota was two). Quota queries can also return a result of cardinality less than the specified quota (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 G), therefore, Hugh Darwen and I suggest a shorthand for expressing them, according to which the foregoing query might be expressed thus in **Tutorial D:**

```
( ( RANK P BY ( DESC WEIGHT AS W ) ) WHERE W ≤ 2 ) { ALL BUT W }
```

SQL has something similar.

7.15 This formulation does the trick:

```
SUMMARIZE SP { SNO , QTY } PER ( S { SNO } ) : { SDQ := SUM ( QTY ) }
```

But the following formulation, using EXTEND and image relations, is surely to be preferred:

```
EXTEND S { SNO } : { SDQ := SUM ( ‼SP { QTY } ) }
```

Here for interest is an SQL analog:

```
SELECT SNO , ( SELECT COALESCE ( SUM ( DISTINCT QTY ) , 0 ) AS SDQ
FROM   S
```

7.16
```
EXTEND S : { NP := COUNT ( ‼SP ) , NJ := COUNT ( ‼SJ ) }
```

```
JOIN { S , SUMMARIZE SP PER ( S { SNO } ) : { NP := COUNT ( PNO ) } ,
           SUMMARIZE SJ PER ( S { SNO } ) : { NJ := COUNT ( JNO ) } }
```

```
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
```

7.17 For a given supplier number, *sno* say, the expression !!SP denotes a relation with heading {PNO,QTY} and body consisting of those (*pno*,*qty*) pairs that correspond in SP to that supplier number *sno*. Call that relation *ir* (for image relation). By definition, then, for that supplier number *sno*, the expression !!(!!SP) is shorthand for the following:

```
( ( ir ) MATCHING RELATION { TUPLE { } } ) { ALL BUT }
```

This expression in turn is equivalent to:

```
( ( ir ) MATCHING TABLE_DEE ) { PNO , QTY }
```

And *this* expression reduces to just *ir*. Thus, "!!" is idempotent (i.e., !!(!!*r*) is equivalent to !!*r* for all *r*), and the overall expression

```
S WHERE ( !!(!!SP) ) { PNO } = P { PNO }
```

is equivalent to:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

("Get suppliers who supply all parts").

7.18 No, there's no logical difference.

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* INTERSECT *r2*).

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 (*r*, say), !!*r2* becomes !!*r*, and it denotes TABLE_DEE for every tuple in *r*.

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."

7.22 In SQL, typically in a cursor definition; in **Tutorial D** (where ORDER BY is spelled just ORDER), in a special operator ("LOAD"), not discussed further in this book, that retrieves a specified relation into an array (of tuples). Note that both of these situations are ones in which we're leaving the relational context and moving out into the external environment, as it were.

7.23 The tables on page 228 have no doubly underlined columns because the relations they represent are both of cardinality one, by definition. If we think of those relations as the current values of two relvars—defined by the SQL expressions SELECT COUNT(*) AS X FROM S and SELECT COUNT (DISTINCT STATUS) AS Y FROM S, respectively—then those relvars are each subject to a constraint to the effect that their cardinality must be one, and hence they each have an empty key (see the answer to Exercise 5.23 in Chapter 5). And empty keys obviously can't be shown using the double underlining convention.

As for the table on page 246, it doesn't represent a relation at all. Therefore it can't be the current value of any relvar; thus, the concept of having a key at all simply doesn't apply, and the double underlining convention therefore doesn't apply either.

Chapter 8

SQL and Constraints

A foolish consistency is the hobgoblin of little minds.
—Ralph Waldo Emerson:
"Self Reliance" (1841)

I've touched on the topic of integrity constraints here and there 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 variables in the database (i.e., relvars) specifically.[1] 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.
- A database constraint further constrains the values that can appear in a given database (where by "further" I mean over and above the constraints already imposed by the pertinent type constraints).

As usual, in what follows I'll discuss these ideas 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 variables in the database 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.

---

[1] As noted in Chapter 5, constraints constrain updates and updates apply to variables, not values, so it does make sense to talk of a constraint "applying to" some variable.

A note on terminology: Let constraint *C* apply to relvar *R* (e.g., *C* might be the constraint that a certain subset of the heading of *R* constitutes a key for *R* and thus has the uniqueness property). Then we say that relvar *R is subject to* constraint *C*; equivalently, we say that constraint *C holds* in relvar *R*. Further, let *r* be a relation of the same type as *R*. If evaluating constraint *C* on relation *r* yields TRUE, we say that *r satisfies C*; otherwise we say that *r violates C*. Of course, if *r* violates *C*, it can't be assigned to *R*; at all times, therefore, the current value of *R* satisfies all constraints to which *R* is subject, necessarily and by definition.

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. Now, 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 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,[2] 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 carefully, however, that there's no suggestion that the specified possible representation is the same as any physical representation, whatever that happens to be; it might be or it might not, but either way it makes no difference to the user.

[2] A minor exception to this rule arises in connection with **Tutorial D**'s type inheritance support, but that exception need not concern us here.

- 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, one of the reasons the possrep concept is required in the first place is precisely to serve as a vehicle for formulating type constraints, as I think the example suffices to show.

  Here now is a slightly more complicated example:

```
TYPE POINT
     POSSREP CARTESIAN { X RATIONAL , Y RATIONAL
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 } ;
```

Type POINT denotes geometric 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 both of type RATIONAL; and there's a CONSTRAINT specification that says (in effect) that the only points we're interested in are those that lie on or inside a circle with center the origin and radius 100 (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 at that time. Tacitly, however, I was assuming the 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 issues raised by the QTY and POINT examples.

Recall from Chapter 2 that scalar types, at least, have certain 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 just to make it clear that 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 simplify 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 from this point forward, barring explicit statements to the contrary.

2. `QTY ( A + B )`

   The expression denoting the argument to a QTY selector invocation can be as complex as we like, of course, 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 in fact we already know from Chapter 2). In fact, all literals are selector invocations, but "most" selector invocations aren't literals; for example, QTY(A+B) isn't.

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 of type QTY (specified by means of an arbitrarily complex expression of that type), 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 RATIONAL , Y RATIONAL CONSTRAINT ... } ;
```

Now continuing with the examples:

4. `POINT ( 5.7 , -3.9 )`

   This expression is a POINT selector invocation (actually a POINT literal).

5. `THE_X ( P )`

   This expression returns the RATIONAL value that's the X coordinate of the cartesian possible representation of the point that's the current value of 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. Temperatures provide another example—again, we might want to define two possreps, to reflect the fact that temperatures can be possibly represented in either degrees Celsius or degrees Fahrenheit. Further details don't belong in a book of this nature; I'll just note for the record that SQL has no analogous feature.

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 rather a 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 it 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. Since, ultimately, the only way any expression can yield a value of type

T is by means of some invocation of some selector for type *T*, 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.[3] 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. (As an aside, I note that this constraint on attribute QTY is an example of what's sometimes called, albeit only informally, 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 type QTY doesn't have any proper subtypes. Subtypes in general are beyond the scope of this book.)

Given the foregoing SQL definition, all available integers (including negative ones!) 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, if column QTY in base table SP is defined to be of type QTY instead of type INTEGER, then you might need to extend the definition of that table as follows (note the CONSTRAINT specification in the last line):

```
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) ) ) ;
```

[3] 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 word in computing contexts.

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 indicated in Chapter 2, in fact, the situation regarding selectors and THE_ operators in SQL is much 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 the 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 SQL's approach to 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).

> *Aside:* 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 absurdities such as nonsquare squares (by which I mean, more precisely, values of a user defined type SQUARE whose sides are of different lengths and are thus not in fact squares at all). For further explanation of such solecisms, see the book *Databases, Types, and the Relational Model: The Third Manifesto,* by Hugh Darwen and myself (see Appendix G). *End of aside.*

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 logically 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, therefore, let me stay with the "longhand" forms. Here are some examples (**Tutorial D** on the left and SQL on the right as usual).

Example 1:

```
CONSTRAINT CX1 IS_EMPTY
  ( S WHERE STATUS < 1
      OR    STATUS > 100 ) ;
```

```
CREATE ASSERTION CX1 CHECK
 ( NOT EXISTS
     ( SELECT * FROM S
       WHERE  STATUS < 1
       OR     STATUS > 100 ) ) ;
```

Constraint CX1 says: Supplier 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"). Now, all such usages ought really to be deprecated, 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. (By the same token, there's no such thing as a tuple variable, or row variable, in a relational database.) However, the terms can sometimes be convenient, and so—somewhat against my own better judgment—I'll be using them occasionally myself in what follows.

Recall now that as mentioned in a footnote in Chapter 7, certain constraints can alternatively be formulated in terms of the aggregate operator AND. In fact this observation applies to tuple constraints specifically. Here by way of example is such a formulation for constraint CX1:

```
CONSTRAINT CX1 AND ( S , STATUS ≥ 1 AND STATUS ≤ 100 ) ;
```

This formulation says, in effect, that the expression STATUS ≥ 1 AND STATUS ≤ 100 must evaluate to TRUE for every tuple in S. As you can see, the desired constraint ("Status values must be greater than or equal to 1 and less than or equal to 100") is stated a little more directly with this formulation than it was with the IS_EMPTY version, where it had to be stated in the negative ("Status values mustn't be less than 1 or greater than 100"). More generally, the AND aggregate operator invocation

```
AND ( rx , bx )
```

means, loosely speaking, that the boolean expression *bx* must evaluate to TRUE for every tuple in the relation that's the result of evaluating the relational expression *rx*.

Can we do the same kind of thing in SQL? Well, SQL's analog of AND is called EVERY. Here's an SQL formulation of constraint CX1 that makes use of that operator:

```
CREATE ASSERTION CX1 CHECK
   ( ( SELECT COALESCE ( EVERY ( STATUS >= 1 AND STATUS <= 100 ) ,
                         TRUE )
       FROM   S ) = TRUE ) ;
```

As you can see, however, this formulation isn't nearly as user friendly as the **Tutorial D** version, for at least two reasons:

- First, EVERY, unlike **Tutorial D**'s AND, returns null, not TRUE, if its argument is empty; hence the need for that COALESCE.

- Second, I pointed out in Chapter 7 that SQL doesn't really support aggregate operators anyway, and the present example brings that point home. To be specific, the parenthesized subexpression SELECT ... FROM S is, of course, a table expression; hence it denotes, not a truth value as such, but rather a one-row, one-column table that contains such a truth value. In fact, that subexpression, parentheses included, is a scalar subquery. As explained in Chapter 2, then, asking for that subquery and the literal value TRUE to be tested for equality causes a double coercion to occur; in other words, the truth value is effectively extracted from the table and then tested to see whether it's equal to TRUE.

The net of this discussion is that EVERY isn't nearly as useful for the formulation of row constraints in SQL as AND is for the formulation of tuple constraints in **Tutorial D**.

Aside: The foregoing might be a little unfair to SQL. To be specific, I *think*—according to my own reading of the standard—that it would be possible to simplify the example by omitting both the COALESCE and the explicit comparison with TRUE, thereby reducing the CHECK clause portion of the assertion to just:

```
CHECK ( ( SELECT EVERY ( STATUS >= 1 AND STATUS <= 100 ) FROM S ) ) ;
```

But these simplifications (if they're legitimate) do rely on several aspects of SQL that are, to put matters politely, hardly very respectable. First of all, note that the double enclosing parentheses are necessary—the outer parentheses enclose a subquery, which requires parentheses of its own. Second, the subquery in question is in fact a *scalar* subquery, and the table it returns gets doubly coerced to the single value—actually a truth value—in the single column of the single row of the table in question (see Chapter 12). Third, if the EVERY invocation in fact returns a null, that null is considered to stand for the truth value UNKNOWN (see Chapter 4). Fourth, if the boolean expression in a CHECK clause evaluates to UNKNOWN, that UNKNOWN gets coerced to TRUE! (See the answer to Exercise 8.20g for further discussion of this last point.) Speaking for myself, therefore, I would far rather include both the COALESCE and the comparison with TRUE, in the interest of explicitness if nothing else. *End of aside.*

Example 2:

```
CONSTRAINT CX2 IS_EMPTY
  ( S WHERE CITY = 'London'
      AND   STATUS ≠ 20 ) ;
```

```
CREATE ASSERTION CX2 CHECK
 ( NOT EXISTS
     ( SELECT * FROM S
       WHERE  CITY = 'London'
       AND    STATUS <> 20 ) ) ;
```

Constraint CX2 says: Suppliers in London must have status 20. Unlike constraint CX1, this constraint involves two distinct attributes; however, 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 or row constraint). Here for interest are AND and EVERY formulations (though the advantages of such a formulation aren't so clear here as they were with constraint CX1):

```
CONSTRAINT CX2
  AND ( S , CITY ≠ 'London'
            OR STATUS = 20 ) ;
```

```
CREATE ASSERTION CX2 CHECK
( ( SELECT COALESCE
      ( EVERY ( CITY <> 'London' OR
                STATUS = 20 ) ,
                TRUE )
    FROM S ) = TRUE ) ;
```

Example 3:

```
CONSTRAINT CX3
    COUNT ( S ) =
        COUNT ( S { SNO } ) ;
```

```
CREATE ASSERTION CX3 CHECK
     ( UNIQUE ( SELECT SNO
                FROM   S ) ) ;
```

Constraint CX3 says: Every supplier has a unique supplier number; in other words, {SNO} is a superkey—actually, of course, it's a key—for relvar S (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; unlike those constraints, however, this one 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. Points arising:

- 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 clearly preferable, at least from a human factors point of view. I show the longhand form merely to make the point that such shorthands are indeed, in the final analysis, just shorthands.[4]

[4] In SQL, that shorthand would typically involve a specification of the form UNIQUE(SNO) as part of the CREATE TABLE for table S. The semantics of such a specification are explained by the standard as follows (I've adapted the standard's own generic phrasing to apply to the specific case at hand): "The constraint UNIQUE(SNO) is not satisfied if and only if EXISTS (SELECT * FROM S WHERE NOT (UNIQUE (SELECT SNO FROM S))) is true." I hope that's perfectly clear. Note the reference to the SQL UNIQUE operator, discussed in the present chapter in the next bullet item.

- As you can see, the SQL formulation of constraint CX3 involves an invocation of the SQL UNIQUE operator. That operator returns TRUE if and only if every row within its argument table is distinct; in the example, therefore, the UNIQUE invocation returns TRUE if and only if no two rows in table S have the same supplier number. Note, incidentally, that the SELECT expression in that invocation must—for once—definitely *not* specify DISTINCT! (Why not?) I'll have more to say about SQL's UNIQUE operator in Chapter 10.

Here for interest is an SQL formulation of constraint CX3 that more closely resembles the **Tutorial D** formulation:[5]

```
CREATE ASSERTION CX3 CHECK
   ( ( SELECT COUNT ( ALL SNO ) FROM S ) =
     ( SELECT COUNT ( DISTINCT SNO ) FROM S ) ) ;
```

Example 4:

```
CONSTRAINT CX4
   COUNT ( S { SNO } ) =
   COUNT ( S { SNO , CITY } ) ;
```

```
CREATE ASSERTION CX4 CHECK
( ( SELECT COUNT ( SNO )
    FROM   S ) =
  ( SELECT COUNT ( * )
    FROM ( SELECT SNO , CITY
           FROM   S ) ) ) ;
```

Constraint CX4 says: Whenever two suppliers have the same supplier number, they also have the same city. In other words, a certain functional dependency (FD) holds in relvar S—namely, an FD from {SNO} to {CITY}. Now, as I'm sure you know (and as in fact we saw in Chapter 5), that FD would more usually be expressed like this:

```
{ SNO } → { CITY }
```

Here's a precise definition:

> **Definition:** Let *A* and *B* be subsets of the heading of relvar *R*. Then the *functional dependency* (FD) *A* → *B* holds in *R* 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 dependency

[5] But is this SQL formulation valid? As you can see, it involves an equality comparison in which the comparands are denoted by subqueries. Since subqueries evaluate to tables, it appears we're trying to test two tables for equality—yet we saw in Chapter 3 that SQL doesn't directly support table equality comparisons. See Exercise 12.5 in Chapter 12 for further discussion.

is basically just another integrity constraint (though, like constraint CX3, it isn't a tuple or row constraint).

Of course, as noted in Chapter 5, the fact that relvar S is subject to the particular FD {SNO} → {CITY} 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. This hypothetical new constraint (which is *not* satisfied by our usual sample values, please note) is clearly an FD:

```
{ CITY } → { STATUS }
```

It can thus be stated in the style of constraint CX4 (see Exercise 8.22 at the end of the chapter).

Now, you might be thinking 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, most 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" here, I really mean *fully normalized*. Normalization as such is beyond the scope of this book (it's covered in depth in the book *Database Design and Relational Theory: Normal Forms and All That Jazz*, which is a companion to the present book—see Appendix G). Of course, relational (or SQL) statements and expressions will work regardless of whether the relvars (or tables) are fully normalized or not. But I should at least point out that those statements and expressions will often be easier to formulate (and, contrary to popular opinion, will often perform better too) if the relvars or tables are fully normalized. However, normalization as such is primarily a database design issue, not a relational model or SQL issue.

Example 5:

```
CONSTRAINT CX5 IS_EMPTY
   ( ( S JOIN SP )
     WHERE STATUS < 20
     AND   PNO = 'P6' ) ;
```

```
CREATE ASSERTION CX5 CHECK
   ( NOT EXISTS
       ( SELECT *
         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 (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 multirelvar constraint. Thus, constraints CX1-CX4 were single relvar constraints, while constraint CX5 is a multirelvar constraint. All of these

terms are somewhat deprecated, however, for reasons to be discussed in the next chapter, in connection with what's called *The Principle of Interchangeability*.

Example 6:

```
CONSTRAINT CX6
   SP { SNO } ⊆ S { SNO } ;
```

```
CREATE ASSERTION CX6 CHECK
  ( NOT EXISTS
      ( SELECT SNO FROM SP
        EXCEPT CORRESPONDING
        SELECT SNO FROM S ) ) ;
```

Constraint CX6 says: Every supplier number in SP must appear in S. As you can see, the **Tutorial D** formulation involves a relational inclusion comparison; SQL doesn't directly support such comparisons, however, and so we have to indulge in some circumlocution in the SQL formulation. Be that as it may, however, 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.[6] Unlike **Tutorial D,** however, SQL has a feature according to which any such constraint can alternatively be specified as part of the definition of some base table—in other words, as a *base table constraint*. For example, here again is an SQL formulation (using CREATE ASSERTION) 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 slightly different form as a base table constraint as part of the definition of base table SP, like this:

```
CREATE TABLE SP
     ( ... ,
       CONSTRAINT CX5 CHECK  /* "base table" constraint */
                ( PNO <> 'P6' OR ( SELECT STATUS FROM S
                                   WHERE  SNO = SP.SNO ) >= 20 ) ) ;
```

[6] Except that (as you'll recall from Chapter 2) constraints in SQL are supposed not to contain "possibly nondeterministic expressions," a rule that could cause serious problems in practice. See Chapter 12 for further discussion.

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 in fact 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 be useful for row constraints (i.e., constraints that can be checked for an individual row in isolation), because it's a little simpler than its CREATE ASSERTION counterpart is, in this particular case. 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.

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.[7] A column constraint in SQL 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 SQL 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.) **Recommendation:** Specify constraints declaratively whenever you can. In practice, however, many constraints (perhaps most) will, regrettably, 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

[7] Not to be confused with attribute constraints (see the end of the section "Type Constraints," earlier in the chapter).

affairs represents a serious defect in today's products, and it needs to be remedied, urgently.[8]

TRANSACTIONS

Despite the SQL defects 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 multirelvar constraint checking is *deferred* to end of transaction ("commit time"). I want to argue, however, that all checking should be immediate, and deferred checking—which is supported in the SQL standard, and also in at least one SQL product to my knowledge—is a logical mistake. In order to explain this perhaps 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.[9] All I want to do here is briefly review the so called *ACID properties* of transactions. ACID is an acronym, standing for atomicity – consistency – isolation – durability, where:

- *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. *Note:* A database state is consistent if and only if it satisfies all defined integrity constraints (*consistency* in this context is just another word for integrity).
- *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.

---

[8] Chapter 11 of the book *Applied Mathematics for Database Professionals*, by Lex de Haan and Toon Koppelaars (highly recommended, by the way), goes into great detail on what's involved in writing your own constraint enforcement code. See Appendix G.

[9] The standard reference (also highly recommended) is *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter. Again, see Appendix G.

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. Rather, 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 I do (viz., 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 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. And 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 for completeness I'll give the other arguments as well. 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 mean quite 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* have certainly communicated, and so they 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. *Note:* Similar remarks apply if *TX1* (a) sees an inconsistency and therefore assigns an incorrect value to some local variable *V* and then (b) transmits the value of that variable *V* to some outside user (since local variables aren't, and can't possibly be, subject to the jurisdiction of the transaction management subsystem).

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 when 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 multirelvar constraint with another. For example, suppose we have two virtual relvars, or 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 are subject to 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 fact that every supplier has exactly one city—i.e., the FD {SNO} → {CITY} holds in base relvar S—together with the fact that any given city is necessarily either equal to London or not equal to London. But suppose we made LS and NLS base relvars and then defined their union as a view called S. Then the constraint *would* have to be stated explicitly:

```
CONSTRAINT CX7 IS_EMPTY
   ( LS { SNO } JOIN
     NLS { SNO } ) ;
```

```
CREATE ASSERTION CX7 CHECK
  ( NOT EXISTS
      ( SELECT *
        FROM   LS , NLS
        WHERE  LS.SNO = NLS.SNO ) ) ;
```

Now what was previously a single relvar constraint on base relvar S ("supplier numbers are unique") has become a multirelvar constraint instead.[10] Thus, if we agree, as most writers do, that single relvar constraints must be checked immediately, we must surely agree that multirelvar constraints must be checked immediately as well (since, logically, there's no real difference between the two, as the example demonstrates).

Fifth and last, there's an optimization technique called *semantic* optimization (it involves expression transformation, but I deliberately didn't mention it in the discussion of that topic in Chapter 6). By way of example, consider the expression (SP JOIN S){PNO}. Now, the join here is based on the correspondence between a foreign key in a referencing relvar, SP, and the target 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}. Note carefully, however, 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 mentioned therefore won't be valid. But in the case at hand 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. And a

[10] Well, it can still be described as a single relvar constraint with the revised design, but that single relvar constraint is a single relvar constraint on a *view* (view S). See Chapter 9 for a discussion of view constraints in general.

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).[11] 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 commercial products do comparatively little 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. Why? Because, as we've just 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) 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.[12] If any such check fails, changes to the database, if any, caused by the offending statement must be undone and an exception raised.

---

[11] The constraint must be stated declaratively, however; obviously there's no way the optimizer can "understand" and exploit constraints that have been specified procedurally (and so we have here another strong reason for requiring declarative constraint support).

[12] If despite everything I've said you're still bothered by this idea, please refer to the section titled "Eventual Consistency" in Appendix F for further clarification.

BUT DOESN'T SOME CHECKING HAVE TO BE DEFERRED?

The arguments of the previous section notwithstanding, conventional wisdom is that multirelvar constraint checking, at least, does have to be deferred, typically 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, each 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 rather than **Tutorial D** precisely because checking *is* immediate in **Tutorial D** and the conventional solution to the problem therefore doesn't work in **Tutorial D** (nor is it needed, of course). What is that conventional solution? *Answer:* We defer the checking of the constraint to commit time,[13] and we make sure the two UPDATEs are part of the same transaction, as in this SQL code:

```
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 the end of the 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 rows for S1 and P1 do exist), it would get the answer *yes*.

---

[13] In case you're wondering how that 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 declared to be (a) either DEFERRABLE or NOT DEFERRABLE, and if DEFERRABLE then (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. (Of course, the constraint(s) in question must have been defined to be DEFERRABLE for SET CONSTRAINTS to apply.) COMMIT forces all DEFERRABLE constraints into immediate mode; if some 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 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 of the individual assignments are evaluated.
- Second, those individual assignments (to the variables on the left sides) are executed.
- Third, all pertinent integrity constraints are checked.

(Actually this definition requires a slight refinement in the case where two or more of the individual assignments specify the same target variable, but that refinement needn't concern us here.) 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 the sequence in which they're executed is irrelevant; in fact, you can think of them as being executed in parallel, or "simultaneously"). Moreover, since multiple assignment is defined to be a semantically atomic 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 state for the record here that *all* statements are semantically atomic in the relational model. In fact, most statements are syntactically atomic too; multiple assignment is an exception, because it's semantically atomic but not syntactically so. *End of aside.*

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 "simultaneously," as part of a single operation. Second, the ability to update (for example) certain join views—see Chapter 9—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 (indeed, 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*[14]—which is precisely the case illustrated by the foregoing example, and precisely what we need in order to avoid having to do deferred integrity 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, if nothing else. All I'm saying is that transactions aren't the "unit of integrity" they're usually supposed to be.

Recommendation: In SQL, use immediate checking whenever you can. Given the state of today's products, however, some checking (especially for constraints that involve more than one table) will almost certainly have to be deferred. In such a case, you should do whatever it takes—which in practice might mean terminating the transaction—to force the check to be done before executing any operation that relies, or 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 something like this:

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

[14] I'm told, however, that this functionality is likely to be provided in some future version of the standard.

In an ideal world, then, this predicate would serve as "the criterion for acceptability of updates" on relvar S—that is, it would dictate whether a given update operation on that relvar can be accepted. But of course 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,[15] 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 requesting some update to be done) that there's a supplier S6 named Lopez with status 30 and city Madrid, then 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 cause any integrity constraint to be violated. Assuming it doesn't, the system will accept the user's assertion, will perform the requested update, *and will treat what the user said as true from that point forward* (until such time as the user tells the system, by requesting 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 ($m \geq 0$) be all of the database constraints, single relvar or multirelvar, 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

---

[15] 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 relvar predicates).

```
( 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 (i.e., $m = 0$), then the default is just TRUE.[16]

Now let *RC* be the (total) 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 the relvar constraint for any relvar to evaluate to FALSE.*

Now let *DB* be a database, and let *R1*, *R2*, ..., *Rn* ($n \geq 0$) 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:**

> *No update operation must ever cause the database constraint for any database to evaluate to FALSE.*

Observe in particular that, in accordance with my position that all integrity checking must be immediate, **The Golden 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 ones that are obviously false, like 1 = 0) can be shown to be "true" in an inconsistent system. *Note:* In case you're still not convinced, I refer you to the further discussion of this issue in Chapter 10.

---

[16] "Unlikely" is right; *every* relvar is supposed to be subject to a key constraint at the very least.

MISCELLANEOUS ISSUES

There are a number of further points to do with integrity that I need to cover somewhere 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). To see that this is so, consider 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 the suppliers 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, unequivocally, 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's 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 that rule as a "generic integrity constraint." However, it should be clear by now that it's somewhat different in kind from the constraints we've been examining in this chapter. It's really what might be called 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.

Fourth, I remind you from Chapter 5 that update operators are always set level, and hence that constraint checking mustn't be done until all of the updating has been done; i.e., a set level update mustn't be treated as a sequence of individual tuple level updates (or row level updates, in

SQL). I also said in that chapter that the SQL standard does conform to this requirement, but that products might not. Indeed, the last time I looked, there was at least one major product that didn't conform but (on foreign key constraints, at least) did "inflight checking" instead. One problem with this state of affairs is that it can lead to undesirable and possibly complex prohibitions against certain operations. For example, suppose there's a cascade delete rule from suppliers to shipments. Then the product in question won't allow the following apparently innocuous, and reasonable, DELETE statement:

```
DELETE
FROM   S
WHERE  SNO NOT IN
     ( SELECT SNO
       FROM   SP ) ;
```

(an attempt to delete suppliers with no shipments).

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_STATUS } )
         JOIN
       ( S  { SNO , STATUS } RENAME { STATUS AS NEW_STATUS } ) )
     WHERE OLD_STATUS > NEW_STATUS ) ;
```

Explanation: I'm adopting the convention that a primed relvar name such as S′ refers to the pertinent relvar as it was immediately prior to the update under consideration. Constraint CX9 thus says: If we join the old value of S and the new one on {SNO} and then restrict the result of that join to just those tuples where the old status is greater than the new one, that restriction 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). *Note:* There might be good reasons for that lack of support, however. See the answer to Exercise 8.26g at the end of the chapter for further discussion.

Last, I hope you agree from everything we've covered in this chapter that constraints are absolutely vital—and yet they seem to be very poorly supported in current products (even though the integrity support in the SQL standard, as opposed to those commercial products, is actually not all that bad). Indeed, the whole business of integrity seem to be underappreciated at best, if not completely misunderstood, in the industry at large. Thus, the emphasis in practice always seems to be on *performance*, *performance*, *performance*; other objectives, such as ease of use,

physical 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.[17]

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 results we're getting from it are 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

8.1 Define the terms *type constraint* and *database constraint*. When are such constraints checked? What happens if the check fails?

8.2 State **The Golden Rule**. Is it true that this rule can be violated if and only if some specific single relvar constraint is violated?

8.3 What do you understand by the following terms?—*assertion*; *attribute constraint*; *base table constraint*; *column constraint*; *multirelvar 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?

8.4 Distinguish between possible and physical representations.

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 ) )`

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 earlier chapters, but now you should be able to be more specific in your answers.

---

[17] I don't mean to suggest here that system enforcement of constraints implies bad performance; in fact, I think it ought to improve performance. (Not to mention the fact that user enforcement is highly nontrivial, and very likely to be incorrect! As mentioned in an earlier footnote, the book by Lex de Haan and Toon Koppelaars, *Applied Mathematics for Database Professionals*, gives a good idea of what's involved in such enforcement.) All I mean is, there tends to be a huge emphasis in vendor development effort on performance issues, to the exclusion of other matters such as data integrity.

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.

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.

8.9 Define supplier numbers as a **Tutorial D** user defined type. You can assume 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 are numerals denoting a decimal integer in the range 1 to 9999. State any assumptions you make regarding the availability of operators to help with your definition.

8.10 A line segment is a straight line connecting two points in the euclidean plane. Give a corresponding **Tutorial D** type definition.

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?

8.12 Can you think of a type for which different possreps might have different numbers of components?

8.13 Which operations might cause constraints CX1-CX9 from the body of the chapter to be violated?

8.14 Does **Tutorial D** have anything directly analogous to SQL's base table constraints?

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.

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?

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.

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?

8.19 Can you give either a **Tutorial D** database constraint or an SQL assertion that's exactly equivalent to the specification KEY {SNO} for relvar S?

8.20 Give an SQL formulation of constraint CX8 from the body of the chapter.

8.21 Using either **Tutorial D** or SQL or both, 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. The total quantity of parts supplied by suppliers in London must be greater than the corresponding total for 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.

8.22 Suppose there's a constraint in effect that says if two suppliers are in the same city, they must have the same status; in other words, suppose relvar S is subject to the functional dependency {CITY} → {STATUS}. (I mentioned this possibility in the discussion of constraint

CX4 in the body of the chapter.) Do either of the following **Tutorial D** CONSTRAINT statements accurately represent this constraint?

```
CONSTRAINT CX22a
    COUNT ( S { CITY } ) = COUNT ( S { CITY , STATUS } ) ;

CONSTRAINT CX22b
    S = JOIN { S { ALL BUT STATUS } , S { CITY , STATUS } } ;
```

8.23 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 final "AND TRUE"?

8.24 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*.

8.25 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).

8.26 Using the syntax sketched for transition constraints in the section "Miscellaneous Issues," write transition constraints to express the following requirements:

a. The total shipment quantity for a given part can never decrease.

b. Suppliers in Athens can move only to London or Paris, and suppliers in London can move only to Paris.

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?)

8.27 Investigate any SQL product that might be available to you. What semantic optimization does it support, if any?

8.28 Why do you think SQL fails to support type constraints? What are the consequences of this state of affairs?

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 system defined types?

8.30 Show that any arbitrary UPDATE can be expressed in terms of DELETE and INSERT.

ANSWERS

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 whenever some selector for type *T* is invoked; if the check fails, the selector invocation fails on a type constraint violation. *Subsidiary exercise:* What do you think should happen if the type constraint for type *T* evaluates to FALSE at the time type *T* is defined? (*Answer:* This state of affairs isn't necessarily an error, but the type in question will be empty. See the answer to Exercise 2.18 in Chapter 2.)

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 update statement that attempts to assign a value to any of the pertinent relvars. If the check fails, the update 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.

8.2 **The Golden Rule** states (in effect) that no update operation must ever cause any database constraint to evaluate to FALSE, and hence that no update operation must ever cause any relvar constraint to evaluate to FALSE either, a fortiori. However, a (total) relvar constraint might evaluate to FALSE, not because some single relvar constraint is violated, but rather because some multirelvar constraint is violated. The point is hardly significant, however, given that—as mentioned in the body of the chapter and explained in more detail in Chapter 9—which relvar constraints are single relvar and which multirelvar is somewhat arbitrary anyway.

8.3 *Assertion* is SQL's term for a constraint specified via CREATE ASSERTION. An *attribute constraint* is a specification 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 such a base table definition). A *column constraint* is an SQL constraint that's specified as part of a column definition within a base table definition. A *multirelvar constraint* is a database constraint that mentions two or more distinct relvars. A *referential constraint* (also known as a *foreign key 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 TRUE and all of the relvar constraints for relvars in *DB*.[18] *"The" (total) relvar constraint* for relvar *R* is the logical AND of TRUE and all of the database constraints that mention *R*. 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.*

8.4 See the section "Type Constraints" in the body of the chapter.

8.5 a. The integer 345. b. The value of a variable called QTY (which must be of type QTY).

8.6 See the body of the chapter.

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.

8.8 By definition, there's no way to impose a constraint in SQL that's exactly equivalent to the one given in the previous answer, even if we define an explicit type, because SQL doesn't support type constraints. But we could define a database constraint to the effect that cities in table S specifically are limited to those same eight values, and likewise for cities in table P. One approach to such a scheme involves defining a base table C ("cities") as follows:

```
CREATE TABLE C ( CITY VARCHAR(20) , UNIQUE ( CITY ) ) ;
```

We could then "populate" this table with the eight city values:

[18] But see the answer to Exercise 8.23 below.

```
INSERT INTO C ( CITY ) 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 such a 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. *Note:* SQL's "domains"—see Chapter 2—could help with this approach (if they're supported, of course!), because they could allow the pertinent constraint to be written just once and shared by all pertinent columns. For example (in outline):

```
CREATE DOMAIN CITY AS VARCHAR(20)
       CONSTRAINT ... CHECK ( VALUE IN ( 'London'    ,
                                         'Paris'     ,
                                         'Rome'      ,
                                         'Athens'    ,
                                         'Oslo'      ,
                                         'Stockholm' ,
                                         'Madrid'    ,
                                         'Amsterdam' ) ) ;
```

Now we can define the CITY columns in tables S and P to be of "domain CITY" instead of type VARCHAR(20), and they'll then "automatically" be subject to the required constraint.

Another approach 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.

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.

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. Note, incidentally, that an SQL analog of the foregoing type definition wouldn't be able to use BEGIN and END as names of the corresponding attributes—*attributes* being (most unfortunately!) SQL's term for components of what it calls a "structured type"—because BEGIN and END are reserved words in SQL. (It would, however, be able to use the *delimited identifiers* "BEGIN" and "END" for the purpose. A delimited identifier in SQL is an arbitrary string of characters—including, possibly, the string of characters that forms an SQL reserved word—enclosed in what SQL calls double quotes, or in other words conventional quotation marks.)

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; however, those constraints must all be logically equivalent. For example:

```
TYPE POINT
     POSSREP CARTESIAN { X RATIONAL , Y RATIONAL
                         CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 }
     POSSREP POLAR { RHO RATIONAL , THETA RATIONAL
                     CONSTRAINT RHO ≤ 100.0 } ;
```

Whether some shorthand could be provided that would effectively allow us to specify the constraint just once instead of once per possrep is a separate issue (a language design issue, in fact) and is beyond the scope of this book.

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).

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

8.14 This exercise is a little unfair, since you aren't supposed to be an expert in **Tutorial D!** Be that as it may, the answer is yes for KEY and FOREIGN KEY constraints, no for other constraints. *Note:* There's no particular reason why the answer shouldn't be yes for other constraints too, if it were thought desirable; however, any temptation to intermingle (and thereby muddle, *à la* SQL) specification of the pertinent relation type and specification of such constraints should be firmly resisted. Also, we'd have to be careful over what it might mean for such a "base relvar" constraint if the base relvar to whose definition it's attached happens to be empty (see the answer to Exercise 8.16 below).

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. See Chapter 10 for further explanation.

8.16 The formal reason has to do with the fact that FORALL is defined to return TRUE when the applicable "range" is an empty set; again, see Chapter 10 for further explanation. **Tutorial D** has nothing directly analogous to base table constraints in general and thus doesn't display analogous behavior.

8.17 As a base table constraint on table S:

```
CREATE TABLE S
  ( ... ,
    CONSTRAINT CX5
       CHECK ( STATUS >= 20 OR SNO NOT IN ( SELECT SNO
                                            FROM   SP
                                            WHERE  PNO = 'P6' ) ) ) ;
```

As a base table constraint on table P:

```
CREATE TABLE P
  ( ... ,
    CONSTRAINT CX5
       CHECK ( NOT EXISTS ( SELECT *
                            FROM   S NATURAL JOIN SP
                            WHERE  STATUS < 20
                            AND    PNO = 'P6' ) ) ) ;
```

Observe in this latter formulation that the constraint specification makes no reference to the base table whose definition it forms part of. Thus, the very same specification could form part of the definition of absolutely any base table whatsoever. (It's essentially identical to the CREATE ASSERTION version, anyway.)

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.

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 on keys—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.[19]

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 ) ;
```

Note the need for an AS specification to accompany the subquery in the outer FROM clause here, even though the name it introduces is never referenced. See the discussion in the section on EXTEND in Chapter 7 if you need to refresh your memory regarding this point.

[19] No such rule exists in SQL, however. What's more, any implementation that tried to impose such a rule would be in violation of the standard!—i.e., the SQL standard explicitly permits "keys" to be declared that the user and the system both know to be proper superkeys. The "justification"—such as it is—for this state of affairs is beyond the scope of this book.

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 ) ;
```

Or:

```
CONSTRAINT CXA
   AND ( P , COLOR ≠ 'Red' OR WEIGHT < 50.0 ) ;

CREATE ASSERTION CXA CHECK
   ( NOT EXISTS ( SELECT *
                  FROM   P
                  WHERE  COLOR = 'Red' AND WEIGHT >= 50.0 ) ) ;
```

Or:

```
CREATE ASSERTION CXA CHECK (
   ( SELECT COALESCE ( EVERY ( COLOR <> 'Red' OR WEIGHT < 50.0 ) , TRUE )
     FROM   S ) = TRUE ) ;
```

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    SNO NOT IN
                       ( SELECT SNO FROM SP
                            WHERE  PNO = 'P2 ' ) ) ) ;
```

c.
```
CONSTRAINT 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 * FROM S WHERE CITY = 'Athens' ) ) ;
```

e.
```
CONSTRAINT CXE IS_NOT_EMPTY ( S WHERE CITY = 'London' ) ;

CREATE ASSERTION CXE CHECK
   ( EXISTS ( SELECT * FROM S WHERE CITY = 'London' ) ) ;
```

f.
```
CONSTRAINT CXF IS_NOT_EMPTY ( P WHERE COLOR = 'Red' AND WEIGHT < 50.0 ) ;

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 ) ;
```

Note: The foregoing formulations allow relvar S to be empty without violating the required constraint. But suppose the SQL formulation were simplified thus:

```
CREATE ASSERTION CXG CHECK
   ( ( SELECT AVG ( STATUS ) FROM S ) ≥ 10 ) ;
```

Now if relvar S is empty, the AVG invocation returns null, and the comparison "null ≥ 10" returns UNKNOWN. Now, we saw in Chapter 4 that (to quote) "queries in SQL retrieve data for which the expression in the WHERE clause evaluates to TRUE, not to FALSE and not to UNKNOWN"; in other words, UNKNOWN effectively gets coerced to FALSE in the context of a query. But if the same thing happens in the context of a constraint like the one under discussion, the effect is that the constraint is considered to be satisfied. In such a context, in other words, UNKNOWN is coerced to TRUE instead of FALSE!

To pursue the point a moment longer, suppose (a) we execute a CREATE ASSERTION saying that shipment quantities must be greater than zero (QTY > 0), and then (b) we execute the following sequence of SQL statements:

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

SELECT * FROM SP WHERE QTY > 0 ;
```

The INSERT will succeed—in the constraint, the expression QTY > 0 will evaluate to UNKNOWN, which will be coerced to TRUE—but the inserted row won't appear in the result of the SELECT. (In fact, knowing that shipment quantities are supposed to be greater than zero, the user would be within his or her rights to expect that SELECT to be logically equivalent to just SELECT * FROM SP.) At the very least, therefore, the user will see a violation of *The Assignment Principle* in this example. To repeat something I said in the answer to Exercise 4.14 in Chapter 4, I regard this state of affairs as yet another of the vast—infinite?—number of absurdities that nulls inevitably seem to give rise to.

h.
```
CONSTRAINT CXG
   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 CASE ) ;
```

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 : { SC := ( (‼SP) JOIN S ) { CITY } } )
                                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. The interesting thing about this one (or one of the interesting things, at any rate) is that it's ambiguous. It might mean that every individual London supplier must supply more different kinds of part than every individual Paris supplier; or it might mean that the number of different kinds of parts supplied by London suppliers considered en masse

must be greater than the number of different kinds of parts supplied by Paris suppliers considered en masse; and there might be other interpretations, too. The following formulations assume the second of these interpretations, but the whole question of ambiguity is revisited in Chapter 11.

```
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 NATURAL JOIN SP
    WHERE  CITY = 'London' ) >
  ( SELECT COUNT ( DISTINCT PNO ) FROM S NATURAL JOIN SP
    WHERE  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 ) , 0 ) FROM S NATURAL JOIN SP
     WHERE  CITY = 'London' ) >
   ( SELECT COALESCE ( SUM ( QTY ) , 0 ) FROM S NATURAL JOIN SP
     WHERE  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 ) ) ;
```

8.22 Constraint CX22a certainly suffices (it's directly analogous to the formulation I gave for CX4 in the body of the chapter). As for constraint CX22b: Well, let's see if we can *prove* it does the job. First of all, to simplify the discussion, let's agree to ignore supplier names, since they're irrelevant to the matter at hand. Then we need to show, first, that if the FD {CITY} → {STATUS} holds, then S is equal to the join of its projections on {SNO,CITY} and {CITY,STATUS}; second, if S is equal to the join of its projections on {SNO,CITY} and {CITY,STATUS}, then the FD {SNO} → {CITY} holds. Denote S{SNO,CITY} and S{CITY,STATUS} by SC and CT, respectively, and denote JOIN{SC,CT} by J. Adopting an obvious shorthand notation for tuples, then, we have for the first part of the proof:

- Let $(s,c,t) \in S$. Then $(s,c) \in SC$ and $(c,t) \in CT$, and so $(s,c,t) \in J$; so $S \subseteq J$.

- Let $(s,c,t) \in J$. Then $(s,c) \in SC$; hence $(s,c,t') \in S$ for some t'. But $t = t'$ thanks to the FD, so $(s,c,t) \in S$ and hence $J \subseteq S$. It follows that $S = J$.

Turning to the second part:

- Let both (*s*,*c*,*t*) and (*s'*,*c*,*t'*) ∈ S. Then (*s*,*c*) ∈ SC and (*c*,*t'*) ∈ CT, so (*s*,*c*,*t'*) ∈ J; hence (*s*,*c*,*t'*) ∈ S. But {SNO} is a key for S and so *t* = *t'* (because certainly (*s*,*c*,*t*) ∈ S); hence the FD {CITY} → {STATUS} holds.

It follows that constraint CX22b does indeed represent the FD, as required. Note carefully, however, that it does so only because we were able to appeal (in the second part of the proof) to the fact that {SNO} is a key for relvar S; it would not correctly represent the desired FD, absent that key constraint.

8.23 It guarantees that the constraint is satisfied by an empty database (i.e., one containing no relvars). Note, however, that it's logically unnecessary, because—as we saw in Chapter 7—TRUE is in fact the identity value for logical AND; thus, the default "total database constraint" for an empty database is simply TRUE, anyway.

8.24 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)!

8.25 We need a multiple assignment (if we are to do the delete in a single statement as requested). Let the supplier number of the specified supplier be *Sx*. Then:

```
DELETE S WHERE SNO = Sx , DELETE SP WHERE SNO = Sx ;
```

The individual assignments (DELETEs) can be specified in either order.

8.26 These constraints can't be expressed declaratively in either SQL or **Tutorial D,** since neither of those languages currently has any direct support for transition constraints. Triggered procedures can be used, but details of triggered procedures are beyond the scope of this book. However, here are possible formulations using the "primed relvar name" convention discussed briefly in the section "Miscellaneous Issues" in the body of the chapter:

a.
```
CONSTRAINT CXA IS_EMPTY
    ( P WHERE SUM ( ‼SP , QTY ) > SUM ( ‼SP' , QTY ) ) ;
```

b.
```
CONSTRAINT CXB
    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' ) ;
```

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. *Note:* An analogous remark applies to transition constraints in general. In other words, such constraints provide a certain degree of protection against unintentional mistakes, but they don't and can't provide protection against deliberately malicious acts.

8.27 *No answer provided.*

8.28 SQL fails to support type constraints for a rather complicated reason having to do with its approach to type inheritance. The specifics are beyond the scope of this book; as noted in the body of the chapter, however, you can find further details in the book *Databases, Types, and the Relational Model: The Third Manifesto,* by Hugh Darwen and myself (see Appendix G). 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, or even −1000).

8.29 In principle they all apply—though **Tutorial D** in particular deliberately provides no way of specifying constraints, other than a priori ones, for either nonscalar or system defined types.

8.30 The generic expansion of an arbitrary UPDATE in terms of DELETE and INSERT can be inferred by straightforward generalization from the following simple, albeit somewhat abstract, example. Let relvar *R* have just two attributes, *X* and *Y*. Consider the following UPDATE on *R*:

```
UPDATE R WHERE X = x : { Y := y } ;
```

Let the current ("old") value of *R* be *r*. Define *d* and *i* as follows:

```
d  =  { t : t ∈ r AND t.X = x }
```

```
i  =  { t' : EXISTS t ∈ d ( t.X = t'.X ) AND t'.Y = y }
```

Then the original UPDATE is logically equivalent to the following assignment:

```
R  :=  r MINUS ( d UNION i ) ;
```

Or equivalently to the following multiple assignment:

```
DELETE R d , INSERT R i ;
```

(The DELETE and INSERT here can be specified in either order.)

Chapter 9

SQL and Views

They're concerned by adverse publicity and that I have to move more into public eye.
Problem is to define first the exact view we want to project.

—H. R. Haldeman:
The Haldeman Diaries: Inside the Nixon White House (1994)

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; in other words, 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—rather, 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 (see the previous bullet item), I should explain that all virtual relvars are derived but some derived ones aren't virtual. See the section "Views and Snapshots," later in this 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" (more precisely, it's a named relational expression).

As usual, in what follows I'll discuss these ideas in both relational and SQL terms. But talking of SQL, 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 fact, the most important thing about a view is precisely that it's a table (just as, in mathematics, the most important thing about, say, the union of two sets is precisely that it's a set). 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 the SQL language itself. Indeed, it could be

argued that the very names of the operators CREATE TABLE and CREATE VIEW in SQL are and always were 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 the database should "always" be accessed through views, see the section "SQL Column Naming" 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 now I 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 views are *restriction* views—their value at any given time is a certain restriction of the value at that time of relvar S. Some syntax issues:

- The parentheses in the SQL examples are unnecessary but not wrong; I include them for clarity. The parentheses in the **Tutorial D** examples are required.
- CREATE VIEW in SQL allows a parenthesized commalist of view 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 "SQL Column Naming" instead. For example, the foregoing view 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 we're telling the system 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—but only if!—SQL 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've 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 UNION NLS ) ;
```

Note: In order to guarantee that this design is logically equivalent to the original one, we would also have to state and enforce certain additional constraints—including 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 for simplicity. See the sections "Views and Constraints" and "Update Operations" later for further consideration of such matters.

Be that as it may, 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 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.[1]) 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. In other words, views should "look and feel" just like base relvars so far as users are concerned.

Here are some implications of this principle:

- As in fact I've already suggested, 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 this position isn't really tenable.) See the section "Views and Constraints," later.

- In particular, views have keys (and so I should perhaps have included some key specifications in my examples of views 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. 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 keys in general, but let that pass).

- Many SQL products, and the SQL standard, provide some kind of "row ID" feature.[2] If that feature is available for base tables but not for views—which in practice is quite likely—then it violates *The Principle of Interchangeability*. (It probably violates *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 have to be prohibited. 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 at least some of the major SQL products—as some kind of *object* ID in the object oriented sense, then they *are* prohibited, very

---

[1] I've touched on this notion before in this book—see the answer to Exercise 3.7 in Chapter 3 (though at that point I wasn't using the term *information equivalence* as such). See also Exercise 9.13.

[2] In the standard, that feature goes by the name of *REF types* and *reference values* (see Chapter 2).

definitely! 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 relvar and multirelvar constraints is more apparent than real (and the terminology is therefore deprecated, somewhat, for that very reason). Indeed, an example in that chapter—essentially the same London vs. non London suppliers example, in fact—showed that the very same constraint (viz., "suppliers numbers are unique") could be a single relvar constraint with one design for the database and a multirelvar 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*. Again, see the section "Update Operations," later.

Relation Constants

You might have noticed that, in the formal definition I gave at the beginning of the present section for what a view was, I said the defining expression had 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 AS
     ( SELECT temp.*
       FROM ( VALUES ( 'S1' , 'Smith' , 20 , 'London' ) ,
                     ( 'S2' , 'Jones' , 10 , 'Paris'  ) ,
                     ( 'S3' , 'Blake' , 30 , 'Paris'  ) ,
                     ( 'S4' , 'Clark' , 20 , 'London' ) ,
                     ( 'S5' , 'Adams' , 30 , 'Athens' ) )
              AS temp ( SNO , SNAME , STATUS , CITY ) ) ;
```

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*. To 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 referred to as a *self-defining* symbol—that denotes a constant (as in fact we already know from Chapter 2).

- 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 brief discussion I'll take the term *constant* to mean a named constant specifically, for brevity.

- Constants can be of any type you like, naturally, but relation constants (i.e., constants of some relation type) are my major focus here. Now, **Tutorial D** doesn't currently allow users to define their own 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 relation constant or "relcon" functionality along the lines sketched above. In fact, **Tutorial D** does already provide two system defined relcons: namely, TABLE_DUM and TABLE_DEE, both of which are extremely important, as we know. But that's the only "relcon" support it provides, and SQL doesn't provide any at all. It's true that (as we've seen) such support can be simulated by means of the conventional view mechanism; however, there's a serious logical difference involved here, and I don't think it helps the cause of understanding to pretend that constants are variables.

VIEWS AND PREDICATES

The Principle of Interchangeability says that views are supposed to "look and feel" just like base relvars. It follows that (a) a view must have a relvar predicate, and further that (b) the parameters to that predicate must 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 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 value London. (Precisely for this reason, in fact, a more realistic version of view LS would probably project away the CITY attribute. I prefer not to do this here, in order to keep the example and corresponding discussion as simple as possible.)

In similar fashion, the predicate for view NLS can be stated thus:

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 (a) users should be able to operate on views as if they were base relvars and (b) 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 defined" 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 ) ;   /* LS is the "London suppliers" view */
```

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 in principle quite straightforward. By way of example, suppose we issue this SQL query on the London suppliers view LS (and here I deliberately show all dot qualifications explicitly):

```
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 first be simplified—see the section "Expression Transformation" in Chapter 6—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 foregoing procedure works precisely because of the relational closure property. Closure implies among other things that wherever we're allowed to use the name of a variable to denote the value of the variable in question—for example, in a query—we can always replace that name by a more general expression (just so long as that expression denotes a value of the appropriate type, of course). 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. *Note:* For obvious reasons, the foregoing procedure (i.e., for implementing read-only operations on views) is known as the *substitution* procedure.

Incidentally, it's worth noting that the substitution procedure didn't always work in early versions of SQL—to be specific, in versions prior to SQL:1992—and 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 SST
       FROM   S
       GROUP  BY CITY ) ;
```

Now a query:

```
SELECT CITY
FROM   V
WHERE  SST > 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      /* warning: invalid !!! */
GROUP  BY CITY
```

(This expression is invalid because SQL doesn't allow "set function" invocations like SUM(STATUS) to be used in the WHERE clause in this manner.)

Now, the standard has been fixed in this regard, as you probably know;[3] however, it doesn't follow that the products have!—and as a matter of fact, the last time I looked (admittedly some time ago now) there was at least one major product that 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. (It probably won't perform very well, either. Not to mention the point that, in any case, such a technique clearly won't work for updates as opposed to retrievals.)

VIEWS AND CONSTRAINTS

To repeat, *The Principle of Interchangeability* says that views are supposed to "look and feel" just like base relvars. It follows 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 defining expression. By way of example, consider view LS once again. That view is a restriction of relvar S—i.e., its 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 constraint 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. For

---

[3] For the example under discussion, according to my own reading of the standard, the substitution procedure now yields an expression along the following lines: SELECT CITY FROM S WHERE (SELECT AST FROM (SELECT CITY, SUM (STATUS) AS AST FROM S GROUP BY CITY)) > 25.

one thing, the system might not be "intelligent" enough to carry out the inferences needed to determine for itself the constraints that apply to some view; for another, such explicit declarations can at least serve documentation purposes (i.e., they can help 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, then, 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. 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 S.*
       FROM   S
       WHERE  S.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 always 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—in other words, for any table that can be defined by some arbitrarily complex table expression (which is to say, any table at all).[4] 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, 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,[5] and declaring it explicitly for view LS might conceivably lead to the same constraint being checked twice. (But it should still be stated as part of the view documentation, somehow, because it's certainly part of the semantics of the view.)

---

[4] Any table, that is, so long as the definition of the table in question doesn't involve a possibly nondeterministic expression, a complication I choose to ignore for now. See Chapter 12 for further discussion.

[5] A more accurate statement is: The specification KEY{SNO} applies to view LS *as a logical consequence of* the fact that it applies to base relvar S. Note, however, that the two specifications don't mean the same thing—the one for view LS means suppliers numbers are unique with respect to London suppliers, the one for base relvar S means they're unique with respect to *all* suppliers.

Perhaps more to the point, there definitely are situations where declaring view constraints explicitly could be a good idea. Here's an example, expressed in 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 ) ) ;
```

The tables have predicates as follows:[6]

- FDH: *Flight FLIGHT leaves at hour HOUR for destination DESTINATION.*
- DFGP: *On day DAY, flight FLIGHT with pilot PILOT leaves from gate GATE.*

They're subject to the following constraints (expressed here in a kind of pseudo logical style):

```
IF ( f1,n1,h ) , ( f2,n2,h ) ∈ FDH AND
   ( d,f1,g,p1 ) , ( d,f2,g,p2 ) ∈ DFGP
THEN f1 = f2 AND p1 = p2 /* and n1 = n2, incidentally */

IF ( f1,n1,h ) , ( f2,n2,h ) ∈ FDH AND
   ( d,f1,g1,p ) , ( d,f2,g2,p ) ∈ DFGP
THEN f1 = f2 AND g1 = g2 /* and n1 = n2, incidentally */
```

Explanation: The first of these constraints says:

a. If two rows of FDH, one for flight *f1* (with destination *n1*) and one for flight *f2* (with destination *n2*), have the same HOUR *h*, and

b. If two rows of DFGP, one each for the FLIGHTs *f1* and *f2* from those two FDH rows, have the same DAY *d* and GATE *g*, then

[6] The tables are rather obviously not very well designed, and for that reason you might think I'm "stacking the deck" in an attempt to make my point seem more convincing than it really is. So I'd like to say the example isn't really mine at all; rather, it's a lightly edited version of one from Joe Celko's article "Back to the Future" (*Database Programming & Design 4*, No. 12, December 1991).

c. Those two FDH rows must be one and the same and those two DFGP rows must be one and the same. In other words, if we know the HOUR, DAY, and GATE, then the FLIGHT and PILOT (and DESTINATION) are determined.

The second constraint is analogous:

a. If two rows of FDH, one for flight *f1* (with destination *n1*) and one for flight *f2* (with destination *n2*), have the same HOUR *h*, and

b. If two rows of DFGP, one each for the FLIGHTs *f1* and *f2* from those two FDH rows, have the same DAY *d* and PILOT *p*, then

c. Those two FDH rows must be one and the same and those two DFGP rows must be one and the same. In other words, if we know the HOUR, DAY, and PILOT, then the FLIGHT and GATE (and DESTINATION) 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 OR
                      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
     ( SELECT * FROM FDH NATURAL JOIN DFGP ,
       UNIQUE ( DAY , FLIGHT )            /* this is      */
       UNIQUE ( DAY , HOUR , GATE ) ,     /* hypothetical */
       UNIQUE ( DAY , HOUR , PILOT ) ) ;  /* syntax !!!!! */
```

Explanation: The join of FDH and DFGP is on {FLIGHT} and is a one to many join, and it has heading {FLIGHT,DESTINATION,HOUR,DAY,GATE,PILOT}. Moreover, since {DAY,FLIGHT} is a key for the many side of the join (i.e., relvar DFGP), it's clearly a key for the result. But as we saw from our earlier analysis of the two stated constraints, the following functional dependencies also hold in that result:

```
{ DAY , HOUR , GATE } → { FLIGHT , PILOT , DESTINATION }

{ DAY , HOUR , PILOT } → { FLIGHT , GATE , DESTINATION }
```

It follows that {DAY,HOUR,GATE} and {DAY,HOUR,PILOT} are also keys for the result of the join. Hence the UNIQUE specifications shown.

Unfortunately, however, the foregoing solution isn't available to us, because SQL doesn't allow UNIQUE specifications (or constraint specifications of any kind, of course) on views. But we can and should at least specify those hypothetical view constraints in terms of suitable assertions, as follows:

```
CREATE VIEW V AS ( SELECT * FROM FDH NATURAL JOIN DFGP ) ;

CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , FLIGHT FROM V ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE FROM V ) ) ;

CREATE ASSERTION VCX3
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT FROM V ) ) ;
```

In fact, of course, we don't actually need to define the view V in order to define these constraints—we could simply replace the references to view V in the UNIQUE expressions in the constraints by the defining expression for V, like this:[7]

```
CREATE ASSERTION VCX1
       CHECK ( UNIQUE ( SELECT DAY , FLIGHT
                        FROM   FDH NATURAL JOIN DFGP ) ) ;

CREATE ASSERTION VCX2
       CHECK ( UNIQUE ( SELECT DAY , HOUR , GATE
                        FROM   FDH NATURAL JOIN DFGP ) ) ;
```

[7] If you look carefully, you'll see I'm not *exactly* replacing those references to V by the defining expression for V. The reason is that (as we saw in Chapter 6) SQL requires an explicit JOIN invocation like FDH NATURAL JOIN DFGP to have a "SELECT * FROM" prefix if it appears at the outermost level of nesting but allows it not to have such a prefix otherwise.

```
CREATE ASSERTION VCX3
       CHECK ( UNIQUE ( SELECT DAY , HOUR , PILOT
                        FROM   FDH NATURAL JOIN DFGP ) ) ;
```

Note: I didn't mention the point in Chapter 8, but **Tutorial D** does provide direct support for saying the relation denoted by some relational expression is required to satisfy some key constraint.[8] By way of illustration, here are **Tutorial D** analogs of assertions VCX1, VCX2, and VCX3:

```
CONSTRAINT VCX1 ( FDH JOIN DFGP ) KEY { DAY , FLIGHT } ;

CONSTRAINT VCX2 ( FDH JOIN DFGP ) KEY { DAY , HOUR , GATE } ;

CONSTRAINT VCX3 ( FDH JOIN DFGP ) KEY { DAY , HOUR , PILOT } ;
```

UPDATE OPERATIONS

Note: Much of the discussion in this section is based on material from my book View Updating and Relational Theory: Solving the View Update Problem *(O'Reilly, 2013). Refer to Appendix G for further details.*

I claimed earlier that *The Principle of Interchangeability* implies that views must be updatable. 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 S JOIN P—a many to many join, observe—of relvars S and P on the basis of their sole common attribute 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 a tuple" or "deleting a tuple" in the relational model (updates, like all relational operations, are always set level). But to be too pedantic about such matters in the present discussion would get in the way of understanding, probably.

Well, even if were true—which in fact it isn't, as I'll show later—that updates can't be done on a view like S JOIN P, let me point out that some updates can't be done on some base relvars, either. For example, inserting a tuple into base relvar SP will fail if the SNO value in that tuple doesn't currently exist in base relvar S. Thus, updates on base relvars can always fail on integrity constraint violations—*and the same is true for updates on views*. So it isn't that some views are inherently nonupdatable; rather, it's just that some updates on some views—like some updates on some base relvars—fail on integrity constraint violations (i.e., violations of **The Golden Rule**).

---

[8] The same goes for foreign key constraints, as a matter of fact. See the paper "Inclusion Dependencies and Foreign Keys" (mentioned in Appendix G).

Let's look at a detailed example—not the S JOIN P example (I'll get back to that one later) but the "London vs. non London suppliers" example once again, involving views LS and NLS. By *The Principle of Interchangeability*, the behavior of these two relvars, and indeed that of relvar S also, shouldn't depend on which relvars if any are base ones and which if any are views. Until further notice, therefore, let's suppose all three are base relvars:

```
VAR S   BASE RELATION { ... } KEY { SNO } ;
VAR LS  BASE RELATION { ... } KEY { SNO } ;
VAR NLS BASE RELATION { ... } KEY { SNO } ;
```

As the definitions show, {SNO} is a key for each of these relvars. (As a matter of fact, {SNO} in each of relvars LS and NLS is a foreign key, referencing the key {SNO} in relvar S, though I haven't bothered to show these constraints explicitly.) The relvars are also clearly subject to the following additional constraints:[9]

```
CONSTRAINT ... LS  = ( S WHERE CITY = 'London' ) ;
CONSTRAINT ... NLS = ( S WHERE CITY ≠ 'London' ) ;
```

What's more, these constraints taken singly or together imply certain additional ones, as follows:

```
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;

CONSTRAINT ... S = UNION { LS , NLS } ;
CONSTRAINT ... IS_EMPTY ( JOIN { LS { SNO } , NLS { SNO } } ) ;
```

The first two of these additional constraints are self-explanatory; the third says every supplier is represented in either LS or NLS, and the fourth says no supplier is represented in both. (In other words, the union in the third constraint is actually a disjoint union, and the join in the fourth constraint is actually an intersection.)

Now, in order to ensure these constraints remain satisfied when updates are done, certain *compensatory actions* (or rules) need to be in effect. In general, a compensatory action is an additional update, over and above some update that's requested by the user, that's performed automatically by the DBMS, precisely in order to avoid some integrity violation that might otherwise occur. Cascade delete is a typical example (see Chapter 5).[10] In the case at hand, in fact, it should be clear that "cascading" is exactly what we need to deal with DELETE operations in particular. First, deleting tuples from either LS or NLS clearly needs to "cascade" to cause

---

[9] Formally, these two constraints are *equality dependencies* (EQDs). In general, an EQD is an expression of the form *rx* = *ry*, where *rx* and *ry* are relational expressions of the same type; it can be read as "The relations denoted by *rx* and *ry* are equal" (in other words, they're one and the same relation).

[10] Cascade delete in particular is usually thought of as applying to foreign key constraints specifically, but the concept of compensatory actions is actually more general—it applies to constraints of all kinds. Also, don't get the idea that such actions must always take the form of simple "cascades"; while all of the examples examined in the present subsection do happen to take that form, more complicated cases might well require actions of some less straightforward form.

those same tuples to be deleted from S; so we might imagine a couple of compensatory actions—actually cascade delete rules—that look something like this (hypothetical syntax):

```
ON DELETE ls FROM LS : DELETE ls FROM S ;

ON DELETE nls FROM NLS : DELETE nls FROM S ;
```

Second, deleting tuples from S clearly needs to "cascade" to cause those same tuples to be deleted from whichever of LS or NLS they appear in:

```
ON DELETE s FROM S : DELETE ( s WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( s WHERE CITY ≠ 'London' ) FROM NLS ;
```

Aside: Given that an attempt (via DELETE, as opposed to I_DELETE) to delete a nonexistent tuple has no effect—see Chapter 5—this latter rule could be simplified to just this:

```
ON DELETE s FROM S : DELETE s FROM LS , DELETE s FROM NLS ;
```

However, the original formulation is perhaps preferable, inasmuch as it's clearly more specific. *End of aside*.

Analogously, we'll need some compensatory actions ("cascade insert rules") for INSERT operations:

```
ON INSERT ls INTO LS : INSERT ls INTO S ;

ON INSERT nls INTO NLS : INSERT nls INTO S ;

ON INSERT s INTO S : INSERT ( s WHERE CITY = 'London' ) INTO LS ,
                     INSERT ( s WHERE CITY ≠ 'London' ) INTO NLS ;
```

As for explicit UPDATE operations, they can be regarded, at least in the case at hand, as a DELETE followed by an INSERT; in other words, the compensatory actions for UPDATE are just a combination of the actions for DELETE and INSERT, loosely speaking.[11] For example, consider the following UPDATE on relvar S:

```
UPDATE S WHERE SNO = 'S1' : { CITY := 'Oslo' } ;
```

What happens here is this:

[11] In fact this state of affairs holds true in all cases, not just for the particular example under consideration (see Exercise 8.30 in Chapter 8), though the details might not always be entirely straightforward. For further explanation, see the book mentioned earlier, *View Updating and Relational Theory: Solving the View Update Problem* (again, see Appendix G).

1. The existing tuple for supplier S1 is deleted from relvar S and (thanks to the cascade delete rule from S to LS) from relvar LS also.

2. Another tuple for supplier S1, with CITY value Oslo, is inserted into relvar S and (thanks to the cascade insert rule from S to NLS) into relvar NLS also. In other words, the tuple for supplier S1 has moved from relvar LS to relvar NLS!—now speaking *very* loosely, of course.

   Suppose now that the original UPDATE had been directed at relvar LS rather than relvar S:

```
UPDATE LS WHERE SNO = 'S1' : { CITY := 'Oslo' } ;
```

Now what happens is this:

1. The existing tuple for supplier S1 is deleted from relvar LS and (thanks to the cascade delete rule from LS to S) from relvar S also.

2. An attempt is made to insert another tuple for supplier S1, with CITY value Oslo, into relvar LS. This attempt fails, however, because it violates the constraint on that relvar that the CITY value must always be London. So the update fails overall; the first step (viz., deleting the original tuple for supplier S1 from LS and S) is undone, and the net effect is that the database remains unchanged.

And now I come to my real point: *Everything I've said in this discussion so far applies pretty much unchanged if some or all of the relvars concerned are views.* For example, suppose as we originally did that S is a base relvar and LS and NLS are views:

```
VAR S   BASE RELATION { .................. } KEY { SNO } ;
VAR LS  VIRTUAL ( S WHERE CITY = 'London' ) KEY { SNO } ;
VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' ) KEY { SNO } ;
```

Now consider a user who sees only views LS and NLS, but wants to be able to behave as if those views were actually base relvars. Of course, that user will be aware of the corresponding relvar predicates, which as we saw earlier are essentially as follows:

- LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (which is London).*

- NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (which isn't London).*

That same user will also be aware of the following constraints (as well as the fact that {SNO} is a key for both relvars):

```
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( JOIN { LS { SNO } , NLS { SNO } } ) ;
```

However, the user won't be aware of any of the compensatory actions as such, precisely because that user isn't aware that LS and NLS are actually views of relvar S; in fact, the user won't even be aware of the existence of relvar S (which is why the user is also unaware of the constraint that the union of LS and NLS is equal to S). But updates by that user on relvars LS and NLS will all work correctly, just as if LS and NLS really were base relvars.

What about a user who sees only view LS, say (i.e., not view NLS and not base relvar S), but still wants to behave as if LS were a base relvar? Well, that user will certainly be aware of the pertinent relvar predicate and the following constraint:

```
CONSTRAINT ... IS_EMPTY ( LS WHERE CITY ≠ 'London' ) ;
```

Clearly, this user mustn't be allowed to insert tuples into this relvar, nor to update supplier numbers within this relvar, because such operations have the potential to violate constraints of which this user is unaware (and must be unaware). Again, however, there are parallels with base relvars as such: With base relvars in general, it'll be the case that certain users will be prohibited from performing certain updates on certain relvars (e.g., consider a user who sees only base relvar SP and not base relvar S). So this state of affairs doesn't in and of itself constitute a violation of *The Principle of Interchangeability*, either.

One last point: Please understand that I'm *not* suggesting that the DBA should have to specify, explicitly, all of the various constraints and compensatory actions that apply in connection with any given view. Au contraire: In many cases if not all, I believe the DBMS should be able to determine those constraints and actions for itself, automatically, from the pertinent view definitions.[12]

From the foregoing discussion, I hope you can see that it's not that updates are intrinsically impossible on views; rather, it's just that some updates on some views fail on a violation of **The Golden Rule**. It follows that in order to support updates on a view *V* properly, the system needs to know the total relvar constraint, *VC* say, that applies to *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 realize, 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 (and this is true of the standard as well as the major products). I'll have more to say regarding the specifics of SQL's view updating support in the

[12] Again I refer you to the book *View Updating and Relational Theory: Solving the View Update Problem* for further discussion.

next subsection but one ("More on SQL"). First, however, I want discuss one particular aspect of that support in some detail: namely, the CHECK option.

The CHECK Option

Consider the following SQL INSERT on view LS ("London suppliers") from the previous subsection:

```
INSERT INTO LS ( SNO , SNAME , STATUS , CITY )
       VALUES  ( 'S6', ............. , 'Madrid' ) ;
```

This INSERT maps to:

```
INSERT INTO S  ( SNO , SNAME , STATUS , CITY )
       VALUES  ( 'S6', ............. , 'Madrid' ) ;
```

(The only change is to 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; however, precisely because it doesn't satisfy the defining expression for view LS, it won't be visible through that view. From the perspective of that view, in other words, the new row just drops out of sight (alternatively, we might say the INSERT is a "no op"—again, from the perspective of the view). Actually, however, what's happened from the perspective of view LS is that *The Assignment Principle* has been violated! (Recall from Chapter 5 and elsewhere that *The Assignment Principle* states that after assignment of value *v* to variable *V*, the comparison *v* = *V* must evaluate to TRUE.)

Now, I hope it goes without saying that the foregoing behavior is logically incorrect. It wouldn't be 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 defining expression for that view. **Recommendation:** Specify WITH CASCADED CHECK OPTION on view definitions whenever possible. Be aware, however, that SQL permits such a specification only if it regards the view as updatable,[13] and (as we'll see in the next subsection) not all logically updatable views are regarded as such in SQL.

Note: The alternative to CASCADED is LOCAL, but don't use it. (The reason I say this is that the semantics of LOCAL are bizarre in the extreme—so bizarre, in fact, that (a) I don't want to waste time and space and energy attempting to explain them here, and in any case (b) it's hard to see why anyone would ever want such semantics. Indeed, it's hard to resist the suspicion that LOCAL was included in the standard originally for no other reason than to allow certain flawed

[13] And then only if the view defining expression isn't possibly nondeterministic (see Chapter 12). Incidentally, note the implication here that SQL does allow updates on "possibly nondeterministic views," and the further implication that SQL is thus apparently quite willing to allow certain updates to have unpredictable results! This state of affairs strikes me as odd, given that (as far as I know) the rationale for not allowing possibly nondeterministic expressions in constraints was precisely to avoid updates having unpredictable results.

implementations, extant at the time, to be able to claim conformance.) 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 extract (which is quoted verbatim from the 2003 version of the standard, SQL:2003) 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 <query expression body> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.
>
> b) If *QE1* simply contains a <query expression body> *QEB* that specifies UNION ALL, then:
>
> i) *QEB* 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 *QEB*, 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 <query term> that specifies INTERSECT.
>
> d) *QE2* is updatable.

Here's my own gloss on the foregoing extract:

- First of all, it doesn't even seem to make sense, at least on the face of it. To be specific, the opening sentence says, in effect, that four conditions a), b), c), and d) have to be satisfied "for every ... *QE2* that is simply contained in *QE1*"—yet item b) in particular has nothing to do with *QE2* (indeed, it doesn't even mention it).

- Next, even if I'm wrong and the extract does make sense, note that it states 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 portions of the standard.

- All of those rules rely on a variety of additional concepts and constructs—updatable columns, leaf generally underlying tables, <query term>s, and so on—that are in turn defined in still further portions of the standard.

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 proper support for (a) constraint inference, (b) **The Golden Rule,** and (c) *The Assignment Principle*, and thanks also to the fact that SQL permits (d) nulls and (e) duplicate rows. And the picture is complicated still further by the fact that SQL identifies four distinct cases: A view in SQL can be *updatable*, *potentially updatable*, *simply updatable*, or *insertable into*. Now, the standard does define these terms formally, but it 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.[14] 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's therefore possible that DELETE and INSERT might not be inverses of each other. Both of these facts, if facts they are, I regard as 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 defining expression is either (a) a simple SELECT expression (not a UNION, INTERSECT, or EXCEPT involving two such expressions) 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).

---

[14] The asymmetry here is intuitively odd. For example, an argument might be made—not by me!—that you can do DELETEs but not INSERTs on a union view, because the delete rule is obvious (delete from both operands) but the insert rule isn't (do we insert into both operands or just one—and if just one, which?). But an exactly analogous argument would surely say you can do INSERTs but not DELETEs on an intersection view (?). In other words, if some views are "deletable from but not insertable into," then surely others must be "insertable into but not deletable from."

- The SELECT clause in that SELECT expression implicitly or explicitly specifies ALL, not DISTINCT.

- After expansion of any "asterisk style" items, every item in the SELECT item commalist is a simple column name (possibly dot qualified, and possibly with a corresponding AS specification), and no such item appears more than once.

- The FROM clause in that SELECT expression takes the form FROM *T* [AS ...], where *T* is the name 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.

The S JOIN P Example

Now I'd like to come back to the S JOIN P example. The truth is, the example discussed in detail earlier (London vs. non London suppliers) was so simple that I suspect some readers might still be harboring doubts about my general claim—my claim, that is, that all views are updatable, modulo only possible **Golden Rule** violations. In an attempt to buttress that claim further, therefore, I want to examine a case that, historically, many people have regarded as "impossible": to be specific, the many to many join case, of which S JOIN P is an example.

First let me simplify matters somewhat by eliminating considerations that are irrelevant to my purpose. To be specific, let's assume, purely for the purposes of the present discussion, that relvar S has just two attributes, SNO and CITY, and relvar P also has just two attributes, PNO and CITY. Now let's define their join as a view called SCP:

```
VAR SCP VIRTUAL ( S JOIN P ) KEY { SNO , PNO } ;
```

Sample values are shown in Fig. 9.1. *Note:* As you can see from that figure, I've dropped from our usual sample values any supplier whose city isn't also a part city and any part whose city isn't also a supplier city. Please note, however, that I don't intend to maintain this simplification throughout the discussion that follows; that is, I'm not going to assume for the purposes of that discussion that every supplier city has to be a part city and vice versa, even though I do need to simplify the presentation somewhat for space and other reasons.

S

SNO	CITY
S1	London
S2	Paris
S3	Paris
S4	London

P

PNO	CITY
P1	London
P2	Paris
P4	London
P5	Paris
P6	London

SCP

SNO	CITY	PNO
S1	London	P1
S1	London	P4
S1	London	P6
S2	Paris	P2
S2	Paris	P5
S3	Paris	P2
S3	Paris	P5
S4	London	P1
S4	London	P4
S4	London	P6

Fig. 9.1: Relvars S, P, and SCP–sample values

As in the case of London vs. non London suppliers, the first thing I'm going to do is see what happens if we think of view SCP as just another base relvar, living alongside the base relvars in terms of which it's defined. Clearly, then, the following constraint holds:

```
CONSTRAINT ... SCP = S JOIN P ;
```

Now let's consider some updates on relvar S (since the roles played by suppliers and parts are clearly symmetric in this example, there's no need to consider updates on relvar P as well). First an INSERT:[15]

```
INSERT ( S5 , Athens ) INTO S ;
```

It should be clear that this INSERT does exactly what it says, no more and no less. And the same goes for the following DELETE, which removes the tuple just inserted:

```
DELETE ( S5 , Athens ) FROM S ;
```

But what about this INSERT?—

```
INSERT ( S7 , Paris ) INTO S ;
```

The point here is, of course, that there are some parts in Paris, viz., parts P2 and P5. Thus, this INSERT can and will succeed, just so long as it additionally has the effect of inserting the following tuples into relvar SCP:

```
( S7 , Paris , P2 )
( S7 , Paris , P5 )
```

[15] In keeping with our simplifying assumption that it makes sense to talk about inserting and deleting individual tuples, in this section I'm going to use a kind of pseudocode style—which I trust is self-explanatory—for INSERT and DELETE operations.

Moreover, the following DELETE can and will now succeed, just so long as it has the additional effect of removing those extra tuples from relvar SCP:

```
DELETE ( S7 , Paris ) FROM S ;
```

From these examples and others like them, I hope it's clear that the following compensatory actions are appropriate:

```
ON INSERT s INTO S : INSERT ( P JOIN s ) INTO SCP ;
ON INSERT p INTO P : INSERT ( S JOIN p ) INTO SCP ;

ON DELETE s FROM S : DELETE ( P JOIN s ) FROM SCP ;
ON DELETE p FROM P : DELETE ( S JOIN p ) FROM SCP ;
```

I turn now to updates on the join SCP. The compensatory action (or rule) for INSERT is fairly obvious:

```
ON INSERT i INTO SCP :
   INSERT i { SNO , CITY } INTO S ,
   INSERT i { PNO , CITY } INTO P ;
```

Note: I say this rule is obvious, but its consequences might not be, at least not immediately. Let's look at a couple of examples, using the sample values from Fig. 9.1:

1. Suppose we insert (S9,London,P1) into SCP. This INSERT will cause (S9,London) to be inserted into S but will have no effect on P, because (P1,London) already appears in P. But inserting (S9,London) into S will cause the insert rule for S to come into play, and the net effect will be that (S9,London,P4) and (S9,London,P6) will be inserted into SCP in addition to the originally requested tuple (S9,London,P1).

2. Suppose we insert (S7,Paris,P7) into SCP. The net effect will be to insert (S7,Paris) into S, (P7,Paris) into P, and all of the following tuples into SCP:

```
( S7 , Paris , P7 )

( S7 , Paris , P2 )
( S7 , Paris , P5 )

( S2 , Paris , P7 )
( S3 , Paris , P7 )
```

Now, the foregoing insert rule (for inserts on SCP) might loosely be characterized as "Insert S subtuples unless they already exist and insert P subtuples unless they already exist." Thus, intuition and symmetry both suggest that the delete rule (for deletes on SCP) should be

"Delete S subtuples unless they exist elsewhere and delete P subtuples unless they exist elsewhere."[16] Formally:

```
ON DELETE d FROM SCP :
   DELETE ( ( S MATCHING d ) NOT MATCHING SCP ) FROM S ,
   DELETE ( ( P MATCHING d ) NOT MATCHING SCP ) FROM P ;
```

Again let's consider some examples, using the sample values from Fig. 9.1:

1. Suppose we delete all tuples from SCP where the city is Paris. This DELETE will cascade to delete the tuples for suppliers S2 and S3 from relvar S and the tuples for parts P2 and P5 from relvar P.

2. Suppose we delete all tuples for supplier S1 from SCP. This DELETE will cascade to delete the tuple (S1,London) from relvar S but will have no effect on relvar P, because SCP still contains some tuples where the city is London—to be specific, the tuples (S4,London,P1), (S4,London,P4), and (S4,London,P6).

3. Suppose we attempt to delete just the tuple (S1,London,P1) from SCP. This attempt must fail; since SCP contains other tuples for both supplier S1 and part P1, the attempted DELETE has no effect on relvars S and P, and so if it were allowed to succeed we would have a **Golden Rule** violation on our hands (to be specific, SCP would no longer be equal to the join of S and P).

I'll leave it as an exercise for you to show that, given the foregoing insert and delete actions, explicit UPDATEs all work as intuitively expected.

And now, as I'm sure you've been expecting, I'm going to claim that everything I've been saying in the foregoing discussion applies pretty much unchanged if some or all of the relvars concerned are views. In particular, let S and P be base relvars as usual and let SCP be a view:

```
VAR S   BASE RELATION { .................. } KEY { SNO } ;
VAR P   BASE RELATION { .................. } KEY { PNO } ;
VAR SCP VIRTUAL ( S JOIN P ) KEY { SNO , PNO } ;
```

Now consider a user who sees only relvar SCP (the view). As far as that user is concerned, then, that view will behave in all respects exactly as if it were a base relvar (though it's only fair to point out that the behavior in question won't be entirely straightforward, as I'll explain in a moment). The predicate is:

---

[16] Some writers have suggested that the "and" in that informal characterization should really be "or," meaning in terms of the S JOIN P example that (e.g.) deleting Paris tuples from SCP can and should be achieved by deleting Paris tuples just from S or just from P instead of from both. Arguments for and against this position are considered in detail in *View Updating and Relational Theory: Solving the View Update Problem*. For the purposes of the present discussion, I'll stay with the rule as stated here.

Supplier SNO and part PNO both have city CITY.

The user will be aware of this predicate, and aware also of the fact that {SNO,PNO} is a key. Moreover, the user will also be aware that the following functional dependencies (FDs) hold as well:

```
{ SNO } → { CITY }
{ PNO } → { CITY }
```

These FDs are effectively inherited from relvars S and P, respectively.

Now, I didn't point this out before, but in fact the following *multivalued* dependencies (MVDs) also hold in SCP (and the user will be aware of these MVDs, too):

```
{ CITY } →→ { SNO } | { PNO }
```

I don't want to get into details about MVDs in general (they're discussed in depth in the book *Database Design and Relational Theory: Normal Forms and All That Jazz*, a companion to the present book). All I want to say here is that (a) the fact that these particular MVDs hold mean that relvar SCP isn't in fourth normal form (4NF),[17] and (b) together, these MVDs are equivalent to the following constraint:

```
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

(i.e., SCP is equal at all times to the join of its projections on {SNO,CITY} and {PNO,CITY}).

Since SCP isn't in 4NF, there are bound to be situations where updating it turns out to be a little awkward. (Incidentally, note that this observation is valid regardless of whether SCP is a base relvar or a view.) Let me be more specific. First of all, updates in general must abide by those MVDs, of course. Second, INSERTs in particular are subject to the following rule:

```
ON INSERT i INTO SCP :
   INSERT ( SCP JOIN i { SNO , CITY } ) INTO SCP ,
   INSERT ( SCP JOIN i { PNO , CITY } ) INTO SCP ;
```

This rule might look a little complicated, but it's basically just a combination of the earlier rules for INSERTs on S, P, and SCP, revised to eliminate references to S and P as such.

Aside: Observe the implication that such a rule ought to, and indeed does, apply even in the case where SCP is a base relvar and relvars S and P don't exist (or are hidden); indeed, we could have arrived at this rule by considering just relvar SCP in isolation. Observe

[17] As a matter of fact it isn't even in second normal form (2NF), thanks to those FDs {SNO} → {CITY} and {PNO} → {CITY}. But it's the violation of 4NF that leads to the relvar's characteristic behavior as a many to many join.

further that I don't give a corresponding delete rule. In fact, however, DELETEs on SCP will always fail unless they request, explicitly or implicitly, deletion of all tuples for some particular supplier(s) and/or deletion of all tuples for some particular part(s). E.g., given the sample values shown in Fig. 9.1, a request to delete just the tuple (S1,London,P1) will fail, while a request to delete all Paris tuples will succeed (as indeed we saw earlier in both cases). *End of aside*.

I'd like to close this rather lengthy section on view updating by repeating something I said earlier (because I think it's important): Please understand that I'm *not* suggesting that the DBA should have to specify, explicitly, all of the various constraints and compensatory actions that apply in connection with any given view. On the contrary, I believe that (in many cases if not all) the DBMS should be able to determine those constraints and actions for itself, automatically, from the pertinent view definitions.

WHAT ARE VIEWS FOR?

So far in this chapter, I've been tacitly assuming 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 defining expression *exp*. Thus, that user can use the name *V* wherever the expression *exp* is intended; however, such uses are basically just shorthand, and are explicitly understood to be just shorthand by the user in question. (What's more, the user in question is unlikely to request any updates on *V*—though if such updates are requested, they must perform as expected, of course.)

- 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 *exp*; 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, then, 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 isn't important here) 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 saw earlier, 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 } ;
```

(Note that now I've specified D_UNION instead of UNION, for explicitness.) Any expression that previously referred to base relvar S will now refer to view S instead. Thus, 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 S. See Exercise 9.8 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 perhaps 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 "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. Of course, other REFRESH options are possible too: for example, EVERY MONDAY, EVERY 5 MINUTES, EVERY MONTH, and so on.

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 such 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;[18] as we've seen, operations on views are supposed to be implemented by mapping them into suitable operations on the underlying relvars. Thus, "materialized view" is simply a contradiction in terms. Worse yet, the unqualified term *view* is now often taken to mean a "materialized view" specifically—again, at least in some circles—and so we're in danger of no longer having a good term to mean a view in the original sense. In this book 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

9.1 Define a view consisting of supplier-number : part-number pairs for suppliers and parts that aren't colocated. Give both **Tutorial D** and SQL definitions.

9.2 Let view LSSP be defined as follows (SQL):

---

[18] 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 for implementation reasons.

```
CREATE VIEW LSSP
  AS ( SELECT SNO , SNAME , STATUS , PNO , QTY
       FROM   S NATURAL JOIN SP
       WHERE  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?

9.3 What key(s) does view LSSP from Exercise 9.2 have? What's the predicate for that view?

9.4 Given the following **Tutorial D** view definition—

```
VAR HP VIRTUAL ( P WHERE WEIGHT > 14.0 ) KEY { PNO } ;
```

—show the converted form after the substitution procedure has been applied for each of the following expressions and statements:

a. `HP WHERE COLOR = 'Green'`

b. `( EXTEND HP : { W := WEIGHT + 5.3 } ) { PNO , W }`

c. `INSERT HP RELATION { TUPLE { PNO 'P9' , PNAME 'Screw' , WEIGHT 15.0 , COLOR 'Purple' , CITY 'Rome' } } ;`

d. `DELETE HP WHERE WEIGHT < 9.0 ;`

e. `UPDATE HP WHERE WEIGHT = 18.0 : { COLOR := 'White' } ;`

9.5 Give SQL solutions to Exercise 9.4.

9.6 Give as many reasons as you can think of for wanting to be able to declare keys for a view.

9.7 Using either the suppliers-and-parts database or any other 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.

9.8 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 noted that such a replacement wasn't a completely trivial matter. Why not?

9.9 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?*

9.10 Distinguish between views and snapshots. Does SQL support snapshots? Does any product that you're aware of?

9.11 What's a "materialized view"? Why is the term deprecated?

9.12 Consider the suppliers-and-parts database, but ignore relvar P 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 of 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?

9.13 Following on from the previous exercise: In the body of the chapter, I said two database designs were information equivalent if they represented the same information (meaning that for every query on one, there's a logically equivalent query on the other). But can you pin down this notion more precisely?

9.14 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?

ANSWERS

9.1
```
VAR NON_COLOCATED VIRTUAL
    ( ( S { SNO } JOIN P { PNO } ) NOT MATCHING ( S JOIN P ) )
    KEY { SNO , PNO } ;

CREATE VIEW NON_COLOCATED
  AS ( SELECT SNO , PNO
       FROM   S , P
       WHERE  S.CITY <> P.CITY
       /* UNIQUE ( SNO , PNO ) */ ) ;
```

9.2 Substituting the view definiting expression for the view reference in the outer FROM clause, we obtain:

```
SELECT DISTINCT STATUS , QTY
FROM ( SELECT SNO , SNAME , STATUS , PNO , QTY
       FROM   S NATURAL JOIN SP
       WHERE  CITY = 'London' ) AS LSSP
WHERE  PNO IN
     ( SELECT PNO
       FROM   P
       WHERE  CITY <> 'London' )
```

This simplifies (potentially!) to:

```
SELECT DISTINCT STATUS , QTY
FROM   S NATURAL JOIN SP
WHERE  CITY = 'London'
AND    PNO IN
     ( SELECT PNO
       FROM   P
       WHERE  CITY <> 'London' )
```

9.3 The sole key is {SNO,PNO}. The predicate is: *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in London, and supplies part PNO in quantity QTY.*

9.4 Note that a. and b. are expressions, the rest are statements.

a. `( P WHERE WEIGHT > 14.0 ) WHERE COLOR = 'Green'`

This expression can be simplified to:

```
P WHERE WEIGHT > 14.0 AND COLOR = 'Green'
```

The simplification is worth making, too, because the first formulation implies (or at least suggests) two passes over the data while the second implies just one.

b.
```
( EXTEND ( P WHERE WEIGHT > 14.0 ) :
                  { W := WEIGHT + 5.3 } ) { PNO , W }
```

c.
```
INSERT ( P WHERE WEIGHT > 14.0 )
       RELATION { TUPLE { PNO 'P9' , PNAME 'Screw' , WEIGHT 15.0 ,
                                     COLOR 'Purple' , CITY 'Rome' } } ;
```

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, although the corresponding syntax isn't currently supported in **Tutorial D** (neither for assignment in general nor for INSERT in particular). *Note:* Similar but not identical remarks apply to parts d. and e. below.

d.
```
DELETE ( P WHERE WEIGHT > 14.0 ) WHERE WEIGHT < 9.0 ;
```

This syntax is currently illegal, although oddly enough the following (which is obviously logically equivalent to that just shown) is legal:

```
DELETE P WHERE WEIGHT > 14.0 AND WEIGHT < 9.0 ;
```

Of course, this DELETE is actually a "no op," because WEIGHT > 14.0 AND WEIGHT < 9.0 is a logical contradiction. Do you think the optimizer would be able to recognize this fact?

e.
```
UPDATE ( P WHERE WEIGHT > 14.0 ) WHERE WEIGHT = 18.0 :
                                          { COLOR := 'White' } ;
```

Again this syntax is currently illegal, but the following is legal:

```
UPDATE P WHERE WEIGHT > 14.0 AND WEIGHT = 18.0 :
                                    { COLOR := 'White' } ;
```

Do you think the optimizer would be able to recognize the fact that the restriction condition WEIGHT > 14.0 here can be ignored?

9.5 Here first is an SQL version of the view definition from Exercise 9.4:

```
CREATE VIEW HP AS
     ( SELECT PNO , PNAME , COLOR , WEIGHT , CITY
       FROM   P
       WHERE  WEIGHT > 14.0
       /* UNIQUE ( PNO ) * / ) ;
```

For parts a.-e., I first show an SQL analog of the **Tutorial D** formulation, followed by the expanded form:

a.
```
SELECT HP.PNO , HP.PNAME , HP.COLOR , HP.WEIGHT , HP.CITY
FROM   HP
WHERE  HP.COLOR = 'Green'

SELECT HP.PNO , HP.PNAME , HP.COLOR , HP.WEIGHT , HP.CITY
FROM ( SELECT PNO , PNAME , COLOR, WEIGHT , CITY
       FROM   P
       WHERE  WEIGHT > 14.0 ) AS HP
WHERE  HP.COLOR = 'Green'
```

I leave further simplification, here and in subsequent parts, as a subsidiary exercise (barring explicit statements to the contrary).

b.
```
SELECT PNO , WEIGHT + 5.3 AS W
FROM   HP

SELECT HP.PNO , HP.WEIGHT + 5.3 AS W
FROM ( SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.CITY
       FROM   P
       WHERE  P.WEIGHT > 14.0 ) AS HP
```

c.
```
INSERT INTO HP ( PNO , PNAME , WEIGHT , COLOR , CITY )
       VALUES ( 'P9' , 'Screw' , 15.0 , 'Purple' , 'Rome' ) ;

INSERT INTO ( SELECT P.PNO , P.PNAME , P.WEIGHT , P.COLOR , P.CITY
              FROM   P
              WHERE  P.WEIGHT > 14.0 ) AS HP
       VALUES ( 'P9' , 'Screw' , 15.0 , 'Purple' , 'Rome' ) ;
```

The remarks regarding **Tutorial D** in the solution to Exercise 9.4c apply here also, mutatis mutandis.

d.
```
DELETE FROM ( SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.COLOR
              FROM   P
              WHERE  P.WEIGHT > 14.0 ) AS HP
WHERE HP.WEIGHT < 9.0 ;
```

This 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 ;
```

(As noted in the answer to Exercise 9.4d, this DELETE is actually a "no op.")

e.
```
UPDATE ( SELECT P.PNO , P.PNAME , P.COLOR , P.WEIGHT , P.COLOR
           FROM   P
           WHERE  P.WEIGHT > 14.0 ) AS HP
SET    COLOR = 'White'
WHERE  HP.WEIGHT = 18.0 ;
```

Syntactically valid equivalent:

```
UPDATE P
SET    COLOR = 'White'
WHERE  WEIGHT = 18.0 AND WEIGHT > 14.0 ;
```

9.6 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, so the user of a view needs to know what keys that view has. Explicitly declaring those keys is the obvious way to make that information available.

- The DBMS might be unable to infer keys for itself (this is almost certainly the case, in general, 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 infer keys for itself, explicit declarations would at least enable the system to check that its inferences 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 inferences.

- As shown 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 some 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?

9.7 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. (What are those constraints exactly?) So we could make those projections base relvars and make the join a view. See also the answer to Exercise 9.12.

9.8 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: */

LS  := ( S WHERE CITY = 'London' ) ;

NLS := ( S WHERE CITY ≠ 'London' ) ;

/* drop the old relvar: */

DROP VAR S ;

/* create the desired view: */

VAR S VIRTUAL ( LS D_UNION NLS ) KEY { SNO } ;
```

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 simply be taken as referring to the view S instead.[19] However, 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 populate this relvar:

[19] 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.

```
SS := S { SNO } ;
```

(This assignment assumes that relvar S hasn't been dropped yet. Alternatively, we could assign to SS the union of LS{SNO} and NLS{SNO}.)

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.

9.9 a. *No answer provided*—except to note that if it's hard 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., but more so. c. Ditto.

9.10 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 last part of 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.)

9.11 "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 relational model is concerned—and it's leading us into a situation in which we no longer have a clear term for a concept that we did have a clear term for, originally. It should be firmly resisted. (I realize I've probably already lost this battle, but I'm an eternal optimist.) 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.

9.12 First, here's a definition of Design b. in terms of Design a. (pertinent constraints included):

```
VAR SSP VIRTUAL ( S JOIN SP )
    KEY { SNO , PNO } ;

VAR XSS VIRTUAL ( S NOT MATCHING SP )
    KEY { SNO } ;

CONSTRAINT B_FROM_A IS_EMPTY ( SSP { SNO } JOIN XSS { SNO } ) ;
```

(Constraint B_FROM_A and the specified key constraints are together what we would have to tell the user if we wanted the user to think of relvars SSP and XSS as base relvars, not views.) 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 } ;

CONSTRAINT A_FROM_B IS_EMPTY ( SP NOT MATCHING S ) ;
```

Given these constraints, the designs are information equivalent. But Design a. is superior, because the relvars in that design are in fifth normal form (5NF). By contrast, relvar SSP in Design b. isn't even in second normal form; as a consequence, it displays redundancy and is thereby subject to certain "update anomalies." Consider also what happens with Design b. if some supplier ceases to supply any parts, or used not to supply any but now does. Further discussion of the problems with Design b. is beyond the scope of this book; I just note that (as the example suggests) database design disciplines like normalization can help with the task of choosing "the best" design from a set of designs that are information equivalent.

Incidentally, I note in passing that—given that {SNO} is a key for relvar S—constraint A_FROM_B here shows another way of formulating a referential constraint. In practice, of course, it would be simpler just to include the following foreign key specification as part of the definition of relvar SP:

```
FOREIGN KEY { SNO } REFERENCES S
```

9.13 The following discussion relies on the fact that (as Appendix A explains in more detail) databases are really variables—i.e., we really need to draw a distinction between database values and database variables, analogous to that between relation values and relation variables. Let *DBD1* and *DBD2* be (logical) database designs; let *DB1* and *DB2* be database variables conforming to *DBD1* and *DBD2*, respectively; and let *db1* and *db2* be the current values of *DB1* and *DB2*, respectively. Further, let there exist mappings *M12* and *M21*—i.e., sequences of relational algebra operations, loosely speaking—that transform *db1* into *db2* and *db2* into *db1*, respectively. Then *db1* and *db2* are information equivalent, meaning that for every expression involving only relations from *db1*, there's an expression involving only relations from *db2* that evaluates to the same result (and vice versa).

Now let database variables *DB1* and *DB2* be such that for every possible value *db1* of *DB1* there exists an information equivalent value *db2* of *DB2* (and vice versa). Then *DB1* and *DB2* per se are information equivalent, as are the corresponding designs *DBD1* and *DBD2*.

Now let database variables *DB1* and *DB2*, as well as their current values *db1* and *db2*, be information equivalent. Let *U1* be an update on *DB1* that transforms *db1* into *db1'*. Then there must exist an update *U2* on *DB2* that transforms *db2* into *db2'*, such that *db1'* and *db2'* are information equivalent. Note that the remarks of this paragraph apply in particular to the case in which *DB1* consists only of base relvars and *DB2* consists only of views of relvars in *DB1*.

Finally, let database variables *DB1* and *DB2*, as well as their current values *db1* and *db2*, *not* be information equivalent. Then there must exist an expression involving only relations from *db1* with no counterpart involving only relations from *db2* (or vice versa), and there must exist an update on *DB1* with no counterpart on *DB2* (or vice versa) —speaking somewhat loosely in both cases. Again note that the remarks of this paragraph apply in particular to the case in which *DB1* consists only of base relvars and *DB2* consists only of views of relvars in *DB1*.

9.14 (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 (and the mapping specification in particular needs to be hidden from the user).

Chapter 10

SQL and Logic

Logic takes care of itself;
all we have to do is look and see how it does it.

—Ludwig Wittgenstein:
Tractatus Logico-Philosophicus (1922)

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 than an algebraic one, 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, of course, to consider the relevant 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 based on the algebra (which is why there aren't many references to **Tutorial D** in this chapter), and Query-By-Example and QUEL (see Appendix G) are both 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 (the latter explicitly, the former perhaps a little less so); indeed, such a goal was the prime motivation for the introduction of the SQL "IN subquery" construct.[1] 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 consequence is that today some aspects of SQL are "algebra like," some are "calculus like," and some are neither—with the further consequence that, as I mentioned in passing in Chapter 6, most queries, constraints, and so on that can be expressed in SQL at all can in fact be expressed in numerous different ways.

---

[1] The name SQL originally stood for *Structured Query Language*; the idea behind that name was that SQL queries typically consisted of subqueries nested inside other subqueries (i.e., that was the "structure" being alluded to). More specifically, SQL allowed such subqueries—i.e., SELECT – FROM – WHERE expressions, loosely speaking—to appear nested inside the WHERE clause of another such expression, recursively. But that was all! The ability to nest subqueries elsewhere (in particular, in the SELECT and FROM clauses) wasn't part of SQL as originally defined, nor was it added to the language until nearly 20 years later, as part of SQL:1992.

Aside: The goal mentioned in the previous paragraph—the goal, that is, of making SQL different from both the algebra and the calculus—was based on what I regard as a fundamental misconception: namely, the idea that the algebra and calculus were both somewhat "user hostile." But that perception, I believe, derived from 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 numerous writers and languages have demonstrated, to wrap that semantics up in syntax that's very user friendly indeed. *End of aside.*

WHY DO WE NEED LOGIC?

Logic is useful because (among other things) it's a great aid to clear and precise thinking. By contrast, everyone would surely agree that natural language is often vague and ambiguous. The following piece by Robert Graves and Alan Hodge illustrates the point delightfully:[2]

From the Minutes of a Borough Council Meeting:

Councillor Trafford took exception to the proposed notice at the entrance of South Park: "No dogs must be brought to this Park except on a lead." He pointed out that this order would not prevent an owner from releasing his pets, or pet, from a lead when once safely inside the Park.

The Chairman (Colonel Vine): What alternative wording would you propose, Councillor?

Councillor Trafford: "Dogs are not allowed in this Park without leads."

Councillor Hogg: Mr. Chairman, I object. The order should be addressed to the owners, not to the dogs.

Councillor Trafford: That is a nice point. Very well then: "Owners of dogs are not allowed in this Park unless they keep them on leads."

Councillor Hogg: Mr. Chairman, I object. Strictly speaking, this would keep me as a dog-owner from leaving my dog in the back-garden at home and walking with Mrs. Hogg across the Park.

Councillor Trafford: Mr. Chairman, I suggest that our legalistic friend be asked to redraft the notice himself.

---

[2] Thanks to Lauri Pietarinen of Relational Consulting Oy, Helsinki, Finland, for drawing this splendid example to my attention. The example is included and analyzed in detail in Ernest Nagel: "Symbolic Notation, Haddocks' Eyes, and the Dog-Walking Ordinance," in James Newman (ed.), *The World of Mathematics, Vol. 3*. Mineola, N.Y.: Dover Publications (2000).

Councillor Hogg: Mr. Chairman, since Councillor Trafford finds it so difficult to improve on my original wording, I accept. "Nobody without his dog on a lead is allowed in this Park."

Councillor Trafford: Mr. Chairman, I object. Strictly speaking, this notice would prevent me, as a citizen, who owns no dog, from walking in the Park without first acquiring one.

Councillor Hogg (with some warmth): Very simply, then: "Dogs must be led in this Park."

Councillor Trafford: Mr. Chairman, I object: This reads as if it were a general injunction to the Borough to lead their dogs into the Park.

Councillor Hogg interposed a remark for which he was called to order; upon his withdrawing it, it was directed to be expunged from the Minutes.

The Chairman: Councillor Trafford, Councillor Hogg has had three tries; you have had only two ...

Councillor Trafford: "All dogs must be kept on leads in this Park."

The Chairman: I see Councillor Hogg rising quite rightly to raise another objection. May I anticipate him with another amendment: "All dogs in this Park must be kept on the lead."

This draft was put to the vote and carried unanimously, with two abstentions.

Note: I can't resist pointing out that the final draft is *still* ambiguous—it could logically be interpreted to mean that all dogs in the park must be kept on the same (i.e., the one and only) lead. But enough of dogs ... Let's move on.

SIMPLE AND COMPOUND PROPOSITIONS

Recall from Chapter 5 that, in logic, a proposition is something that evaluates unequivocally to either TRUE or FALSE. Here are some examples:

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`

Of these, Nos. 1, 4, and 5 are true and Nos. 2 and 3 are false—though in the case of No. 5 we do need to be rather careful over what exactly we mean by "between"! (To be a little more precise about the matter, what I mean by it is this: If we denote the average distances of

Mercury, Venus, and Earth from the sun by m, v, and e, respectively, then $m < v < e$.) Be that as it may, a good informal test for whether something, p say, is a valid proposition is to ask whether "Is it true that p?" is a sensible question. For example, "Is it true that $2 + 3 > 7$?" is certainly a sensible question, even though the answer is no. To check your understanding of this point, which of the following do you think are legal propositions? (You might want to check the answers at the end of the chapter before reading further.)

- Bach is the greatest musician who ever lived.
- What's the time?
- Supplier S2 is located in some city, x.
- Some countries have a female president.
- All politicians are corrupt.
- Supplier S1 is located in Paris.
- We both have the same favorite author, x.
- Nothing is heavier than lead.
- It will rain tomorrow.
- Supplier S6's city is unknown.

By the way, there's an important point of detail here (which I'm mostly going to ignore—I mention it only to head off at the pass, as it were, certain criticisms that persons trained in formal logic might be tempted to level at this chapter). The truth is, a proposition isn't really a declarative sentence as such; rather, it's the assertion made by that sentence. For example, "It's hot" and "Il fait chaud" are clearly distinct sentences, but they both assert the same proposition. That said, I'll continue to assume from this point forward for simplicity that a proposition is indeed just a declarative sentence. Analogous remarks apply to predicates also (see later).

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, sometimes written "⇒"), and IF AND ONLY IF (also known as IFF, or BI-IMPLIES, or IS EQUIVALENT TO, and

sometimes written "⇔" 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 )`

I've introduced some 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 required. 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 one or more propositions as input and return another proposition as 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*, or *composite*. And the truth value of a compound proposition can be determined from the truth values of its 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

Neither style is more correct than the other; it's just that sometimes one is more convenient, sometimes the other is. Anyway, let's take a closer look at one of the foregoing compound propositions (number 9, to be specific). Here it is again:

```
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 logical 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, observe that there are exactly 16 dyadic connectives altogether, corresponding to the 16 possible dyadic truth tables (just four of which are shown above). *Note:* Exercise 10.1 asks you to draw all of those truth tables, and it might be worth having a go at that exercise right now.

- 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*. Similar remarks apply to all of the other connectives.

  Aside: The foregoing example (concerning AND) is perhaps a little misleading, in that it could be argued that "I was seriously disappointed" means different things in the two statements quoted. In the first, it means "I was seriously disappointed in the status quo"; in the second, it means "I was seriously disappointed in the the outcome of the vote." If this analysis is correct, it would be strictly incorrect to symbolize the two statements as *p* AND *q* and *q* AND *p*, respectively; although the two *q*'s are the same, the two *p*'s aren't. But the example is valuable nevertheless, in that it does at least show—not for the first

time, perhaps—that we have to be rather careful in mapping natural language utterances to their symbolic logic counterparts. *End of aside.*

- Of the 16 available dyadic connectives, the one called IMPLIES has behavior that most closely resembles that of implication as understood in natural language. For example, "if Mars has two moons, then Mars has at least one moon" is a valid implication, both in logic and in 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 by means of a truth table 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 (check the truth table), because the antecedent is false; yet whether Mars has two moons clearly 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 (which is 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, as you can see from the truth table. And, just incidentally, this equivalence serves to illustrate the point that the connectives NOT, AND, OR, IMPLIES, and IF AND ONLY IF aren't all primitive, since 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.[3] (*Exercise:* Check

[3] Conventional logic (i.e., so called two-valued logic, 2VL) is thus *truth functionally complete*. In general, a logic is truth functionally complete if and only if every possible connective can be defined in terms of the given ones. As noted in Exercise 4.11 in Chapter 4, truth functional completeness is an extremely important property; a logic without it would be like an arithmetic that was missing certain operations (the operation of addition, say) and would thus be of extremely limited utility.

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 negative, and consider the following SQL expression:

```
SELECT ...
FROM   ...
WHERE  X >= 0 AND SQRT ( X ) <= 100 ...
```

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.

Contrapositives

Consider again the database constraint discussed above in connection with logical implication:

```
IF ( COLOR = 'Red' ) THEN ( CITY = 'London' )
```

Let me now point out that this expression is logically equivalent to the following one:

```
IF NOT ( CITY = 'London' ) THEN NOT ( COLOR = 'Red' )
```

This latter expression is the *contrapositive* of the previous one. In general, in fact, we have the following equivalence:

```
IF p THEN q  ≡  IF NOT q THEN NOT p
```

As a matter of fact we've seen several examples of contrapositives in this book already. For example:

- Chapter 6 stated, in connection with certain operators such as join, that "attributes with the same name must be of the same type; i.e., attributes of different types must have different names" (slightly paraphrased). Each half of this statement is the contrapositive of the other.

- The discussion of constraint CX1 in Chapter 8 (effectively, though perhaps a little tortuously) relied on the fact that each of the following expressions—

  ```
  IF STATUS < 1 OR STATUS > 100 THEN FALSE
  ```

  and

  ```
  IF TRUE THEN STATUS ≥ 1 AND STATUS ≤ 100
  ```

  —is the contrapositive of the other.

- Chapter 8 also stated that "*correct* implies *consistent* (but not the other way around), and *inconsistent* implies *incorrect* (but not the other way around)." Again, each half of this statement is the contrapositive of the other.

Note that it's strictly expressions of the form IF *p* THEN *q*—that is, logical implications—to which the contrapositive notion applies. Be that as it may, here's a question for you: How many of the following expressions are logically distinct?

- `IF ( WEIGHT > 17.0 ) THEN ( CITY ≠ 'Paris' )`
- `IF ( CITY = 'Paris' ) THEN ( WEIGHT ≤ 17.0 )`
- `( WEIGHT ≤ 17.0 ) OR ( CITY ≠ 'Paris' )`
- `NOT ( ( CITY = 'Paris' ) AND ( WEIGHT > 17.0 ) )`

Well, I hope you can see that all four of these expressions in fact say the same thing. However, I think you'll agree also that this fact isn't immediately obvious! Let's take a closer look. Let's use *p* and *q* to denote the subexpressions (WEIGHT > 17.0) and (CITY ≠ 'Paris'), respectively. The four expressions become:

- IF *p* THEN *q*
- IF NOT *q* THEN NOT *p*
- NOT *p* OR *q*
- NOT ((NOT *q*) AND *p*)

Now I think it's a little easier to see that the four are indeed all equivalent to one another.[4] So the example demonstrates two things: First (to repeat), the equivalences aren't always obvious; second, introducing symbols like *p* and *q* allows us to manipulate the expressions in a

[4] Easier, yes, but it's still necessary to appeal to certain laws of transformation that I haven't yet explained (though they're intuitively obvious). See Chapter 11 for further discussion.

purely formal manner and makes it easier to see what's really going on (easier to see the forest as well as the trees, one might say). I'll have more to say about such matters in the next chapter.

SIMPLE AND COMPOUND PREDICATES

Consider the following statements:[5]

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* or *placeholders*. As a consequence, the statements aren't propositions (i.e., they aren't unequivocally either true or false), precisely because they do 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 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 paragraph immediately following makes clear.

Now, we can substitute *arguments* for those 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 unequivocally either true or false (in fact, of course, it's true). But the original statement as such ("*x* is a star") is, to say it again, not itself a proposition. Rather, it's a *predicate*, which—as you'll recall from Chapter 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 the predicate is 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* doesn't.

Note: Recall from Chapter 5 that logicians speak not of invoking a predicate but rather of *instantiating* it. (As a matter of 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 5.18 in

---

[5] As noted in Chapter 5, statements in logic aren't the same thing as statements in a programming language; in some respects, in fact, a statement in logic is more like a programming language *expression*, at least inasmuch as it denotes a value (a truth value, of course). In logic contexts, therefore, I'll use the terms *statement* and *expression* (both in this chapter and the next) more or less interchangeably—and I apologize if this usage on my part leads to any confusion.

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's 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." This example differs from the previous one ("*x* is a star") in that it involves two parameters, *x* and *m*. (By way of example, 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. And a proposition is a niladic predicate.

Next, 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*, or *composite*. Here's an example of a compound predicate:

```
16. ( 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* (one of which is referenced twice and the other once only).

Rules of Inference

It's a bit of a digression from my main purpose in this chapter, but as an aside I can now give a (somewhat loose) definition of *predicate logic*. Logic in general can be defined as *the science, or scientific study, of the methods and principles used in valid reasoning*. And predicate logic in particular can be defined as a formal system involving predicates and connectives and the inferences that can be made using such predicates and connectives. Observe, therefore, that predicate logic involves certain *rules of inference*—i.e., rules by which additional truths can be derived (or proved) from established truths. The additional truths are called theorems, and the established truths are either axioms or theorems that have previously been proved.

One important inference rule is called *modus ponens:* If we know that *p* is true, and if we also know that IF *p* THEN *q* is true, then we can infer that *q* is true. For example, given the truth of both "I have no money" and "If I have no money, then I will have to wash dishes," we can infer the truth of "I will have to wash dishes."

Another important inference rule is *modus tollen:* If we know that IF *p* THEN *q* is true, but we also know that *q* is false, then we can infer that *p* is false. This rule is relevant to the process of database integrity checking. Conceptually, what happens is this: When an update is

requested, the proposed new database value is checked against known integrity constraints; if the proposition expressed by some constraint—see Chapter 8—now evaluates to FALSE, that proposed new value must also represent falsehood, and so the update must be rejected.

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 (a process known to logicians as *instantiation*). 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 that can be substituted for 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 also a proposition, and it means: "All possible argument values a that can be substituted for 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 persons, in the examples). I'll come back to this latter point in the section "Relational Calculus," later.

The term used in logic for constructs 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 ("∃") and FORALL by an upside down A ("∀"). I use the keywords EXISTS and FORALL here for readability.

> *Aside:* At this point, one reviewer asked whether a quantifier is just another connective. No, it isn't. Let *p*(*x*) and *q*(*x*) be predicates, each with a single parameter *x*. Then *p*(*x*) and *q*(*x*) can be combined in various ways by means of connectives (as in, e.g., *p*(*x*) AND *q*(*x*)), 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*(*x*)) or FORALL *x* (*q*(*x*)), or even EXISTS *x* (*p*(*x*) AND *q*(*x*))—has the effect of converting the predicate concerned into something else (viz., a proposition). Thus, there's a clear logical difference between the two concepts. (Though I should add that, at least in the database context, the quantifiers can in fact be *defined in terms of* certain connectives. I'll explain this point later, in the section "More on Quantification.") *End of aside.*

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 unequivocally either true or false; in fact, it's a monadic predicate—it has a single parameter, *y*. Suppose we invoke this predicate with argument Steve. We obtain:

```
EXISTS x ( x is taller than Steve )
```

This statement *is* a proposition (and if there exists at least one person—Arnold, say—who's taller than Steve, then it evaluates to TRUE). 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 and to TRUE otherwise (think about it!).

There are several lessons to be learned from this simple 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 ) )
```

  However, I hope you can see 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 bullet item 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 bullet item.) I've also shown a precise natural language interpretation in each case. Note that those interpretations are all logically different!—in particular, some of them evaluate to TRUE and some to FALSE. Please note, however, that I've had to assume in connection with certain of those evaluations that there does exist at least one person "in the universe," as it were. I'll come back to this assumption in the section "More on Quantification," later.

```
EXISTS x ( EXISTS y ( x is taller than y ) )
```

  Meaning: Somebody is taller than somebody; 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 for present purposes I'll assume 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 (I apologize for the repetition, but the point is important): Even though five out of six of the foregoing propositions do 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 so far been calling parameters are more usually known in logic as *free variables*—and quantifying over a free variable, using either EXISTS or FORALL, converts that free variable 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*,[6] we obtain:

```
EXISTS x ( x is taller than y )
```

Now *y* is free (still) but *x* is bound. And if we now quantify existentially over *y* as well, we obtain:

```
EXISTS x EXISTS y ( x is taller than y )
```

Now *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

[6] Existentially just to be definite. Quantifying universally instead would make no difference to the point I'm making here.

conventional programming; 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 here that the variable *x* is constrained to "range over" the set of integers. Again, I'll come back to this question of "ranges" 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 and logically 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 isn't:

```
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 predicate EXISTS *y* (*y* > 3) AND *y* < 0, for example, it's the first two *occurrences* of the *reference* to *y* that are 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,[7] even though such talk is really quite sloppy. Be on your guard for confusion in this area!

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.

[7] Even, sometimes, in logic textbooks, where the practice really ought to be deprecated.

RELATIONAL CALCULUS

Essentially everything I've discussed 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 all and only those possible values of the proto tuple that satisfy the predicate (i.e., make it evaluate to TRUE). 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 the one in that (SNO,STATUS) pair and with part number P2.[8]

---

[8] *A couple of remarks on terminology:* First, 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). Second, I've referred to the construct following the keyword WHERE as a predicate. Syntactically speaking, however, it's just a boolean expression—but of course a boolean expression can be regarded as the concrete representation of a predicate. In general, I tend to use "predicate" when I'm discussing logic as such (including relational calculus), but "boolean expression" when I'm discussing some concrete language such as **Tutorial D** or SQL.

Note the use of dot qualified names in this example (in both the proto tuple and the predicate). I won't go into details, however, because dot qualified names will be familiar to you from SQL, I'm sure. 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' )
```

As this example indicates:

- First, SQL does support range variables, though it doesn't usually refer to them by that name:[9] The specifications S AS SX and SP AS SPX serve to define such variables, and those variables are then explicitly referenced elsewhere in the overall expression by means of dot qualified names such as SX.SNO, SPX.SO, and so on. *Note:* In practice, such AS specifications and such explicit range variable references are often omitted, at least in simple queries. I'll explain exactly how such omissions are possible in Chapter 12, when I discuss range variables in SQL in more detail.

- Second, and perhaps more important, SQL 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.[10] *Note:* 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 typically 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. Again, 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

---

[9] Actually the standard does, but products typically don't—they tend to use the term *correlation name* instead (see Chapter 12 for further discussion), or else some wildly inappropriate term such as *alias* or *table alias* or (surely the most grotesque of the many I've seen) *join variable*.

[10] 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.

model is based on). In 3VL, the existential quantifier can return three different results: TRUE, FALSE, and UNKNOWN (where UNKNOWN is "the third truth value"; 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.[11] 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. Example 3 in the next chapter is a case in point. *End of aside.*

Let's look at another example. Consider the query "Get names of suppliers who supply all parts." I'll assume we have the same range variables SX and SPX as before, but I'll also define another one (PX) ranging over P:

```
RANGEVAR PX RANGES OVER P ;

{ SX.SNAME } WHERE FORALL PX ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

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. 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 names of suppliers who supply at least one red part"):

```
{ SX.SNAME } WHERE EXISTS PX ( PX.COLOR = 'Red' AND
                               EXISTS SPX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

I'm assuming here that we have the same range variables available to us as we had in the earlier examples; in fact, I'll continue to make that same assumption throughout the rest of the chapter.

---

[11] To be a little more precise about the matter: Suppose *tx* denotes a nonempty restriction of some table *T* and the restriction condition evaluates to UNKNOWN for every row in *T*; then EXISTS (*tx*) ought logically to return UNKNOWN but will actually return TRUE, in SQL.

By the way, here's another possible formulation of the foregoing query:

```
{ SX.SNAME } WHERE EXISTS PX ( EXISTS SPX ( PX.COLOR = 'Red' AND
                                            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 of this concept (observe that the definition involves some recursion):

> **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 (a) $n \geq 0$; (b) each *Qi* ($i = 1, 2, ..., n$) is either EXISTS or FORALL; and (c) 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, and the easiest to write, in many cases (not all).

More on Range Variables

From what I've 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 and more conventional meaning (see Chapter 2). For such reasons, I won't adopt that usage in this book.[12]

Now I can say a little more about the syntax of relational calculus expressions:

---

[12] In practice, the term *tuple calculus* is used mainly to distinguish the version of the relational calculus discussed in the present chapter from the *domain* calculus, which is a version of the relational calculus in which the variables range over domains—i.e., types—instead of relations. But there's no need to discuss the domain calculus in this book; if you want to know more, you can find a detailed explanation in my book *An Introduction to Database Systems* (see Appendix G).

- 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 specification 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 in my examples 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. 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 for 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 or possibilities; however, I'm definitely 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, etc., have been defined as range variables over S; PX, PY, etc., have been defined as range variables over P; and SPX, SPY, 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 another 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.

```
{ SA := SX.SNO , SB := SY.SNO }
                  WHERE SX.CITY = SY.CITY AND SX.SNO < SY.SNO
```

Note the introduction of result attribute names SA and SB in this example. Incidentally, this example provides a good illustration of the point that some queries are more easily formulated in the calculus than they are in the algebra (if you recall, an algebraic formulation for this query, rather more complicated than the calculus formulation just shown, was given in the section "Formulating Expressions One Step at a Time" in Chapter 6).

Example 2: Get names of suppliers who supply at least one Paris part.

```
{ SX.SNAME } WHERE EXISTS SPX ( EXISTS PX ( SX.SNO = SPX.SNO AND
                                            SPX.PNO = PX.PNO AND
                                            PX.CITY = 'Paris' ) )
```

Example 3: Get names of 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 names of suppliers who don't supply part P2.

```
{ SX.SNAME } WHERE NOT ( EXISTS SPX ( SPX.SNO = SX.SNO AND
                                      SPX.PNO = 'P2' ) )
```

The outer parentheses in this example (i.e., the ones enclosing the expression following NOT) might not be needed in practice; indeed, I'll often omit such parentheses in later examples.

Incidentally, the predicate in the foregoing formulation isn't in prenex normal form, precisely because of that opening NOT. It would be possible to replace it by one that is, like this—

```
{ SX.SNAME } WHERE FORALL SPX ( SPX.SNO ≠ SX.SNO OR SPX.PNO ≠ 'P2' )
```

—but I don't think this alternative formulation is as "natural" as the non PNF version; that is, I think this example illustrates the point that a PNF formulation isn't always the one that comes most readily to mind.

Example 5: For each shipment, get full shipment details, including total shipment weight.

```
{ SPX , SHIPWT := PX.WEIGHT * SPX.QTY } 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, and probably image relations also.

Example 6: For each part, get the part number and the total shipment quantity.

```
{ PX.PNO , TOTQ := SUM ( SPX WHERE SPX.PNO = PX.PNO , QTY ) }
```

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 , TOTQ := SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO ) }
```

Answer: Because duplicate quantities would be eliminated before the sum is computed.

Example 7: Get 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 in formulating constraints. The first eight are based on, and use the same numbering as, the constraint examples 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 ≥ 1 AND SX.STATUS ≤ 100 ) ;
```

Note: SQL allows a constraint like this one to be simplified by (in effect) eliding both the explicit use of (a) the range variable and (b) 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 >= 1 AND STATUS <= 100 )
```

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 (i.e., {SNO} is a key, or rather a superkey, for relvar S).

```
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 ) ) ;
```

This formulation isn't very elegant, to say the least! I'll come back to this example and give a better formulation of it in the next section.

Example 4: Whenever two tuples in relvar S have the same supplier number, they also have the same city (in other words, the functional dependency {SNO} → {CITY} holds in relvar S).

```
CONSTRAINT CX4 FORALL SX ( FORALL SY ( IF SX.SNO = SY.SNO
                                       THEN SX.CITY = SY.CITY ) ) ;
```

As noted in Chapter 8, this constraint is actually a logical consequence of the fact that {SNO} is a superkey for relvar S. If this latter constraint is stated, therefore, constraint CX4 needn't be.

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 ) ) ;
```

As with Example 3, I'll have more to say about this 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 FORALL SX ( FORALL PX
   ( IF SX.SNO = 'S1' AND PX.PNO = 'P1' THEN 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. (By contrast, the expression EXISTS SX (FALSE) *always* evaluates to FALSE.

Conversely, the expression FORALL SX (FALSE) evaluates to TRUE if and only if SX ranges over an empty relation—see the discussion of empty ranges in the next section—while the expression FORALL SX (TRUE) always evaluates to TRUE.)

MORE ON QUANTIFICATION

There are a number of further issues I need to discuss regarding quantification in particular.

We Don't Need Both Quantifiers

It's easy to see that any predicate that can be expressed in terms of EXISTS can 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"; of course, this predicate is in fact a simple proposition). 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 all of the above 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 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 (users often have trouble with it); double negation, as in this example, 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 persons 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 colloquially, "No person over fifty feet tall works for IBM"—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 expression FORALL *x* (...) evaluates to TRUE if there are no *x*'s, *regardless of what appears inside the parentheses*. For example, "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 the foregoing state of affairs is that certain queries will 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.

Incidentally, the foregoing example serves as a good illustration of the point that while logic is certainly necessary as a foundation for database systems, it might not be sufficient. For example, how do you think an only child should respond to the question "Are your siblings all boys?" The logically correct answer is, of course, *yes* (though I observe that *yes* is the logically correct answer to the question "Are your siblings all girls?" as well). In practice, however, we would surely expect some more informative response, along the lines of "Well, actually I don't have any siblings." In other words—and now reverting to database systems as such—it might be nice if the system, as well as simply giving its responses as such, could also explain those responses if and when asked to do so.

Defining EXISTS and FORALL

As you might have already 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 (equivalently, if $n = 0$), 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.

So we see that 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.[13]

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. Also (and much more to the point, perhaps), the

---

[13] To elaborate: Consider by way of example the proposition EXISTS *x* ($p(x)$), where *p* is a predicate with just one parameter, *x*. If *x* ranges over an infinite set, then any attempt to use an "iterated OR" algorithm for evaluating the proposition will inevitably be flawed, since the algorithm might never terminate (it might never find the one value of *x* that satisfies *p*). Likewise, any attempt to use an "iterated AND" algorithm for FORALL *x* ($p(x)$) will also inevitably be flawed, since again the algorithm might never terminate (it might never find the one value of *x* that fails to satisfy *p*).

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.[14]

Other Kinds of Quantifiers

While it's true that the EXISTS and FORALL quantifiers are far and away the most important ones 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?

---

[14] *A note on syntax:* Recall from Chapter 7 that **Tutorial D** supports the aggregate operators AND and OR, thereby allowing us to write, e.g., AND (SP, QTY > 0), to express the fact that QTY values in relvar SP must be greater than zero. The discussions of the present section suggest that more "user friendly" names for these operators might well be FORALL and EXISTS, respectively. For example, the expression FORALL (SP, QTY > 0) does read quite well from an intuitive point of view. Likewise, EXISTS (SP, QTY > 250) seems to be an intuitively pleasing way of expressing the fact that at least one QTY value in relvar SP must be greater than 250.

As another exercise, what does the following predicate mean?

```
FORALL x ( UNIQUE y ( x has social security number y ) )
```

Here's how UNIQUE might be used in the formulation of constraints. Recall the formulation I gave earlier for constraint CX3 ("every supplier has a unique 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 ) ) ;
```

A much better formulation would clearly be as follows:

```
CONSTRAINT CX3 FORALL SX ( UNIQUE SY ( SX.SNO = SY.SNO ) ) ;
```

("For all suppliers SX, there's exactly one supplier SY with the same supplier number.") For example, if SX denotes the tuple for supplier S4, say, then SY must also denote the tuple for supplier S4—in other words, SX and SY must denote the very same tuple—in order for the constraint to be satisfied.

By way of another example, 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 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 even more indirect than its support for EXISTS is. 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. Note that it follows from this definition that the operator certainly returns TRUE if its argument table has either just one row or no rows at all.[15] And it further follows that, whereas the logic expression

```
UNIQUE x ( p ( x ) )
```

[15] By contrast, the UNIQUE quantifier gives FALSE if its range is empty.

means "There exists *exactly* one argument value *a* corresponding to the parameter *x* such that *p*(*a*) evaluates to TRUE," the (very approximate!) SQL analog—

```
UNIQUE ( SELECT k FROM T AS ... WHERE p ( x ) )
```

—where *k* denotes an arbitrary constant value, say the integer 0—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." For example, given our usual sample value for relvar S, the SQL expression

```
UNIQUE ( SELECT 0 FROM S AS SX WHERE SX.CITY = 'Athens' )
```

returns TRUE, while the SQL expression

```
UNIQUE ( SELECT 0 FROM S AS SX WHERE SX.CITY = 'Paris' )
```

returns FALSE.[16]

All of that being said, I won't attempt to give an SQL formulation here for constraint CX6 that uses UNIQUE—I'll leave it to Chapter 11 (see Example 10 in that chapter).

As you can see, the foregoing examples are designed to exploit the fact that SQL retains duplicates in the result of a SELECT expression if DISTINCT isn't specified. Of course, I've suggested elsewhere in this book—in Chapter 4, to be specific—that DISTINCT should "always" be specified. In contexts like the one under discussion, however, DISTINCT must definitely *not* be specified (right?).

Of course, I don't mean to suggest that the argument expression in an SQL UNIQUE invocation must always be of the form "SELECT *k* FROM ...," where *k* denotes some constant. By way of a counterexample, here repeated from Chapter 8 is one possible SQL formulation of the constraint that distinct suppliers must have distinct supplier numbers:

```
CREATE ASSERTION CX3 CHECK ( UNIQUE ( SELECT SNO FROM S ) ) ;
```

Recall now that SQL also uses the keyword UNIQUE in key constraints. For example, the CREATE TABLE for table S includes the following specification:

```
UNIQUE ( SNO )
```

You can think of this specification as shorthand for the following (which could be part of a more general base table constraint or a CREATE ASSERTION statement):

---

[16] It follows that AT_MOST_ONE (or perhaps NO_DUPS) would be a better name for the SQL operator than UNIQUE, at least in a context like the one under discussion. (Come to that, AT_LEAST_ONE might be a better name than EXISTS, too, both for the existential quantifier as such and for SQL's analog of that quantifier.)

```
CHECK ( UNIQUE ( SELECT SNO FROM S ) )
```

Aside: To repeat something I said (in a footnote) in Chapter 8, what the standard actually says in this connection is as follows (more or less): "The constraint UNIQUE (SNO) is not satisfied if and only if EXISTS (SELECT * FROM S WHERE NOT (UNIQUE (SELECT SNO FROM S))) evaluates to TRUE." Well, it seems to me this definition could surely be simplified, thus: "The constraint UNIQUE (SNO) is satisfied if and only if UNIQUE (SELECT SNO FROM S) evaluates to TRUE." Now, I dare say there's a good reason for what seems to me the standard's excessive circumlocution here, but whatever it is certainly escapes me. Perhaps it has to do with nulls, in which case I'm not interested. *End of aside*.

SQL also uses the keyword UNIQUE in MATCH expressions. Here's an example ("Get suppliers who supply exactly one part"):[17]

```
SELECT SX.*
FROM   S AS SX
WHERE  SX.SNO MATCH UNIQUE
     ( SELECT SPX.SNO
       FROM   SP AS SPX )
```

But this usage too is basically just shorthand. For example, the example just shown is equivalent to the following—

```
SELECT SX.*
FROM   S AS SX
WHERE  UNIQUE ( SELECT SPX.SNO             /* i.e., there's AT  */
                FROM   SP AS SPX           /* MOST one shipment */
                WHERE  SPX.SNO = SX.SNO )  /* for supplier SX   */
AND    EXISTS ( SELECT SPX.SNO             /* ... and there's   */
                FROM   SP AS SPX           /* also AT LEAST one */
                WHERE  SPX.SNO = SX.SNO )
```

Incidentally, note that the UNIQUE invocation here is indeed of the form "UNIQUE (SELECT *k* FROM ...)" where *k* denotes a constant value, because the boolean expression in the inner WHERE clause makes SPX.SNO constant with respect to "the current row" of table S, and hence with respect to each evaluation of the outer WHERE clause. Of course, SPX.SNO denotes different constants with respect to different evaluations of this latter clause.

SOME EQUIVALENCES

In this section I offer a few remarks regarding certain equivalences that might have already occurred to you (indeed, I've touched on some of them myself from time to time at earlier

[17] Note that in this context, by contrast, the SQL keyword UNIQUE does mean *exactly* one.

points). First of all, recall the **Tutorial D** 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 ) )`
- `FORALL x ( p ) ≡ IS_EMPTY ( X WHERE NOT ( p ) )`

(I'm assuming here that the variable *x* ranges over a set called *X*.)

Actually, SQL's support for EXISTS—and 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 EXISTS in SQL is evaluated by first evaluating the expression that denotes its sole argument, and then 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; various optimizations are possible, but they're irrelevant to the present discussion.)

And now I can explain why SQL doesn't directly 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 sense. For example, consider the hypothetical SQL 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 that hypothetical FORALL operator is being 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 further 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, 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 (the ones involving COUNT, I mean) 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 zero; rather, we want it to stop counting, for any given supplier, as soon as it finds the first shipment for that supplier. In other words, we'd really like some optimization to be done. Writing code that effectively *requires* a certain optimization to be done is usually not a good idea! **Recommendation:** Be careful over the use of COUNT, therefore; in particular, don't use it where EXISTS would be more logically correct. *End of aside.*

Relational Completeness

Every operator of the relational algebra has a precise definition in terms of logic. (I didn't call this point out explicitly before, but it should be obvious that the definitions I gave in Chapters 6 and 7 for join and the rest can be stated, perhaps a little more precisely, 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's an expression of the relational calculus that's logically equivalent to—i.e., has the same semantics as—that algebraic expression. In other words, the relational calculus is at least as "powerful" (better: *expressive*) as the relational algebra: Anything that can be expressed in the algebra can also be expressed in the calculus.

Now, it might not be obvious, but actually the opposite is true too—that is, for every expression of the calculus, there's an expression of the algebra that's logically equivalent to that calculus expression. Thus, the algebra is at least as expressive as the calculus, and so the two formalisms are logically equivalent: Both are what's called *relationally complete*.[18] Relational completeness is a basic measure of the expressive capability of a language; if a language is

[18] Don't confuse relational completeness with any other kind of completeness: in particular, with truth functional completeness, mentioned in an earlier footnote.

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 iterative loops or branching. 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.

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 like 1 = 0) 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 (a) relvar S is currently nonempty; (b) there's a constraint to the effect that there must always be at least one part; but (c) 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 query is evaluated directly, the result will be empty, because the expression in the WHERE clause evauates to FALSE. Alternatively, if the system (or the user) observes that there's a constraint that says that EXISTS PX (TRUE) must evaluate to TRUE—or, in SQL, that SELECT * FROM P must return a nonempty result—that WHERE clause can be replaced by one saying simply WHERE TRUE, and the result will then be all suppliers. 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 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 some proposition that relies on *p* in some way 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 a row 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 some 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 this 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 quantifiers are 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 who disagree with me here; that is, there are many 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 another book—*Logic and Databases: The Roots of Relational Theory* (Trafford, 2007)—and I'd like to quote the concluding remarks from that earlier discussion here:

> Surely it's worth investing a little effort up front in becoming familiar with [basic logic] 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 plan to demonstrate in the next chapter.

EXERCISES

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?

10.2 Let *p* and *q* stand for arbitrary propositions. Prove that

```
( ( NOT p ) AND ( p OR q ) ) IMPLIES q
```

is a tautology. (Recall from Chapter 4 that a *tautology* in logic is an expression that's guaranteed to evaluate to TRUE, regardless of the values of any variables involved. Likewise, a *contradiction* in logic is an expression that's guaranteed to evaluate to FALSE, regardless of the values of any variables involved.)

10.3 Again let *p* and *q* denote arbitrary propositions. Prove that the following is a tautology:

```
NOT ( p AND q )  ≡  ( NOT p ) OR ( NOT q )
```

10.4 (*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.

10.5 Consider the predicate "*x* is a star." (a) First, 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?* (b) Second, if the argument *the sun* is substituted for *x*, is the predicate satisfied? If not, why not? And what about the argument *the moon?*

10.6 Consider the predicate "*x* has two moons." If the argument *Jupiter* is substituted for *x*, is the predicate satisfied? Justify your answer.

10.7 Here's constraint CX1 once again from Chapter 8:

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Now, in Chapter 8, I said the relvar name "S", in the expression IS_EMPTY (S WHERE ...), was acting as a *designator*. But isn't it actually a parameter? If not, what's the difference?

10.8 (*Repeated from the body of the chapter.*) What query does the following expression represent? And do you think that query is a "sensible" one?

```
{ SX.SNAME } WHERE EXISTS SPX ( FORALL PX ( SPX.SNO = SX.SNO AND
                                            SPX.PNO = PX.PNO ) )
```

10.9 (*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.

10.10 Prove that AND and OR are associative.

10.11 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?[19] I remind you that the symbol "⇒" means *implies*; the symbol "≡" means *is equivalent to*. Note too that $A \Rightarrow B$ and $B \Rightarrow A$ are together the same as $A \equiv B$ (in other words, $(A \Rightarrow B \text{ AND } B \Rightarrow A) \equiv (A \equiv B)$ is a tautology).

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) )`

10.12 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) )  ≡  FORALL y EXISTS x ( p(x,y) )`

f. `EXISTS y FORALL x ( p(x,y) )  ⇒  FORALL x EXISTS y ( p(x,y) )`

---

[19] The term *valid* is something of a loaded word in logical contexts. I'm using it in these exercises simply to mean that the statement in question is true, regardless of what values are assigned to any variables involved (in other words, the statement in question is a tautology).

10.13 Where possible and reasonable, give relational calculus solutions to exercises from Chapters 6-9.

10.14 Consider this query: "Get cities in which either a supplier or a part is located." Can this query be expressed in the relational calculus? If not, why not?

10.15 Is SQL relationally complete? *Note:* To prove it is, you need to show that for every expression of the relational algebra there exists a semantically equivalent expression in SQL. Alternatively, to prove it isn't, you need to show there exists at least one expression of the relational algebra for which no such SQL equivalent exists.

10.16 Is prenex normal form always achievable?

ANSWERS

First of all, in the body of the chapter I asked which of the following natural language sentences were legal propositions. My own answers are as follows (but note that some of them, at least, might be open to debate):

- Bach is the greatest musician who ever lived. *Yes*. *Subsidiary exercise:* Is the proposition true?
- What's the time? *No*. It doesn't make sense to ask "Is it true that *p?*" where *p* is "What's the time?"
- Supplier S2 is located in some city, *x*. My own answer here is *yes*, but you might well argue that it should be *no*. In fact, the example is a good illustration of the ambiguity problem discussed near the beginning of the chapter; my answer is based on the assumption that the phrase "some city, *x*" could be abbreviated to just "some city" without changing the overall meaning of the sentence,[20] but you might reasonably argue the opposite. Compare and contrast "We both have the same favorite author, *x*" (see below).
- Some countries have a female president. *Yes*.
- All politicians are corrupt. *Yes*.

[20] In other words, I'm claiming that *x* here is a bound variable, and hence that the sentence could be rephrased thus: "There exists some city, say *x*, such that supplier S2 is located in city *x*."

- Supplier S1 is located in Paris. *Yes* (though it's false, given our usual sample values).
- We both have the same favorite author, *x*. *No*. Here I'm assuming that the phrase "the same favorite author, *x*" couldn't be abbreviated to just "the same favorite author" without changing the overall meaning of the sentence (but you could reasonably argue the opposite). If my assumption is correct, then *x* is a variable (better: a *parameter*), and until we know what argument is to be substituted for that parameter—i.e., until we know what that *x* stands for—we can't assign a truth value to the sentence. But when we do know—i.e., when we substitute Jane Austen, say, for *x*—then the sentence does become a proposition.
- Nothing is heavier than lead. *Yes*.
- It will rain tomorrow. *No*. It doesn't make sense to ask "Is it true that it will rain tomorrow?"—at least, not if you're expecting an answer (yes or no) that's guaranteed to be unequivocally correct.
- Supplier S6's city is unknown. *Yes*. See Appendix C for further discussion of propositions like this one.

10.1 See the answer to Exercise 4.10 in Chapter 4.

10.2 Consider the following truth table:

p	*q*	NOT *p* (= *x*)	*p* OR *q* (= *y*)	*x* AND *y*	(*x* AND *y*) IMPLIES *q*
T	T	F	T	F	T
T	F	F	T	F	T
F	T	T	T	T	T
F	F	T	F	F	T

Since the final column contains T (true) in every position, the given expression is indeed a tautology.

10.3 This is one of De Morgan's laws. It's proved in Chapter 11.

10.4 First of all, it's easy to see that we don't need both AND and OR, because

```
p AND q  ≡  NOT ( NOT ( p ) OR NOT ( q ) )
```

and

```
p OR q  ≡  NOT ( NOT ( p ) AND NOT ( q ) )
```

These equivalences are easily established by means of truth tables, as in the answer to Exercise 10.2 above. What they show is that each of AND and OR can be defined in terms of the other, together with NOT. It follows that we can freely use both AND and OR 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                  /* identity     */
c(p)  ≡  NOT ( p )          /* NOT          */
c(p)  ≡  p OR NOT ( p )     /* always TRUE  */
c(p)  ≡  p AND NOT ( p )    /* always FALSE */
```

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
c(p,q)  ≡  q
c(p,q)  ≡  NOT ( p )
c(p,q)  ≡  NOT ( q )
c(p,q)  ≡  p AND q
c(p,q)  ≡  p OR q
c(p,q)  ≡  p AND NOT ( q )
c(p,q)  ≡  p OR NOT ( q )
c(p,q)  ≡  NOT ( p ) AND q
c(p,q)  ≡  NOT ( p ) OR q
c(p,q)  ≡  NOT ( p ) AND NOT ( q )
c(p,q)  ≡  NOT ( p ) OR NOT ( q )
c(p,q)  ≡  p AND NOT ( p ) AND q AND NOT ( q )
c(p,q)  ≡  p OR NOT ( p ) OR q OR 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 (and compare them with those given in the answer to Exercise 4.10 in Chapter 4).

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 helpfully 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 AND q      ≡  ( p ↓ p ) ↓ ( q ↓ q )
p OR q       ≡  ( p ↓ q ) ↓ ( p ↓ q )
```

For example, let's take a closer look at the case of *p* AND *q* (I'll leave the other two cases to you):

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 (*p*↓*p*)↓(*q*↓*q*) is equivalent to *p* AND *q*, because its first, second, and final columns are identical to the corresponding 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, AND, and OR, the overall conclusion follows.

10.5 (a) "The sun is a star" and "the moon is a star" are both propositions, though the first is true and the second false. (b) *The sun* satisfies the predicate, *the moon* doesn't.

10.6 This exercise points up the question of ambiguity once more. If the predicate means *x* has exactly two moons, then clearly Jupiter doesn't satisfy it. But if it means *x* has at least two moons (and maybe more), then Jupiter does satisfy it. Once again, then, we see how logic forces us—or does its best to force us, at any rate—into thinking clearly and saying exactly what we mean.

10.7 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, in fact—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).

10.8 "Get names of suppliers such that there exists a shipment—a *single* shipment, that is—that links them to every part." The query probably isn't very sensible. To be specific, (a) if relvar P contains exactly one tuple and relvar SP contains *n* tuples, linking *n* distinct suppliers to that single part, then it'll return the names of those *n* suppliers; (b) if relvar P doesn't contain exactly one tuple, then it'll return an empty result.

10.9 The following SQL expressions are deliberately meant to be as close to their relational counterparts (as given in the body of the chapter) as possible. See Chapter 11 for further discussion.

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 names of 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  PX.COLOR = 'Red'
              AND    SX.SNO = SPX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

Example 3: Get names of 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 names of 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 ( SPX.QTY ) , 0 )
                  FROM   SP AS SPX
                  WHERE  SPX.PNO = PX.PNO ) AS TOTQ
FROM   P AS PX
```

Example 7: Get 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
```

10.10 The following truth table shows (how, exactly?) 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	F	F	F	F
T	F	T	F	F	F	F
T	F	F	T	T	T	T
F	T	T	F	F	F	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Note: The formal name for AND is *conjunction*, and its operands (i.e., the terms being ANDed together) are called *conjuncts*. Similarly, the formal name for OR is *disjunction*, and its

operands are called *disjuncts*. Further, since (a) AND and OR are commutative as well as associative and (b) they both possess an identity value, we can define *n*-adic versions, as follows:

- AND {*p1*,*p2*,...,*pn*} is defined to be equivalent to

  ```
  ( p1 ) AND ( p2 ) AND ... AND ( pn ) AND TRUE
  ```

  If none of the conjuncts (i.e., the *p*'s) involves any ANDs, the expression overall is said to be in *conjunctive normal form* (CNF).

- OR {*p1*,*p2*,...,*pn*} is defined to be equivalent to

  ```
  ( p1 ) OR ( p2 ) OR ... OR ( pn ) OR FALSE
  ```

  If none of the disjuncts (i.e., the *p*'s) involves any ORs, the expression overall is said to be in *disjunctive normal form* (DNF).

Note: It's clear from these definitions that if *n* = 1, then *n*-adic AND and OR both return *p1*; if *n* = 0, they return TRUE and FALSE, respectively.

10.11 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., the fact that exactly one integer is equal to zero doesn't imply 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:* (Valid!) equivalences and implications like those under discussion here (and in the next exercise) 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. See Chapter 11 for further discussion.

10.12 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.

10.13 I give solutions only in those cases—in fact, in just a few cases, from Chapters 6 and 7 only—where there's some significant point to be made regarding the solution in question. For cross reference purposes, I show the numbers of the original exercises in *italics*. *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 on no attributes:

```
{ } WHERE EXISTS ( SX )
```

The relational calculus isn't usually 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 usually considered as having a direct counterpart to **Tutorial D**'s D_UNION or I_MINUS, but there's no reason in principle why it shouldn't. *Note:* The relational calculus counterpart to regular UNION is illustrated in the answer to Exercise 10.14 below.

10.13 (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 , SCT := COUNT ( SPX WHERE SPX.PNO = PX.PNO ) }`

7.8 A relational calculus analog of the **Tutorial D** expression SP GROUP { } AS X is:

```
{ SPX , X := { } }
```

7.11 `{ SX , PNO_REL := { SPX.PNO } WHERE SPX.SNO = SX.SNO }`

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.*

10.14 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.

10.15 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 (because, as noted in Chapter 6, all of the other algebraic operators discussed in this book can be defined in terms of these five),[21] and (b) the operands to those SQL expressions can be arbitrarily complex SQL expressions in turn. So let's give it a try.

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.[22] 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 such naming. Furthermore—provided those 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 mentioned above:

---

[21] Except for TCLOSE—but TCLOSE wasn't included in the original definition of what it meant to be relationally complete, anyway. *Note:* You might be wondering about some of the other operators from Chapter 7 (e.g., EXTEND, SUMMARIZE, and GROUP and UNGROUP). In fact, Hugh Darwen and I show in our book *Databases, Types, and the Relational Model: The Third Manifesto* (see Appendix G) that these operators can indeed be defined in terms of restrict, project, etc., at least in principle.

[22] To state the matter a little more precisely: An SQL analog of the algebraic expression *R* RENAME {*A* AS *B*} is the (extremely inconvenient!) SQL expression SELECT *X*, *Y*, ..., *Z*, *A* AS *B* FROM *R* (where *X*, *Y*, ..., *Z* are all of the columns of *R* 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).

```
Algebra                  SQL

R WHERE bx               SELECT * FROM R WHERE bx

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

R1 MINUS R2              SELECT * FROM R1
                         EXCEPT CORRESPONDING
                         SELECT * FROM R2
```

Moreover, (a) *R*, *R1*, and *R2* in the SQL expressions shown above are all table expressions (in the sense in which I've been using this latter term throughout this book up to this point), and (b) if we take any of those expressions and enclose it in parentheses, what results is a table expression in turn.[23] It follows that SQL is indeed relationally complete. Or is it? Unfortunately, the answer is *no*. The reason is that there's a slight (?) glitch in the foregoing argument—SQL fails to support projection on no columns at all, because it doesn't support empty commalists in the SELECT clause. As a consequence, it doesn't support TABLE_DEE or TABLE_DUM, and therefore it isn't relationally complete after all ... but it "nearly" is.

10.16 No, it isn't—though textbooks on logic usually claim the opposite, and in practice it's "usually" achievable. Here's an example of a relational calculus expression where the predicate in the WHERE clause can't be replaced by a PNF equivalent:

```
SX WHERE EXISTS SPX ( SPX.SNO = SX.SNO ) OR SX.CITY = 'Athens'
```

The paper "A Remark on Prenex Normal Form" (see Appendix G) explains the situation in detail.

[23] I choose to overlook the fact that SQL would actually require such a table expression, when used in any of the contexts under discussion, to be accompanied by a pointless range variable definition.

Chapter 11

Using Logic to Formulate SQL Expressions

There is science, logic, reason; there is thought verified by experience.
And then there is California.[1]

—Edward Abbey:
A Voice Crying in the Wilderness (1989)

In Chapter 6, I described the process of expression transformation as it applied to expressions of the relational algebra; to be specific, 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. Restriction distributes over union, intersection, and difference

b. Projection distributes over union but not over intersection or 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 any 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. But 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 (i.e., relational calculus) formulation of, e.g., some query or constraint and map it systematically into an SQL equivalent. And while the SQL formulations so obtained can sometimes be hard to understand, we know they're correct, because of the systematic manner in which they've been obtained. Hence the subtitle of this book: *How to Write Accurate SQL Code.*

---

[1] I remark, for what it's worth, that SQL and the relational model and SQL are both essentially products of California.

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 and elsewhere 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* instead 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. 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 from *X1* by substituting an occurrence of *x2* for the occurrence of *x1* in question. Then *X1* and *X2* are logically and semantically equivalent; hence, *X1* can be rewritten as *X2*. By way of a simple example, consider the following 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 (thanks to the distributivity of AND over OR—see later) 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' )
```

And there might indeed be some small advantage to this transformation (quite apart from the fact that it saves keystrokes), because it makes it obvious that the STATUS column needs to be tested once per row instead of twice.

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 have much to say about its use there. Take a moment (if you need to) to check the truth tables and convince yourself 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 the discussions is on such expressions—i.e., on pieces of program text, in effect—rather than on logic per se. 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). You can think of a boolean expression as a concrete representation of some predicate.

- *The double negation law* (also known as *the involution 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 Chapter 10, but you probably learned about them in school. In any case, they make obvious 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 for the first law:

p	*q*	*p* AND *q*	NOT (*p* AND *q*)	NOT *p*	NOT *q*	(NOT *p*) OR (NOT *q*)
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Since the columns for NOT (*p* AND *q*) and (NOT *p*) OR (NOT *q*) are identical, the validity of the first law follows. Proof of the validity of the second law is analogous (exercise for the reader).

- *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 (as indeed I did mention in passing) 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 restriction, 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. What I didn't point out there, however, is that—as I'm sure you can see—it's really just an application of De Morgan's laws to EXISTS and FORALL expressions specifically (recall from that chapter that EXISTS and FORALL can be regarded as 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, interchanging the two sides (but it's the same thing, logically):

```
( NOT p ) OR q  ≡  NOT ( p AND NOT q )
```

And now using the implication law:

```
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 complicated.

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 unnecessary parentheses (in other words, applying some further simple transformations):

```
CONSTRAINT BTCX1 CHECK ( COLOR <> 'Red' OR CITY = 'London' )
```

Note: I've said that SQL doesn't support logical implication (IF … THEN …) as such. That's true. But it does support CASE expressions, and so this first example might alternatively be formulated in SQL as follows:

```
CONSTRAINT BTCX1 CHECK ( CASE
                            WHEN COLOR = 'Red' THEN CITY = 'London'
                            ELSE TRUE
                         END ) ;
```

In general, the logical implication IF *p* THEN *q* can be mapped into the SQL CASE expression CASE WHEN *sp* THEN *sq* ELSE TRUE END, where *sp* and *sq* are SQL analogs of *p* and *q*, respectively.[2] For simplicity, however, I'll ignore this possibility in future examples.

[2] What would happen if we omitted that ELSE TRUE?

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 tacitly 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, etc., denote variables that range over the current value of relvar (or table) P; names of the form SX, SY, etc., denote variables that range over the current value of relvar (or table) S; and so on.[3] Details of how such variables are defined—in logic, I mean, not in SQL—are unimportant for present purposes, and I won't bother to show them. In SQL, they're defined by means of AS specifications, 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 add or drop parentheses as and when I feel it's desirable to do so, 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:

---

[3] I'm being sloppy here. The phrase "range over table P" ought really to be "range over the table value that's the current value of the table variable called P" (and similarly for "range over table S," of course). Of course, SQL has no explicit notion of table values vs. table variables anyway.

```
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, NOT maps to NOT (unsurprisingly).
- 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*. Of course, mapping *bx* to *sbx* might require further (recursive) application of these rules.
- 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 SQL 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 names of parts whose weight is different from that of every part in Paris." Here's a straightforward relational calculus formulation:

```
{ PX.PNAME } WHERE FORALL PY ( IF PY.CITY = 'Paris'
                               THEN PY.WEIGHT ≠ PX.WEIGHT )
```

This expression can be read 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 read 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:[4]

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

[4] 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 here regardless of whether or not DISTINCT was specified. If so, then the products in question would seem to have a bug in this area.

Paris; the query is unanswerable. But SQL gives us an answer anyway ... To be specific, the subquery following the keyword EXISTS evaluates to an empty table for every part PX represented in P; the NOT EXISTS therefore evaluates to TRUE for every such part PX; and the expression overall therefore incorrectly returns all part names in table P.

> *Aside:* As explained in Chapter 4, this is 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 discussion of such matters, refer to the paper "Why Three- and Four-Valued Logic Don't Work" (see Appendix G). *End of aside.*

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 about the matter, "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.

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 (in fact I touched on this very point in a footnote in Chapter 7). See the further remarks on this topic at the end of the present 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 ) )
```

("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 ) )
```

Recall now from Chapter 7 that if 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). So the result given our usual sample values 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 they did in this example, is to abstract a little by introducing symbolic names for subexpressions (I did briefly mention this point in the previous chapter, but now I want to get more specific). 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 )
```

As I said in the previous chapter, 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 ) )
```

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 (producing the same result as before):

```
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 said earlier, however, we know they're correct, because of the systematic manner in which they've been derived.[5]

[5] It's worth pointing out in passing that the tactic of introducing names for subexpressions is reminiscent, somewhat, of the use of WITH in simplifying complex expressions as discussed in Chapter 6. But there's a difference: For WITH, the subexpressions in question are required to be *closed*, whereas no such requirement applies in the present context. Indeed, all we're doing in the present context is, in effect, simple text substitution, which is not what happens with WITH.

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. Let's denote the restriction P WHERE COLOR = 'Purple' by the symbol *pp*. Also, let's simplify the query at hand—"Get full supplier details for suppliers who supply all purple parts"—such that it asks for supplier numbers only, instead of full supplier details. Then it might be thought (see Chapter 7) that the query could be represented by the following algebraic expression:

```
SP { SNO , PNO } DIVIDEBY pp { PNO }
```

With our usual sample data values, however, relation *pp*, and hence the projection of *pp* on {PNO}, are both empty (because there aren't any purple parts), and the foregoing expression therefore returns the supplier numbers S1, S2, S3, and S4. As noted earlier, however, 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 currently 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 explicitly, intended to solve.

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  PX.CITY <> SX.CITY
       AND    EXISTS
            ( SELECT *
              FROM   SP AS SPX
              WHERE  SPX.SNO = SX.SNO
              AND    SPX.PNO = PX.PNO ) )
```

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 condition 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 }
```

(If *t* is "the current tuple" from relvar S, then the left comparand of the "=" comparison here is a relation containing just the CITY value from that tuple *t*, and the right comparand is a relation containing CITY values for all parts supplied by the supplier corresponding to that tuple *t*.)

EXAMPLE 7: DEALING WITH AMBIGUITY

As we saw in Chapter 10, 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.) Alternatively, the query might mean:

> Get suppliers SX such that (a) SX supplies at least two distinct parts and (b) for all pairs of distinct parts PX and PY, if SX supplies both of them, 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 pinpoint and help 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. Anyway, here's the result:

SNO	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

At this point, I'd like to remind you of another transformation law that's sometimes useful: *the contrapositive law* (I discussed this one at some length in the previous chapter). 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 good 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 getting wet," then "The streets aren't getting 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 them both.

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

There's still a little more to be said about Example 7. 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 (right?). 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? *Note:* See Example 12 for further discussion of GROUP BY and HAVING.

EXAMPLE 9: ANOTHER VARIATION

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 AS PC
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	PC
S3	Blake	30	Paris	Paris

Exercise: Is that DISTINCT necessary in this example?

EXAMPLE 10: UNIQUE QUANTIFICATION

Recall this example from Chapter 10 (a logical formulation of the constraint that there's exactly one supplier for each shipment):

```
CONSTRAINT CX6 FORALL SPX ( UNIQUE SX ( SX.SNO = SPX.SNO ) ) ;
```

Recall too 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 an obvious trap for the unwary here.) Instead, it maps to:

```
UNIQUE ( SELECT k FROM S AS SX WHERE ( sbx ) )
AND
EXISTS ( SELECT * FROM S AS SX WHERE ( sbx ) )
```

where *k* denotes some arbitrary constant value.[6] (The UNIQUE invocation says there's *at most* one, the EXISTS invocation says there's *at least* one—where by "one" I mean one row in table S for which the boolean expression *sbx* evaluates to TRUE.) So constraint CX6 might map to:

```
CREATE ASSERTION CX6 CHECK
     ( NOT EXISTS
         ( SELECT *
           FROM   SP AS SPX
           WHERE  NOT UNIQUE
                ( SELECT SX.SNO
                  FROM   S AS SX
                  WHERE  SX.SNO = SPX.SNO )
           OR     NOT EXISTS
                ( SELECT SX.SNO
                  FROM   S AS SX
                  WHERE  SX.SNO = SPX.SNO ) ) ) ;
```

Note: As in one of the examples in Chapter 10, the UNIQUE invocation here—even though it might not look like it—is in fact of the form UNIQUE (SELECT *constant* FROM ...), thanks to the boolean expression in the inner WHERE clause.[7]

Aside: Given that {SNO} is a key for S, it would in fact be possible to omit that portion of constraint CX6 as just formulated that requires there to be *at most* one matching supplier.

---

[6] Actually we could employ the same trick in mapping EXISTS—i.e., we could define EXISTS SX (*bx*) as mapping to EXISTS (SELECT *k* FROM S AS SX WHERE (*sbx*)), instead of EXISTS (SELECT * FROM S AS SX WHERE (*sbx*)). For symmetry I've done exactly this in the SQL formulation of constraint CX6 that follows.

[7] A similar remark applies to the EXISTS invocation, of course.

However, this fact doesn't change the overall message of the present section and discussion. *End of aside*.

Incidentally, I think this example illustrates very well my claim that the SQL formulations produced by the techniques I'm describing in this chapter can be hard to understand. The foregoing SQL assertion might be transcribed into stilted natural language like this:

There exists no shipment such that either there's not at most one corresponding supplier or there's not at least one corresponding supplier.[8]

Well, I don't know about you, but I think it's far from immediately obvious that this extremely tortuous sentence is logically equivalent to the following one:

Every shipment has exactly one corresponding supplier.

By the way, there's another equivalence we might appeal to here—the logic expression UNIQUE SX (*bx*) is clearly equivalent (as we saw in Chapter 10) to:

```
COUNT ( SX WHERE ( bx ) ) = 1
```

As a result we can simplify the foregoing SQL CREATE ASSERTION to:

```
CREATE ASSERTION CX6 CHECK
     ( NOT EXISTS
         ( SELECT *
           FROM   SP AS SPX
           WHERE  ( SELECT COUNT ( * )
                    FROM   S AS SX
                    WHERE  SX.SNO = SPX.SNO ) <> 1 ) ) ;
```

Here for interest is yet another SQL formulation, one that uses neither UNIQUE nor COUNT. Try to convince yourself it's correct.

```
CREATE ASSERTION CX6 CHECK
     ( NOT EXISTS
         ( SELECT *
           FROM   SP AS SPX
           WHERE  NOT EXISTS
                ( SELECT *
                  FROM   S AS SX
                  WHERE  SX.SNO = SPX.SNO
                  AND    NOT EXISTS
```

[8] Note that the "or" in this sentence is implicitly inclusive. Would it be more correct if it were exclusive? Would it make any difference?

```
( 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 *tsq*, where:

- *rx* is a row expression.
- *tsq* is a table subquery. (Subqueries of all kinds are discussed further in Chapter 12.)
- θ is any of the usual scalar comparison operators supported in SQL ("=", "<>", "<", "<=", ">", ">=") followed by one of the keywords ALL, ANY, or SOME. *Note:* As mentioned in a footnote in Chapter 7, SOME is just an alternative spelling for ANY in this context.

The semantics are as follows:

- An ALL comparison returns TRUE if and only if the corresponding comparison without the ALL returns TRUE for all of the rows in the table represented by *tsq*. If that table is empty, the ALL comparison returns TRUE.[9]
- An ANY comparison returns TRUE if and only if the corresponding comparison without the ANY returns TRUE for at least one of the rows in the table represented by *tsq*. If that table is empty, the ANY comparison returns FALSE.

Here's an example ("Get names of parts whose weight is greater than that of every blue part"):

---

[9] And that TRUE is logically correct! This behavior is certainly a little surprising, given that SQL's EVERY "set function" incorrectly returns null, not TRUE, if its argument is empty. (EVERY is, of course, the "set function" analog of ALL in the context under discussion.) The reason for the inconsistency is that—as perhaps you've guessed—SQL's ALL or ANY comparisons were defined before nulls were added to the language. (Is there a moral here?) Analogous remarks apply to ANY comparisons also, mutatis mutandis.

```
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 *tsq* 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—(a) they're error prone, and in any case (b) their effect can always be achieved by other methods. As an illustration of point (a), consider the fact that a perfectly idiomatic English language formulation of the foregoing query might well use *any* in place of *every*—"Get names of 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 (a) and (b), 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 *is not equal to any* part city"—but that's not what it means. Instead, it's logically equivalent[10] to the following ("Get names of suppliers where there's at least one part in a different city"):

```
SELECT DISTINCT SNAME
FROM   S
WHERE  EXISTS ( SELECT *
                FROM   P
                WHERE  P.CITY <> S.CITY )
```

Result:

[10] Or is it? What if supplier or part cities could be null?

SNAME
Smith
Jones
Jones
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 certainly (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 table below summarizes the possibilities in this regard. Note in particular from the table that =ANY and <>ALL are equivalent to IN and NOT IN, respectively, and so these two are important exceptions to the overall recommendation to avoid ALL and ANY comparisons in general; I mean, you can certainly use IN and NOT IN whenever you want to, and you can spell them =ANY and <>ALL, respectively, if you like. (Personally, I think IN and NOT IN are much clearer than their alternatives, but it's your choice.) By contrast, =ALL and <>ANY have no analogous equivalents; however, expressions involving those operators can always be replaced by expressions involving EXISTS instead, as already noted.

	ANY	ALL
=	IN	
<>		NOT IN
<	< MAX	< MIN
<=	<=MAX	<=MIN
>	> MIN	> MAX
>=	>=MIN	>=MAX

Caveat: Unfortunately, the transformations involving 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 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 names of 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' )
```

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.[11]

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 names of 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:

[11] 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.

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT <ANY ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.CITY = 'Paris' )
```

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 in connection with Example 8, 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 formulation:

```
{ PX.PNO , PX.CITY ,
           TPQ := SUM ( SPX.QTY WHERE SPX.PNO = PX.PNO , QTY ) }
                        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? *Answer:* Yes, it is, at least according to the standard, though it used not to be. (I did mention this point in Chapter 7, but I'll repeat it here for convenience.) Let *S* be a SELECT expression with a GROUP BY clause, and let column *C* be referenced in the SELECT clause of *S*. In earlier versions of SQL, then, *C* had to be one of the grouping columns (or be referenced inside a "set function" invocation, but let's agree to ignore that possibility for simplicity). In the current version, by contrast, it's required only that *C*—or {*C*}, rather—be functionally dependent on the grouping columns.

- Do you think the GROUP BY / HAVING formulation is easier to understand? (Debatable.)

- 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

11.1 If you haven't already done so, complete the exercises included inline in the body of the chapter.

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.

11.3 Try applying the techniques described in this chapter to some genuine SQL problems from your own work environment. *Note:* This exercise is important. The techniques described in this chapter can seem a little daunting or hard to follow at first. In order to become familiar and comfortable with them, therefore, there's really no substitute for "getting your hands dirty" and applying them for yourself.

11.4 Let relvar EMP have attributes ENO and HEIGHT and predicate *Employee ENO has height HEIGHT*. Here's a relational calculus formulation of the quota query (see Exercise 7.14) "Get the employee number for the three shortest employees":

```
{ EX.ENO } WHERE COUNT ( EY WHERE EY.HEIGHT < EX.HEIGHT ) < 3
```

And here's a fairly direct transliteration of this expression into SQL:

```
SELECT  EX.ENO
FROM    EMP AS EX
WHERE ( SELECT COUNT ( * )
        FROM   EMP AS EY
        WHERE  EY.HEIGHT < EX.HEIGHT ) < 3
```

Here by contrast are three GROUP BY / HAVING expressions:

```
SELECT EX.ENO
FROM   EMP AS EX , EMP AS EY
WHERE  EX.HEIGHT >= EY.HEIGHT
GROUP  BY EX.ENO
HAVING 3 <= COUNT ( * )

SELECT EX.ENO
FROM   EMP AS EX , EMP AS EY
WHERE  EX.HEIGHT > EY.HEIGHT
GROUP  BY EX.ENO
HAVING 3 > COUNT ( * )

SELECT EX.ENO
FROM   EMP AS EX , EMP AS EY
WHERE  EX.HEIGHT > EY.HEIGHT
OR     EX.ENO = EY.ENO
GROUP  BY EX.ENO
HAVING 3 >= COUNT ( * )
```

Do you think these expressions are easier to understand than the relational calculus expression? More to the point, do they accurately represent the desired query? Also, what happens in each case if there aren't exactly three shortest employees?

11.5 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?

11.6 In this chapter, I've discussed techniques for mapping relational calculus expressions into SQL equivalents. However, the mapping process was always carried out "by hand," as it were. Do you think it could be mechanized?

ANSWERS

11.1 First of all, you were asked several times in the body of the chapter whether it was necessary to worry about the possibility that the tables involved might include duplicate rows or nulls or both. But I categorically refuse—and so, I would like to suggest politely, should you—to waste any more time worrying about such matters. Avoid duplicates, avoid nulls, and then the transformations will all work just fine (and so will many other things, too).

That said, let me now give solutions to a couple of the more significant inline exercises:

(*From the end of the section on Example 7.*) Here's 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 does this formulation differ 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
               ( 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 ) ) )
```

(*From the end of the section on Example 12.*) You were asked to 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.

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 but the GROUP BY / HAVING formulations aren't. By contrast, the non GROUP BY / non HAVING formulations *are* symmetric.

11.2 *No answer provided.*

11.3 *No answer provided* (obviously).

11.4 First of all, the exercise asked if you think the GROUP BY / HAVING expressions are easier to understand than the relational calculus expression (or the direct SQL transliteration of that expression). Only you can answer this question, of course, but I'm pretty sure the answer for most people would have to be *no*.

Second, the exercise also asked if those GROUP BY / HAVING expressions accurately represent the desired query. *Answer:* The third one does; by contrast, the first returns all employee numbers in EMP and the second returns no employee numbers at all.

Third, the exercise also asked what happens in each case if there aren't exactly three shortest employees. I'll leave this one to you!

11.5 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).

11.6 Well, I certainly don't see why not.

Chapter 12

Miscellaneous SQL Topics

I explained that we are calling the White Paper "Open Government"
because you always dispose of the difficult bit in the title.
It does less harm there than on the statute books.
—Sir Humphrey Appleby, in *Open Government*
(first episode of the BBC TV series *Yes Minister*,
by Antony Jay and Jonathan Lynn, 1981)

This last chapter is something of a potpourri; it discusses a few SQL features and related matters that, for one reason or another, don't fit very neatly into any of the previous chapters. For purposes of reference, it also gives a simplified BNF grammar for SQL table expressions and boolean expressions.

Also, this is as good a place as any to define two terms you need to watch out for. The terms in question are *implementation defined* and *implementation dependent*, and they're both used heavily 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 doesn't even have to be documented (it might vary from release to release, or even more frequently). An example is the full effect of an ORDER BY clause, if the specifications in that clause fail to specify a total ordering. By way of example, consider the effect of the SQL expression SELECT SNO FROM S ORDER BY CITY on our usual suppliers relation. As noted in Chapter 7, this expression can return the five supplier numbers in any of the following sequences (and which particular sequence you get with any particular product or at any particular time is implementation dependent):

- S5 , S1 , S4 , S2 , S3
- S5 , S4 , S1 , S2 , S3
- S5 , S1 , S4 , S3 , S2
- S5 , S4 , S1 , S3 , S2

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. In particular, it's probably acceptable at the outermost level of a SELECT – FROM – WHERE expression in what the standard calls "direct" (i.e., interactive) SQL, or in other words in an interactive query. 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 specify the pertinent columns explicitly, by name. A similar remark applies to view definitions also. (On the other hand, if you adopt the strategy suggested under the discussion of column naming in Chapter 3 of always accessing the database via views—the "operate via views" strategy—then it might be safe to use "SELECT *" wherever you like, other than in 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 makes use of 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
```

DOT QUALIFICATION

References to column names in SQL can usually be dot qualified by the name of the applicable range variable (see the next section). As you know, however, SQL does allow 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, especially if the overall table expression involves any nested subqueries or explicit joins.[1] 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 such 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 within a base table definition (CREATE TABLE, also ALTER TABLE)
- A key or foreign key specification
- The column name commalist, if specified (but it shouldn't be—see Chapter 8), in CREATE VIEW
- The column name commalist, if specified (but it usually shouldn't be—see the next section), following the definition of a range variable

[1] A detailed though not necessarily exhaustive discussion of this issue can be found on pages 141-144 of the book *A Guide to the SQL Standard* (4th edition), by Hugh Darwen and myself (see Appendix G).

- The column name commalist in JOIN ... USING
- The column name commalist, if specified (and it should be—see Chapter 5), on INSERT
- The left side of a SET assignment on UPDATE

It might help to note that most of the contexts listed above are ones in which no range variable, as such, is available for dot qualification use anyway. The point is, however, that an unsuspecting user might expect to be able to use table names as qualifiers in these contexts,[2] on the grounds—I suppose—that SQL often uses table names as if they were range variable names anyway, as explained in the section immediately following.

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 tuples in some relation (or the set of rows in some table, in SQL terms). In SQL, such variables are defined by means of AS specifications in the context of either a FROM clause or an explicit JOIN (see the BNF grammar, later). Here's a simple example of the FROM case:

```
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.* However, 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*.[3]

---

[2] As in, for example, UPDATE S SET S.STATUS = S.STATUS + 1 (the second S.STATUS here is legal but the first isn't).

[3] Actually, the current version of the standard—viz., SQL:2011—does sometimes use the the term *range variable*, but it doesn't do so either exclusively or systematically, and *correlation name* still seems to be the preferred term.

Incidentally, it's worth noting that 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 (note the text in ***bold italics***):

```
SELECT S.SNO
FROM   S AS S
WHERE  S.STATUS > 15
```

In this latter formulation, the "S" dot qualifiers and the "S" in the 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.[4]

Now, the BNF grammar defined later in this chapter refers to the items in the commalist in a FROM clause—i.e., the items following the keyword FROM itself—as *table specifications*.[5] The expressions denoting the table operands in an explicit JOIN are also table specifications. So let *ts* be such a table specification. Then, if the portion of *ts* consisting of a table expression as such in fact consists of a table subquery (see the next section), then *ts* must include an associated AS specification—even if the range variable introduced by that AS specification is never explicitly mentioned anywhere else in the overall expression. Here's a JOIN example:

```
( SELECT SNO , CITY FROM S ) AS temp1
  NATURAL JOIN
( SELECT PNO , CITY FROM P ) AS temp2
```

Here's another example (this one is repeated from Chapter 7):

```
SELECT PNO , GMWT
FROM ( SELECT PNO , WEIGHT * 454 AS GMWT
       FROM   P ) AS temp
WHERE  GMWT > 7000.0
```

For interest, here's this latter example repeated with all implicit qualifiers made explicit:

---

[4] 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."

[5] The standard calls them table *references*, but that term is hardly 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 the value of the variable in question. But an SQL "table reference" isn't a special case of a table expression—not in the sense in which the 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 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 that defines column names for the table the range variable ranges over, as in the following example (see the last two lines):

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

The introduced column names here (SNO, SNAME, STATUS, SCITY, PNO, PNAME, COLOR, WEIGHT, and PCITY) 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).[6] However, it shouldn't be necessary to introduce column names in this way 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.[7] Be aware, however, that SQL's name scoping rules for such variables can be quite hard to understand (but this is true regardless of whether the variables in question are explicit or implicit).

Caveat: As noted in Chapter 10, many SQL texts refer to range variable names, or correlation names, as *aliases*—sometimes, more specifically, *table* aliases—and describe them as if they were just alternative names for the tables they range over. Here's a typical quote: "You couldn't write this query without using aliases because the table names are identical." 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 it's strongly deprecated on that account. Be on your guard against this sloppy manner of speaking (and/or thinking).

---

[6] As the example suggests, the column name commalist in a range variable definition is required, somewhat annoyingly, to be exhaustive—there's no way to rename just some of the columns concerned and not others. Also, note the need here to be fully cognizant of SQL's rules regarding left to right column ordering in the result of the explicit JOIN!

[7] I'll omit them from most of my own examples in the remainder of this chapter, however, because (a) using explicit range variables might distract from the main point I'm trying to make with those examples and (b) those examples are all fairly simple, anyway.

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. Note, however, that (as mentioned in Chapters 1 and 6) the expression *tx* can't be an explicit JOIN expression. Thus, for example,

```
( A NATURAL JOIN B )
```

isn't a legal subquery.[8] By contrast, the following expression *is* a legal subquery:

```
( SELECT * FROM A NATURAL JOIN B )
```

Subqueries fall into three broad categories, though the syntax is the same in every case. The details, partly repeated from earlier chapters, are as follows:

- 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 logically required. 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*). *Note:* If *rsq* doesn't denote a table with just one row, then (a) if it denotes a table with more than one row, an error is raised; (b) if it denotes a table with no rows at all, then that table is treated as if it contained just one row, where the row in question contains a null in every column position.

- A *scalar subquery* is a subquery appearing in a position where a scalar expression is logically required. 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*). *Note:* If *ssq* doesn't denote a table with just one row and just one column, then (a) if it denotes a table with more than one column, an error is raised (probably at compile time); (b) if it denotes a table with one column but more than one row, an error is raised (probably at run time); (c) if it denotes a table with one column and no rows at all, then that table is treated as if it contained just one row, where the row in question contains a single null.

The following examples involve, in order, a table subquery, a row subquery, and a scalar subquery:

---

[8] It was legal in SQL:1992 but became illegal in SQL:2003.

```
SELECT SNO
FROM   S
WHERE  CITY IN
     ( SELECT CITY              /* table subquery */
       FROM   P
       WHERE  COLOR = 'Red' )

UPDATE S
SET  ( STATUS , CITY ) =
     ( SELECT STATUS , CITY     /* row subquery */
       FROM   S
       WHERE  SNO = 'S1' )
WHERE  CITY = 'Paris' ;

SELECT SNO
FROM   S
WHERE  CITY =
     ( SELECT CITY              /* scalar subquery */
       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 what's called a "correlated reference" to some "outer" table. In the following example, the parenthesized expression following the keyword IN is a correlated subquery—a correlated table subquery, in fact—because it includes a correlated reference to the outer table S (the query is "Get names of suppliers who supply part P1," and the correlated reference, viz., S.SNO, appears in the very last line):

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  'P1' IN
      ( SELECT PNO              /* correlated table subquery */
        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              /* noncorrelated table subquery */
       FROM   SP
       WHERE  PNO = 'P1' )
```

Here's another example showing the use of a correlated subquery (a correlated scalar subquery, in the SELECT clause this time), repeated from the section "Summarization" in Chapter 7:

```
SELECT S.SNO , ( SELECT COUNT ( PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

The query is "For each supplier, get the supplier number and a count of the number of parts supplied by that supplier." Given the sample values in Fig. 1.1 in Chapter 1, the result looks like this:

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

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 specifically and (b) includes a reference to an "outer" table that's defined by a table specification appearing earlier within that same FROM clause. For example, here's another possible formulation of the query just illustrated ("For each supplier, get the supplier number and the number of parts supplied by that supplier"):

```
SELECT S.SNO , temp.PCT
FROM   S , LATERAL ( SELECT COUNT ( 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 "lateral" subquery yields exactly one value—namely, the applicable count—for each SNO value in table S).

Now, there's something going on here that you might be forgiven for finding a bit confusing. The items in a FROM clause are table specifications, and so they denote tables. In the example, though, the particular table specification that begins with the keyword LATERAL—more precisely, what remains of that table specification if the keyword LATERAL is removed—looks more like what might be called a *scalar* specification, or more precisely a scalar subquery; certainly it could used as such, should the context demand such an interpretation (e.g., in a SELECT clause, as indeed we saw above in the previous formulation of this query). In fact, however, it's a table subquery. The table it denotes, for a given value of S.SNO, is called

temp; that table has just one column, called PCT, and just one row, and hence in fact contains a single scalar value. Then the expression *temp*.PCT in the SELECT clause causes that scalar value to become the contribution of table *temp* to the applicable result row (just as the expression S.SNO in that same SELECT clause causes the applicable SNO value to become the contribution of table S to that result row).

Following on from the foregoing rather complicated explanation, I feel bound to add that it's not exactly clear why "lateral" subqueries are needed anyway. Indeed, as we already know, the foregoing example can easily be reformulated in such a way as to avoid the apparent need for any such thing. Here again is that reformulation:

```
SELECT S.SNO , ( SELECT COUNT ( PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

Briefly, what's happened here is that 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 apparently no longer needed. 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 (this point was discussed in more detail in an aside in the section "Summarization" in Chapter 7).

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 the SQL context, in other words, it's either 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. *Note:* The term is sometimes used (though not in this book!) to refer to an update request also. It's also used on occasion to refer to the natural language version of some retrieval or update request.

"POSSIBLY NONDETERMINISTIC" EXPRESSIONS

As we saw in Chapter 2, an SQL table expression is "possibly nondeterministic" if it might give different results on different evaluations, even if the database hasn't changed in the interim. Here's the standard's own definition:

> A <query expression> or <query specification> is *possibly nondeterministic* if an implementation might, at two different times where the state of the SQL-data is the same, produce results that differ by more than the order of the rows due to General Rules that specify implementation dependent behavior.

Actually this definition is a trifle odd, inasmuch as tables—which is what <query expressions>s and <query specifications>s are supposed to produce—aren't supposed to have an ordering to their rows anyway. But let's overlook this detail; the important point is that, as noted in Chapter 2, "possibly nondeterministic" expressions aren't allowed in integrity constraints,[9] a state of affairs that could have serious practical implications if true.

The standard's rules for labeling a given table expression "possibly nondeterministic" are quite complex, and full details are beyond the scope of the present discussion. However, a table expression *tx* is certainly considered to be "possibly nondeterministic" if any of the following is true:[10]

- *tx* is a union, 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:

  a. The SELECT item commalist is preceded by the keyword DISTINCT.

  b. *C* involves a MAX or MIN invocation.

  c. *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) a reference to a grouping column of type character string or (b) a MAX or MIN invocation in which the argument is of type character string.

- *tx* is a JOIN expression and either or both of the operand expressions is possibly nondeterministic.

Note, however, that these rules are certainly stronger than they need be. For example, suppose that (a) NO PAD applies to the pertinent collation and (b) no two characters from the pertinent character set are "distinct, considered equal" according to that collation. Then, e.g., SELECT MAX(*C*) FROM *T*, where column *C* of table *T* is of the character string type in question, is surely well defined.

---

[9] Nor in view definitions, if WITH CHECK OPTION is specified.

[10] What follows represents my own understanding and paraphrasing of the pertinent text from SQL:1992 (except that I've taken into account certain minor revisions made in subsequent versions of the standard). More important, I follow SQL:1992 here in talking about character string types only. The rules have since been extended to include as possibly nondeterministic (a) expressions involving data of certain user defined types and (b) expressions involving invocations of certain user defined operators (*routines*, to use the standard's term). Further details are beyond the scope of this book.

EMPTY SETS

The empty set is the set containing no elements. This concept is both 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 contain an empty row expression commalist.
- The SQL "set functions" all return null if their argument is empty (except for COUNT(*) and COUNT, which correctly return zero in such a situation).
- 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 for UNION CORRESPONDING, INTERSECT CORRESPONDING, and EXCEPT CORRESPONDING can't be empty (though it can be for NATURAL JOIN).
- A row can't have an empty set of components.

A SIMPLIFIED BNF GRAMMAR

For purposes of reference, it seems appropriate to close this chapter, and the main part of this book, with a simplified BNF grammar for the standard dialect of SQL—not for the whole of the language, of course, but at least for SQL table expressions and boolean expressions.[11] The grammar is deliberately conservative, in that it fails to define as valid certain expressions that are so, according to the SQL standard. (However, I don't believe it defines as valid any expressions that aren't so according to that standard.) To be more specific, constructs that I've previously advised you not to use—including in particular everything to do with nulls and 3VL—are deliberately omitted; so too are certain somewhat esoteric features (e.g., recursive queries). Also, for reasons explained in Chapter 1, almost all of the syntactic categories in what follows have names that differ somewhat from their counterparts in the standard. The following simplifying abbreviations are used:

```
exp     for     expression
spec    for     specification
```

All syntactic categories of the form <... *name*> are assumed to be <*identifier*>s and are defined no further here. The category <*scalar exp*> is also left undefined, though it might help to recall in particular that:

- A scalar subquery is a legal scalar expression.
- Most "row expressions" that occur in practice are actually scalar expressions.
- Boolean expressions are scalar expressions too.

Table Expressions

As you can see, the grammar in this subsection begins with a production for <*with exp*>, a construct not mentioned (at least, not as such) in the body of the book. I introduce this syntactic category in order to capture (among other things) the fact that join expressions can't appear without being nested inside some other table expression—but it does mean that the construct referred to throughout earlier parts of the book as a table expression doesn't directly correspond to anything defined in the grammar! (I mean, there's no production for a syntactic category called <*table exp*>.) I apologize if you find this state of affairs confusing, but it's the kind of thing that always happens when you try to define a grammar for a language that violates orthogonality.[12]

---

[11] Appendix D gives a BNF grammar for relational expressions (also relational assignment) in **Tutorial D**.

[12] It might help to point out that, loosely speaking, explicit joins can't appear at the outermost level of nesting, while WITH specifications can't appear anywhere else.

Note that the productions defined in this subsection agree with the SQL standard in giving INTERSECT higher precedence than UNION; thus, for example, the table expression *t1* INTERSECT *t2* UNION *t3* is understood as (*t1* INTERSECT *t2*) UNION *t3* and not as *t1* INTERSECT (*t2* UNION *t3*). But it's probably better always to specify parentheses explicitly in such expressions, anyway.

```
<with exp>
    ::=   [ <with spec> ] <nonjoin exp>

<with spec>
    ::=   WITH <name intro commalist>

<name intro>
    ::=   <table name> AS <table subquery>

<table subquery>
    ::=   <subquery>

<subquery>
    ::=   ( <nonjoin exp> )

<nonjoin exp>
    ::=   <nonjoin term>
        | <nonjoin exp> UNION [ DISTINCT ]
                              [ CORRESPONDING ] <nonjoin term>
        | <nonjoin exp> EXCEPT [ DISTINCT ]
                               [ CORRESPONDING ] <nonjoin term>

<nonjoin term>
    ::=   <nonjoin primary>
        | <nonjoin term> INTERSECT [ DISTINCT ]
                         [ CORRESPONDING ] <nonjoin primary>

<nonjoin primary>
    ::=   TABLE <table name>
        | <table selector>
        | <select exp>
        | ( <nonjoin exp> )

<table selector>
    ::=   VALUES <row exp commalist>

<row exp>
    ::=   <scalar exp>
        | <row selector>
        | <row subquery>

<row selector>
    ::=   [ ROW ] ( <scalar exp commalist> )

<row subquery>
    ::=   <subquery>
```

```
<select exp>
    ::=   SELECT [ DISTINCT ] [ * | <select item commalist> ]
             FROM <table spec commalist>
              [ WHERE <boolean exp> ]
                [ GROUP BY <column name commalist> ]
                  [ HAVING <boolean exp> ]
```

The <*column name commalist*> in the GROUP BY clause can optionally be enclosed in parentheses. If it is, then (and only then)—unlike all other commalists mentioned in this grammar—it can also be empty.

```
<select item>
    ::=   <scalar exp> [ AS <column name> ]
        | <range variable name>.*

<table spec>
    ::=   <table name> [ AS <range variable name> ]
        | [ LATERAL ] <table subquery> AS <range variable name>
        | <join exp>
        | ( <join exp> )

<join exp>
    ::=   <table spec> CROSS JOIN <table spec>
        | <table spec> NATURAL JOIN <table spec>
        | <table spec> JOIN <table spec> ON <boolean exp>
        | <table spec> JOIN <table spec>
                       USING ( <column name commalist> )
```

Boolean Expressions

Note that the productions defined in this subsection agree with the SQL standard in giving AND higher precedence than OR; thus, for example, the boolean expression *c1* AND *c2* OR *c3* is understood as (*c1* AND *c2*) OR *c3* and not as *c1* AND (*c2* OR *c3*). But it's probably better always to specify parentheses explicitly in such expressions, anyway.

```
<boolean exp>
    ::=   <boolean term>
        | <boolean exp> OR <boolean term>

<boolean term>
    ::=   <boolean factor>
        | <boolean term> AND <boolean factor>

<boolean factor>
    ::=   [ NOT ] <boolean primary>

<boolean primary>
    ::=   <boolean literal>
        | <boolean variable name>
        | <boolean column name>
        | <condition>
        | ( <boolean exp> )
```

```
<boolean literal>
    ::=   TRUE | FALSE

<condition>
    ::=   <simple comparison exp>
        | <between exp>
        | <like exp>
        | <in exp>
        | <match exp>
        | <all or any exp>
        | <exists exp>
        | <unique exp>

<simple comparison exp>
    ::=   <row exp> <simple comp op> <row exp>

<simple comp op>
    ::=   = | < | <= | > | >= | <>

<between exp>
    ::=   <row exp> [ NOT ] BETWEEN <row exp> AND <row exp>

<like exp>
    ::=   <scalar exp> [ NOT ] LIKE <scalar exp> [ ESCAPE <scalar exp> ]
```

The <*scalar exp*>s must denote character strings. For ESCAPE, that string must be of length one.

```
<in exp>
    ::=   <row exp> [ NOT ] IN <table subquery>
        | <row exp> [ NOT ] IN ( <row exp commalist> )

<match exp>
    ::=   <row exp> MATCH [ UNIQUE ] <table subquery>

<all or any exp>
    ::=   <row exp> <scalar comp op> <all or any> <table subquery>

<all or any>
    ::=   ALL | ANY | SOME

<exists exp>
    ::=   EXISTS <table subquery>

<unique exp>
    ::=   UNIQUE <table subquery>
```

EXERCISES

12.1 According to the BNF grammar given in the body of the chapter, which of the following are legal as “stand alone” table expressions—in other words, as “with expressions,” which is to

say, table expressions not nested inside another table expression—and which not, syntactically speaking? (*A* and *B* are table names, and you can assume the tables they denote satisfy the requirements for the operator in question in each case.)

a. `A NATURAL JOIN B`

b. `A INTERSECT B`

c. `SELECT * FROM A NATURAL JOIN B`

d. `SELECT * FROM A INTERSECT B`

e. `SELECT * FROM ( A NATURAL JOIN B )`

f. `SELECT * FROM ( A INTERSECT B )`

g. `SELECT * FROM ( SELECT * FROM A INTERSECT SELECT * FROM B )`

h. `SELECT * FROM ( A NATURAL JOIN B ) AS C`

i. `SELECT * FROM ( A INTERSECT B ) AS C`

j. `TABLE A NATURAL JOIN TABLE B`

k. `TABLE A INTERSECT TABLE B`

l. `SELECT * FROM A INTERSECT SELECT * FROM B`

m. `( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )`

n. `( SELECT * FROM A ) AS AA INTERSECT ( SELECT * FROM B ) AS BB`

What do you conclude from this exercise? Perhaps I should remind you that, relationally speaking, intersection is a special case of natural join.

12.2 Take another look at the expressions in Exercise 12.1. In which of those expressions would it be syntactically legal to replace *A* or *B* or both by "table literals" (i.e., appropriate VALUES expressions)?

12.3 Let X and Y both be of the same character string type and be subject to the same collation; let PAD SPACE apply to that collation (not recommended, of course); and let X and Y have the values '42' and '42 ', respectively (note the trailing space in the second of these). Then we know from Chapter 2 that although X and Y are clearly distinct, the expression X = Y gives TRUE. But what about the expression X LIKE Y?

12.4 Given our usual sample values, what do the following expressions return?

a.
```
SELECT DISTINCT STATUS
FROM   S
WHERE  STATUS BETWEEN 10 AND 30
```

```
b.  SELECT DISTINCT CITY
    FROM   S
    WHERE  CITY LIKE 'L%'

c.  SELECT DISTINCT CITY
    FROM   S
    WHERE  CITY BETWEEN 'Paris' AND 'Athens'
```

12.5 The following is intended to be an SQL expression of type BOOLEAN. Is it legal?

```
( SELECT CITY FROM S WHERE STATUS < 20 )
  =
( SELECT CITY FROM P WHERE WEIGHT = 14.0 )
```

12.6 In the body of the chapter I recommended circumspection in the use of asterisk notation in the SELECT clause. For brevity, however, I didn't always follow my own advice in this respect in earlier chapters. Take a look through those chapters and see if you think any of my uses of the asterisk notation were unsafe.

12.7 Consider any SQL product available to you. Does that product support (a) the UNIQUE operator, (b) explicit tables, (c) lateral subqueries, (d) possibly nondeterministic expressions?

12.8 With regard to possibly nondeterministic expressions, recall that SQL prohibits the use of such expressions in integrity constraints. Take another look at the examples in Chapter 8 (and/or the answers to those exercises). Do any of those examples or answers involve any possibly nondeterministic expressions? If so, what can be done about it?

12.9 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 been a long way from being exhaustive). But every product on the market departs from the standard in various ways, either by omitting some standard features or by introducing proprietary features of its own or (almost certainly in practice) both. Again, consider any SQL product available to you. Identify as many departures from the standard in that product as you can.

ANSWERS

12.1 For convenience I repeat the original expressions (or would-be expressions) below:

a. *A* NATURAL JOIN *B* : Illegal

b. *A* INTERSECT *B* : Illegal

c. SELECT * FROM *A* NATURAL JOIN *B* : Legal

d. `SELECT * FROM A INTERSECT B` : Illegal

e. `SELECT * FROM ( A NATURAL JOIN B )` : Legal

f. `SELECT * FROM ( A INTERSECT B )` : Illegal

g. `SELECT * FROM ( SELECT * FROM A INTERSECT SELECT * FROM B )` : Illegal

h. `SELECT * FROM ( A NATURAL JOIN B ) AS C` : Illegal

i. `SELECT * FROM ( A INTERSECT B ) AS C` : Illegal

j. `TABLE A NATURAL JOIN TABLE B` : Illegal

k. `TABLE A INTERSECT TABLE B` : Legal

l. `SELECT * FROM A INTERSECT SELECT * FROM B` : Legal

m. `( SELECT * FROM A ) INTERSECT ( SELECT * FROM B )` : Legal

n. `( SELECT * FROM A ) AS AA INTERSECT ( SELECT * FROM B ) AS BB` : Illegal

You were also asked what you conclude from this exercise. One thing I conclude 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 NATURAL JOIN counterparts. I remark also that if we replace INTERSECT by NATURAL JOIN in the last two expressions, then the legal one becomes illegal and vice versa! That's because, believe it or not, the expressions

```
( SELECT * FROM A )
```

and

```
( SELECT * FROM B )
```

are considered to be subqueries in the context of NATURAL JOIN but not that of INTERSECT. (In other words, a subquery is a SELECT expression enclosed in parentheses, loosely speaking, but a SELECT expression enclosed in parentheses isn't necessarily a subquery.)

12.2 The effects are as follows: Expression b. was previously illegal but becomes legal; expressions c., e., k., l., and m. were previously legal but become illegal; and the others were all illegal anyway and remain so. What do you conclude from *this* exercise?

12.3 It gives FALSE. Note, therefore (to spell the point out), it's possible in SQL for two values to be "equal" and yet not "like" each other! (Lewis Carroll, where are you?)

12.4 Expression a. gives:

STATUS
10
20
30

(The point here is that BETWEEN is inclusive, not exclusive, and so 10 and 30 are both included in the result. Does this state of affairs accord with your own intuitive understanding of the meaning of *between?*) Expression b. gives:

CITY
London

And expression c. gives:

CITY

London *isn't* included in the result. The reason is that the expression

```
y BETWEEN x AND z
```

is shorthand for

```
x <= y AND y <= z
```

The problem here is that the natural language expression "*y* is between *x* and *z*" is symmetric in *x* and *z* (i.e., switching *x* and *z* has no effect on the meaning), while the same is not true for the SQL expression "*y* BETWEEN *x* AND *z*." In a nutshell, BETWEEN in SQL doesn't mean the same as *between* in natural language.

12.5 First of all, observe that both comparand expressions are subqueries, and they therefore evaluate to tables. Now, those tables both have exactly one column, a fact that can be determined at compile time. What's more, given our usual sample values, they also both have exactly one row; the subqueries are therefore scalar subqueries, and the overall comparison is thus legal (a double coercion occurs on both sides, and the net effect is that two scalar values

are compared). But suppose the WHERE clause in the second subquery had specified 12.0 instead of 14.0. Given our usual sample values, the comparison overall would then no longer be legal (it would fail at run time), because the second subquery would now be a table subquery instead of a scalar one.

12.6 *No answer provided.*

12.7 *No answer provided.*

12.8 *No answer provided.*

12.9 *No answer provided.*

Appendix A

The Relational Model

When we try to pick out anything by itself,
we find it hitched to everything else in the universe.

—John Muir:
My First Summer in the Sierra (1911)

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 very strong claim![1] My argument goes like this:

- First of all, we saw in Chapter 5 that a database, despite the name, isn't really just a collection of data; rather, it's a collection of "true facts," or (rather more respectably, since "facts" are supposed to be true by definition) *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, let's say, "Joe" is a value of type NAME and "50K" is a value of type MONEY).

- But we don't want to record just any old propositions; rather, we want to record all of those propositions that happen to be true instantiations of certain *predicates*. In the case of "Joe's salary is 50K," for example, the pertinent predicate is "*N*'s salary is *M*," where *N* is a value of type NAME and *M* is a value of type MONEY.

- In other words, we want to record the *extension* of the predicate "*N*'s salary is *M*," which we can do in the form of a set of ordered pairs.

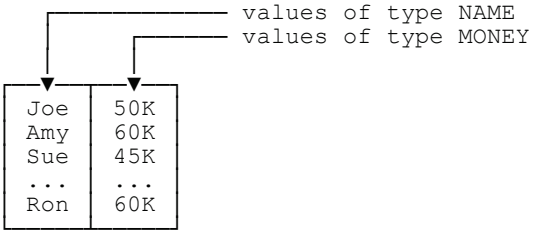
---

[1] Of course, one immediate objection to that claim is that there are clearly many nonrelational databases in existence already: hierarchic databases, CODASYL databases, and so on. True enough—but note carefully that those older databases were never meant to be application neutral and fully general purpose; rather, they were typically built to serve some specific application. As a consequence, they don't *and can't* provide all of the functionality that a truly general purpose database is supposed to provide—ad hoc query, view support, full data independence, flexible security and integrity controls, and so forth. In fact, I would go further; I would argue that, to the extent they fail to be fully relational, even modern SQL databases are subject to some of these same criticisms. In other words, I regard nonrelational databases in general as little more than *application specific data stores*, and I'm very tempted to say they shouldn't be called databases, as such, at all.

- But a set of ordered pairs is, precisely, a binary relation, in the mathematical sense of that term. Here's the definition:

  Definition: A (mathematical) 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:

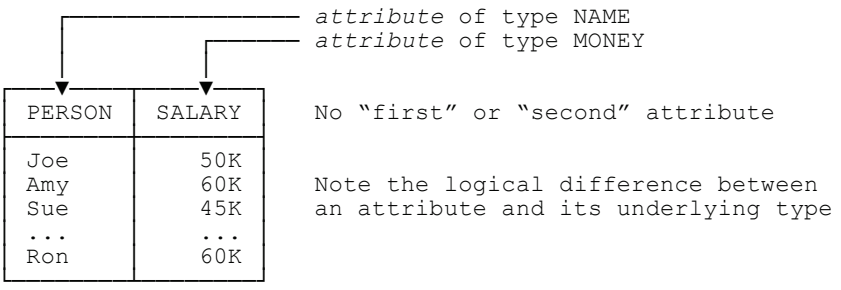


  So we can regard this picture as depicting a subset of the cartesian product of the set of all names ("type NAME") and the set of all money values ("type MONEY"), in that order. *Note:* As an aside, I remark that this particular example is not just a relation but a *function*, because each person has just one salary. A function is a special case of a binary relation.

Given the argument so far, then, we can see we're talking about some fairly humble (but very solid) beginnings. However, in 1969-1970, Codd realized that:

- We need to deal with *n-adic,* not just dyadic, predicates and propositions (e.g., the proposition "Joe has salary 50K, works in department D4, and was hired in year 1993," which is an instantiation of the predicate "*N* has salary *M*, works in department *D*, and was hired in year *Y*"). 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 by the concept of *attributes* (identified by name), and let's redefine the relation concept accordingly. The example now looks like this:



From this point forward, then, you can take the term *relation* to mean a relation in this revised and extended 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 begin to see now 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 as the relational model. Indeed, I seriously question whether they deserve to be called models at all.[2] The hierarchic and network models in particular never really existed in the first place!—as abstract models, I mean, preceding any implementations. Instead, they were invented *after the fact*; that is, hierarchic and network products were built first, and the corresponding models were defined afterward, 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

[2] Which is why I set them all enclosed in quotation marks. I'll drop those 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.

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.[3] 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 have claimed there isn't a unique relational model, either. I'll deal with that argument in a few moments.)

> *Aside:* The following quote from "The Object Oriented Database System Manifesto," by Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik (Proc. 1st International Conference on Deductive and Object Oriented Databases, Kyoto, Japan, 1989) lends weight to my suggestion that in the case of the object oriented model, at least, implementations came first and the model itself was—or indeed, the quote rather strongly suggests, should be (?)—defined afterward:
>
> With respect to the specification of the system, we are taking a Darwinian approach: We hope that, out of the set of experimental prototypes being built, a fit model will emerge. We also hope that viable implementation technology for that model will evolve simultaneously.
>
> What the authors seem to be saying here is that the code should be written first, and then that a model might possibly be developed later by abstracting from that code. If I'm right on this—I mean, if this interpretation is correct—then I have to say I find the position an extraordinary one; I mean, surely it's better to know what we're trying to do before we do it? *End of aside.*

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, those 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).[4] As a consequence, they're harder to use and understand, and they give implementers far less freedom—far less than the relational model does, I mean—to adopt inventive or creative approaches to questions of implementation.

So what of those claims to the effect that there are several relational models, too? One example of such a claim can be found in the book *Joe Celko's Data and Databases: Concepts in Practice* (Morgan Kaufmann, 1999), where the author, Joe Celko, says this:

---

[3] My own opinion, for what it's worth, is that the semistructured model and the object model are just the old hierarchic model warmed over and the old network model warmed over, respectively.

[4] Actually I think these remarks are rather charitable; in my opinion, those other models are really little more than slightly abstract, but otherwise ad hoc, *storage structures* that have been elevated above their station and will not stand the test of time.

> 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 says are six "different relational models."[5]

Now, I wrote an immediate response to these claims when I first encountered them. Here's a lightly 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.)

I'll have a little more to say about those noneuclidean geometries in the next section. Meanwhile, 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*, 2nd 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), and it's discussed in a little more detail in Appendix G of the present book. 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

---

[5] Here for the record is, verbatim, the way those "six models" are labeled in Celko's book: 1. Chris Date = No Duplicates, No NULLs. 2. E. F. Codd, RM Version I. 3. E. F. Codd, RM Version II. 4. SQL-92 = Duplicates, One NULL. 5. Duplicates, One NULL, Non-1NF Tables. 6. Rick Snodgrass = Temporal SQL.

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 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 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 explicit relational comparisons—but the axioms (which are basically those of classical set theory and 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 probably 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. What's more—as with mathematics in general—those new theorems and results can be proved and discovered by anyone who's competent to do so.[6] The relational model began as the brainchild of one man, but now belongs to the world.

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.

---

[6] "I see relational theory as simply a body of theory to which many people are contributing in different ways" (E. F. Codd, in an interview in *Data Base Newsletter 10*, No. 2, March 1982).

- 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 useful 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).

THE SIGNIFICANCE OF THEORY

Note: The bulk of this section consists of an abbreviated and slightly revised version of material from an interview I did in 2005 (published in my book Date on Database: Writings 2000-2006, *Apress, 2006).*

The relational model, whatever else it might be, is certainly a theory, and so I'd like to say a few words about the significance of theory in general before getting into details of the relational model in particular. As I said in the preface to this book, it's an article of faith with me that *theory is practical*. The purpose of relational theory in particular 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. Thus, I believe that, in the relational context specifically, departures from the underlying theory are A Big Mistake.

Unfortunately, however, the term "theory" has two quite different meanings. In common parlance, it's almost pejorative—"oh, that's just your theory." Indeed, in such contexts it's effectively just a synonym for *opinion* (and the adverb *merely*—it's *merely* your opinion—is often implied, too). But to a scientist, the term has a very different meaning. To a scientist, a theory is a set of ideas or principles that explain some set of observable phenomena, such as the motion of the planets. Of course, when I say I it explains something, I mean it does so coherently: It fits the facts, as it were. Moreover (and very importantly), it doesn't just explain something, it also makes predictions—predictions that can be tested and (at least in principle) can be shown to be false. And if any of those predictions do indeed turn out to be false, then we

move on: Either we modify the existing theory, or we adopt a new one. That's the scientific method:

1. We observe certain phenomena, empirically.
2. We construct a theory or hypothesis to explain those phenomena.
3. We use that theory to make predictions.
4. We test the accuracy of those predictions.
5. Based on the results of those tests, we refine our theory (or reject it, in extreme cases).
6. And we iterate.

That's how the Copernican system replaced epicycles; how Einstein's cosmology replaced Newton's; how general relativity replaced special relativity; and so on. Incidentally, Carl Sagan has a nice observation in this regard:

> In science it often happens that scientists say, "You know, that's a really good argument, my position is mistaken," and then they actually change their minds, and you never hear that old view from them again. They really do it. It doesn't happen as often as it should, because scientists are human and change is sometimes painful. But it happens every day. I cannot recall the last time something like that happened in politics or religion.

Anyway, I now claim the relational model is indeed a theory in the scientific sense. More specifically, I claim it's a mathematical theory. Now, mathematical theories are a little special, in a way. First of all, the observed phenomena they're supposed to explain tend to be rather abstract—not nearly as concrete as something like the motion of the planets, for example. Second, the predictions they make are essentially the theorems that can be proved within the theory; thus, those "predictions" can be falsified only if there's something wrong with the premises, or axioms, on which the theorems are based. But even this does happen from time to time! For example, in euclidean geometry, you can prove that every triangle has angles that sum to 180 degrees. So if we ever found a triangle that didn't have this property, we would have to conclude that the premises—the axioms of euclidean geometry—must be wrong. And in a sense exactly that happened: Triangles on the surface of a sphere (for example, on the surface of the Earth) turned out to have angles that sum to more than 180 degrees. And the problem turned out to be the euclidean axiom regarding parallel lines. Riemann replaced that axiom by a different one and thereby defined a different (but equally valid) kind of geometry.

In the same kind of way, the theory that's the relational model *might* be falsified in some way—but I think it's pretty unlikely, because (as I said in the previous section) the premises on

which the relational model is based are essentially those of set theory and predicate logic, and those premises have stood up pretty well for a very long time.

So, to get to the real point of this section: Given that the relational model is a scientific theory, the question is whether that theory is really important. Of course, my own answer to this question is *yes*. In fact, I'd like to turn the question on its head. First of all, database management is a field in which some solid theory certainly does exist. Furthermore, we know the value of that theory; we know the benefits that accrue if we follow that theory. We also know there are costs associated with not following that theory (we might not know exactly what those costs are—I mean, it might be hard to quantify them—but we do know there are going to be costs).

If you're traveling on an airplane, you'd like to be sure it's been constructed in accordance with the principles of physics and aerodynamics. If you live or work in a high rise building, you'd like to be sure it's been constructed in accordance with sound engineering and architectural principles. In the same kind of way, if you're using a DBMS, wouldn't you like to be sure it's been constructed in accordance with solid database principles? If it hasn't, you know things will go wrong. And while it might be hard to say exactly *what* will go wrong, and it might be hard to say whether things will go wrong in a major or minor way, you *know*—it's guaranteed—that things will go wrong.

So I don't think people should be asking "What's the business value of implementing the relational model?" Rather, I think they should be asking, or perhaps trying to explain, what the business value is of *not* implementing it. In other words, those who ask "What's the value of the relational model?" are basically saying "What's the value of database theory?"—and I hereby challenge them to tell me what the value is of *not* abiding by that theory.

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 reasonably precise: so much so, 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 much 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 truth is, those three aspects of the model are inextricably intertwined. For example, the relvars (structural piece) in any given database will be subject to a variety of integrity constraints (integrity piece), and those constraints will be expressed using a variety of relational operators (manipulative piece). Moreover, those relvars will (or should) have been designed in accordance with relational design theory, which likewise involves all three pieces. And of course they'll be subject to update, which again involves all three pieces.

- That "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 (still argued on occasion) that domains and types might somehow be different things. Regarding nulls, incidentally, I note that Codd invented 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 languages and 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 and/or detailed 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.

  Aside: On the other hand, I think it could fairly be argued that the foregoing 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, even in Codd's very first paper. *End of aside*.

  Without further ado, then, let me give my own preferred definition.

  Definition: The relational model consists of five components:

  1. An open ended collection of scalar types, including type BOOLEAN in particular[7]

[7] As explained in Chapter 2, the relational model doesn't rely on the distinction between scalar and nonscalar types in any formal sense. I appeal to it here (as elsewhere in this book) merely as an aid to intuition.

2. A relation type generator and an intended interpretation for relations of types generated thereby

3. Facilities for defining relation variables of such generated relation types

4. A relational assignment operator for assigning relation values to such relation variables

5. A relationally complete (but otherwise 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. First, however, a word of caution. My epigraph to this appendix, by John Muir, bears repeating here: "When we try to pick out anything by itself, we find it hitched to everything else in the universe" (often quoted in the form "Everything is connected to everything else"). John Muir was referring to the natural world, of course, but he might just as well have been talking about the relational model. The fact is, the various features of the relational model are highly interconnected—remove just one of them, and the whole edifice crumbles. Translated into concrete terms, this metaphor means that if we build a "relational" DBMS that fails to support some aspect of the model, the resulting system (which shouldn't really be called relational, anyway) will be bound to display behavior on occasion that's certainly undesirable, and possibly unforeseeable. I can't stress the point too strongly: Every feature of the model is there *for solid practical reasons*; if we choose to ignore some detail, then we do so at our own peril.[8]

Scalar Types

Scalar types can be either system defined or user defined, in general; thus, a means must be available for users to define their own scalar types (this requirement is implicit in the fact that the set of scalar types is open ended). A means must therefore also be available for users to define their own operators, since types without operators are useless. The set of system defined scalar types is required to include type BOOLEAN (the most fundamental type of all, containing precisely two values, viz., the truth values TRUE and FALSE), but a real system will surely support others as well (INTEGER, CHAR, etc.). Support for type BOOLEAN implies support for the usual logical operators (NOT, AND, OR, etc.) as well as other operators, system or user defined, that return boolean values. In particular, the equality comparison operator "=" (which is a boolean operator by definition) must be available in connection with every type, nonscalar types included, 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:

---

[8] I've already said it's misleading to think of the relational model as structure plus integrity plus manipulation because all three aspects are inextricably intertwined. Well, the same goes for the five components to be described in the following subsections, of course (at least to some extent).

Definition: 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.

Aside: The following is a logical consequence of the foregoing definition that can be very helpful in practice. Let *Op* be an operator with a parameter *P*; let *P* be of type *T*, so that the argument corresponding to *P* in any given invocation of *Op* is also of type *T*; and let *v1* and *v2* be values of type *T*. If two successful invocations of *Op* that are identical in all respects except that the argument corresponding to *P* is *v1* in one invocation and *v2* in the other are somehow distinguishable in their effect, then *v1* = *v2* will (in fact, must) evaluate to FALSE. *End of aside.*

Let *T* be some scalar type.[9] Associated with type *T*, then, there's at least one selector operator, with the properties that (a) every invocation of that operator returns a value of type *T* and (b) every value of type *T* is returned by some invocation of that operator (more specifically, by some corresponding literal—recall that a literal is a special case of a selector invocation).

Relation Types

The relation type generator allows users to specify individual relation types as needed (in particular, as the type of 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 be the current value of some relvar—then the predicate in question is the relvar predicate for that relvar. Relvars in particular are interpreted in accordance with *The Closed World Assumption* (see later in this appendix, also Chapter 5 and Appendix C).

Let *T* be some relation type. Associated with *T*, then, there's a relation selector operator, with the properties that (a) every invocation of that operator returns a relation of type *T* and (b) every relation of type *T* is returned by some invocation of that operator (more specifically, by some relation literal). Also, since the equality comparison operator "=" is available in connection with every type, it's available in connection with type *T* in particular. So too is the relational inclusion 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*.

[9] Actually this paragraph applies to all types, scalar or otherwise.

Relation Variables

As noted above, one use—a particularly important one—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; all other kinds of variables, scalar variables or tuple variables or any other kind, are prohibited. (In programs that access such a database, by contrast, they're not prohibited—in fact, they're almost certainly required.)

The statement that the database contains nothing but relvars is one possible formulation of what Codd originally called *The Information Principle*, though it's not 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, and it seems to me that any violation of it must be seen as serious.[10] In particular, 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 everything we need in this respect, and it doesn't give us anything we don't need.

I'd like to pursue this point a moment longer. In general, it's axiomatic that if we have *n* different ways of representing data, then we need *n* different sets of operators. For example, if we had arrays as well as relations, we'd need a full complement of array operators as well as a full complement of relational ones.[11] 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, of course, *n* does equal one).

What's more, not only does the relational model give us just one construct, the relation itself, for representing data, but that construct is—to quote Codd himself (see the section "Objectives of the Relational Model," later in this appendix)—*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

---

[10] It goes without saying that object databases, XML databases, and more generally nonrelational databases of any kind, do all violate it, necessarily.

[11] We'd also have to choose which data we wanted to represent as relations and which as arrays, probably without any good guidelines to help us in making such choices. And what about the database catalog? Would it contain relations, or arrays? Or a mixture?

introducing more operators as well. In fact, SQL is living proof of this observation. For example, SQL has nine different union operators (and ought by rights to have 18, if not 27), while the relational model has just one.

> *Aside:* Perhaps I should explain these last remarks. First of all, SQL supports six different unions for tables as such—UNION DISTINCT, UNION DISTINCT CORRESPONDING, UNION DISTINCT CORRESPONDING BY, and three variants on these in which DISTINCT is replaced by ALL. The funny thing is, the one kind of union it doesn't support for tables as such is true bag union! For the record, here's the definition: Let *b1* and *b2* be bags; let *x* appear exactly *n1* times in *b1* and exactly *n2* times in *b2*; and let *b* be the bag union of *b1* and *b2*. Then *x* appears exactly *n* times in *b*, where *n* = *n1* (if *n1* ≥ *n2*) or *n2* (otherwise). So SQL ought by rights to support BAG or some such keyword as an alternative to DISTINCT and ALL, giving us three more unions. Nine so far. Next, SQL supports two different unions for what it calls "multiset values" (as opposed to tables), viz., MULTISET UNION DISTINCT and MULTISET UNION ALL; however, it ought really to support seven more possibilities here too (involving BAG, CORRESPONDING, and so on), at least if those "multiset values" are multisets of rows specifically. Now we're up to 18. Finally, SQL also supports a union "set function" (for use in summarization), though it calls it not UNION but FUSION. FUSION has no variants, but by rights the same possibilities that apply to tables should apply here too (again, if the "multiset values" in question are multisets of rows specifically). Total: 27.[12] *End of aside.*

As you can see, then, *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," even though we usually talk about it as if it were. Rather, it's a *variable*. After all, it can certainly be updated—and that means it's a variable by definition! Logically speaking, in other words, the database in its entirety is one (typically rather large) variable in itself, which we might call a *dbvar*. I'll elaborate on this concept in the section "Database Variables," later in this appendix.

---

[12] To all of the foregoing I'd like to add a comment Hugh Darwen once made to me (in a private communication): "UNION CORRESPONDING was added to SQL in 1992, presumably to fill some perceived gap in functionality. Suppose it had been part of the language as originally defined; when if ever would the need have emerged to introduce a UNION based on left to right column ordering instead?" I note too that questions like this one apply to a whole host of constructs that have been added to SQL since it was first defined.

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. The operators INSERT, DELETE, and UPDATE (likewise D_INSERT and I_DELETE) 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 (see Chapter 8) and (b) must abide by both *The Assignment Principle* (see Chapter 2) and **The Golden Rule** (see Chapter 8 again).

Relational Operators

The "generic relational operators" are the operators that make up the relational algebra (or something logically equivalent to the 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 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, 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 in question will be just a subexpression of some boolean expression, typically but not invariably an IS_EMPTY invocation)
- Serving as a basis for research 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. To repeat from Chapter 10:

- 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 (or the calculus).

- 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 branching or iterative loops.

- 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.

DATABASE VARIABLES

Note: This section consists of a revised version of material from Appendix D ("What Is a Database?") from the book Databases, Types, and the Relational Model: The Third Manifesto, *by Hugh Darwen and myself (see Appendix G).*

I mentioned in the previous section that databases are really variables (as I said in that section, if a database can be updated, then it's a variable by definition). In other words, we can draw a distinction between database values and database variables, precisely analogous to the one we already draw between relation values and relation variables. As a matter of fact, we—i.e., Darwen and myself—did draw exactly such a distinction in the first version of *The Third Manifesto*. As we said at the time, more or less:

> The first version of this *Manifesto* distinguished databases per se (i.e., database values) from database variables ... It suggested that the unqualified term *database* be used to mean a database value specifically, and it introduced the term *dbvar* as shorthand for "database variable." While we still believe this distinction to be a valid one, we found it had little direct relevance to other aspects of the *Manifesto*. We therefore decided, in the interest of familiarity, to revert to more traditional terminology. [*In other words, we went on to use the term "database" to mean a database variable rather than a database value, and we didn't use the terms "database variable" or "dbvar" at all.*]

And of course I've done the same thing—I mean, I've used the term *database* in the traditional way, and I haven't used the terms *database variable* or *dbvar* at all—throughout the body of the present book. However, the most recent (i.e., third) edition of the *Manifesto* book, after quoting the foregoing text, goes on to say:

> Now this bad decision has come home to roost! With hindsight, it would have been much better to "bite the bullet" and adopt the more logically correct terms *database value* and *database variable* (or dbvar), despite their lack of familiarity.

That same book gives arguments in support of this position, of course, but I don't need to repeat those arguments here; the simple fact is, a database simply *is* a variable (its value changes over time), regardless of whether we call it a "dbvar" or just a database.

Now, it follows from the foregoing that when we "update some relvar" (within some database), what we're really doing is updating the pertinent dbvar. (For clarity, I'll adopt the term *dbvar* for the remainder of the present section.) For example, the **Tutorial D** statement

```
DELETE SP WHERE QTY < 150 ;
```

"updates the shipments relvar SP" and thus really updates the entire suppliers-and-parts dbvar (the "new" database value for that dbvar being the same as the "old" one except that certain shipment tuples have been removed). In other words, while we might say a database "contains variables" (viz., the applicable relvars), such a manner of speaking is only approximate, and in fact quite informal. A more formal and more accurate way of characterizing the situation is this:

A dbvar is a tuple variable.

The tuple variable in question has one attribute for each relvar in the dbvar (and no other attributes), and each of those attributes is relation valued. In the case of suppliers and parts, for example, we can think of the entire dbvar as a tuple variable of the following tuple type:

```
TUPLE { S  RELATION { SNO CHAR , SNAME CHAR ,
                      STATUS INTEGER, CITY CHAR } ,
        P  RELATION { PNO CHAR , PNAME CHAR ,
                      COLOR CHAR , WEIGHT RATIONAL , CITY CHAR } ,
        SP RELATION { SNO CHAR , PNO CHAR , QTY INTEGER } }
```

Suppose we call the suppliers-and-parts dbvar (or tuple variable, rather) SPDB. Then the DELETE statement shown above might be regarded as shorthand for the following tuple assignment:

```
SPDB := TUPLE { S   ( S FROM SPDB ) ,
                P   ( P FROM SPDB ) ,
                SP  ( ( SP FROM SPDB ) WHERE NOT ( QTY < 150 ) ) } ;
```

Explanation: The expression on the right side of this assignment is a tuple selector invocation, and it denotes a tuple with three attributes called S, P, and SP, each of which is relation valued. Within that tuple, (a) the value of attribute S is the current value of relvar S; (b) the value of attribute P is the current value of relvar P; and (c) the value of attribute SP is the current value of relvar SP, minus those tuples for which the quantity is less than 150.

In sum: A dbvar is a tuple variable, and a database (i.e., the value of some given dbvar at some given time) is a tuple. What's more, given a relational assignment of the form

```
R := rx
```

(where *R* is a relvar reference—i.e., a relvar name—denoting a relvar in the database and *rx* is a relational expression), that relvar reference *R* is really a *pseudovariable* reference (see the paragraph immediately following). In other words, that relational assignment is shorthand for a tuple assignment that "zaps" one component of the corresponding dbvar (which is, to repeat, really a tuple variable). It follows that "relation variables" (at least, relation variables in the database) aren't really variables at all; rather, they're a convenient fiction that gives the illusion that the database—or the dbvar, rather—can be updated in a piecemeal fashion, individual relvar by individual relvar.

A note on pseudovariables: Essentially, a pseudovariable reference consists of an operational expression appearing in the target position within an assignment operation. For example, let X be a variable of type CHAR, and let 'Middle' be the current value of X. Then the assignment SUBSTR(X,2,1) := 'u' has the effect of "zapping" the second character position within X, replacing the *i* by a *u*. The expression on the left side of that assignment is a pseudovariable reference. *Note:* The paper "On the Logical Differences Between Types, Values, and Variables" (see Appendix G) discusses the pseudovariable concept in detail.

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 DBA

4. To introduce a theoretical foundation, albeit modest, into database management (a field sadly lacking in solid principles and guidelines)

5. To merge the fact retrieval and file management fields in preparation for the addition at a later time of inferential services in the commercial world

6. To lift database application programming to a new level—a level in which sets (and more specifically relations) are treated as operands instead of being processed element by element

I'll leave it to you to judge to what extent you think the relational model meets these objectives. Myself, I think it does pretty well.

SOME DATABASE PRINCIPLES

In Chapter 1, I said I was interested in principles, not products, and we've encountered several principles at various points in the book. Here I collect them together for ease of reference.

- *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.

  Aside: The Information Principle is closely related to another important concept, viz., the concept of *essentiality*. To elaborate briefly: Let *DM* be a data model in the first sense of that term (see Chapter 1) and let *DS* be a data structure provided by *DM*. Let *dm* be a data model in the second sense of that term (again, see Chapter 1), created using the facilities of *DM*, and let *dm* include an occurrence *ds* of *DS*. Let *db* be a database conforming to *dm*. If removal from *db* of the data corresponding to *ds* would cause a loss of information from *db*, then *ds* is essential in *dm* (and, loosely, *DS* is essential in *DM*). Clearly, then, relational systems provide just one essential data construct, viz., the relation itself. By contrast, nonrelational systems provide numerous different ways of representing information essentially, including (e.g.) pointers, record ordering, repeating groups, and so on and so forth. *End of aside*.

- *The Closed World Assumption:* Let relation *r* correspond to predicate *P*. If tuple *t* appears in *r*, then the proposition *p* corresponding to *t* is assumed to be true; conversely, if tuple *t* plausibly could appear in *r* but doesn't, then the proposition *p* corresponding to *t* is assumed to be false. *Note:* In Chapter 5 I explained *The Closed World Assumption* in terms of relvars, not relations, but the definition just given is slightly more general. Note that it applies to relations that are the current values of relvars in particular, but it isn't limited to such relations. In particular (and importantly), it applies to relations that are the results of queries.

- *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 the database constraint for any database 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.[13] *Note:* I didn't mention this principle earlier in the book, but I appealed to it tacitly on many occasions. It can alternatively be stated thus: *Every entity has its own unique identity*. In the relational model, such identities are represented in the same way as everything else—namely, by means of attribute values (see *The Information Principle* above)—and numerous benefits accrue from this simple fact.

WHAT REMAINS TO BE DONE?

Despite everything I've said in this appendix so far, I don't want to leave you with the impression that we won't continue to make progress, or there isn't still work to be done, in this important field. 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!

To elaborate: First, 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 at all. 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 (at least, not in any mainstream product), 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*

[13] So here we have another reason—a somewhat philosophical reason, perhaps—for rejecting the notion of duplicates.

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, primarily, Hugh Darwen and myself).[14] We believe such an implementation would serve as a solid basis on which to build so many other things—for example, so called "object/relational" DBMSs; spatial and/or temporal DBMSs; DBMSs used in connection with the World Wide Web; and "rule engines" (also known as "business logic servers"), which some 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 much more 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 the process of determining that constraint *rc* be mechanized?

- Can we inject more science into the database design process? In particular, can we come up with a precise and operationally useful characterization of the notion of redundancy? *Note:* The book *Database Design and Relational Theory: Normal Forms and All That Jazz* (see Appendix G) offers some proposals in this connection.

- 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? *Note:* Appendix C of the present book offers some suggestions in this regard.

---

[14] In this connection, we'd also like to see an implementation that's more sophisticated in certain respects than most current SQL implementations typically are. More specifically, we'd like to see an implementation based on *The TransRelational™ Model* (see Appendix G).

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.[15] In other words, what's really going on here is again a raising of the level of abstraction (rather as 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 *Time and Relational Theory: Temporal Data in the Relational Model and SQL* (see Appendix G), Hugh Darwen, Nikos Lorentzos, and I—building on original work by Lorentzos—introduce *interval types* as a basis for supporting temporal data in a relational framework. For example, consider the "temporal relation" in Fig. A.1 below, 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*]

Fig. A.1: A relation with an interval valued attribute

[15] As a matter of fact, Darwen and I show in our *Manifesto* book that every algebraic operator discussed in this book, with the sole exception of TCLOSE, can be expressed in terms of just two primitives, *remove* (which is basically "project over all attributes but one") and either *nand* or *nor* (which are basically algebraic analogs of the logical operators with the same names—see the answer to Exercise 10.4 in Chapter 10).

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 (i.e., conventional) algebraic operators as described in this book. Once again, then, what's fundamentally going on is a raising of the level of abstraction.

Two further points on this topic: First, our 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 all of this temporal support—and 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, it seems to me that database design and the like will still necessarily be based on solid relational principles. At least, I certainly hope so.)

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 truly 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,[16] 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 kind of 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.

---

[16] Yes, it really was thought of in such terms. Here's a quote from the very first paper on the language we now know as SQL (see Appendix G): "Examples of such users are accountants, engineers, architects, and urban planners. It is for this class of users that [SQL] is intended."

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 say it again, we do realize that SQL databases and applications are going to be with us for quite a long time—to suggest 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—even things like duplicates and nulls—without having to support such concepts directly in the underlying relational language.

Appendix B

SQL Departures from the Relational Model

Between the idea
And the reality ...
Falls the Shadow

—T. S. Eliot:
The Hollow Men (1925)

In this appendix I summarize, mainly for purposes of reference and with little by way of additional commentary, some of the ways in which SQL—by which I mean, as always in this book, the standard version of that language, except where otherwise noted—departs from the relational model. Now, I know there are those who will quibble over specific items in what follows; it's not easy to compile a list of this kind, especially if it's meant to be orthogonal (i.e., if an attempt is made to keep the various items all independent of one another). But I don't think such quibbling is important. What's important is the cumulative effect, which quite frankly I think is overwhelming.[1]

- SQL fails to distinguish adequately between table values and table variables.
- SQL tables aren't the same as relations (or relvars), because they either permit or require, as the case may be:

  a. Duplicate rows

  b. Nulls

[1] I remind you too, one more time, that *All logical differences are big differences*.

c. Left to right column ordering

d. Anonymous columns

e. Duplicate column names

f. Pointers

g. Hidden columns (at least in some products, though not in the standard as such)

Note that all of these differences constitute violations of *The Information Principle* (see Appendix A).

- SQL has no proper table literals.
- SQL often seems to think views aren't tables.
- SQL tables—views included!—must have at least one column (no support for TABLE_DEE and TABLE_DUM).
- SQL has no support for empty rows or empty keys. (In fact, SQL suffers from numerous defects in connection with empty sets, as documented in Chapter 12.)
- SQL has no explicit table assignment operator.
- SQL has no explicit multiple table assignment a fortiori (nor does it have an INSERT / DELETE analog).
- SQL violates *The Assignment Principle* in numerous different ways (some but not all of them having to do with nulls).
- SQL violates **The Golden Rule** in numerous different ways (some but not all of them having to do with nulls).

- SQL has no proper "table type" notion. As a consequence, its support for table type inference (i.e., determining the type of the result of some table expression) is very incomplete.
- SQL has no "=" operator for tables; in fact, it has no proper table comparison operators, as such, at all.
- SQL supports "reducible keys" (i.e., it allows proper superkeys to be declared as keys).
- SQL's union, intersection, and join operators aren't commutative.
- SQL's union, intersection, and join operators aren't idempotent.
- SQL's intersection operator isn't a special case of SQL's natural join operator.
- SQL has no proper aggregate operators.
- Numerous SQL operators are "possibly nondeterministic."
- SQL supports various row level operators (cursor updates, row level triggers).
- Although the SQL standard doesn't, the dialects of SQL supported in various commercial products do sometimes refer to certain storage level constructs (e.g., indexes).
- SQL's view definitions include mapping information as well as structural information.[2]
- SQL's support for view updating is weak, ad hoc, and incomplete.
- SQL fails to distinguish properly between types and representations.
- SQL's "structured types" are sometimes encapsulated and sometimes not. (This issue wasn't discussed in the body of this book.)
- SQL fails to distinguish properly between types and type generators.

---

[2] To be fair, this criticism applies to **Tutorial D** too at the time of writing.

- Although the SQL standard does support type BOOLEAN, commercial SQL products typically don't.
- SQL's support for "=" is seriously deficient. To be more specific, SQL's "=" operator:
  a. Can give TRUE even when the comparands are clearly distinct[3]
  b. Can fail to give TRUE even when the comparands aren't clearly distinct
  c. Can have user defined, and hence arbitrary, semantics (for user defined types)
  d. Isn't supported at all for the system defined type XML
  e. In some products, isn't supported for certain other types as well
- SQL is based on three-valued logic (sort of), whereas the relational model is based on two-valued logic.
- SQL isn't relationally complete.

The foregoing list is not exhaustive.

[3] One consequence of this point is that two rows can be duplicates of each other without being identical. And one consequence of *that* point is that the definition of (e.g.) SQL's UNION DISTINCT operator has to look something like this: Let tables *t1* and *t2* be the union operands; let *r* be a row that's a duplicate of some row in *t1* and a duplicate of some row in *t2*; then the result table contains (a) exactly one duplicate of every such row *r* and (b) no row that's not a duplicate of some such row *r*. (And even this definition is incomplete, of course—we need additional rules specifying the column names and types of that result table.) Contrast the simple relational definition of union in Chapter 6.

Appendix C

A Relational Approach to Missing Information

Missing so much and so much

—Frances Cornford:
To a Fat Lady Seen from the Train (1910)

The book *Database Explorations: Essays on The Third Manifesto and Related Matters*, by Hugh Darwen and myself (see Appendix G), describes a variety of approaches to the problem of missing information, all of which avoid the use of, or apparent need for, SQL-style nulls. The present appendix is based in part on a chapter from that book, and it describes one of those approaches in detail. The approach in question is known as *the decomposition approach*, because it involves decomposing, in a variety of ways, relvars that might appear to require nulls (or something like them) into ones that don't. In other words, the emphasis is on designing the database in such a way as to avoid a perceived need for nulls. As a consequence, the approach:

- Has no notion of null or any other construct that's allowed to appear wherever a value is expected and yet isn't itself a value;
- Relies exclusively on classical two-valued logic (2VL), instead of three-valued logic (3VL) or, more generally, *n*-valued logic (*n*VL) for some $n > 2$;
- Abides by *The Information Principle*—see Appendix A—in that, at all times, the database contains relations and nothing but relations; and
- Is capable of dealing with missing information of any number of different kinds.

Note: Before going any further, I should mention that the approach I'm going to be describing is similar but not identical to one proposed by David McGoveran in 1994 in a series of papers with the overall title "Nothing from Nothing" (again, see Appendix G).

Consider Fig. C.1, which shows a version of our usual suppliers table in which certain information is missing (indicated in the figure, as in Chapters 1 and 4, by shading the pertinent entries). Note that I can't say the figure shows a *relation*, precisely because of those shaded

entries; hence my use of the term *table*, and the related terms *column* and *row*, here and throughout much of this appendix.

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

Fig. C.1: Table S—sample "value" (?)

Now, I said in Chapter 5 that the predicate for suppliers was as follows:

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

For present purposes, however, I'll simplify this predicate slightly by dropping the bit about the supplier being under contract. The predicate becomes:

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

Observe now that this predicate is at best approximate. It would be appropriate if it weren't for those shaded entries. After all, the following—obtained from the predicate by substituting values from the row in Fig. C.1 for supplier S1, which has no shaded entries—is a meaningful instantiation of it (i.e., it's a meaningful proposition):

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

But if we substitute values from the row for supplier S2, we obtain:

Supplier S2 is named Jones, has status 10, and is located in city ▒▒▒▒▒▒.

And this certainly isn't a meaningful instantiation, or proposition; in fact, it doesn't make sense at all.

Another interesting question is: What are the data types for columns STATUS and CITY? (I'm assuming here for the sake of the example, and I'll continue to assume throughout the rest of this appendix, that shaded entries don't appear, and won't ever appear, in the other two columns, SNO and SNAME.) In SQL in particular, the shaded entries in columns STATUS and CITY can be interpreted as meaning the pertinent entries are null; elsewhere in this book, however, I've said the (SQL) data types of those columns are INTEGER and VARCHAR(20),

respectively, and null certainly isn't a value of either type INTEGER or type VARCHAR(20). In fact, of course, null isn't a value at all, and so it can't be said to be of any type at all.[1]

From this preliminary discussion, it should be clear that what we need to do is get rid of those shaded entries. Two kinds of decomposition, vertical and horizontal, can be used to achieve this goal.

VERTICAL DECOMPOSITION

The first step in that process of getting rid of those shaded entries is to apply *vertical decomposition* to produce a set of tables with the property that no table ever has more than one column with any such entries. (Note that vertical decomposition in this sense—vertical because the dividing lines in the decomposition are between columns, so to speak—is reminiscent of what we do when we do classical normalization.) For the table in Fig. C.1, the result of this step is the collection of tables SN, ST, and SC as shown in Fig. C.2. *Note:* For obvious reasons I use T, not S, as an abbreviation for STATUS throughout this appendix.

SN

SNO	SNAME
S1	Smith
S2	Jones
S3	Blake
S4	Clark

ST

SNO	STATUS
S1	20
S2	10
S3	
S4	

SC

SNO	CITY
S1	London
S2	
S3	Paris
S4	

Fig. C.2: Vertically decomposing table S

Now, the "obvious" (?) predicates for the tables in Fig. C.2 are as follows:

- SN: *Supplier SNO is named SNAME.*
- ST: *Supplier SNO has status STATUS.*
- SC: *Supplier SNO is located in city CITY.*

In fact the "obvious" predicate is indeed the correct one in the case of table SN. But those "obvious" predicates for tables ST and SC are still only approximate, because of those shaded entries—and that's why we need horizontal decomposition, which I'll get to in the next section.

---

[1] In SQL, by contrast, it's considered to be of *every* type. To quote the standard: "Every data type includes a special value, called the *null value* … [that] is neither equal to any other value nor not equal to any other value." Note that phrase "null value," bythe way! Since the crucial point about nulls is that they're not values, that phrase is simply a contradiction in terms.

First, however, note that each of tables SN, ST, and SC has just two columns. But this state of affairs is a fluke, in a way; it's a direct result of my choice of example. If the example were different—e.g., if we knew that column STATUS, as well as columns SNO and SNAME, will never contain any shaded entries—then the appropriate vertical decomposition would be as shown in Fig. C.3 below. *Note:* I've assumed in Fig. C.3, just for the sake of this revised example (but in accordance with our usual sample values), that suppliers S3 and S4 have status 30 and 20, respectively, instead of those STATUS values being missing as they are in Fig. C.2.

SNT

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

SC

SNO	CITY
S1	London
S2	▒▒▒▒▒▒
S3	Paris
S4	▒▒▒▒▒▒

Fig. C.3: Vertically decomposing table S, if every supplier has a known status

HORIZONTAL DECOMPOSITION

In horizontal decomposition, the dividing lines in the decomposition are between rows (so to speak) instead of between columns. The basic motivation for such decomposition is this: We shouldn't try to use the same table to represent two or more different predicates. For example, consider table SC again, as shown in either Fig. C.2 or Fig. C.3. In that table, the row for supplier S1 means: *Supplier S1 is located in London*. By contrast, the row for supplier S2 means: *We don't know where supplier S2 is located* (at any rate, let's agree that's what it means for the time being). So different rows correspond to different predicates, and the "obvious" predicate I gave for SC earlier—*Supplier SNO is located in city CITY*—doesn't in fact apply to every row.

Now, we might try a different predicate, perhaps like this (note the OR, which I've shown in uppercase bold for emphasis):

Supplier SNO is located in city CITY ***OR*** *we don't know where supplier SNO is located.*

But this predicate doesn't work either. If we try to instantiate it with values (or "values," rather, since ▒▒▒▒▒▒ certainly isn't a value) from the row for supplier S2, we get:

Supplier S2 is located in city ▒▒▒▒▒▒ ***OR*** *we don't know where supplier S2 is located.*

And the first half of this sentence—Supplier S2 is located in city ▒▒▒▒▒▒—still makes no sense, because ▒▒▒▒▒▒ isn't a legitimate city name and can't legitimately be substituted as an

argument for the CITY parameter in the putative predicate. So what we need to do is break the predicate into two separate pieces, as it were (more precisely, we need to break the two disjuncts apart, *disjunct* being—as you'll recall from the answer to Exercise 10.10 in Chapter 10—the correct term for a sentence that's OR'ed with another such); in other words, we need to apply horizontal decomposition to table SC, to obtain one table for each of those disjuncts. The result of this step is the tables shown in Fig. C.4.

SC

SNO	CITY
S1	London
S3	Paris

SUC

SNO
S2
S4

Fig. C.4: Horizontally decomposing table SC

As you can see, we now have two tables: (a) an abbreviated version of table SC (for which I've retained that same name SC, for convenience), containing just the original SC rows that had no shaded entries in column CITY; and (b) another table SUC ("suppliers with an unknown city"), containing just the original SC rows that did have shaded entries in column CITY—except that the CITY column in that table, if we kept it, would contain nothing but shaded entries, and so we can discard it without losing any information. The predicates for these two tables are as follows:

- SC: *Supplier SNO is located in city CITY.*
- SUC: *We don't know where supplier SNO is located.*

Observe in particular that the predicate for this new version of table SC has two parameters, SNO and CITY, and that table has two columns accordingly; by contrast, the predicate for table SUC has just one parameter, SNO, and that table has just one column accordingly.

Of course, we can and should perform an analogous horizontal decomposition on table ST from Fig. C.2. The result is shown in Fig. C.5.

ST

SNO	STATUS
S1	20
S2	10

SUT

SNO
S3
S4

C.5: Horizontally decomposing table ST

The predicates for the tables in Fig. C.5 are as follows:

- ST: *Supplier SNO has status STATUS.*
- SUT: *We don't know supplier SNO's status.*

Note: Each of tables SN (Fig. C.2), SC and SUC (in Fig. C.4), and ST and SUT (in Fig. C.5) is in fact "truly relational," and so I could now switch to my preferred terminology of relations, tuples, and attributes. For simplicity, however, I'll continue to use the terminology of tables, rows, and columns throughout the remainder of this appendix.

WHAT DO THE SHADED ENTRIES MEAN?

Let's ignore status values for the moment and concentrate on cities. So far, then, I've said that shaded entries in the CITY column as shown in, e.g., Fig. C.2 mean *we don't know* the applicable supplier city—i.e., the supplier does have a city, but we don't know what it is. But our not knowing is only one of many possible reasons why we might not be able to use a genuine city name as some entry in that column. For example, it might be that the notion of having a city simply doesn't apply to some suppliers (perhaps they conduct their business entirely online). If so, we might say, *very* loosely, that table SC, with those shaded entries in the CITY column (i.e., table SC as shown in Fig. C.2), has a predicate looking something like this:

> *Supplier SNO is located in city CITY* ***OR*** *we don't know where supplier SNO is located* ***OR*** *supplier SNO isn't located anywhere.*

Note, therefore, that those shaded entries now potentially have two distinct interpretations: Some of them mean we don't know the applicable city, others mean the property of having a city doesn't apply. So, again, we apply horizontal decomposition, this time to obtain three tables: SC (suppliers with a known city), SUC (suppliers with an unknown city), and SNC (suppliers with no city). If we assume for the sake of the example that supplier S2 has an unknown city and supplier S4 doesn't have a city at all, the result of this decomposition is as shown in Fig. C.6.

SC

SNO	CITY
S1	London
S3	Paris

SUC

SNO
S2

SNC

SNO
S4

Fig. C.6: Horizontally decomposing table SC, allowing for suppliers with no city

The predicates are:

- SC: *Supplier SNO is located in city CITY.*
- SUC: *We don't know where supplier SNO is located.*
- SNC: *Supplier SNO doesn't have a location.*

In other words, the decomposition approach allows us to represent as many different kinds of missing information as we like. To be specific, if there are *n* distinct reasons for supplier cities to be missing, there'll be *n* + 1 tables having to do with suppliers and cities. Two possible objections to the approach thus immediately spring to mind:

1. Aren't some queries going to get awfully complex? For example, suppose we just want to retrieve everything in the database having to do with suppliers (the analog of "SELECT * FROM S" in SQL); aren't we going to have to do a lot of joins, or (worse) outer joins?

2. Aren't we going to wind up with an awful lot of tables?

I'll come back to the first of these issues in the section "Queries," later. As for the second, well, there are several points I want to make. Let *C* be an SQL column for which nulls are allowed. Then:

- If the nulls in column *C* all represent the same kind of missing information, and if the same is true for all such columns *C*, then the number of tables resulting from the decomposition approach is exactly the same as the number resulting from a good relational design. (To paraphrase something I said earlier, the presence of such a column *C* in a table *T* means table *T* is certainly not a *relational* table. Proper relational design requires elimination of such columns.)

- The situation is worse if the nulls in some such column *C* represent two or more distinct kinds of missing information but proper decomposition isn't done. If it isn't, there'll certainly be fewer tables—but the apparent simplicity of such a design is spurious: Those tables aren't relational, they don't faithfully reflect the real world, they no longer have a clear predicate, and queries are more susceptible to errors of formulation or errors of interpretation or both.

- There's a tactic we might consider, if we want to reduce the number of tables, which I'll illustrate with reference to Fig. C.6. In terms of that example, the tactic would involve

combining tables SUC and SNC into a single table with two columns, SNO and REASON, where REASON indicates the reason why the applicable supplier has no recorded city:

SNO	REASON
S2	*d/k*
S4	*n/a*

But now we have to define appropriate values, and spell out their interpretations, for column REASON (in the example, I've used *d/k* for "don't know" and *n/a* for "not applicable"). In fact, if the decomposition approach requires *n* missing information tables, the combination approach requires *n* missing information reasons. So the combination approach is in some respects no less complex than the decomposition approach. (In this connection, see Exercise C.1 at the end of this appendix.)

CONSTRAINTS

So far, then, our suggested overall design for the running example looks like Fig. C.7 below.

SN

SNO	SNAME
S1	Smith
S2	Jones
S3	Blake
S4	Clark

ST

SNO	STATUS
S1	20
S2	10

SUT

SNO
S3
S4

SC

SNO	CITY
S1	London
S3	Paris

SUC

SNO
S2

SNC

SNO
S4

Fig. C.7: Fully decomposing table S

I'm assuming here, and will continue to assume throughout the rest of this appendix, that there's just one reason why STATUS values might be missing (viz., we don't know the value) and just two reasons why CITY values might be missing (viz., either we don't know the value or no such value exists). Note, however, that the design of Fig. C.7 requires certain constraints to be satisfied in order to hold it together, so to speak. To be specific, the following constraints need to be stated and enforced:

1. Each table has {SNO} as a key.

2. Each row in SN has a matching row in exactly one of ST and SUT, and conversely.

3. Each row in SN has a matching row in exactly one of SC, SUC, and SNC, and conversely.

Of course, the first of these is just a conventional key constraint on each of the six tables; it can thus be expressed by means of conventional KEY specifications. As for the other two, they can easily be expressed in **Tutorial D** using D_UNION, as follows:[2]

```
CONSTRAINT EQD2
    SN { SNO } = D_UNION { ST { SNO } , SUT { SNO } } ;

CONSTRAINT EQD3
    SN { SNO } = D_UNION { SC { SNO } , SUC { SNO } , SNC { SNO } } ;
```

Aside: Actually it might not be a good idea to use D_UNION in a constraint as I've just done. After all, if some update violates the constraint in question, we don't want a run-time failure to occur during constraint checking, we just want the constraint to evaluate to FALSE and the update to be rejected. So constraint EQD2, for example, might better be formulated as follows:

```
CONSTRAINT EQD2 SN { SNO } = ( ST { SNO } UNION SUT { SNO } )
                AND IS_EMPTY ( ST { SNO } JOIN  SUT { SNO } ) ;
```

End of aside.

QUERIES

Now I return to the question I raised earlier: Given a design like that of Fig. C.7, aren't some queries going to get awfully complex? In particular, what's involved with that design in doing a query analogous to the "simple" SQL query SELECT * FROM S?

Before I address that issue, let me first point out that some queries—queries, I venture to suggest, that are more likely to be needed in practice than ones like SELECT * FROM S—are actually easier to formulate with the design of Fig. C.7. As a trivial example, the query "For suppliers for whom CITY is both applicable and known, get supplier numbers and cities" becomes just

---

[2] They can be expressed in SQL, too, though not quite so easily (exercise for the reader). In fact, they're both examples of what are called, for obvious reasons, *equality dependencies* or EQDs. (EQDs were mentioned briefly in Chapter 9, as you might recall.) Note that if an EQD is in effect, spanning two or more tables, then certain updates on just one of those tables will necessarily cause that EQD to be violated. See the discussion of multiple assignment and related matters in Chapter 8.

```
SELECT SNO , CITY
FROM   SC
```

instead of:

```
SELECT SNO , CITY
FROM   S
WHERE  CITY IS NOT NULL
```

What's more, the query "Get suppliers for whom CITY is applicable but unknown" is not only simpler with the design of Fig. C.7, it can't be done at all with the original design of Fig. C.1. (In other words, not only does the design of Fig. C.1 not deal very well with the missing information problem in general, it actually manages to *lose* information!)

Be that as it may, let's now consider the "SELECT * FROM S" question. More precisely, let's see how a respectable version of the table in Fig. C.1 can be obtained from those in Fig. C.7—where by *respectable*, I mean the table will contain proper and informative data values everywhere (no shaded entries! no nulls!), as indicated in Fig. C.8 below.

S

SNO	SNAME	XSTATUS	XCITY
S1	Smith	20	London
S2	Jones	10	*d/k*
S3	Blake	*d/k*	Paris
S4	Clark	*d/k*	*n/a*

Fig. C.8: Revised (respectable) version of table S

Now, however, I'll switch to **Tutorial D** (doing the example in SQL would make it too hard to see the forest for the trees). I'll show the solution a step at a time, using the values from Fig. C.7 as a basis for illustrating the result of each step in turn; then I'll bring all the steps together at the end.

1.
```
WITH ( t1 := EXTEND ST : { XSTATUS := CAST_AS_CHAR ( STATUS ) } ) :
```

t1

SNO	STATUS	XSTATUS
S1	20	20
S2	10	10

```
/* STATUS values are integers,          */
/* XSTATUS values are character strings */
```

2. `WITH ( t2 := t1 { ALL BUT STATUS } ) :`

t2

SNO	XSTATUS
S1	20
S2	10

3. `WITH ( t3 := EXTEND SUT : { XSTATUS := 'd/k' } ) :`

t3

SNO	XSTATUS
S3	*d/k*
S4	*d/k*

4. `WITH ( t4 := UNION { t2 , t3 } ) :`

t4

SNO	XSTATUS
S1	20
S2	10
S3	*d/k*
S4	*d/k*

5. `WITH ( t5 := SC RENAME { CITY AS XCITY } ) :`

t5

SNO	XCITY
S1	London
S3	Paris

6. `WITH ( t6 := EXTEND SUC : { XCITY := 'd/k' } ) :`

t6

SNO	XCITY
S2	*d/k*

7. `WITH ( t7 := EXTEND SNC : { XCITY := 'n/a' } ) :`

t7

SNO	XCITY
S4	*n/a*

8. `WITH ( t8 := UNION { t5 , t6 , t7 } ) :`

t8

SNO	XCITY
S1	London
S2	*d/k*
S3	Paris
S4	*n/a*

9. `WITH ( S := JOIN { SN , t4 , t8 } ) : S`

S

SNO	SNAME	XSTATUS	XCITY
S1	Smith	20	London
S2	Jones	10	*d/k*
S3	Blake	*d/k*	Paris
S4	Clark	*d/k*	*n/a*

Putting all of these steps together and simplifying slightly, we have:

```
WITH ( t1 := EXTEND ST : { XSTATUS := CAST_AS_CHAR ( STATUS ) } ,
       t2 := t1 { ALL BUT STATUS } ,
       t3 := EXTEND SUT : { XSTATUS := 'd/k' } ,
       t4 := UNION { t2 , t3 } ,
       t5 := SC RENAME { CITY AS XCITY } ,
       t6 := EXTEND SUC : { XCITY := 'd/k' } ,
       t7 := EXTEND SNC : { XCITY := 'n/a' } ,
       t8 := UNION { t5 , t6 , t7 } ,
       S  := JOIN { SN , t4 , t8 } ) :
S
```

Now, it's certainly true that this expression looks a little complicated (or tedious, at any rate), and it might look even more so if I hadn't formulated it a step at a time, using WITH. However:

- Various shorthands could be defined, if desired, that could be used to simplify it.

- I frankly doubt whether tables such as that in Fig. C.8 would ever be wanted much in practice anyway, except perhaps as the basis for some kind of periodic report.

- In any case, the complexity, such as it is, can always be concealed by making the table a view.

MORE ON PREDICATES

Note: This section is based on material from Chapter 4 ("The Closed World Assumption") of my book Logic and Databases: The Roots of Relational Theory *(see Appendix G).*

In this section, I show how it's possible to get "don't know" answers out of a database without nulls, even if there aren't any tables like table SUC (suppliers with an unknown city) that explicitly represent the fact that something is unknown. For simplicity, suppose our database consists in its entirety of just table SC (suppliers with a known city), as shown in Fig. C.9 below.

SC

SNO	CITY
S1	London
S3	Paris

Fig. C.9: Table SC (suppliers with a known city)

Now consider the following query on table SC:

Is supplier S1 in London?

In **Tutorial D:**[3]

```
( SC WHERE SNO = 'S1' AND CITY = 'London' ) { }
```

Clearly, this expression evaluates to either TABLE_DEE or TABLE_DUM (TABLE_DEE if supplier S1 is in London, TABLE_DUM otherwise). Note, therefore, that—as I mentioned briefly in Chapter 3—TABLE_DEE and TABLE_DUM can be interpreted as *yes* and *no*, respectively. Note too the implicit appeal to *The Closed World Assumption!*—in effect, we're

[3] I have to use **Tutorial D** here, not SQL, because the example under discussion is a yes/no query, as we'll see in a moment. As a consequence, it relies on the special relations TABLE_DEE and TABLE_DUM (the only relations of degree zero—see Chapter 3), and SQL doesn't support those relations.

saying that if the row (S1,London) fails to appear in table SC, we're allowed to conclude that *it's not the case that* supplier S1 is in London.

Now, I said previously that the predicate for table SC was *Supplier SNO is located in city CITY*. But it isn't—not really. To see why not, consider what happens if some user tries to introduce a new row into the table, perhaps as follows:

```
INSERT SC RELATION { TUPLE { SNO 'S6' , CITY 'Madrid' } } ;
```

In effect, the user here is telling the system there's a new supplier, S6, with city Madrid. Now, the system obviously has no way of knowing whether the user's assertion is true; as I explained in Chapter 8, all it can do (and does do) is check that the requested insertion, if performed, doesn't cause any integrity constraints to be violated. If it doesn't, then the system accepts the row, *and interprets it as representing a true fact from this point forward*.

We see, therefore, that rows in table SC don't necessarily represent actual states of affairs in the real world; rather, they represent *what the user tells the system* about the real world, or in other words the user's *knowledge* of the real world. Thus, the predicate for relvar SC isn't really just *Supplier SNO is located in city CITY*; rather, it's ***We know that*** *supplier SNO is located in city CITY*. And the effect of a successful INSERT is to make the system aware of something the user already knows. Thus, the database doesn't contain the real world (of course not); what it contains is, rather, *the system's knowledge of* the real world. And the system's knowledge is derived in turn from the user's knowledge (of course!—there's no magic here).[4]

So when we pose a query to the system, by definition that query can't be a query about the real world; instead, it is—it has to be—a query about the system's knowledge of the real world. For example, consider again the query discussed above: *Is supplier S1 in London?* This rather imprecise natural language formulation has to be understood as shorthand for the following more accurate one:

Do we know that *supplier S1 is in London?*

In practice, of course, we almost never talk in such precise terms; we usually elide qualifiers like "Do we know that" (or "According to the system's knowledge, is it true that," or "Does the database say that," and so on). But even if we do elide them, we certainly need to understand that, conceptually, they're there—for otherwise we'll be really confused. (Though perhaps I should add that such confusions aren't exactly unknown in practice.)

It follows from the foregoing discussion that the **Tutorial D** expression I showed earlier—

```
( SC WHERE SNO = 'S1' AND CITY = 'London' ) { }
```

[4] Even the terms *know* and *knowledge* might be a little strong in contexts such as those at hand (the terms *believe* and *beliefs* might be better)—but I'll stay with *know* and *knowledge* for the purposes of the present discussion.

—doesn't really represent the query *Is supplier S1 in London?* after all. Rather, it represents the query *Do we know that supplier S1 is in London?* And, appealing again to *The Closed World Assumption,* it follows further that:

- If the result is TABLE_DEE (*yes*), it means we do know supplier S1 is in London.
- If the result is TABLE_DUM (*no*), it means *we don't know whether* supplier S1 is in London. And that's a "don't know" answer if ever you saw one.

Of course, if a row for supplier S1 does appear in the table but the CITY value in that row isn't London, we know that supplier S1 *isn't* in London (I'm appealing here to the fact that {SNO} is a key for table SC). Putting it all together, then, we have the following:

- If a row for supplier S1 appears in table SC and the CITY value in that row is London, it means yes, we know supplier S1 is in London.
- If a row for supplier S1 appears in table SC but the CITY value in that row is something other than London, it means no, we know supplier S1 isn't in London.
- And if no row for supplier S1 appears in table SC at all, it means we don't know whether supplier S1 is in London.

Given *The Closed World Assumption*, then, we can formulate queries that return a true / false / unknown answer, and we *don't* need nulls or 3VL to do so. Here's a **Tutorial D** formulation for the example under discussion:

```
( EXTEND ( SC WHERE SNO = 'S1' AND CITY = 'London' ) { } :
                              { RESULT := 'true    ' } ) { RESULT }
  UNION
( EXTEND ( SC WHERE SNO = 'S1' AND CITY ≠ 'London' ) { } :
                              { RESULT := 'false   ' } ) { RESULT }
  UNION
( EXTEND ( RELATION { TUPLE { SNO 'S1' } } MINUS SC { SNO } ) { } :
                              { RESULT := 'unknown' } ) { RESULT }
```

As you can see, this expression takes the form *a* UNION *b* UNION *c*, where each of *a*, *b*, and *c* is a table of just one column, called RESULT. Moreover, it should be clear that exactly one of *a*, *b*, and *c* contains just one row and the other two contain no rows at all. The overall result is thus a one-column, one-row table; the single column, RESULT, is of type character string, and the single row contains the appropriate RESULT value. And the trick—though it isn't really a trick at all—is that the RESULT value is a character string, not a truth value. As a consequence, there's no need to get into the 3VL quagmire in order to formulate queries that can yield "true," "false," or "unknown" answers, if that's what we really want.

For completeness, here's an SQL analog of the foregoing **Tutorial D** expression:

```
SELECT 'true    ' AS RESULT
FROM ( SELECT *
       FROM   SC
       WHERE  SNO = 'S1'
       AND    CITY = 'London' ) AS POINTLESS1
UNION  CORRESPONDING
SELECT 'false   ' AS RESULT
FROM ( SELECT *
       FROM   SC
       WHERE  SNO = 'S1'
       AND    CITY <> 'London' ) AS POINTLESS2
UNION  CORRESPONDING
SELECT 'unknown' AS RESULT
FROM ( VALUES ('S1' )
       EXCEPT
       SELECT SNO
       FROM   SC ) AS POINTLESS3
```

Incidentally, if you're wondering about those AS *POINTLESS* specifications in this SQL expression, I remind you from Chapter 7 (or Chapter 12) that SQL has a syntax rule to the effect that a table subquery in the FROM clause *must* be accompanied by an explicit AS clause that defines an associated range variable, even if that range variable is never explicitly referenced anywhere in the overall expression. Note also that specifying CORRESPONDING on the EXCEPT in the final portion of this expression would actually be incorrect! It could be made correct by replacing the specification VALUES ('S1') by an expression of the form SELECT DISTINCT 'S1' AS SNO FROM *T* where *T* is some arbitrary—but nonempty—named table.

EXERCISES

C.1 Give an appropriate predicate for the table shown near the top of page 500 (with columns SNO and REASON).

C.2 Give SQL versions of constraints EQD2 and EQD3 from the section "Constraints" in the body of the appendix.

C.3 Give an SQL version of the **Tutorial D** expression near the end of the section "Queries" in the body of the appendix.

C.4 Why would it be incorrect to specify CORRESPONDING on the EXCEPT in the final portion of the SQL expression at the end of the section immediately preceding these exercises?

ANSWERS

C.1 REASON is either 'd/k' or 'n/a', and if REASON is 'd/k' then we don't know where supplier SNO is located, and if REASON is 'n/a' then supplier SNO doesn't have a location.

C.2 Here's an SQL version of constraint EQD2 (only; constraint EQD3 is essentially similar, of course).

```
CREATE ASSERTION EQD2 CHECK
     ( NOT EXISTS ( SELECT SNO
                    FROM   ST
                    WHERE  SNO IN ( SELECT SNO
                                    FROM   SUT ) )
       AND
       NOT EXISTS ( SELECT SNO
                    FROM   SUT
                    WHERE  SNO IN ( SELECT SNO
                                    FROM   ST ) )
       AND
       NOT EXISTS ( SELECT SNO
                    FROM   SN
                    WHERE  SNO NOT IN ( SELECT SNO
                                        FROM   ST
                                        UNION  CORRESPONDING
                                        SELECT SNO
                                        FROM   SUT ) )
       AND
       NOT EXISTS ( SELECT SNO
                    FROM ( SELECT SNO
                           FROM   ST
                           UNION  CORRESPONDING
                           SELECT SNO
                           FROM   SUT ) AS POINTLESS
                    WHERE  SNO NOT IN ( SELECT SNO
                                        FROM   SN ) ) ) ;
```

C.3

```
WITH t1 AS ( SELECT SNO , STATUS ,
                    CAST ( STATUS AS CHAR ( 3 ) ) AS XSTATUS
             FROM   ST ) ,
     t2 AS ( SELECT SNO , XSTATUS
             FROM   t1 ) ,
     t3 AS ( SELECT SNO , 'd/k' AS XSTATUS
             FROM   SUT ) ,
     t4 AS ( SELECT SNO , XSTATUS
             FROM   t1
             UNION  CORRESPONDING
             SELECT SNO , XSTATUS
             FROM   t3 ) ,
     t5 AS ( SELECT SNO , CITY AS XCITY
             FROM   SC ) ,
     t6 AS ( SELECT SNO , 'd/k' AS XCITY
             FROM   SUC ) ,
```

```
t7 AS ( SELECT SNO , 'n/a' AS XCITY
        FROM   SNC ) ,
t8 AS ( SELECT SNO , XCITY
        FROM   t5
        UNION  CORRESPONDING
        SELECT SNO , XCITY
        FROM   t6
        UNION  CORRESPONDING
        SELECT SNO , XCITY
        FROM   t7 ) ,
S  AS ( SELECT SNO , SNAME , XSTATUS , XCITY
        FROM   SN NATURAL JOIN t4 NATURAL JOIN t8 )
SELECT SNO , SNAME , XSTATUS , XCITY
FROM   S
```

C.4 Because CORRESPONDING means “match on column names” and the single column in the table produced by the expression VALUES('S1') doesn’t have a name.

Appendix D

A Tutorial D Grammar

I never use a big, big D—

—W. S. Gilbert:
HMS Pinafore (1878)

For purposes of reference, this appendix gives a BNF grammar for **Tutorial D** relational expressions and assignments. (Nonrelational operations—including aggrgegate operations in particular—are mostly omitted, as are definitional operations such as those used to define types, base relvars, views, and constraints.) The following are also omitted:

- TUPLE FROM, because it doesn't return a relation
- THE_ operators and <*attribute name*> FROM, because these operators too don't return relations (except in the unusual special case where the specified possrep component or attribute, as applicable, happens to be relation valued)
- DIVIDEBY and SUMMARIZE, because (as explained in Chapter 7) these operators are both somewhat deprecated

Also, the grammar is simplified in certain respects. In particular, it makes no attempt to say where image relations can and can't be used, nor does it pay any attention to operator precedence rules. (As a result of this latter point, certain constructs permitted by the grammar—for example, the expression *r1* MINUS *r2* MINUS *r3*—are potentially ambiguous. Additional syntax rules are needed to resolve such issues, but such rules are omitted here. Of course, parentheses can always be used to guarantee a desired order of evaluation anyway.) A few points of detail:

- The following simplifying abbreviations are used:

```
exp     for     expression
op      for     operator
comp    for     comparison
```

- All syntactic categories of the form <... *name*> are assumed to be <*identifier*>s and are defined no further here.

- The categories <*tuple exp*> and <*boolean exp*> are also left undefined—though it might help to recall in particular that a relational comparison is a special case of a boolean expression.

- As usual, all of the various commalists mentioned in what follows are allowed to be empty.

Relational Expressions

```
<relation exp>
    ::=   <with exp> | <nonwith exp>

 <with exp>
    ::=   WITH ( <name intro commalist> ) : <relation exp>

<name intro>
    ::=   <relvar name> := <relation exp>

<nonwith exp>
    ::=   <image exp> | <relation op> | ( <relation exp> )

<image exp>
    ::=   ‼<relvar name> | ‼( < relation exp > ) | ( <image exp> )

<relation op>
    ::=   <relation selector> | <monadic op> | <dyadic op> | <n-adic op>

<relation selector>
    ::=   RELATION [ <heading> ] { <tuple exp commalist> }
        | TABLE_DUM | TABLE_DEE
```

Note: In the first format, a <*heading*> must be specified if the <*tuple exp commalist*> is empty.

```
<heading>
    ::=   { <attribute commalist> }

<attribute>
    ::=   <attribute name> <type name>

<monadic op>
    ::=   <relvar name> | <rename> | <where> | <project>
        | <extend> | <group> | <ungroup> | <tclose>

<rename>
    ::=   <relation exp> RENAME { <renaming commalist> }

<renaming>
    ::=   <attribute name> AS <attribute name>

<where>
    ::=   <relation exp> WHERE <boolean exp>

<project>
    ::=   <relation exp> { [ ALL BUT ] <attribute name commalist> }

<extend>
    ::=   EXTEND <relation exp> : { <attribute assign commalist> }

<attribute assign>
    ::=   <attribute name> := <exp>
```

Note: An alternative form of <*attribute assign*>, syntactically identical to a <*relation assign*> except that the pertinent <*attribute name*> appears in place of the target <*relvar name*> in that <*relation assign*>, is also supported if the attribute in question is relation valued.

```
<group>
    ::=   <relation exp>
            GROUP { [ ALL BUT ] <attribute name commalist> }
                                                AS <attribute name>

<ungroup>
    ::=    <relation exp> UNGROUP <attribute name>

<tclose>
    ::=    TCLOSE ( <relation exp> )

<dyadic op>
    ::=   <relation exp> <dyadic op name> <relation exp>
```

```
<dyadic op name>
    ::=   UNION | D_UNION | XUNION | INTERSECT | MINUS | I_MINUS
        | JOIN | TIMES | MATCHING | NOT MATCHING

<n-adic op>
    ::=   <n-adic op name> [ <heading> ] { <relation exp commalist> }
```

Note: A <*heading*> must be specified if the <*relation exp commalist*> is empty.

```
<n-adic op name>
    ::=   UNION | D_UNION | XUNION | INTERSECT | JOIN | TIMES

<relation comp>
    ::=   <relation exp> <relation comp op> <relation exp>

<relation comp op>
    ::=   = | ≠ | ⊆ | ⊂ | ⊇ | ⊃
```

Assignments

```
<relation assignment>
    ::=   [ WITH ( <name intro commalist> ) : ]
                        <relation assign commalist> ;

<relation assign>
    ::=   <relvar name> := <relation exp>
        | <insert> | <d_insert> | <delete> | <i_delete> | <update>

<insert>
    ::=   INSERT <relvar name> <relation exp>

<d_insert>
    ::=   D_INSERT <relvar name> <relation exp>

<delete>
    ::=   DELETE <relvar name> <relation exp>
        | DELETE <relvar name> [ WHERE <boolean exp> ]

<i_delete>
    ::=   I_DELETE <relvar name> <relation exp>

<update>
    ::=   UPDATE <relvar name> [ WHERE <boolean exp> ] :
                               { <attribute assign commalist> }
```

Appendix E

Summary of Recommendations

Do as I say, not as I do.
—mid 16th century English proverb

In this appendix I present for purposes of reference a brief summary of the recommendations from Chapters 1-12. The page numbers against the various items show where the individual recommendations are discussed in the body of the text.

- Don't use SQL like a simple access method. (*Page 16*)
- Avoid the use of any SQL construct that references physical access paths such as indexes. (*Page 17*)
- Don't use *table* to mean a base table specifically unless your intended meaning is clear from the context. (*Page 24*)
- Don't think of views as if they were somehow different from tables. (*Page 24*)
- Avoid coercions wherever possible. (*Page 62*)
- Ensure that columns with the same name are of the same type. (*Page 62*)
- Avoid type conversions where possible. When they can't be avoided, do them explicitly if you can. (*Page 62*)
- Don't use PAD SPACE. (*Page 64*)
- Avoid possibly nondeterministic expressions. (*Page 65*)
- Don't use "typed tables," reference values, REF types, or any SQL construct related to these features. (*Page 67*)
- If you must talk about nulls, call them nulls and not "null values." (*Page 84*)

- Don't use the comparison operators "<", "<=", ">", and ">=" on rows of degree greater than one. (*Page 88*)

- Use AS specifications whenever necessary (and possible) to give proper column names to columns that otherwise (a) wouldn't have a name at all or (b) would have a name that wasn't unique. (*Pages 97, 179*)

- If two columns represent the same kind of information, give them the same name wherever possible. (*Page 98*)

- Never write code that relies on left to right column positioning. (*Pages 99-100*)

- Avoid duplicates. Make sure you know when SQL eliminates duplicates without you asking it to; when you do have to ask, make sure you know whether it matters if you don't; when it does matter, specify DISTINCT; and never specify ALL. (*Page 120*)

- Avoid nulls: (*Page 125*)

  a. Specify NOT NULL, explicitly or implicitly, for every column in every base table.

  b. Don't use the keyword NULL anywhere other than in the context of such a NOT NULL specification.

  c. Don't use the keyword UNKNOWN in any context whatsoever.

  d. Don't omit the ELSE clause from a CASE expression unless omitting it makes no logical difference.

  e. Don't use NULLIF.

  f. Don't use the keywords OUTER, FULL, LEFT, and RIGHT on JOIN (except, just possibly, in connection with COALESCE).

  g. Don't use union join.

  h. Don't specify PARTIAL or FULL on MATCH.

  i. Don't use MATCH on foreign key constraints.

  j. Don't use IS DISTINCT FROM.

k. Don't use IS TRUE, IS NOT TRUE, IS FALSE, or IS NOT FALSE.

l. Do use COALESCE on every scalar expression that might "evaluate to null" without it.

- Don't use DELETE or UPDATE through a cursor unless you can be certain that integrity constraint problems will never arise in connection with such use. (*Page 143*)
- Avoid operations that are inherently row level (e.g., row level triggers). (*Pages 144, 155-156*)
- Specify target columns explicitly on INSERT. (*Page 148*)
- Don't define as a key some column combination that you know to be reducible. (*Page 150*)
- Use UNIQUE and/or PRIMARY KEY specifications to ensure that every base table has at least one key. (*Page 151*)
- Ensure that foreign key columns have the same name as the corresponding key columns wherever possible. (*Page 154*)
- Don't use triggers if they violate *The Assignment Principle*. (*Page 155*)
- Don't use any operation that violates the relational closure property if you want the result to be amenable to further relational processing. (*Page 177*)
- Use NATURAL JOIN in preference to other methods of formulating a join (but make sure columns with the same name are of the same type). (*Page 186*)
- If you use JOIN ON, make sure columns with the same name are of the same type, and make sure you rename columns appropriately. (*Page 186*)
- If you use JOIN USING, make sure columns with the same name are of the same type. (*Page 187*)
- If you use CROSS JOIN, make sure there aren't any common column names. (*Page 187*)
- For UNION, INTERSECT, and EXCEPT, make sure corresponding columns have the same name and type. (*Page 188*)

- For UNION, INTERSECT, and EXCEPT, always specify CORRESPONDING if possible. If it isn't possible, then make sure columns line up properly. Preferably avoid use of the BY option, unless it makes no difference anyway. (*Page 188*)

- If you use GROUP BY or HAVING, make sure the table you're summarizing is the one you really want to summarize. (*Page 242*)

- Be on the lookout for the possibility that some summarization is being done on an empty set, and use COALESCE wherever necessary. (*Page 242*)

- Where possible, use database constraints to make up for SQL's lack of support for type constraints. (*Page 287*)

- Specify constraints declaratively whenever you can. (*Page 294*)

- Use immediate constraint checking whenever you can. (*Page 301*)

- If checking has to be deferred on some constraint, make sure the check is done before doing any operation that might rely on the constraint being satisfied. (*Page 301*)

- In CREATE VIEW, don't use the option that allows you to specify the view column names immediately following the view name itself. (*Page 325*)

- Specify WITH CASCADED CHECK OPTION on view definitions whenever possible. (*Pages 325, 341*)

- Specify constraints that apply to views (e.g., key constraints) in the form of comments—typically on the view definition. (*Page 332*)

- 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. (*Page 351*)

- Be careful over the use of COUNT; in particular, don't use it where EXISTS would be more logically correct. (*Page 396*)

- Use the techniques described in Chapter 11, at least for formulating "complex" SQL expressions. (*Pages 412ff*)

- Don't use ALL or ANY comparisons. (*Page 433*)

- Don't use "SELECT *" at the outermost level in a cursor definition or view definition, or more generally in any position where the meaning of that "*" might be subject to change. (*Page 444*)

- Favor the use of explicit range variables, especially in "complex" expressions. (*Page 448*)

Well ... after this rather lengthy list of admonitions, it seems only right to close this appendix by reminding you of what in Chapter 1 I called the overriding rule:

You can do what you like, so long as you know what you're doing.

Appendix F

NoSQL and Relational Theory

Just say no!

—Nancy Reagan (1982)

Note: This appendix is based in part on material from two papers that originally appeared in the NoCOUG Journal (the journal of the Northern California Oracle User Group):

- C. J. Date and Hugh Darwen: "No to SQL! No to NoSQL!" (interview by Iggy Fernandez), *NoCOUG Journal 27*, No. 3 (August 2013), *www.nocoug.org/Journal/NoCOUG_Journal_201308.pdf.*

- C. J. Date: "Some Comments on Iggy Fernandez's Paper *The Rise and Fall of the NoSQL Empire*," *NoCOUG Journal 29*, No. 2 (May 2015), *www.nocoug.org/Journal/NoCOUG_Journal_201505.pdf.*

The material is reused here by permission.

I said in Chapter 1 that we'd be concerned in this book with principles, not products, and foundations, not fashion or fads. In this appendix, however, I'm going to go back on that promise a little. As I'm sure you know, in recent years there's been a major upsurge of interest in what are known generically as *NoSQL systems*. And, as so often happens when something suddenly becomes fashionable for some reason, this new development has been surrounded by a fair degree of confusion. In this appendix, I'd like to try to clear up some of that confusion, if I can. In particular, I'd like to try to clarify the relationship—to the extent that any such relationship exists—between the NoSQL movement and what is after all a major topic for the present book, viz., relational theory.

The first confusion arises in connection with the very name "NoSQL." Naïvely, what I thought when I first encountered that name was "Aha! So at last people have begun to realize what a difficult language SQL really is, and they want something better to access their databases with." But, of course, that's not what the NoSQL advocates are saying at all.[1] Their quarrel isn't with the SQL language as such; rather, it's with existing SQL products, which, they claim, all fail to provide adequate performance and/or adequate scalability and/or adequate data

[1] Actually it was, originally. The first use of the term, by Carlo Strozzi in 1995, was as the name of a relational DBMS that was explicitly intended to support something that wasn't SQL (see *www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page*).

availability. Well, they might be right in these claims—in fact, let's assume they are, for the sake of the present discussion at any rate. But of course those deficiencies are deficiencies of the products in question, not deficiencies of the SQL language as such. Indeed, some NoSQL advocates are now saying that "NoSQL" doesn't really mean "No SQL," it means "Not Only SQL," with the implication that the products in question might still support SQL as such in addition to something else (i.e., something that's explicitly not SQL).

A second confusion arises from the fact that the name "NoSQL"doesn't refer to just one product, or one kind of product. Rather, it's an umbrella term, used to refer to at least four different things: viz., document, key-value, column, and graph systems.[2] These different systems can be characterized very briefly as follows:

- *Document systems:* The data in the database consists of a collection of XML or other documents. Documents can be retrieved in their entirety or searched via some document query language (e.g., XPath or Xquery, in the case of XML).

- *Key-value systems:* The database contains nothing but uninterpreted "blobs" (see Chapter 2), each with its own unique ID. The blobs are the values and the IDs are the keys, and the entire collection of key-value pairs is referred to as a dictionary, map, or associative array. All access is by key.

- *Column systems:* The database contains (typically) just one giant table, made up of rows and columns as usual.[3] The rows are identified by key as usual. However, the columns aren't columns in the SQL sense—they don't contain just values, they contain key-value pairs instead, and there's no requirement that every row contain the same kinds of such pairs (i.e., the rows aren't homogeneous). Access is by row key and column key within row key.

- *Graph systems:* Graph systems are somewhat different in kind from the other three. The database consists of nodes and edges, which to a first approximation you can think of as representing entities and relationships, respectively. Typically, a query language is provided that allows the user to enter the graph at some specific node and then traverse edges in the graph to some desired target.

So what is it that unites these systems?—what is it that they have in common, to justify referring to them all by the same blanket label? Mostly, it seems they're united in what they're not. To be specific, they're obviously not conventional SQL systems, as that term is generally

---

[2] The industry tends to call them not systems but "stores," but (as with calling the DBMS a database, a usage I objected to in Chapter 2) this nomenclature seems to confuse the stored data as such with the software system that manages it.

[3] Since this appendix has something of an SQL flavor (or a NoSQL flavor, rather), throughout most of the discussion I'll stick with the SQL terminology of tables, rows, and columns.

understood. Unfortunately, however, I would have to say they're also united in the extensive and demonstrable lack of understanding of the relational model displayed by their promoters. (The remainder of this appendix provides evidence in support of this complaint on my part.) And the first three categories, at least, are also united in their failure to distinguish properly between a model and its implementation (this criticism might not apply to graph systems, or at least not so much or so obviously). And there are some further, somewhat interrelated, points of commonality too (though again these points might not apply to graph systems). To be specific:

- The systems typically reject the ACID properties of transactions, replacing them by a set of so called "BASE properties" instead. BASE is a somewhat contrived acronym, standing for Basic Availability, Soft state, Eventual consistency. I'll have more to say about these properties in the section "Eventual Consistency," later.

- They typically keep several copies of the stored data (*data replication*), typically at several different sites, in order to provide improved availability.

- They also typically employ *functional segmentation* and *sharding* in order to improve availability still further. I'll discuss these issues in the next two sections.

- Because the data structures they support are so simple, they might not require a schema.[4] Of course, without a schema the DBMS has no knowledge of any internal structure the data might possess, in which case it probably won't be able to support anything very sophisticated by way of a query language.

- There's no join support.[5] Thus, it might be necessary to do several separate queries in order to obtain some desired result. Here's what Wikipedia has to say in connection with this point: "NoSQL queries are often faster than traditional SQL queries, so the cost of having to do additional queries [might] be acceptable." Myself, I would have thought this point could be open to dispute.

---

[4] The term *schema* is much used in the database industry, in the SQL world in particular. However, I've avoided it in this book because, to be frank, it isn't always clear exactly what it means. Sometimes it's used as if it were just a synonym for *database design* (either logical or physical). Sometimes it's used to mean a set of definitional statements, such as CREATE TABLE in SQL, that correspond to such a design (but then how do statements like ALTER TABLE and DROP TABLE fit in with that definition?). Also, in the more specific form *relation schema*, it's sometimes used to mean a relvar or relation heading, or a relvar or relation heading together with the constraints that apply to the relvar or relation in question, or sometimes a relvar or relation heading together with just the pertinent key constraint(s). And so on.

[5] The book *Graph Databases*, by Ian Robinson, Jim Webber, and Emil Efrem (O'Reilly, 2013), gives the following justification for this state of affairs: "[Query] execution times increase as the size of tables and the number of joins gow (so called *join pain*)." Well, this is obviously true—but in a well architected system, that growth should be linear (though I grant it's probably not linear in today's mainstream SQL products). See my book, mentioned in Appendix G, *Go Faster! The TransRelational™ Approach to DBMS Implementation*.

Aside: Before I go on to elaborate on some of the foregoing ideas, perhaps I should offer some explicit examples of quotes to back up my claim that the promoters of these systems seem not to understand the relational model properly. Here are a few (sources omitted to protect the guilty):

- "Relational databases lack relationships."
- "Relationships do exist in the vernacular of relational databases, but only as a means of joining tables."
- "Join tables add accidental complexity; they mix business data with foreign key metadata."
- "[This product] goes a step beyond even relational databases by lifting up the connections to be first-class citizens. It does this by allowing the relationships ... to have an arbitrary number of properties."
- "The relational database exposes a tabular view of the world. But our [application] is filled with hierarchical and connected data."

And so on. *End of aside.*

FUNCTIONAL SEGMENTATION

The basic idea of functional segmentation, or just *segmentation* for short, is very simple: The overall business enterprise is divided up into a set of more or less independent functional pieces or *segments*, and then all of the data for a given segment is stored at its own site, in its own database. For example, a computer manufacturer might have a hardware products segment, a software products segment, an education and training segment, a publications segment, and so on, each with its own database. The big advantage of such a scheme is that it avoids having a single point of failure in the system, so that a user of, say, the education database will be unaffected if the publications database goes offline.

Of course, segmentation as just described isn't exactly a novel concept (many years ago we used to call those separate databases *subject databases*).[6] But more to the point, let me now observe that—as I'm sure you've already realized—there's no conflict whatsoever between such segmentation and relational theory. Segmentation is merely a physical database design decision, and it could perfectly well be applied in the relational context if it were thought desirable to do so. In other words, there's absolutely no reason why a desire for segmentation should require the rejection of any relational concepts (or SQL concepts, come to that).

---

[6] They're also highly reminiscent of what in Appendix A I called *application specific data stores*.

Note: The complete set of subject databases for the enterprise could be regarded as a single distributed database. However, distributed transactions can be expensive, and for that reason they aren't allowed in NoSQL systems. As a consequence, there's a possibility that those separate subject databases might sometimes become at least temporarily inconsistent—for example, a document in the publications database might contain a reference to a product the hardware division has stopped making. I'll return to this point in the section "Eventual Consistency" below.

SHARDING

Sharding can be regarded as an extended form of what's more usually known as *horizontal decomposition*, as this latter term is used in conventional database design contexts. Suppose the computer manufacturer's education database from the previous section contains tables that look like this (in outline):

```
COURSE    ( CNO , TITLE , ... )              KEY ( CNO )
OFFERING  ( CNO , ONO , LOCATION , ... )     KEY ( CNO , ONO )
TEACHER   ( CNO , ONO , TNO , ... )          KEY ( CNO , ONO , TNO )
STUDENT   ( CNO , ONO , SNO , GRADE , ... )  KEY ( CNO , ONO , SNO )
```

Note the hierarchic structure here: There's a one to many relationship from courses to course offerings, another from such offerings to teachers, and another from such offerings to students.[7] In particular, each of the four tables has {CNO} (course number) as its key or part of its key. Thus, the tables can each be partitioned on the basis of course numbers, such that the rows from all four tables with CNO in the range 1-100 (say) form one "shard," the rows with CNO in the range 101-200 form another, and so on. Each shard is then kept in its own database, stored at its own site.

Observe now that—once again as I'm sure you've realized—there's no conflict here with relational theory. Like segmentation, sharding is just a physical database design decision, and it can perfectly well be applied in the relational context if it were thought desirable to do so. Thus, there's no reason why a desire to carry out such sharding should require the rejection of any relational concepts (or SQL concepts, come to that). Note too that the complete set of shards can be regarded as a single distributed database, though as previously noted distributed transactions aren't allowed in NoSQL systems.

[7] It's worth noting that functional segmentation as discussed in the previous section will tend to produce such hierarchic structures naturally, and even if it doesn't it can be forced to.

EVENTUAL CONSISTENCY

Now let me get back to that BASE acronym. B and A stand for basic availability, meaning, loosely, that "most of the data is available most of the time." S stands for soft state, apparently meaning that the stored data doesn't always have to be consistent (though exactly how it comes to have that meaning I really couldn't say). And E stands for eventual consistency, meaning that inconsistencies if any will be resolved at some future time ("eventually"), not necessarily at the time of the pertinent update, nor even necessarily at the end of the transaction. It's this last item, eventual consistency, that I now want to discuss.

Let me begin by saying that I have no problem with the broad intent of this notion, nor with the business requirement that the NoSQL supporters are aiming at in this connection. However, I do think I have a rather different perspective on what's really going on here—different, that is, from the perspective usually presented when "eventual consistency" is discussed in the NoSQL literature—and it's that different perspective that I'd now like to explain.

- To say a database (be it distributed or otherwise) is consistent merely means, formally speaking, that the database in question conforms to all declared integrity constraints. Now, it's crucially important that databases *always* be consistent in this sense; indeed, a database that's not consistent in this sense, at some particular time, is like a logical system that contains a contradiction. Well, actually, that's exactly what it is—a logical system that contains a contradiction. And in a logical system that contains a contradiction, you can prove anything; for example, you can prove that 1 = 0. (In fact, as we saw in Chapter 8, you can *prove* that you can prove that 1 = 0 in such a system.) What this means in database terms is that if the database is inconsistent in the foregoing sense, you can never trust the answers you get to queries (they may be false, they may be true, and you have no way in general of knowing which they are); all bets are off. As far as declared constraints are concerned, in other words, the system simply *must* do the checking whenever a pertinent update occurs; there's no alternative, because (to say it again) not to do that checking is to risk having a database for which all bets are off. In other words, immediate constraint checking is logically required.

- But consistency in the foregoing formal sense isn't necessarily the same thing as consistency as conventionally (and informally) understood—meaning consistency as understood in conventional real world terms, outside the world of databases.[8] Suppose there are two items *A* and *B* in the database that, in the real world, we believe should have the same value. They might, for example, both be the selling price for some given commodity, stored twice (perhaps at different sites) because data replication is being used to improve availability. If *A* and *B* in fact have different values at some given time, we

[8] I explained in Chapter 8 how the term *isolation* as used in the database context (in particular, in the context of the ACID properties of transactions) doesn't mean quite the same thing as it does in ordinary English. What I'm saying now is that an analogous remark applies to consistency as well, mutatis mutandis.

might certainly say, informally, that there's an inconsistency in the stored data at that time. But that "inconsistency" is an inconsistency as far as the system is concerned *only if the system has been told that A and B are supposed to be equal*—i.e., only if "$A = B$" has been declared as a formal integrity constraint. If it hasn't, then (a) the fact that $A \neq B$ at some time doesn't in itself constitute a consistency violation as far as the system is concerned, and (b) importantly, the system will nowhere rely on an assumption that *A* and *B* are equal.

- Thus, if all we want is for *A* and *B* to be equal "eventually"—i.e., if we're content for that requirement to be handled in the application layer, outside the DBMS—all we have to do as far as the database system is concerned is omit any declaration of "$A = B$" as a formal constraint. No problem, and in particular no violation of relational theory.

THE FERNANDEZ INTERVIEW

I've claimed in this appendix so far that three of what are generally regarded as the defining characteristics of NoSQL systems—segmentation, sharding, and eventual consistency (or replication together with eventual consistency, rather)—are entirely compatible with relational theory. Thus, it seems appropriate to close the discussion with a lightly edited version of the pertinent portion of the interview conducted by Iggy Fernandez with Hugh Darwen and myself, where some of these matters are examined in a little more depth. *Note:* The interview was originally published in *NoCOUG Journal 27*, No. 3 (August 2013), *www.nocoug.org/Journal/NoCOUG_ Journal_201308.pdf*. What follows is essentially the whole of the "No to NoSQL!" section of that interview. My thanks to Iggy Fernandez and NoCOUG for allowing me to republish this material in the present book.

Fernandez: The archetypal NoSQL product is Dynamo from Amazon.com. The 2007 ACM paper by Amazon.com states: *"Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers …There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key-only interface to meet the requirements of these applications."* The Dynamo paper is where the popular claim originated that NoSQL products are faster, more scalable, and more available than relational products in certain clearly delineated scenarios such as online shopping carts. But is there any merit to the claim at all?

Date: First off, let me make it very clear that I know almost nothing about Dynamo as such. But if the statement is correct that it provides "a simple primary-key-only interface to meet the requirements of [certain rather simple] applications"—well, fine. I have no problem with that. If there's a class of applications that (a) are important for some pragmatic reason and (b) require only a limited subset of the system's full functionality, then I think it's perfectly reasonable for the system to provide a special interface tailored to just those requirements. Why, that's exactly what IBM did, with its Fast Path option in IMS! In a relational system, that special interface would support a carefully chosen subset of the full relational interface, and the implementation would be able to take advantage of the fact that the interface is circumscribed in just such a way. It might be able to make use of its own special stored data formats as well. And—just to spell the point out—I see no reason why the provision of that special interface and those special stored data formats should have any negative effect at all on users who want to use the regular "full function" relational interface.

That said, if there's a suggestion that Amazon's various disaster scenarios, regarding tornados and the rest, are somehow more of a problem for relational systems than they are for nonrelational ones, then of course I reject that suggestion 100 percent. As so often, I strongly suspect that what's going on here is some kind of confusion between what truly relational systems ought to be capable of and what today's mainstream SQL products can actually do. If today's SQL products fail to meet Amazon's requirements, well, that might be a valid criticism of those products—but it's not a valid criticism of relational systems in general.

To sum up: I do think we should discard SQL, as quickly as we can, and replace it by something better. Unfortunately, however, most of the people who currently want to discard SQL (or, at least, those who are most vocal about doing so) seem to want to do so for the wrong reasons. And there's a strong likelihood that they'll replace it, not by something better, but by something worse.

Darwen: So Amazon uses a Dynamo key value to access a giant blob whose structure is understood only by the applications. In that case they are happy to forgo all of those advantages given by Codd in 1974, not all of which are properly delivered by SQL products in any case.[9] If the people at Amazon are satisfied that Dynamo provides everything they need, and they feel they have good reason to reject the use of any SQL products, then who are we to argue? The indictment seems to be of SQL products, not relational databases.

It's easy to understand why the mainstream SQL products might be too "heavy" for Amazon's needs. Those products have become extremely cluttered up with all sorts of features that would be of little or no use in the Amazon scenario: baroque support for user defined data types, pointers (in the form of REF values), BLOBs and CLOBs, subtables and supertables, sequence generators, datalinks, locators, system versioned tables, and on and on.

---

[9] The advantages Darwen is referring to here are basically what Codd claimed as his objectives for introducing the relational model (see the section "Objectives of the Relational Model" in Appendix A).

Rel (*dbappbuilder.sourceforge.net/Rel.html*) is an implementation, by Dave Voorhis of the University of Derby, U.K., of the relational language **Tutorial D** that we (Chris and I) devised for teaching and illustrative purposes. *Rel* is a very simple DBMS that meets all of the criteria for being fully relational. If *Rel*, or one of the other prototype implementations of *The Third Manifesto*, were dressed up sufficiently for commercial purposes—including in particular the performance enhancements to be obtained by implementation of established optimization techniques and sophisticated storage structures—then it would be interesting to see if Amazon still preferred Dynamo.

All of that said, we admit that full scalability might be hard to achieve with a fully relational system. That's because we require such a system to support expressions of arbitrary complexity for deriving whatever results might be desired from the database and for expressing whatever integrity constraints might be needed. Let's suppose there are extreme cases where what runs in acceptable time with small databases is simply not feasible with large ones. Such cases would militate against the declaration of certain integrity constraints (constraints that current SQL products don't even support anyway). They also militate against the use of certain queries, in an OLTP context, that SQL products *do* support, but that problem can of course be avoided simply by not doing such queries. We conjecture that such cases would be unlikely to arise in Amazon's shopping scenario. In any case, if they would still prefer Dynamo to a putative souped up *Rel*, then so be it. If the benefits of a relational system aren't all needed in a particular situation, and provision of a more tailored solution in that situation is found to be cost effective, then who could argue that Amazon would have made a bad choice?

By the way, the bogey of scalability is sometimes advanced as justification for failing to provide any solution at all. A pertinent example is the lack of full support for integrity constraints in SQL systems (where something close to full support could be achieved by implementing the ISO standard's CREATE ASSERTION statement, for example). But some databases are quite small and subject to quite infrequent updates. I use *Rel* for several small databases that I maintain for domestic and hobby purposes on my home computer. I have benefited significantly from the ability to define constraints that would be impossible to define in SQL without CREATE ASSERTION.[10] Without those constraints, certain errors by me would have gone undetected, resulting in incorrect databases. I'm typically dealing with relations consisting of a few hundred tuples, and in some cases I'm updating no more than once per month. It seems unfair that small organizations, which have little clout with the SQL vendors, can't also enjoy the benefits of such solutions, just because those solutions might not be practical for large organizations (with deep pockets, therefore listened to by the DBMS vendors) using OLTP on enormous databases. In this connection, one SQL DBMS implementer once told me privately that he agreed with me in principle but tellingly added that supporting CREATE ASSERTION would be very difficult in his product and lack of scalability for some kinds of constraints would give him a good get-out clause!

---

[10] Or equivalent functionality, which as we saw in Chapter 8 is available in the SQL standard in the form of base table constraints.

Fernandez: Another breed of NoSQL products that has gained considerable commercial momentum is "graph systems" such as Neo4J. In "Normalized Database Structure: A Brief Tutorial," Codd carved out a special exception for such products: *"It may be argued that in some applications the problems have an immediate natural formulation in terms of networks. This is true of some applications, such as studies of transportation networks, power-line networks, computer design, and the like. We shall call these network applications … Except in network applications, links should not be employed in the user's data model."* Since the problems addressed by these products (e.g., shortest path) have no solution in relational calculus, do these products have a legitimate case to be nonrelational?

Date: Several points here. Yes, Codd did "carve out a special exception" for what he called network applications. But I'm not sure he was right to do so. As a simple counterexample, a company's organization chart has "an immediate natural formulation in terms of networks" (in fact, often—though not always—in terms of a hierarchy, which is a simple special case). But it doesn't follow that we need a network DBMS (i.e., one that exposes "links" or pointers to the user) in order to deal with corporate organizations, and of course we don't.

Second, I'd like to point out that in the paper you reference, Codd also said this: "[Users often have] occasion to require tables to be printed out or displayed. What could be a simpler, more universally needed, and more universally understood data structure than a table? Why not permit ... users to view all the data ... in a tabular way?"

Third, any graph can always be represented—quite succinctly, in fact—in relational form. As for "shortest path" and other such problems, please note that the relational model is only a minimum requirement. Even if you're right when you suggest that the shortest path problem can't be formulated in relational calculus—I presume you're referring to the fact that the relational calculus as originally defined had no support for the famous "ancestral" problem—well, that's not to say it never will have such support. In fact, a great deal of research has been done on adding such support (and implementing it efficiently, too).

Fourth, let's agree for the sake of the argument that there are some problems that graph-based DBMSs can solve better than relational ones. I don't have an issue with that. My position is this: We know the class of problems for which relational systems are suited is very large—but it's not necessarily universal. But I very much doubt whether any other approach is universal either. So my objection isn't to using, e.g., graph-based DBMSs to solve problems that they solve well; rather, my objection is to attempts to solve by nonrelational means problems that can reasonably, perhaps better, be solved by relational means. In other words, graph-based DBMSs (for example) might well have a role to play, but that role is *not* to take over the entire database world. To repeat something I've said elsewhere: I've never seen a proposal for "taking over the database world"—i.e., for replacing the relational model—in which the person doing the proposing really understood the relational model. Surely, if you want to claim that Technology *A* is no good and needs to be replaced by Technology *B*, then it's incumbent on you to understand Technology *A* in the first place? And, more specifically, to demonstrate that

Technology *B* solves not only all of the problems that Technology *A* does, but also some problem that Technology *A* doesn't?

Darwen: Well, graph DBMSs and the like simply are nonrelational. Of course we don't suggest that *all* databases should be relational—only that all general purpose ones should be. But if you were to ask if it's legitimate to claim that solutions to problems like "shortest path" are inherently unobtainable with a relational system, then the answer is an emphatic "No!" As Chris has effectively already said, having no solution in relational calculus doesn't mean it's impossible for a relational DBMS to provide the necessary operators. For example, **Tutorial D** already includes an operator, TCLOSE, for deriving a relation representing the transitive closure of its operand, a recursive relation. And even SQL includes support—rather elaborate support, in fact—for recursive table expressions in general. Any operator that's closed over relations is admissible in a relational database language.

Now, the proponents of graph databases might wish to argue that their systems can provide much faster solutions to such problems than could ever be obtained by implementations of suitable relational operators. But suppose the graph DBMS were the *engine* of a relational DBMS, such that a relational expression is mapped under the covers to an equivalent expression or procedure in the graph DBMS's language. Wouldn't we then see the relational DBMS performing pretty much as well—or as badly!—as the graph DBMS on its own? And wouldn't its user then be receiving all the benefits claimed for relational DBMSs in general *in addition to* those claimed for graph DBMSs in particular? It's interesting to see in this connection that some of the DBMSs listed in the Wikipedia article "Graph database," notably those available from Oracle, use some form of SQL as their query language.

Fernandez: Another breed of NoSQL products that has gained considerable commercial momentum is the so-called "Big Data" products like Hadoop that aim to process nontransactional data outside the transactional DBMS. It was apparent that the glaring drawback of this class of NoSQL products was the absence of SQL, and so there has been a rush to provide SQL-like functionality in this space, with Impala from Cloudera leading the way. Which leads to the question: Is it kosher to decouple relational algebra and relational calculus from the DBMS as Impala has done?

Date: Before I became "a database person," I was a languages person. I worked for several years at IBM Hursley (in the U.K.), which in those days was the home of PL/I. (Of course, you might never have heard of PL/I, and I'll be the first to admit that as a language it looks a little antiquated now. But it was a big deal at the time—and a big revenue earner for IBM, I might add.) So when I first learned about Ted Codd's relational model, I wanted to add relations and relational operators to PL/I—in order that PL/I would be able to operate on data in a relational database, of course, but not only for that reason; I always thought it would be useful to have "local" relations, meaning ones that weren't in the database, and to be able to operate on those

relations by means of joins and unions and so on. So if that's what you mean by "decoupling relational algebra and relational calculus from the DBMS," then I'm all for it.

But you touch on something else here: "Big Data." Sorry if I'm beginning to sound like a broken record—I guess that metaphor is pretty antiquated too!—but I see no reason why a relational system shouldn't be able to handle "Big Data" perfectly well. Data size is, of course, an implementation concern, not a model concern. The relational model is deliberately silent on all matters of physical implementation. Just because the implementation has to deal with enormous volumes of data, that's no reason, as far as I can see, why the user interface has to be anything other than relational.

Darwen: My observations on graph databases seem applicable here too. Couldn't Impala be thought of as the DBMS and Hadoop as Impala's database engine? Well, up to a point, perhaps, but if Impala users have to use Hadoop itself for certain purposes (perhaps database definition? updating? constraint enforcement?), then we could hardly call Impala a fully relational DBMS, even if its language, as far as it goes, were in keeping with the relational model. In any case, there have been many examples over the years of the need being perceived for a "decoupled" relational front end to a nonrelational system. For example, in the early 1980s a group in IBM (U.K.) developed an SQL front end to IBM's ancient and still running hierarchical system IMS. There can be no objection to such products; on the contrary, if the front end were fully relational (as opposed to SQL), we would encourage them to be provided wherever the need arises.

Appendix G

Suggestions

for Further Reading

A man ought to read just as inclination leads him;
for what he reads as a task will do him little good.

—Samuel Johnson:
quoted in Life of Johnson, *by James Boswell* (1791)

As the title says, this appendix gives some suggestions for further reading. The publications are listed in alphabetical order by author and chronological order within author.[1] *Note:* This book isn't concerned with specific SQL products, and I therefore don't mention any product oriented publications in this appendix. But many such publications exist, and I'm sure you'll want to refer to one or more of them as well if you want to apply the ideas discussed in the present book to some individual project or 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 this book—viz., that SQL is complicated, confusing, and error prone. Here's what Chaudhuri and Weikum have to say on the matter: *"SQL is painful.* A big headache that comes with a database system is the SQL language. It is the union of all conceivable features (many of which are rarely used or should be discouraged 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

[1] I apologize for the number of times my own name appears as either author or coauthor in this list, but given the nature of the material, such a state of affairs is—if you'll pardon me for saying so—more or less inevitable.

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."

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

   This is the paper that first introduced the SQL language (or SEQUEL—Structured English QUEry Language—as it was originally called; the name was subsequently changed for legal reasons). There are some interesting differences between SEQUEL as described in this paper and SQL as generally understood today.[2] Here are some of them:

   - There were no nulls.
   - Although the SELECT clause was supported, the "SELECT *" form didn't exist. Thus, for example, to get all suppliers in London, you just wrote S WHERE CITY = 'London'—and to get all suppliers, you just wrote S.
   - Duplicates were eliminated by default (though not in "set functions").
   - The FROM clause always contained exactly one table name. In other words, what I called in Chapter 6 "the only [form of join] supported in SQL as originally defined" wasn't supported at all in *SEQUEL* as originally defined!
   - The right comparand (but not the left) in a comparison in a WHERE clause was allowed to be a subquery, in which case the comparison was in fact an ANY or ALL comparison. (Note, however, that the term *subquery* didn't exist—the construct was called a *mapping* instead.) ANY was the default, and could only be specified implicitly. IN syntax as such was not supported; "=" (meaning, by default, "=ANY") was used instead.
   - Set comparison operators ("⊆" etc.) were supported.
   - There was no GROUP BY clause as such; instead, GROUP BY could be specified as an option on the FROM clause.

---

[2] Of course, SEQUEL hadn't actually been implemented when this paper was written, nor so far as I know had it even been formally defined. If it had been (either defined or implemented, that is), some of the items in the subsequent bullet list would certainly have had to change, and some of them probably wouldn't have survived—a state of affairs that might explain some of the differences here referred to.

- There was no HAVING clause; "set functions" could be invoked in the WHERE clause instead.

- There were no correlation names. Instead, "blocks" (apparently another term for SELECT – FROM – WHERE expressions, or in other words "mappings" in the sense indicated above) could be labeled, and block labels could be used as dot qualifiers.

- Expressions such as QTY / AVG(QTY)—i.e., expressions involving "set function" invocations in addition to simple column references and the like—were legal in the SELECT clause, and presumably in the WHERE clause also.

- "Blocks" or "mappings" could be combined by means of intersection, union, and difference (and these operators were denoted by conventional mathematical symbols instead of English keywords).

The paper also discusses several perceived differences between SEQUEL and the relational calculus, claiming in every case an advantage for SEQUEL over the calculus. However, the differences and claims in question don't really stand up to careful analysis.

3. 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). *Note:* The first of these papers was reprinted in *ACM SIGMOD Record 38*, No. 1 (March 2009); the second was reprinted in *Milestones of Research—Selected Papers 1958-1982 (CACM 25th Anniversary Issue)*, *CACM 26*, No. 1 (January 1983) and elsewhere.

   Codd's first two papers on the relational model. The 1969 paper was, of course, the very first; 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). The 1970 paper was the first widely available paper on the subject, by Codd or anyone else. In fact, 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. *Note:* For some detailed analysis of both papers, see references [12] and [37]. See also reference [9].

4. 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.

5. 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 include 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.

6. 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 dependencies (FDs) satisfied by the relation *r* denoted by an arbitrary relational expression can be inferred from those holding for the relvar(s) referenced 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 of the present book.

7. Hugh Darwen: *An Introduction to Relational Database Theory*. Frederiksberg, Denmark: Ventus Publishing (2010), available as a free download from *http://bookboon.com*.

   From the preface: "This book introduces you to the theory of relational databases, focusing on the application of that theory to the design of computer languages that properly embrace it. The book is intended for those studying relational databases as part of a degree course in Information Technology (IT)."

8. Hugh Darwen: *SQL: A Comparative Survey*. Frederiksberg, Denmark: Ventus Publishing (2012), available as a free download from *http://bookboon.com*.

   This book is a companion to reference [7], showing how the coding examples in that book can be expressed in SQL. From the preface: "SQL's many deviations from relational database theory are thus exposed and their consequences discussed. Drawing on the author's long experience as a

member of the committee responsible for production of the ISO SQL standard, the book includes copious Historical Notes showing how SQL has evolved from its very beginnings in the 1970s." As you can see, references [7] and [8] between them cover territory somewhat similar to that covered by the present book, albeit from a rather different perspective.

9. Hugh Darwen: "Why Are There No Relational DBMSs?", *www.thethirdmanifesto.com* (2015).

   Darwen's own abstract for this paper reads somewhat as follows: "I describe the circumstances in which I obtained, in 1978, a good answer to a burning question: How can Codd's relational model be properly embraced by a computer language? Considering that the answer to that question was publicly available in 1975, I wonder why it all went wrong and suggest some possible reasons." *Note:* The answer that Darwen refers to here is documented in reference [52] in the present appendix.

10. Hugh Darwen and C. J. Date: "Textbook Treatments of Relational Algebra," *www.thethirdmanifesto.com* (2015).

    This reference isn't so much a technical paper as such, in the usual sense of that term; rather, it consists essentially of an annotated bibliography, listing some 15 database textbooks and offering opinions on the quality of the treatment of the relational model found in those books, with special attention to the treatment of relational algebra in particular.

11. C. J. Date: "Fifty Ways to Quote Your Query," *www.dbpd.com* (July 1998).

    A discussion of redundancy in the SQL language.

12. C. J. Date:*The Database Relational Model: A Retrospective Review and Analysis*. Reading, Mass.: Addison-Wesley (2001).

    From the preface (but lightly edited here): "This book consists in essence of a series of twelve articles, originally published in the print and online portions of the Miller Freeman magazine *Intelligent Enterprise* (*Vol. 1*, Nos. 1-3, and *Vol. 2*, Nos. 1-9, October 1998 onward. The overall title for the series was *30 Years of Relational*, on the grounds that the articles were written, in part, to celebrate the relational model's 30th birthday. The intent was to provide a historical account and impartial analysis of E. F. Codd's (huge!) contribution to the field of database technology. Codd's relational model, represented by a startlingly novel series of research papers appearing over the period 1969-1979, was a revolution at the time, albeit one that was desperately needed. Now, however, it seems that—despite the fact that the entire multibillion dollar database industry

is founded on Codd's original ideas—those ideas are in danger of being ignored or forgotten (or, at best, being paid mere lip service to). Certainly we can observe many examples today of those ideas being flouted in (among other things) database products, database designs, and database applications. It thus seems appropriate to take another look at Codd's original papers, with a view to assessing their true significance and restating (and reinforcing) their message for a new generation of database professionals." *Note:* The papers by Codd considered in this book include "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks" and "A Relational Model of Data for Large Shared Data Banks" (see reference [3]); "Relational Completeness of Data Base Sublanguages" (reference [4]); and several others not mentioned elsewhere in this appendix.

13. C. J. Date: *Go Faster! The TransRelational™ Approach to DBMS Implementation*. Frederiksberg, Denmark: Ventus Publishing (2002, 2011), available as a free download from *http://bookboon.com*.

    A detailed description of The TransRelational™ Model, a novel and radically different implementation technology mentioned in passing at various points in the present book (see, e.g., Appendixes A and F). A short and very incomplete introduction to that technology can also be found in Appendix A of reference [14]. *Note:* By "radically different" here, I mean radically different from the "direct image" style of implementation mentioned in passing in Chapter 1 and found in just about every mainstream SQL product today.

14. 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 [49], [60], and [61].

15. C. J. Date: "Edgar F. Codd (August 23rd, 1923 - April 18th, 2003): A Tribute and Personal Memoir," in reference [23].

16. C. J. Date: "Double Trouble, Double Trouble," in reference [23].

    An extensive and detailed treatment of the problems caused by duplicates. The discussion of duplicates in Chapter 4 of the present book is based in part on an example from this paper.

17. C. J. Date: "What First Normal Form Really Means," in reference [23].

    First normal form has been the subject of much misunderstanding over the years. This paper is an attempt to set the record straight—even to be definitive, as far as possible. The crux of the argument, as indicated in Chapter 2 of the present book, is that the concept of *data value atomicity* (the basis of first normal form as originally defined) has no absolute meaning.

18. C. J. Date: "A Sweet Disorder," in reference [23].

    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 many seem to think. (Many of the recommendations in the present book have to do with techniques for behaving as if the state of affairs in question didn't exist after all.)

19. 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 reference [23].

    The titles say it all.

20. C. J. Date: "Two Remarks on SQL's UNION," in reference [23].

    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) and are caused by its support for (a) coercions and (b) duplicate rows.

21. C. J. Date: "A Cure for Madness," in reference [23].

    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 always 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. *Note:* For simplicity I

choose to ignore the fact that the standard would actually require the subquery in the FROM clause in the first of the foregoing expressions to be accompanied by an AS specification.

22. C. J. Date: "Why Three- and Four-Valued Logic Don't Work," in reference [23].

    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.

23. C. J. Date: *Date on Database: Writings 2000-2006*. Berkeley, Calif.: Apress (2006).

24. C. J. Date: "The Closed World Assumption," in reference [26].

    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 that 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).

25. C. J. Date: "The Theory of Bags: An Investigative Tutorial," in reference [26].

    Among other things, this paper discusses what happens to operators such as join and union when their operands are bags (as they are in SQL, loosely speaking) instead of sets.

26. C. J. Date: *Logic and Databases: The Roots of Relational Theory*. Victoria, BC: Trafford Publishing (2007).

27. C. J. Date: "Inclusion Dependencies and Foreign Keys," in reference [45].

    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 detailed discussion of the logical differences between foreign keys and pointers. (As noted in passing in Chapter 2 of the present book, some writers have claimed that foreign keys are nothing more than traditional pointers in sheep's clothing, but such is not the case.)

28. C. J. Date: "Image Relations," in reference [45].

29. C. J. Date: "*N*-adic vs. Dyadic Operators: An Investigation," in reference [45].

Tutorial D supports *n*-adic versions of several relational operators—union, join, and so on—that are usually considered to be dyadic operators merely. This paper examines the twin questions of (a) what makes it possible to define an *n*-adic version of some dyadic operator and (b) how such *n*-adic versions can sensibly be defined.

30. C. J. Date: "A Remark on Prenex Normal Form," in reference [45].

31. C. J. Date: "Is SQL's Three-Valued Logic Truth Functionally Complete?", in reference [45].

    Among other things, this paper includes a comprehensive description of SQL's support for nulls and three-valued logic.

32. C. J. Date: "A Brief History of the Relational Divide Operator", in reference [45].

33. C. J. Date: *Database Design and Relational Theory: Normal Forms and All That Jazz*. Sebastopol, Calif.: O'Reilly Media Inc. (2012).

    Design theory is the scientific foundation for database design, just as the relational model is the scientific foundation for database technology in general. This book, a companion to the present book, is a tutorial on database design theory (normalization, orthogonality, and related matters) for database professionals.

34. C. J. Date: *View Updating and Relational Theory: Solving the View Update Problem*. Sebastopol, Calif.: O'Reilly Media Inc. (2013).

    The problem of (a) updating base relvars appropriately in order to support updates on views is, abstractly, the same problem as (b) the problem of updating stored data appropriately in order to support updates on base relvars. They just show up at different points in the overall system architecture, that's all. It follows that we must solve this problem, for otherwise we have to give up on the goal of data independence. (Note, therefore, that logical and physical data independence are really the same problem, too; they differ only in that they too show up at different points in the overall architecture.) This book argues that, contrary to popular belief, views should always be updatable, modulo only integrity constraint violations. More particularly, it elaborates on the idea, briefly discussed in Chapter 9, that a fruitful way to think about view updating in general is to consider what would happen if the view in question were defined as a base relvar instead, living alongside (as it were) the base relvar(s) in terms of which it's defined, with constraints interrelating the two.

35. *C. J. Date:* "Some Comments on Iggy Fernandez's Paper *The Rise and Fall of the NoSQL Empire*," *NoCOUG Journal 29*, No. 2 (May 2015), *www.nocoug.org/Journal/NoCOUG_Journal_201505.pdf.*

    See reference [51].

36. C. J. Date: *Relational Theory for Computer Professionals: What Relational Databases Are Really All About*. Sebastopol, Calif.: O'Reilly Media Inc. (2013).

    The book you're looking at right now, *SQL and Relational Theory: How to Write Accurate SQL Code*, discusses relational theory on the assumption that you're already a database professional. This reference [36], by contrast, covers much of the same territory without making that same assumption. It also provides a brief introduction to transaction management and database design theory, matters not discussed in the present book.

37. C. J. Date: "Codd's First Relational Papers: A Critical Analysis," *www.thethirdmanifesto.com* (2015).

    Reference [12] analyzes Codd's early writings with a view to highlighting all the many, many things he got right. By contrast, the present reference focuses on some of the things he at least arguably got wrong (in his first two papers [3], that is).

38. C. J. Date: *The **New** Relational Database Dictionary*. Sebastopol, Calif.: O'Reilly Media Inc. (2015).

    Many of the definitions given in the body of the present book are based on ones in this reference.

39. 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. Numerous features have been added to the standard since that time (including the so called "object/relational" features—see reference [14]), but they're mostly irrelevant so far as the goal of using SQL relationally is concerned. In my not unbiased opinion, therefore, the book is still a reasonably 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.

40. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Boston, Mass.: Addison-Wesley (2007).

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). See also references [42] and [45].

41. C. J. Date and Hugh Darwen: "Multiple Assignment," in reference [23].

42. C. J. Date and Hugh Darwen: "The Third Manifesto," in reference [45].

This paper (Chapter 1 of reference [45]) presents the very latest version of *The Third Manifesto*. It consists for the most part of a revised version of the pertinent chapter from reference [40].

43. C. J. Date and Hugh Darwen: "**Tutorial D**," in reference [45].

This paper provides a comprehensive description of the most recent version of **Tutorial D** (which is essentially the version used in examples in the present book). *Note:* The website *www.thethirdmanifesto.com* gives information regarding a variety of existing **Tutorial D** implementations, as well as other projects related to proposals of *The Third Manifesto*.

44. C. J. Date and Hugh Darwen: "Toward an Industrial Strength Dialect of **Tutorial D,**" in reference [45].

A proposal for upgrading **Tutorial D** to make it more suitable for commercial implementation. Certain of the ideas from this proposal (including in particular image relation and foreign key support) have been assumed in the body of the present book.

45. C. J. Date and Hugh Darwen: *Database Explorations: Essays on The Third Manifesto and Related Matters*. Bloomington, Ind.: Trafford Publishing (2010). Also online at *www.thethirdmanifesto.com*.

46. C. J. Date and Hugh Darwen: "No to SQL! No to NoSQL!" (interview by Iggy Fernandez), *NoCOUG Journal 27*, No. 3 (August 2013), *www.nocoug.org/Journal/NoCOUG_Journal_201308.pdf*.

See Appendix F.

47. C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Time and Relational Theory: Temporal Databases in the Relational Model and SQL*. Waltham, Mass.: Morgan Kaufmann (2003).

This book is a replacement for an earlier book by the same authors, viz., *Temporal Data and the Relational Model* (Morgan Kaufmann, 2003). A very brief indication of what it covers can be found in Appendix A of the present book.

48. C. J. Date and David McGoveran: "Why Relational DBMS Logic Must Not Be Many-Valued," in reference [26].

This paper presents a series of logical arguments in support of the position that database languages should be based (like the relational model, but unlike SQL) on two-valued logic.

49. Ramez Elmasri and Shamkant Navathe: *Fundamentals of Database Systems* (6th edition). Boston, Mass.: Addison-Wesley (2011).

50. 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 sound 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."

51. Iggy Fernandez: "The Rise and Fall of the NoSQL Empire," *NoCOUG Journal 29*, No. 1 (February 2015), *www.nocoug.org/Journal/NoCOUG_Journal_201502.pdf*. *Note:* There was a mistake in this paper as originally printed—the title was given as "The Rise and Fall of the *SQL* Empire" (italics added).

    A good analysis of what NoSQL is all about. Appendix F was influenced by this paper.

52. 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. See also references [9] and [10].

53. G. D. Held, M. R. Stonebraker, and E. Wong: "INGRES—A Relational Data Base System," Proc. NCC *44,* Anaheim, Calif. Montvale, N.J.: AFIPS Press (May 1975).

    There were two major relational prototypes under development in the mid to late 1970s—System R at IBM, and Ingres (originally INGRES, all uppercase) at the University of California at Berkeley. Unlike System R, Ingres was not originally an SQL system; instead, it supported a language called QUEL ("Query Language"), which was based on relational calculus and in many ways was technically superior to SQL. This paper, which was the first to describe the Ingres prototype, includes a preliminary definition of QUEL.

54. Jim Gray and Andreas Reuter: *Transaction Processing: Concepts and Techniques*. San Mateo, Calif.: Morgan Kaufmann (1993).

The standard text on transaction management.

55. 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 of the present book to help with the formulation of complex SQL expressions. It also shows how to implement integrity constraints by means of procedural code (if necessary!—see Chapter 8 of the present book). Recommended.

56. Wilfrid Hodges: *Logic*. London, England: Penguin Books (1977).

A gentle introduction to logic for the layperson.

57. International Organization for Standardization (ISO): *Database Language SQL,* Document ISO/IEC 9075:2008 (2011).

The official SQL standard (2011 version). Note that it is indeed an international standard and not just (as so many seem to think) an American or "ANSI" standard. Note too that although SQL:2011 is the current version of the standard, almost all of the SQL features discussed in the present book were already included in SQL:2008 or SQL:2003 or SQL:1999; in fact, most of them were included in SQL:1992 or even earlier versions.

58. David McGoveran: "Nothing from Nothing" (in four parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

This paper is referenced in Appendix C of the present book.

59. 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).

As mentioned in Chapter 1, the SQL standard has been through several versions over the years—the current version is SQL:2011 [57], the previous version was SQL:2008, the one before that was

SQL:2003, the one before that was SQL:1999, and the one before that was SQL:1992. So far as I know, these two books are the only ones available that cover, between them, any version later than SQL:1992 in detail. Melton was the editor of the SQL standard for many years.

60. Raghu Ramakrishnan and Johannes Gehrke: *Database Management Systems* (3rd edition). New York, N.Y.: McGraw-Hill (2003).

61. Avi Silberschatz, Henry F. Korth, and S. Sudarshan: *Database System Concepts* (6th edition). New York, N.Y.: McGraw-Hill (2009).

62. 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. *Note:* For a less formal introduction, see the book by Hodges [56].

63. Dave Voorhis: *Rel: An Implementation of Date and Darwen's* ***Tutorial D*** *Database Language*, *http://dbappbuilder.sourceforge.net/Rel.html*.

    Downloadable code for a prototype implementation of (a dialect of) **Tutorial D**.

64. 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. (In the interest of accuracy, however, I should note that QBE is really based more on the domain calculus than it is on the tuple calculus, which is the version of the calculus discussed in the body of this book.) Zloof was the original inventor and designer of QBE, and this paper was the first of many by Zloof on the subject.

I n d e x

For alphabetization purposes, (a) differences in fonts and case are ignored; (b) quotation marks are ignored; (c) other punctuation symbols—hyphens, underscores, parentheses, etc.—are treated as blanks; (d) numerals precede letters; (e) blanks precede everything else.