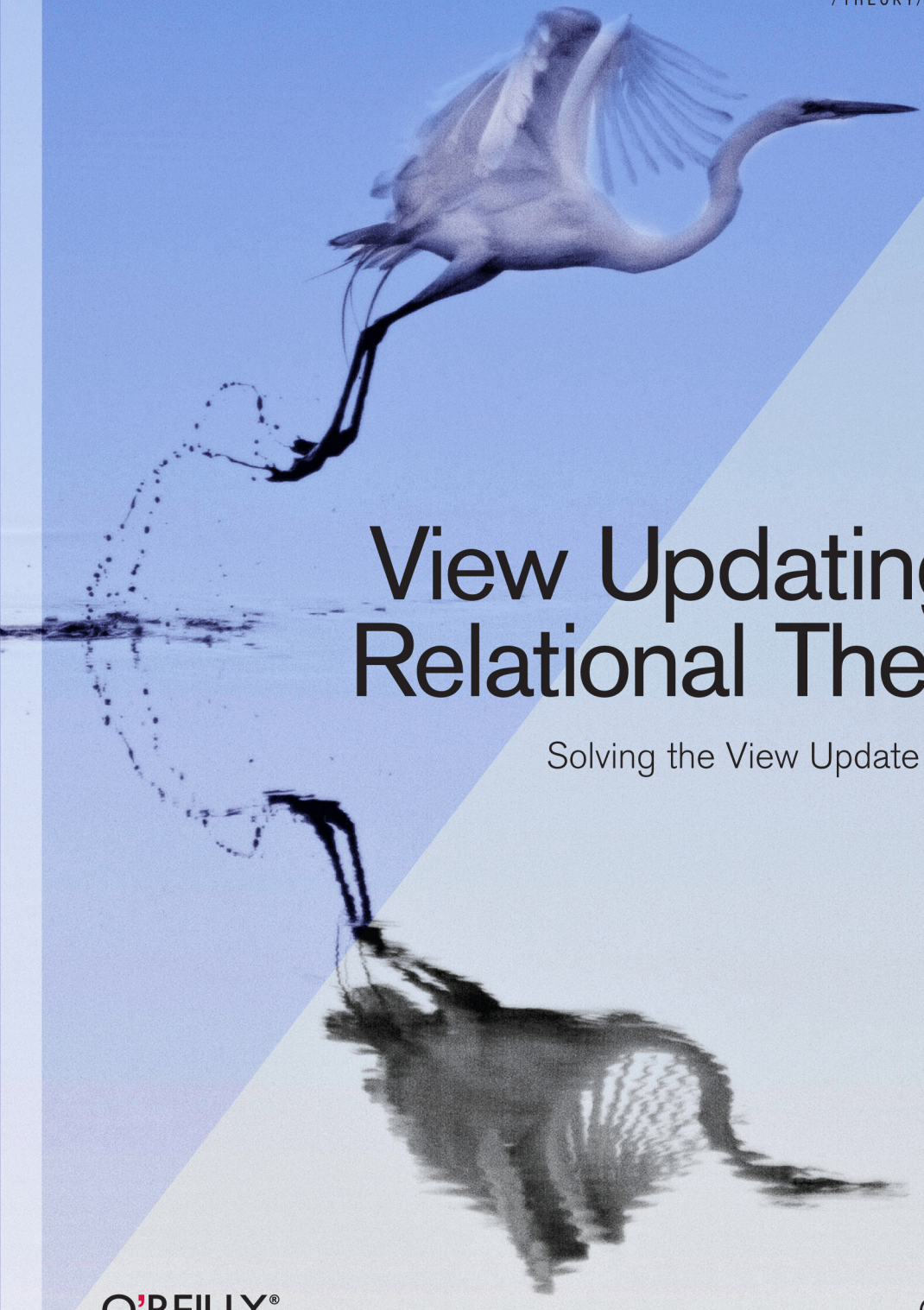
/Theory/In/Practice



View Updating & Relational Theory

Solving the View Update Problem

O'REILLY®

C.J. Date

Databases

View Updating and Relational Theory

Views are *virtual tables*. That means they should be updatable, just as "real" or base tables are. In fact, view updatability isn't just desirable, it's *crucial*, for practical reasons as well as theoretical ones. But view updating has always been a controversial topic. Ever since the relational model first appeared, there has been widespread skepticism as to whether (in general) view updating is even possible.

In stark contrast to this conventional wisdom, this book shows how views, just like base tables, can *always* be updated (so long as the updates don't violate any integrity constraints). More generally, it shows how updating *always* ought to work, regardless of whether the target is a base table or a view. The proposed scheme is 100% consistent with the relational model, but rather different from the way updating works in SQL products today.

This book can:

- Help database products improve in the future
- Help with a "roll your own" implementation, absent such product improvements
- Make you aware of the crucial role of predicates and constraints
- Show you how relational products are really supposed to behave

Anyone with a professional interest in the relational model, relational technology, or database systems in general can benefit from this book.

Chris Date is well known in the database industry for his ability to explain complex technical issues in a clear and understandable fashion. This book is a companion to his previous books for O'Reilly: *SQL and Relational Theory* (2nd edition) and *Database Design and Relational Theory*, both published in 2012.

US $39.99 CAN $41.99
ISBN: 978-1-449-35784-9

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

View Updating and Relational Theory

Solving the View Update Problem

C. J. Date

View Updating and Relational Theory
by C. J. Date

Printed in the United States of America.

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

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

Printing History:
January 2013: First Edition.

Revision History:
2012-12-12 First release
See *http://oreilly.com/catalog/errata.csp?isbn=0636920028437* for release details.

ISBN: 978-1-449-35784-9
[LSI]

Intension extension
Edgar F. Codd
Invented a notion
We now know as views
Now view and base relvar
Exchangeability
Got us all singing
Those view update blues

—Anon.: *Where Bugs Go*

The duke of Ormond took a view yesterday of his troop,
and ordered all that had bay or grey horses to change them for black.

—earliest known example (1693) of view updating,
quoted in the *Oxford English Dictionary* from
"A Brief Historical Relation of State Affairs 1678–1714,"
by Narcissus Luttrell (1857)

A little learning is a dangerous thing;
Drink deep, or taste not the Pierian spring:
There shallow drafts intoxicate the brain,
And drinking largely sobers us again.

—Alexander Pope: *An Essay on Criticism* (1711)

—— ♦♦♦♦♦ ——

To my wife Lindy
and my daughters Sarah and Jennie
with all my love

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold well over 850,000 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 Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, coauthored with Hugh Darwen, 2006)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007)
- From Apress: *The Relational Database Dictionary, Extended Edition* (2008)
- From Trafford: *Database Explorations: Essays on The Third Manifesto and Related Topics* (coauthored with Hugh Darwen, 2010)
- From Ventus: *Go Faster! The TransRelational™ Approach to DBMS Implementation* (2002, 2011)
- From O'Reilly: *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition, 2012)
- From O'Reilly: *Database Design and Relational Theory: Normal Forms and All That Jazz* (2012)

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

This book is the third in a series. Its predecessors were as follows:

- *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition)
- *Database Design and Relational Theory: Normal Forms and All That Jazz*

Both of these books were published by O'Reilly in 2012. The first was aimed at database practitioners of all kinds; it explained the principles of relational theory and used those principles as a basis for recommendations on how to use SQL as if it were a true relational language (a discipline I referred to in that book as "using SQL relationally"). The second was a little more specialized; it was aimed at database professionals with an interest in database design specifically, and it explained the theory of relational database design and showed why that theory was important. And this third book is more specialized too, inasmuch as it also focuses on one specific technical issue—but the issue in question is an extremely important one, one that gets to the heart of how relational database systems really ought to behave (as opposed to the way today's commercial SQL systems actually do behave, for the most part). That issue is *a theory of updating*: a theory that, as the book's title indicates, applies to the updating of views in particular but is actually more general, in that it applies to the updating of "base data" just as much as it does to the updating of views as such. *Note:* Despite this latter state of affairs, I decided to emphasize the updating of views as such in the book's title because it seems to me that, while database practitioners in general believe they understand how updating works when the target is base data, they're typically more than a little skeptical as to whether it really works, or can be made to work, when the target is a view. In fact, view updating as such is a surprisingly controversial topic—which was and is, of course, a strong reason for wanting to write this book in the first place.

With regard to those two earlier books, incidentally, I should probably apologize for the large number of references to them (especially the first one) in the present book. Now, most references in this book to other publications are given in full, as in this example:

> David McGoveran: "Accessing and Updating Views and Relations in a Relational Database," U.S. Patent No. 7,263,512 (August 28th, 2007)

In the case of those previous books of mine in particular, however, I'll refer to them from this point forward by their abbreviated titles alone (viz., *SQL and Relational Theory* and *Database Design and Relational Theory*, respectively).

Aside: I've said I'll be giving references to other publications in full, but actually there aren't many such references anyway. Although numerous papers, articles, and other writings on view updating have appeared over the past 30 years or so, most of them—with the notable exception of certain publications by David McGoveran—advocate approaches that differ fairly drastically from the one described in the present book (see later in this preface for further discussion of this point). For the most part, therefore, I felt it inappropriate to reference them, except for an occasional citation here and there. If you're interested in investigating some of those other approaches in more detail, you can find a short list of pertinent references in Chapter 10 of my book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004). *End of aside.*

I should stress that I do assume throughout what follows that you're familiar with much of what's covered in the *SQL and Relational Theory* book in particular. For example, I certainly assume you know what relations, attributes, and tuples are. Now, I make no apology for this state of affairs, since the present book is aimed at database professionals and database professionals ought really to be familiar with most of what's in that earlier book anyway. In order to make the present book a little more self-contained, however, I do offer in Chapter 2 ("The Technical Context") a brief review of pertinent aspects of that earlier book. I also offer in Chapter 3 ("The View Concept: A Closer Look") a more detailed summary of what views in particular are and how they're supposed to work.

Who Should Read This Book

My target audience is database professionals, or more generally anyone interested in the relational model, relational technology, or relational systems in general. As already indicated, familiarity with the *SQL and Relational Theory* book would be a big help, but I believe the present book has fresh insights to offer regarding relational theory in general, with special reference to view updating in particular. Also, I think it's worth pointing out that it might be possible to use the ideas contained herein to guide a "roll your own" implementation (of view updating, I mean), absent native support on the part of the pertinent DBMS.[1] However, my dearest wish in this regard is that DBMS implementers in particular will read this book and will thereby be motivated to provide some native view update support in their own product. *Note:* I'd also like to mention that I have a live seminar available based on the material in this book. For further details, please go to the website *www.justsql.co.uk/chris_date/chris_date.htm*.

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

Structure of the Book

I've said I assume you know what relations, attributes, and tuples are; more specifically, I assume you know what views are, too, at least in general terms. Views were originally discussed (though not by that name) in Codd's very first paper on the relational model:

> E. F. Codd: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19th, 1969)

Now, the principal rationale for supporting views, as Codd himself foresaw in the paper just referenced, is that they provide the means by which—at least in principle—the important goal of logical data independence can be achieved. (The term *logical data independence* refers to the ability to change the logical design of a database without having to make corresponding changes in the way the database is perceived by users, thereby protecting investment in, among other things, existing user training and existing applications. See Chapter 3 for further discussion.) In other words, the primary *raison d'être*.for views is, precisely, the goal of logical data independence. But if we're to achieve that goal in practice and not just in principle, then it's clear that views have to be updatable.

So view updating is an important problem. As a consequence, it has been the focus of considerable attention for quite some time now (at least 35 years or so), in both commercial and academic environments, and several different approaches have been proposed—even implemented, in some cases. However, the approaches in question all fail to provide a truly satisfactory solution to the problem (not just in my opinion, but also in that of other writers, I hasten to add). In the case of today's mainstream SQL products, for example, the view updating mechanisms are typically both:

- Incomplete, meaning they fail entirely to support updates on certain theoretically updatable views, and also

- Incorrect, meaning even the view updates they do support they implement incorrectly, at least in some cases

(Again, see Chapter 3 for further discussion of these points.) As for the research literature, it seems to me that the writings in question typically overlook certain important factors—factors that are crucial to a systematic, comprehensive, and correct solution to the problem. By contrast, the solution described in detail in this book is indeed, I believe, a "systematic, comprehensive, and correct" one. I also believe (though in this connection I must make it very clear that I'm not an implementer myself) that the proposed solution could be incorporated into a relational DBMS with comparatively modest conceptual extensions to the architecture of the system.

Aside: Note that I do carefully say "a *relational* DBMS" here. As will be seen, the proposed solution relies heavily on the ability to state integrity constraints declaratively (and on the ability of the DBMS to enforce them, of course). For my part, I regard such capabilities as a sine qua non of a truly relational system. As I'm sure you're aware, however, most if not all of today's SQL products are seriously deficient in this area. *End of aside*.

With the foregoing by way of preamble, let me now say something about the way the text is structured:

- Chapter 1 provides a motivating example that illustrates in simple and familiar terms (actually SQL terms) the approach to view updating to be described in detail in subsequent chapters. In particular, it demonstrates that "updating is updating," regardless of whether it's a view or base data that's being updated. That's why, as I said earlier, the book is concerned with what might be called a theory of updating in general—a theory that does apply to views in particular, but applies to base data equally as much.

- Next, as previously mentioned, Chapter 2 offers a brief review of pertinent aspects of relational theory. In particular, it emphasizes the nature of the database per se as "the one true variable" and hence as the proper target for all operations of an updating nature.

- Chapter 3 then describes the view concept and related matters in detail. Of course, I've already said I assume you know what views are in general terms, but this chapter covers a lot of material you might not be so familiar with, material that's essential to a proper understanding of subsequent chapters.

- Chapters 4–13 then discuss, one by one, views based on a variety of familiar (and, in a few cases, possibly not so familiar) relational operators—restriction, projection, join, and so on. Chapter 4 in particular, on restriction views, also introduces by means of examples quite a lot of additional foundation material (in fact, the chapter is in some respects a continuation of Chapter 3). The chapter also gives some idea as to the plan to be followed in the next nine chapters.

- Chapter 14 then investigates the question of combining operations (e.g., what's involved in updating a join of two restrictions, or a union of two joins?), a question that raises some rather intriguing and possibly surprising issues.

- Finally, Chapter 15 presents an approach to resolving certain ambiguities that might arise—or might be claimed to arise, at least—in connection with the scheme described in previous chapters.

- There are also two appendixes. Appendix A goes into detail on certain aspects of the all important *relational assignment* operator. Appendix B contains definitions for purposes of reference of the various relational operators considered in detail in the body of the book.

Note: As the foregoing outline should be sufficient to suggest, the book is definitely meant to be read in sequence as written.

Technical Notes

There are a few further preliminary points I need to cover here. First of all, note that I follow the usual convention throughout this book in using the generic term *update* in lower case to refer to the INSERT, DELETE, and UPDATE operators considered collectively (as well as to what I just referred to as "the all important relational assignment operator"—see Chapter 2). When I want to refer to the UPDATE operator as such, I'll set it in all upper case ("all caps") as just shown. As for the INSERT and DELETE operators, however, where no ambiguity arises, it can be a little tedious always to set them in all caps—especially when they're being used as qualifiers, as in, e.g., "INSERT rule" ("insert rule"?). I've therefore decided to use both forms in this book, letting context be my guide in any given situation (and I won't pretend I've been all that consistent in this respect, either).

Second, please note that I use the term *SQL* to mean the standard version of that language specifically, 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 pronunciation is common in the field), thereby writing things like *an* SQL table, not *a* SQL table. *Note:* The SQL standard has been through several versions, or editions, over the years. The version current at the time of writing is SQL:2011. Here's the formal reference:

> International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:2011 (2011)

Third and last, I need to say something about my use of the term *user*; in particular, I need to explain what I mean by my frequent use of phrases such as "what the user sees" or "the user's perception of the database." In general, you can take the term *user* to refer to either an interactive user[2] or an application programmer or both, as the context demands. As for "what the user sees" and similar phrases, what I'm referring to here is the fact that most users interact, not with the database in its entirety, but rather with some subset of that entire database, defined by what's sometimes called a *subschema*. What's more, thanks to the view mechanism, that subset can and often does involve some logical restructuring. In fact, we can (and I will) assume for

[2] But still someone who knows something about database issues, not a genuine "end user," who might quite reasonably be totally ignorant of such matters.

simplicity, and without loss of generality, that the subset in question consists exclusively of views, even if some of the views in question are effectively identical to the base data from which they're derived. Of course, to the user of that subset, that collection of views *is* the database! In other words, *database* is a relative term, in a sense. Thus, we can usefully, albeit somewhat loosely, define a database, at least for the purposes of this book, to be either a given collection of data—i.e., the given base data—or some specific subset, possibly restructured, of that given collection. *Note:* When I say "somewhat loosely" here, what I have in mind primarily is the fact that a database is more than just data as such—the pertinent integrity constraints need to be taken into account as well, as we'll see in Chapters 2 and 3.

Acknowledgments

I'd like to begin by thanking my wife Lindy once again for her support throughout the production of this book, as well as all of its predecessors. I'd also like to thank my friends and colleagues Hugh Darwen, David Livingstone, and David McGoveran for their detailed and comprehensive reviews of earlier drafts of this book. Those reviewers and their reviews were all very helpful in different ways, but David McGoveran in particular deserves special thanks—first of all, for originally suggesting the basic idea on which the view updating approach described in this book is based; second, for communicating and collaborating with me on this topic many times over the past 20 years or so; and last but not least, for his extensive theoretical work in this area. David also went considerably beyond the call of duty in his review: He not only commented on the text as such, he actually compiled and sent me a series of short essays on various aspects of the subject matter. Those essays were extremely helpful to me in my task of rewriting, and I believe they've resulted in a greatly improved text. Of course, I haven't incorporated all of his suggestions—I don't believe any author ever does act on all of the comments he or she receives from reviewers! But I've tried to do justice to what seemed to me to be the most important and substantive of his comments. Of course, it goes without saying that, as always, any remaining errors are my responsibility.

C. J. Date
Healdsburg, California
2013

Foreword

In the field of relational database theory and practice there have been two particularly thorny and controversial issues, neither of which has been resolved to everybody's satisfaction: the missing information problem and the view updating problem. On the first of these, Chris Date has written copiously over the last 30 years or so; now he tackles the second one head on.

It's not as though he hasn't addressed the subject before, of course. His well known and widely used textbook, *An Introduction to Database Systems,* included material—well, a page or two, at any rate—on the subject in its very first edition, published in 1975. That page count grew to sixteen or so in the eighth edition (2004). His first whole chapter on the subject appeared in the book that started his long running *Relational Database Writings* series, in 1986. In the fourth book in that series, which appeared in 1995, he and David McGoveran gave us two chapters that showed evidence of a major shift in thinking on the issue, based on McGoveran's work. That thinking then further evolved in an appendix in *Databases, Types, and the Relational Model: The Third Manifesto* (2007), through a chapter in *Database Explorations* (2010), and on to the present volume.

The basic idea, first mooted by E. F. Codd in 1969, has never changed. Assume we're given a database consisting, by definition, of (a) some collection of relation variables or *relvars*,[1] together with (b) a set of integrity constraints governing the permissible values of those relvars. Those given relvars are said to be the *base* ones. In general, the chosen design is one of several that could have been chosen to represent exactly the same information. From the chosen design we can derive an alternative one by defining virtual relvars, or *views*, in terms of relational expressions referencing the base relvars. For various reasons, such an alternative design—an alternative view of the database, in effect—might be considered more suitable than the base design for certain users. More importantly, that alternative design might actually exclude parts of the underlying or "real" database that some users have no interest in, or perhaps are not authorized to see. Moreover, if some change to the base design becomes necessary, virtual relvars representing the original design can be defined on the new design, such that existing users' views of the database are immune to the change and potentially unpleasant upheavals are avoided. This is the basic idea behind the well known goal of *logical data independence.*

The thorny issues arise when users express database updates in terms of updates against the virtual relvars they see as constituting their database. How is the DBMS to determine the real updates to the real database that will cause the specified changes to occur in those virtual relvars? And if there are several ways of achieving the desired effect, which one should be chosen? For a simple example, suppose a user of the usual suppliers-and-parts database (described in detail in Chapter 1) sees a virtual relvar, or view, PS that shows only those suppliers that are located in Paris. The defining expression for view PS is, of course, S WHERE CITY = 'Paris'. Now

[1] SQL would call those relvars *tables*. For further explanation of the terminology of relvars and related matters, see Chapter 2.

suppose that same user tells the DBMS to delete the tuple for supplier S2 from that view PS. Should the DBMS assume that supplier S2 no longer exists and delete the underlying tuple from base relvar S? Or should it reject the request as being ambiguous, considering that the same effect could be achieved by replacing supplier S2's CITY value by something other than Paris? Moreover, suppose the user actually knows supplier S2 has moved to London and attempts to effect that change by "updating the tuple" for supplier S2 accordingly in view PS. Should the DBMS accept that update? Now suppose still further that view PS excludes the STATUS attribute. How should the DBMS react to an attempt by that user to insert tuples into that view, given that such tuples must necessarily omit values for STATUS?

These and many more are the kinds of questions Date attempts to answer in the detailed, thorough, careful, methodical analysis he now offers us. He lays out his plan of attack in the first three chapters. He clearly defines what it means for two database designs to be equivalent in the sense of representing the same information, and he then describes the methodology applied in the next ten chapters. That methodology entails examining each of the operators of the relational algebra in turn. For example, that "Paris suppliers only" view PS is what he calls a restriction view—i.e., a virtual relvar defined using just the restriction operator. Likewise, the view that excludes the STATUS attribute from PS is defined using projection. As this latter view is a projection of a restriction, we can infer the effects of updates on it by invoking Date's rules for updating through projection to determine the effects on the underlying restriction, then invoke the rules for updating though restriction to determine the effects on the underlying base relvar S.

Applying the rules for a view whose definition involves several relational operations raises a very interesting and possibly controversial issue that Date addresses in Chapter 14: viz., if two expressions are syntactically distinct but logically equivalent (in the way that, for example, the numerical expressions $x(y+z)$ and $xy+xz$ are syntactically distinct but logically equivalent), should views defined on those expressions necessarily exhibit identical behavior with respect to update operations on them?

Now, some aspects of Date's proposals proved to be controversial when they appeared in the 2007 and 2010 publications I mentioned earlier. For example, should a tuple inserted into a view defined on the union of *R1* and *R2* result in that tuple appearing in both *R1* and *R2*? And should a tuple being deleted from a view defined on the intersection of *R1* and *R2* result in that tuple disappearing from both *R1* and *R2*? I am on record as being one of those who expressed opposition to those particular proposals—this being, I hasten to add, the only serious technical disagreement between Date and myself that has arisen during our long period of collaboration. Those controversial details are retained here and Date has strengthened his rationale for them, though admitting that he might still fail to convince everybody who was against them. For my part, I found that his final chapter, "Ambiguity Revisited," offers an intriguing possibility of light at the end of this particular tunnel. In it he describes in outline an idea, due to David McGoveran, for a radically different approach to the language we use for updating relational databases, effectively replacing—or at least extending—the familiar INSERT, DELETE, and UPDATE operators that have been with us in some form or other since prerelational times.

Among the advantages claimed for this novel approach is that the problems giving rise to the controversy I have mentioned simply do not arise.

Date tells us that he does not expect or even wish this book to be the end of the story on view updating, but he hopes it will provide a firm basis on which the debate can move forward. I think that is exactly what he has provided, and I join him in that hope.

Hugh Darwen
Shrewley, England
2013

Chapter 1

A Motivating Example

Example is always more efficacious than precept
—Samuel Johnson: *Rasselas* (1759)

Examples throughout this book are based for the most part on the familiar (not to say hackneyed) suppliers-and-parts database. I apologize for dragging out this old warhorse yet one more time, but as I've said elsewhere, I believe using the same example in a variety of different publications can be a help, not a hindrance, in learning. In SQL terms,[1] the database contains three tables—more specifically, three base tables—called S ("suppliers"), P ("parts"), and SP ("shipments"), respectively. Sample values are shown in Fig. 1.1.

S

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

P

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

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig. 1.1: The suppliers-and-parts database—sample values

The semantics (in outline) are as follows:

[1] I use SQL and SQL-style syntax in this introductory chapter for reasons of familiarity, despite the fact that it's not really to my taste, and (more to the point, perhaps) despite the fact that it actually makes the motivating example harder to explain properly.

- Table S represents *suppliers under contract*. Each supplier has one supplier number (SNO), unique to that supplier; one name (SNAME), not necessarily unique (though the sample values shown in Fig. 1.1 do happen to be unique); one status value (STATUS); and one location (CITY). *Note:* In the rest of this book I'll abbreviate "suppliers under contract," most of the time, to just *suppliers*.

- Table P represents *kinds of parts*. Each kind of part has one part number (PNO), which is unique; one name (PNAME); one color (COLOR); one weight (WEIGHT); and one location where parts of that kind are stored (CITY). *Note:* In the rest of this book I'll abbreviate "kinds of parts," most of the time, to just *parts*.

- Table SP represents *shipments*—it shows which parts are shipped, or supplied, by which suppliers. Each shipment has one supplier number (SNO); one part number (PNO); and one quantity (QTY). Also, there's at most one shipment at any given time for a given supplier and given part, and so the combination of supplier number and part number is unique to any given shipment. *Note:* In the rest of this book I'll assume QTY values are always greater than zero.

Now I want to focus on table S specifically; for the rest of this chapter, in fact, I'll mostly ignore tables P and SP, except for an occasional remark here and there. Here's an SQL definition for that table S:

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

As I've said, table S is a base table, but of course we can define any number of views "on top of" that base table. Here are a couple of examples—LS ("London suppliers") and NLS ("non London suppliers"):

```
CREATE VIEW LS /* London suppliers */ AS
  ( SELECT SNO , SNAME , STATUS , CITY
    FROM   S
    WHERE  CITY = 'London' ) ;

CREATE VIEW NLS /* non London suppliers */ AS
  ( SELECT SNO , SNAME , STATUS , CITY
    FROM   S
    WHERE  CITY <> 'London' ) ;
```

Sample values for these views corresponding to the value of table S in Fig. 1.1 are shown in Fig. 1.2.

LS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

Fig. 1.2: Views LS and NLS—sample values

Views LS and NLS are the ones I want to use in this initial chapter as the basis for my motivating example. In essence, what I want to do with that example is try to give you some preliminary idea as to why I believe that—contrary to popular opinion and most conventional wisdom in this area—***all views are updatable.*** (Note, however, that I must immediately qualify this very strong claim by making it clear that I'm necessarily speaking rather loosely at this stage. Later chapters will elaborate.)

THE PRINCIPLE OF INTERCHANGEABILITY

So far, then, table S is a base table and tables LS and NLS are views. Observe now, however, that *it could have been the other way around*—that is, I could have made LS and NLS base tables and S a view, like this:

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

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

CREATE VIEW S AS
  ( SELECT SNO , SNAME , STATUS , CITY
    FROM   LS
    UNION
    SELECT SNO , SNAME , STATUS , CITY
    FROM   NLS ) ;
```

Note: In order to guarantee that this design is formally equivalent to the original one, I should really state, and have the DBMS enforce, certain integrity 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 want to ignore such details for the moment. I'll have a lot more to say about such matters in a little while, I promise you.

Anyway, the message of the example is that, in general, which tables are base ones and which ones are views is arbitrary (at least from a formal point of view). In other words, in the case at hand, 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, implying among other things that for any query on one, there's a logically equivalent query on the other. Chapter 3 elaborates on this concept.) And *The Principle of Interchangeability* is a logical consequence of such considerations:

- **Definition:** *The Principle of Interchangeability* states that there must be no arbitrary and unnecessary distinctions between base tables and views; in other words, views should—as far as possible—"look and feel" just like base tables so far as users are concerned.

Here are some implications of this principle:

- As I've already suggested, views are subject to integrity constraints, just like base tables. (We usually think of integrity constraints as applying to base tables specifically, but *The Principle of Interchangeability* shows this position isn't really tenable.)

- In particular, views have keys (and so I ought really to have included some key specifications in my view definitions; unfortunately, however, SQL doesn't permit such specifications).[2] They might also have foreign keys, and foreign keys might refer to them.

- Many SQL products, and the SQL standard, provide some kind of "row ID" feature (in the standard, that feature goes by the name of *REF types* and *reference values*). If that feature is available for base tables but not for views—which in practice is quite likely—then it clearly violates *The Principle of Interchangeability*.

- 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*.

---

[2] Throughout this book I use the term *key*, unqualified, to mean a candidate key, not necessarily a primary key specifically. In fact, **Tutorial D**—see Chapter 2—has no syntax for distinguishing between primary and other keys. For reasons of familiarity, however, I use double underlining in figures like Fig. 1.1 to suggest that the attributes so underlined can be thought of as primary key attributes, if you like.

BASE TABLES ONLY: CONSTRAINTS

One thing that follows from *The Principle of Interchangeability* is that the behavior of tables S, LS, and NLS shouldn't depend on which if any are base tables and which if any are views. Until further notice, therefore, let's suppose they're all base tables:

```
CREATE TABLE S   ( ... , UNIQUE ( SNO ) ) ;
CREATE TABLE LS  ( ... , UNIQUE ( SNO ) ) ;
CREATE TABLE NLS ( ... , UNIQUE ( SNO ) ) ;
```

Now, these tables, like all tables, are clearly subject to a number of constraints. Unfortunately, most of those constraints are quite awkward to formulate in SQL, so I'll content myself for present purposes with stating them in natural language only (and pretty informal natural language at that, for the most part). Here they are:

- {SNO} is a key for each of the tables; also, {SNO} in each of tables LS and NLS is a foreign key, referencing the key {SNO} in table S. *Note:* For an explanation of why I use braces "{" and "}" here, please refer to *SQL and Relational Theory*.[3]

- At any given time, table LS is equal to that restriction of table S where the CITY value is London, and table NLS is equal to that restriction of table S where the CITY value isn't London. Moreover, every row of table LS has CITY value London,[4] and no row of table NLS does.

- At any given time, table S is equal to the union of tables LS and NLS; moreover, that union is *disjoint* (i.e., the corresponding intersection is empty)—no row in S appears in both LS and NLS. To spell the point out in detail: Every row in S also appears in exactly one of LS and NLS, and every row in either LS or NLS also appears in S.

- Finally, the previous constraint and the constraint that {SNO} is a key for all three tables, taken together, imply that every supplier number (not just every row) in S also appears in exactly one of LS and NLS, and every supplier number in either LS or NLS also appears in S.

Of course, as the immediately preceding bullet point illustrates, the foregoing constraints aren't all independent of one another—some of them are logical consequences of others.

---

[3] I remind you from the preface that throughout this book I use "*SQL and Relational Theory"* as an abbreviated form of reference to my book *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition, O'Reilly, 2012).

[4] Precisely because of this fact, a more realistic version of view LS would probably drop the CITY attribute. I choose not to do so here, in order to keep the example simple.

BASE TABLES ONLY: COMPENSATORY ACTIONS

Now, in order to ensure that the constraints outlined in the previous section continue to hold when certain updates are done, certain **compensatory actions** need to be in effect. In general, a compensatory action—also known as a *compensating* action—is an additional update (over and above some update explicitly requested by the user) that's performed automatically by the DBMS, precisely in order to avoid some integrity violation that might otherwise occur.[5] Cascade delete is a typical example.[6] 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. To be specific, deleting rows from either LS or NLS clearly needs to cascade to cause those same rows 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 d FROM LS : DELETE d FROM S ;

ON DELETE d FROM NLS : DELETE d FROM S ;
```

Likewise, deleting rows from S clearly needs to cascade to cause those same rows to be deleted from whichever of LS or NLS they appear in:

```
ON DELETE d FROM S : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( d WHERE CITY <> 'London' ) FROM NLS ;
```

As an aside, I remark that, given that an attempt to delete a nonexistent row has no effect—or so I'm going to assume, at any rate—we could replace each of the expressions in parentheses in the foregoing rule by just *d*. However, the expressions in parentheses are perhaps preferable, at least inasmuch as they're clearly more specific.

Analogously, we'll need some compensatory actions ("cascade insert rules") for INSERT operations:

```
ON INSERT i INTO LS : INSERT i INTO S ;

ON INSERT i INTO NLS : INSERT i INTO S ;

ON INSERT i INTO S : INSERT ( i WHERE CITY = 'London' ) INTO LS ,
                     INSERT ( i WHERE CITY <> 'London' ) INTO NLS ;
```

---

[5] One reviewer asked why I chose the term *compensatory action* for this construct. Well, I should have thought the answer was obvious, but in case it isn't, let me spell it out: The reason I call such actions "compensatory" is because they cause a second update to be done to compensate for the effects of the first (speaking a trifle loosely, of course).

[6] Cascade delete is usually thought of as applying to foreign key constraints specifically; however, the concept of compensatory actions is actually more general and applies to constraints of many kinds.

Note: The concept of cascade insert doesn't usually arise in connection with foreign key constraints, of course, but that's no reason not to support such a concept in general. More important, don't get the idea that compensatory actions must always take the form of simple cascades. While the ones discussed in this introductory chapter do all happen to take that form, more complicated cases are likely to require actions of some less straightforward form, as we'll see in later chapters.

As for UPDATE operations, they can be regarded, at least in the case at hand, as a DELETE and an INSERT taken in combination; as a consequence, the necessary compensatory actions are just a combination of the corresponding delete and insert actions, loosely speaking. For example, consider the following UPDATE on table S:

```
UPDATE S
SET    CITY = 'Oslo'
WHERE  SNO = 'S1' ;
```

What happens here is this:

1. The existing row for supplier S1 is deleted from table S and a new row for that supplier, with CITY value Oslo, is inserted into that same table.

2. The existing row for supplier S1 is deleted from table LS as well, thanks to the cascade delete rule from S to LS, and the new row for that supplier, with CITY value Oslo, is inserted into table NLS as well, thanks to the cascade insert rule from S to NLS. In other words, the row for supplier S1 has "migrated" from table LS to table NLS! (Of course, here I'm speaking very loosely indeed.)

   Suppose now that the original UPDATE had been directed at table LS rather than table S:

   ```
   UPDATE LS
   SET    CITY = 'Oslo'
   WHERE  SNO = 'S1' ;
   ```

Now what happens is this:

1. The existing row for supplier S1 is deleted from table LS.

2. An attempt is made to insert a new row for supplier S1, with CITY value Oslo, into table LS. That attempt fails, however, because it violates the constraint on table LS that the CITY value in that table must always be London. So the update fails overall; the previous step (viz., deleting the original row for supplier S1 from LS) is undone, and the net effect is that the database remains unchanged.

VIEWS: CONSTRAINTS AND COMPENSATORY ACTIONS

Now I come to the real point of this chapter: *Everything I've said in the previous two sections applies pretty much unchanged if some or all of the tables concerned are views.* For example, suppose as we originally did that S is a base table and LS and NLS are views:

```
CREATE TABLE S      ( ............. , UNIQUE ( SNO ) ) ;
CREATE VIEW  LS  AS ( SELECT ... WHERE CITY = 'London' ) ;
CREATE VIEW  NLS AS ( SELECT ... WHERE CITY <> 'London' ) ;
```

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 tables. As far as that user is concerned, then, those tables have semantics 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 is not London).*

That same user will also be aware of the following constraints (note that these constraints make no mention of table S, because the user in question doesn't even know table S exists):

- {SNO} is a key for both LS and NLS.
- Every row in LS has CITY value London, and no row in NLS does.
- No supplier number appears in both LS and NLS.

However, that user won't be aware of any compensatory actions as such, precisely because he or she isn't aware that LS and NLS are actually views of S; indeed, as I've already said, the user isn't even aware of the existence of S (which is why that user is also unaware of the constraint to the effect that the union of LS and NLS is equal to S). But updates by that user on LS and NLS will all work as far as that user is concerned exactly as if LS and NLS really were base tables. Also, of course, updates by that user on LS and NLS will have the appropriate effects on S, even though those effects won't be directly visible to that user.

THERE'S NO MAGIC

Now consider a user who sees only, say, view LS (i.e., not view NLS and not base table S). Presumably this user still wants to be able to behave as if LS were a base table. Of course, this user will certainly know the semantics of that table—

> LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (which is London).*

—and will also be aware of the following constraints:

- {SNO} is a key for LS.
- Every row in LS has CITY value London.

Clearly, this user can't be allowed to insert rows into this table—nor to update supplier numbers within this table—because such operations have the potential to violate constraints of which this user is unaware (and must be unaware).[7] But if LS really were a base table, it would surely be possible to insert rows into it, wouldn't it? Indeed, if it weren't, then the table would always be empty! So doesn't the foregoing state of affairs constitute a violation of *The Principle of Interchangeability?*

In fact it does not. While it's true that this particular user can't be allowed to insert rows into the table, that's not the same as saying *no* user is allowed to do so. The basic reason why this particular user can't insert rows into LS is that this user is seeing only part of the picture, as it were. Contrast a user who does see both LS and NLS, which in combination are information equivalent to the original table S; as we saw in the previous section, such a user certainly can insert rows into LS (and/or NLS). But the user who sees only LS is seeing something that isn't information equivalent to the original table S, and so it's only to be expected that there'll be certain operations that he or she can't be allowed to do.

In closing, it's worth pointing out that even here there are parallels with the situation in which all tables involved really are base tables. That is, even when the tables in question are all base tables, it'll sometimes be the case that certain users will be prohibited from performing certain updates on certain tables. By way of example, consider a user who sees only base table SP and not base table S. Like the user who sees only table LS, that user can't be allowed to perform insert operations, because such operations have the potential to violate constraints of which that user is unaware (and must be unaware)—to be specific, the foreign key constraints from SP to tables S and P.

---

[7] I suppose the user *might* be allowed to perform such operations if he or she is also prepared to accept occasional error messages to the effect that an operation is rejected simply "because the system says so," without further explanation. See Chapter 4 for further discussion of this point.

CONCLUDING REMARKS

This brings me to the end of the discussion of the motivating example. Now, that example is extremely simple, and the conclusions I've drawn from it are perhaps all very obvious; but what I'm suggesting is that thinking of views as base tables "living alongside" the tables in terms of which they're defined is a fruitful way to think about the view updating problem in general—indeed, not just a fruitful way, but a way I believe is *logically correct*.[8] The overall idea is thus as follows:

1. The view defining expressions imply certain constraints. For example, the view defining expression for view LS ("London suppliers") implies a constraint to the effect that LS is equal to that restriction of table S where the CITY value is London.

2. Such constraints in turn imply certain compensatory actions (i.e., actions that need to be performed, over and above updates that are explicitly requested by the user, in order to avoid some integrity violation that might otherwise occur). For example, the constraints on tables S, LS, and NLS imply certain cascade deletes and cascade inserts, as we've seen.

By the way, I'd really like to stress this latter point—the point, that is, that it should be possible for the compensatory actions that apply in a given situation to be determined by the DBMS from the pertinent view defining expression. In other words, what I'm *not* suggesting is that such actions need to be specified explicitly, thereby imposing yet another administrative burden on the already overworked DBA.[9] But this issue, like many others I've touched on briefly in this introductory chapter, will be explored in more detail in later parts of the book.

In closing, let me suggest that if (like most people) you skipped the preface and started straight in on this first chapter, now would be a good time to go back and read the preface, before you move on to the next chapter. Among other things, the preface includes an outline of the structure of the book overall. It also spells out certain important technical assumptions that I'll be relying on in the chapters to come, and hence that you need to be aware of.

---

[8] Acknowledgments here to David McGoveran, who first got me thinking along these lines several years ago.

[9] DBA = database administrator.

Chapter 2

The Technical Context

What I assume you shall assume
—Walt Whitman: *Leaves of Grass* (1885)

The discussions in the previous chapter were based on SQL for reasons of familiarity. Unfortunately, however, SQL really isn't suitable as a basis for the kind of investigation and detailed technical discussion the subject at hand demands. For one thing, the concepts we need to examine often can't be formulated in SQL at all; for another, even when they can, SQL usually manages to introduce so much irrelevant and unnecessary complexity that it becomes hard to see the forest for the trees, as it were. For such reasons, I intend to use as a foundation for the rest of the book, not SQL as such (though I'll still have a few things to say about SQL as such from time to time), but rather a hypothetical language called **Tutorial D**.[1] Now, I believe that language is pretty much self-explanatory; however, a comprehensive description can be found if needed in the book *Databases, Types, and the Relational Model: The Third Manifesto*, by Hugh Darwen and myself (3rd edition, Addison-Wesley, 2006).[2]

As its title suggests, the book just mentioned—referred to hereinafter as just "the *Manifesto* book" for short—also introduces and explains *The Third Manifesto*, a precise though somewhat formal definition of the relational model and a supporting type theory (including, incidentally, a comprehensive model of type inheritance). In that book, we use the name **D** as a generic name for any language that conforms to the principles laid down by the *Manifesto*. Any number of distinct languages could qualify as a valid **D**; sadly, however, SQL isn't one of them. By contrast, **Tutorial D** is a valid **D**, of course; in fact, **Tutorial D** was explicitly designed to be suitable as a vehicle for illustrating and teaching the ideas of the *Manifesto*, a state of affairs that makes it equally suitable for the purposes I propose to use it for in this book. Thus, while I've said that discussions in this book will be based on **Tutorial D**, it would really be more accurate to say they'll be based on the ideas of the *Manifesto* per se. The remainder of this chapter

---

[1] The language is hypothetical only inasmuch as no commercial implementations exist at the time of writing. But prototype implementations do exist and can be accessed via the website *www.thethirdmanifesto.com* (see the footnote immediately following).

[2] **Tutorial D** has been revised and extended somewhat since that book was first published. A description of the revised version (which is close but not identical to the version I'll be using in this book) can be found in another book by Hugh Darwen and myself, *Database Explorations: Essays on The Third Manifesto and Related Topics* (Trafford, 2010), as well as on the website *www.thethirdmanifesto.com* (which, as its name suggests, additionally contains much information concerning the *Manifesto* as such).

consists of a survey of those ideas (ignoring, of course, ones that aren't particularly relevant to our main purpose). In other words, it consists primarily of a summary of material I would really prefer to assume you're familiar with already. Even if you are, however, it's probably a good idea at least to give the chapter a "once over lightly" reading anyway, if only to take note of some of the concepts and terminology I'm going to be relying on heavily in subsequent chapters.

RELATIONS AND RELVARS

To begin with, then, every relation has a *heading* and a *body*, where the heading is a set of attributes and the body is a set of tuples that conform to the heading. For example, referring once again to the suppliers-and-parts database (see Fig. 2.1, a repeat of Fig. 1.1 from Chapter 1), the heading of the suppliers relation is the set {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}—I'm assuming here for definiteness that attributes SNO, SNAME, STATUS, and CITY are defined to be of types CHAR, CHAR, INTEGER, and CHAR, respectively—and the body of that relation is the set of tuples for suppliers S1, S2, S3, S4, and S5. What's more, each of those tuples does conform to the pertinent heading, inasmuch as it contains exactly one value, of the applicable type, for each of the attributes SNO, SNAME, STATUS, and CITY (and, of course, nothing else). *Note:* It would be more precise—and actually more correct—to say that each of those tuples *has* the pertinent heading, because in fact tuples have headings, just as relations do.

S

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

P

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

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig. 2.1: The suppliers-and-parts database—sample values

Every relation also has (or is of) a certain *type*—a certain relation type, to be specific, where the type in question is fully determined by the pertinent heading. Thus, we denote the

type of a given relation in **Tutorial D** by means of the keyword RELATION followed by the applicable heading. For example, here's the type for the suppliers relation:

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

Next, there's a logical difference between relations as such and relation *variables*.[3] Take another look at Fig. 2.1. That figure shows three relations: namely, the relations 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 relations appearing in their place. In other words, S, P, and SP are really variables—relation variables, to be precise—and like all variables they have different values at different times. And since they're relation variables specifically, their values at any given time are, of course, relation values. Note, however, that the legal values for a given relation variable must all be of the same relation type—i.e., they must all have the same heading—and the type and heading in question are thereby also considered to be the type and heading of the relation variable as such.

By way of elaboration on the foregoing ideas, suppose the relation variable S currently has the value shown in Fig. 2.1, and suppose we delete the tuples for suppliers in London:

```
DELETE ( S WHERE CITY = 'London' ) FROM S ;
```

Relation variable S now looks like this:

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

Conceptually, what's happened here is that the old value of S (a relation value, of course) has been replaced in its entirety by a new value (another relation value). Now, the old value (with five tuples) and the new one (with three) are rather similar, in a sense, but they certainly are different values. In fact, the DELETE just shown is logically equivalent to, and is indeed defined to be "shorthand" for,[4] the following *relational assignment:*

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

---

[3] The notion of *logical difference*, much appealed to by Darwen and myself in technical writings (especially ones to do with *The Third Manifesto*) derives from a dictum of Wittgenstein's: ***All logical differences are big differences.***

[4] "Shorthand" in quotation marks because in fact it's longer than what it's supposed to be short for. But that's partly due to the fact that I'm using an unconventional dialect of **Tutorial D** (the actual **Tutorial D** syntax for this DELETE would be DELETE S WHERE CITY = 'London'). The syntax I choose to use for DELETE (and INSERT) in this book is deliberately a trifle verbose, in order that the semantics might be absolutely clear.

Or equivalently:

```
S := S MINUS ( S WHERE CITY = 'London' ) ;
```

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

So 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. Logically speaking, in fact, relational assignment is the only update operator we really need (a point I'll elaborate on in the next section).

To sum up: There's a logical difference between relation values and relation variables. For that reason, I'll distinguish very carefully between the two from this point forward—I'll talk in terms of relation values when I mean relation values and relation variables when I mean relation variables. However, I'll also abbreviate *relation value,* most of the time, to just *relation* (exactly as we abbreviate *integer value* most of the time to just *integer*). And I'll abbreviate *relation variable* most of the time to **relvar**; for example, I'll say the suppliers-and-parts database contains three *relvars* (more specifically, three "real" or base relvars, so called to distinguish them from "virtual" relvars or views).

Base Relvar Definitions

Here for purposes of subsequent reference are **Tutorial D** definitions for the three base relvars in our running example:

```
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 ;
```

Note: For the purposes of this book, I choose to overlook the fact that **Tutorial D** as currently defined doesn't actually include any explicit FOREIGN KEY syntax.

RELATIONAL ASSIGNMENT

The **Tutorial D** syntax for relational assignment as such—i.e., as opposed to one of the shorthands such as INSERT or DELETE—takes the following generic form:

```
R := rx
```

Here *R* is a relvar reference (i.e., a relvar name, syntactically speaking), and *rx* is a relational expression, denoting a relation *r*, say, of the same type as relvar *R*. Now, it's easy to see (thanks to David McGoveran for this observation) that any such assignment is logically equivalent to one of the following form—

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

—where:

- *r* is the "old" value of *R*
- *d* is a set of tuples to be deleted from *R* (the "delete set")
- *i* is a set of tuples to be inserted into *R* (the "insert set")[5]
- $d \subseteq r$ (i.e., *d* is a subset of *r*, meaning we aren't trying to delete any tuples that don't exist)
- *i* and *r* are disjoint (meaning we aren't trying to insert any tuples that do already exist)
- *d* and *i* are disjoint a fortiori
- *d* and *i* are well defined and unique[6]

These points can conveniently be illustrated by means of a Venn diagram (see Fig. 2.2 overleaf). *Explanation:* In that diagram, *r*, *d*, and *i* are as above, and *u* is the universal relation of the pertinent type (in other words, *u* is the relation whose body consists of all tuples with the same heading as *R*, including, of course, those tuples that currently appear in *R*). Note that the difference $u - r$ is the absolute complement of *r* (in other words, $u - r$ is the relation whose body consists of all tuples with the same heading as *R* that don't currently appear in *R*).

---

[5] Technically, of course, *d* and *i* aren't just "sets of tuples" but in fact relations, of the same type as *r*.

[6] See Appendix A for a proof of this claim.

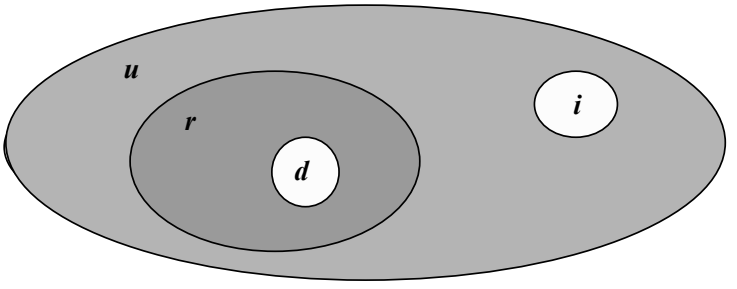


Fig. 2.2: The delete set *d* and the insert set *i*

Of course, the delete set *d* might be empty, in which case the original assignment is effectively a pure INSERT operation. Or the insert set *i* might be empty, in which case it's effectively a pure DELETE operation. Or they might both be empty, in which case the assignment overall effectively degenerates to the "no op" *R* := *R*.

It follows from all of the above that the original assignment *R* := *rx* is in fact logically equivalent to the following *multiple* assignment (see "Multiple Assignment" on page 18 for further discussion of this latter construct):[7]

```
DELETE d FROM R , INSERT i INTO R
```

As a consequence, although I said earlier that assignment as such is really the only update operator we need, we can always (and for intuitive reasons, at least, it turns out to be convenient to do this) think in terms of DELETE and INSERT operations instead. That is, faced with an arbitrary and possibly multiple assignment to relvar *R*—including in particular the case where the assignment in question is formulated as an explicit UPDATE on *R*—I propose that we map that assignment to a DELETE on *R* and an INSERT on *R*, where the delete set *d* and the insert set *i* are well defined, disjoint, and unique. *Note:* Because of this state of affairs, in what follows I'll discuss explicit UPDATE operations only when there's something interesting to say about them. Also, I'll ignore for the most part the practical problem of actually computing *d* and *i*. My primary concern, in this book as in most of my other writings, is always to get the theory right first before worrying about issues of implementation. Of course, I don't mean to suggest by these remarks that I think implementation issues are unimportant. *Au contraire*, in fact—checking feasibility of implementation is crucial to ensuring the correctness of the theory.

[7] The actual **Tutorial D** syntax for this multiple assignment would be DELETE *R d*, INSERT *R i*.

A Note on Syntax

To repeat, the assignment *R* := *rx* is logically equivalent to the following:

```
DELETE d FROM R , INSERT i INTO R
```

Note carefully that it doesn't matter here whether the DELETE or the INSERT is "done first," precisely because *d* and *i* are disjoint. (I'll have more to say in a few moments on this question of the order in which the individual operations are done in the context of a multiple assignment.) Thus, we can equally well say that the original assignment is logically equivalent to either of the following:

```
WITH ( R := R MINUS d ) : INSERT i INTO R

WITH ( R := R UNION i ) : DELETE d FROM R
```

We could even say it's logically equivalent to either of the following:

```
WITH ( DELETE d FROM R ) : INSERT i INTO R

WITH ( INSERT i INTO R ) : DELETE d FROM R
```

Note: If you're not familiar with WITH specifications as illustrated above, I refer you to *SQL and Relational Theory*.

In fact we could go further. **Tutorial D** additionally supports variants on DELETE and INSERT called I_DELETE ("included DELETE") and D_INSERT ("disjoint INSERT"), respectively. With I_DELETE, it's an error to attempt to delete a tuple that doesn't exist (i.e., one that's not present in the first place); likewise, with D_INSERT, it's an error to attempt to insert a tuple that does exist (i.e., one that's already present). Thus, we could actually say the original assignment *R* := *rx* is logically equivalent to either of the following:

```
I_DELETE d FROM R , D_INSERT i INTO R

D_INSERT i INTO R , I_DELETE d FROM R
```

For reasons of simplicity and familiarity, however, I'll stay with the conventional DELETE and INSERT operators throughout the remainder of this book, and I'll assume throughout the text—*please note carefully!*—that an attempt to delete a tuple that doesn't exist isn't an error, and nor is an attempt to insert one that does.

Multiple Assignment

The Third Manifesto requires—and as already mentioned, **Tutorial D** of course supports—a multiple form of assignment, which allows any number of individual assignments to be performed "at the same time." For example:

```
DELETE ( S  WHERE SNO = 'S1' ) FROM S ,
DELETE ( SP WHERE SNO = 'S1' ) FROM SP ;
```

Explanation: First, note the comma separator, which means the two DELETEs are part of the same overall statement. Second, as we know, DELETE is really assignment, and the foregoing "double DELETE" is thus just shorthand for a double assignment of the following general form:

```
S := ... , SP := ... ;
```

This latter statement assigns one value to relvar S and another to relvar SP, both as part of the same overall operation. In outline, the semantics of multiple assignment are as follows:

- First, the source expressions on the right sides of the individual assignments are evaluated.
- Second, those individual assignments are executed.[8]

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 (you can even think of them as being executed in parallel, or "simultaneously," if you like). Moreover, since multiple assignment is considered to be a semantically atomic operation, no integrity checking is performed "in the middle of" any such assignment (indeed, this state of affairs is the main reason why the *Manifesto* requires support for the operation in the first place). *Note:* Integrity constraints are discussed in detail later in this chapter.

Semantics Not Syntax

I've said that every relational assignment is equivalent to a DELETE plus an INSERT, where the delete set *d* and the insert set *i* are well defined, disjoint, and unique. It's important to understand, however, that two distinct assignments can use quite different syntax and yet correspond to the same delete set and same insert set, as I'll now show (thanks to Hugh Darwen for this example).

---

[8] This definition requires some refinement in the case where two or more of the individual assignments specify the same target variable, but that refinement needn't concern us yet. See Appendix A for further explanation.

Consider a relvar R with just two attributes, K and A. Let {K} be the sole key; further, let K and A both be of type INTEGER, and let R contain just two tuples, (1,2) and (3,-2).[9] Now consider the following explicit UPDATE operations:

```
UPDATE R : { K := K + A } ;

UPDATE R : { A := -A } ;
```

Note in particular that the first of these is a "key UPDATE" and the second isn't; thus, if a compensatory action of the form ON UPDATE K … had been defined (which it well might have been, in an SQL context), that action would presumably be invoked in connection with the first UPDATE and not the second. And yet it's easy to see that, given the specified initial value for R, the two UPDATEs are effectively equivalent—in fact, they're both equivalent to this explicit assignment:[10]

```
R := RELATION { TUPLE { K 1 , A -2 } , TUPLE { K 3 , A 2 } } ;
```

In other words, the delete set *d* is the same for both of the original UPDATEs, and so is the insert set *i*. (*Exercise:* What are they, exactly?) Clearly, therefore, what we want is for compensatory actions, if any, to be driven by the applicable delete set and insert set as such, not by the arbitrary choice of syntax in which the pertinent update happens to have been formulated.

INTEGRITY CONSTRAINTS

Every relvar is subject to a set of integrity constraints, or just constraints for short. First of all, as we know from the section "Relations and Relvars," any given relvar is constrained to be of a certain type (more specifically, a certain relation type)—namely, the type specified when the relvar in question is defined. For example, here again is the definition of the suppliers relvar S:[11]

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

As you can see, this definition states explicitly that relvar S is of type RELATION {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}. What's more, it's immediate from that specification that attributes SNO, SNAME, STATUS, and CITY are of types CHAR, CHAR,

---

[9] Here and elsewhere in this book I adopt this simplified notation for tuples in the interest of readability.

[10] The expression on the right side of this assignment is a *relation selector invocation*—in fact, a relation *literal*. Refer to *SQL and Relational Theory* if you need further explanation of such matters.

[11] Relvar S is a base relvar, of course. For views, see Chapter 3.

INTEGER, and CHAR, respectively. *Note:* These latter constraints—the ones on the individual attributes—are examples of what are sometimes called *attribute* constraints.

To repeat, every relvar (and every attribute, a fortiori) is constrained to be of some type. But relvars in general are also subject to numerous additional constraints, constraints that are typically formulated in **Tutorial D** by means of explicit CONSTRAINT statements.[12] The following simple examples are, I hope, self-explanatory (further explanation can be found if you need it in *SQL and Relational Theory*):

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 ) ;

CONSTRAINT CX2 IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ;

CONSTRAINT CX3 IS_EMPTY ( ( S JOIN SP )
                            WHERE STATUS < 20 AND PNO = 'P6' ) ;
```

Now, *The Third Manifesto* requires all constraints to be satisfied at statement boundaries ("immediate checking"). In other words, constraints are checked, logically, at the end of any statement that has the potential to violate them.[13] Loosely, we can say they're checked "at semicolons." Thus, contrary to the SQL standard and certain SQL products, integrity checking is never deferred to end of transaction or COMMIT time.

Finally, note that it's sometimes convenient to talk in terms of "the" (total) constraint—sometimes the total *relvar* constraint, for emphasis—for a given relvar *R*, meaning the logical AND of all of the individual constraints that mention relvar *R*. It's also sometimes convenient to talk in terms of "the" (total) constraint—sometimes the total *database* constraint, for emphasis—for a given database, meaning the logical AND of all of the constraints that mention any relvar in that database.

Updating Is Set At a Time

The point is well known, but worth stressing nevertheless, that updating in the relational model is always set at a time (better: *relation* at a time). Loosely speaking, in other words, INSERT inserts a set of tuples into the target relvar; DELETE deletes a set of tuples from the target relvar; UPDATE updates a set of tuples in the target relvar; and, more generally, relational assignment assigns a set of tuples to the target relvar. Of course, it's true that we often talk in terms of (for example) inserting some individual tuple as such—indeed, I'll do so myself in this book from time to time—but such a manner of speaking is really sloppy (though convenient). Be that as it may, the significance of the point for present purposes is that no integrity constraint checking

---

[12] The exceptions are constraints formulated by means of KEY and FOREIGN KEY specifications—but even these are essentially just shorthand for constraints that can be expressed, albeit more longwindedly, using explicit CONSTRAINT statements. See *SQL and Relational Theory* for further discussion of this point.

[13] I'm simplifying slightly here: Constraints to the effect that some relation selector invocation does indeed yield a relation of the required type are checked "even more immediately," as part of the process of evaluating the relation selector invocation in question. Again, see *SQL and Relational Theory* for further explanation.

must be done until *all* operations of an updating nature (including compensatory actions, if any) have been completed. In other words, a set level update must *not* be treated, logically, as a sequence of individual tuple level updates.

Two Important Principles

Now I'm in a position to articulate two important principles that underpin the whole notion of updating in general (including view updating in particular). The first is known as **The Golden Rule:**

- **Definition: The Golden Rule** states that no database is ever allowed to violate its own total database constraint.

Of course, it's an immediate consequence of this rule is that no relvar is ever allowed to violate its own total relvar constraint, either. All of which simply amounts to saying, as I've already stressed, that constraints must be satisfied at statement boundaries. Note in particular that, to repeat, the rule applies to views just as much as it does to base relvars.

The second principle is called *The Assignment Principle*:

- **Definition:** *The Assignment Principle* states that after assignment of value *v* to variable *V*, the comparison *v* = *V* must evaluate to TRUE.

Now, this principle actually applies to assignments of all kinds—in fact, as I'm sure you realize, it's more or less the *definition* of assignment—but for present purposes it's sufficient to note that it applies to relational assignments in particular. Again note that (of course) the principle applies to views just as much as it does to base relvars.

RELVAR PREDICATES

In Chapter 1, when I introduced the running example (i.e., the suppliers-and-parts database), I said this among other things:

> Table S represents *suppliers under contract*. Each supplier has one supplier number (SNO), unique to that supplier; one name (SNAME), not necessarily unique (though the sample values shown in Fig. 1.1 do happen to be unique); one status value (STATUS); and one location (CITY).

Of course, what in that chapter I called "table S" I'd much rather call "relvar S," but that's not the point I want to make here. Rather, the point is that, first, the foregoing text represents the *meaning* of relvar S as understood by the user; second, the term used in logic for such meanings is *predicate* (or sometimes *interpretation* or *intended interpretation*, but I'll mostly stay with *predicate* in this book). In other words, we can say of every relvar that it has an associated

predicate, called the *relvar predicate* for the relvar in question, and that relvar predicate is, in essence, how the relvar in question is understood by the user. Here by way of example is the way I would prefer to state the predicate for relvar S:

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

And here for the record are the predicates for relvars P and SP:

> P: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY.*
>
> SP: *Supplier SNO supplies part PNO in quantity QTY.*
>
> *Aside:*[14] Perhaps I should say a little more about the way I use the term *predicate* in this book. First of all, you're very likely familiar with the term already, since SQL uses it extensively to refer to boolean or truth valued expressions (it talks about comparison predicates, IN predicates, EXISTS predicates, and so on). However, while this usage on SQL's part isn't exactly incorrect, it does usurp a very general term—one that's extremely important in database contexts—and give it a rather specialized meaning, which is why I prefer not to follow that usage myself.
>
> Second, I should explain in the interest of accuracy that a predicate isn't really a natural language sentence as such, like the ones shown above; rather, it's the assertion made by such a sentence. For example, the predicate for relvar S is what it is, regardless of whether it's expressed in English or Spanish or whatever. For simplicity, however, I'll assume in what follows that a predicate is indeed just a sentence per se, expressed (usually) in some natural language. (Analogous remarks apply to propositions also—see below.)
>
> Finally, I've now explained what I mean by the term; however, you should be aware that, the previous paragraph notwithstanding, there seems to be little consensus, even among logicians, as to *exactly* what a predicate is. In particular, some writers regard a predicate as a purely formal construct that has no meaning in itself, and regard what I've referred to above as the intended interpretation as something distinct from the predicate as such. I don't want to get into arguments about such matters here; for further discussion, I refer you to the article "What's a Predicate?" in *Database Explorations: Essays on The Third Manifesto and Related Topics,* by Hugh Darwen and myself (Trafford, 2010). *End of aside.*

---

[14] This aside is taken, lightly edited, from *Database Design and Relational Theory*. *Note:* I remind you from the preface that throughout this book I use "*Database Design and Relational Theory"* as an abbreviated form of reference to my book *Database Design and Relational Theory: Normal Forms and All That Jazz* (O'Reilly, 2012).

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 for relvar S, 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. We obtain:

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

This latter sentence is in fact a *proposition*, which in logic is a sentence that has no parameters and evaluates unequivocally to either true or false. (In the case at hand, of course, the proposition is a true one, or so at least we assume for the sake of the example.) Which brings me to another important principle, *The Closed World Assumption:*

- **Definition:** Let relvar *R* have predicate *P*. Then *The Closed World Assumption* (CWA) says (a) if tuple *t* appears in *R* at time *T*, then the instantiation *p* of *P* corresponding to *t* is assumed to be true at time *T*; conversely, (b) if tuple *t* has the same heading as *R* but doesn't appear in *R* at time *T*, then the instantiation *p* of *P* corresponding to *t* is assumed to be false at time *T*. In other words, tuple *t* appears in relvar *R* at a given time if and only if it satisfies the predicate for *R* at that time.[15]

For example, with reference to Fig. 2.1 once again, it's currently a "true fact" that supplier S1 supplies part P1 in quantity 300, because the tuple (S1,P1,300) currently appears in relvar SP. But if we were to delete that tuple from that relvar, we would be saying (in effect) that *it's no longer the case that* supplier S1 supplies part P1 in quantity 300.

Now, so far I've discussed predicates in connection with relvars specifically. In fact, however, all of the foregoing notions extend in a natural way to arbitrary relational expressions. For example, consider the projection of suppliers on all attributes but CITY (note the **Tutorial D** syntax for projection):[16]

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

[15] I'd like to stress that "if and only if." It has at least one important corollary. To be specific, let relvars *R1* and *R2* have predicates *P1* and *P2*, respectively; then if *P1* and *P2* are both satisfied by the same tuple *t*, it follows that *t* must appear in both *R1* and *R2*. (As a rule of thumb, it might be a good idea to ensure that *P1* and *P2* are specific enough to preclude the foregoing possibility. Note, however, that we'll see some deliberate violations of this suggested discipline in Chapters 4 and 9-11.)

[16] I remark in passing that we find it convenient to give projection the highest precedence of all of the relational operators in **Tutorial D**.

This expression denotes a relation containing all tuples of the form (*s*,*n*,*t*) such that a tuple of the form (*s*,*n*,*t*,*c*) currently appears in relvar S for some CITY value *c*. In other words, the result contains all and only those tuples that correspond to currently true instantiations of the following predicate:

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

This predicate thus represents the meaning of the expression—the relational expression, that is—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 CITY such that*. *Note:* A possibly clearer way of making the same point (i.e., that the predicate has three parameters, not four) is to observe that the predicate in question is effectively 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, in accordance with the semantics of the relational operations involved in that expression.[17]

> *Aside:* To repeat, the projection of suppliers on all attributes but CITY can be expressed in **Tutorial D** as follows: S {SNO,SNAME,STATUS}. For convenience, however, **Tutorial D** also allows projections to be expressed in terms of the attributes to be removed instead of the ones to be kept. Thus, for example, the expression S {ALL BUT CITY} is equivalent to the one just shown. An analogous remark applies, where it makes sense, to all of the operators in **Tutorial D**. *End of aside.*

Now, relvar predicates as I've described them so far are quite informal; in fact, they're essentially nothing more than a representation in natural language of the fact that the relvar in question has certain attributes (equivalently, is of a certain type). But we could use a somewhat more formal style if we wanted to. For example, given that the suppliers relvar S has attributes SNO, SNAME, STATUS, and CITY, and given also that {SNO} is the sole key for that relvar, we might say the corresponding predicate looks something like this:

---

[17] Thus, we can extend *The Closed World Assumption* as follows: A proposition *p* is considered to be true at a given time if it corresponds to a tuple *t* and that tuple *t* either (a) appears in some database relvar at that time (as already explained) or (b) appears in some relation that can be derived from the relations that are the values of those database relvars at that time. In other words (loosely): Everything stated or implied by the database is true; everything else is false.

```
is_entity  ( SNO ) AND
has_SNAME  ( SNO , SNAME ) AND
has_STATUS ( SNO , STATUS ) AND
has_CITY   ( SNO , CITY )
```

Suppose tuple *t* currently appears in relvar S. Then the foregoing predicate asserts that:

- The SNO value in *t* identifies a certain entity.
- That entity has an SNAME property, the value of which is given by the SNAME value in *t*.
- That entity also has a STATUS property, the value of which is given by the STATUS value in *t*.
- That entity also has a CITY property, the value of which is given by the CITY value in *t*.

Of course, the system still doesn't know the entity in question (the entity identified by SNO) is in fact a supplier in the real world—a supplier "under contract," in fact—nor does it know what it means for something to have an SNAME property or a STATUS property or a CITY property.

In a similar manner, we might say the predicate for the projection of suppliers on all attributes but CITY is:

```
is_entity  ( SNO ) AND
has_SNAME  ( SNO , SNAME ) AND
has_STATUS ( SNO , STATUS ) AND
EXISTS CITY ( has_CITY ( SNO , CITY ) )
```

Finally, I'd like to close this section by confessing that everything I've said so far regarding relvar predicates in general has deliberately been slightly simplified. Of course, I believe the treatment is adequate for present purposes; for further specifics, however—"the true story," as it were, or at least a slightly truer one—I refer you to Chapter 4 ("The Closed World Assumption") of my book *Logic and Databases: The Roots of Relational Theory* (Trafford, 2007).

MATCHING, NOT MATCHING, AND EXTEND

Throughout the bulk of this book, I will of course be discussing the operators of the relational algebra in some detail, and for the most part I'll take it you're reasonably familiar already with the operators in question (though definitions can be found if you need them in Appendix B). But there are certain useful operators, supported by **Tutorial D** in particular, that I do need to appeal to fairly frequently but you might not be familiar with, since they're not discussed in most of the usual textbooks (probably because they're not supported by SQL—at least, not directly). The first is MATCHING. Here's a definition:

- **Definition:** The expression *r1* MATCHING *r2* returns a relation equal to the relation returned by the expression *r1* JOIN *r2*, projected back on the attributes of *r1*.

Here's an example ("Get suppliers who currently supply at least one part"):

```
S MATCHING SP
```

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

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

Tutorial D also supports a negated form called NOT MATCHING. Here's the definition:

- **Definition:** The expression *r1* NOT MATCHING *r2* returns a relation equal to the relation returned by the expression *r1* MINUS (*r1* MATCHING *r2*).[18]

Here's an example ("Get suppliers who currently supply no parts at all"):

```
S NOT MATCHING SP
```

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

SNO	SNAME	STATUS	CITY
S5	Adams	30	Athens

The other operator I want to describe briefly here is EXTEND. EXTEND comes in two forms. Here's a definition of the first:

- **Definition:** Let relation *r* not have an attribute called *A*. Then the expression EXTEND *r* : {*A* := *exp*} returns a relation with heading the heading of *r* extended with attribute *A* and

[18] I remark in passing that the well known relational difference operator—MINUS in **Tutorial D**—is a special case of NOT MATCHING. To be specific, *r1* NOT MATCHING *r2* degenerates to *r1* MINUS *r2* , as you can easily confirm for yourself, if relations *r1* and *r2* are of the same type. In a sense, therefore, NOT MATCHIG is actually more fundamental than MINUS is.

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 *exp* on that tuple of *r*.

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 }
```

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

The second form of EXTEND is aimed primarily at what are sometimes called "what if" queries. In other words, it can be used to explore the effect of making certain changes without actually having to make (and subsequently unmake, possibly) the changes in question. Here's an example ("What if part weights were given in grams instead of pounds?"):

```
EXTEND P : { WEIGHT := WEIGHT * 454 }
```

The point here is that the target attribute in the assignment in braces here isn't a "new" attribute as it was in the previous example; instead, it's an attribute that already exists in the specified relation (that relation being the current value of relvar P, in the case at hand). The expression overall is defined to be equivalent to the following:

```
( ( EXTEND P : { GMWT := WEIGHT * 454 } ) { ALL BUT WEIGHT } )
                                            RENAME { GMWT AS WEIGHT }
```

Note: As you can see, this expansion makes use of the **Tutorial D** RENAME operator. I think that operator is more or less self-explanatory, but you can find a definition if you need it in Appendix B. For further discussion, see *SQL and Relational Theory*.

For completeness, here's a definition of this second form of EXTEND:

- **Definition:** Let relation *r* have an attribute called *A*. Then the expression EXTEND *r* : {*A* := *exp*} returns a 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*.

DATABASES AND DBVARS

This is the final section of this chapter. In it, I want to draw your attention to the important fact that just as there's a logical difference between relation values and relation variables (relvars), so there's also a logical difference between database values and database variables, or what might be called *dbvars*. Here's a quote from the *Manifesto* book:

> The first version of this *Manifesto* distinguished databases per se (i.e., database values) from database variables ... [It also] 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 interests of familiarity, to revert to more traditional terminology.[19]

It follows 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 this section.) For example, the **Tutorial D** statement

```
DELETE ( SP WHERE QTY < 150 ) FROM SP ;
```

"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" (namely, the applicable relvars), such a manner of speaking is really only approximate, and is 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 (or "tuplevar") 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 } }
```

[19] In other words, we went on to use the term *database* to mean sometimes a database variable and sometimes a database value, and we didn't use the terms *database variable* or *dbvar* at all. However, it turned out later that, for reasons I don't want to go into here, this was a very bad decision on our part. (Doing something you know is logically wrong is usually a bad idea.) As for the way I plan to use this terminology in the present book, please see the remainder of this section.

Let's agree, just for the moment, to call the suppliers-and-parts dbvar (or tuplevar, if you prefer) SPDB.[20] Then the DELETE statement shown above can be regarded as shorthand for the following *database* (and hence 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 denotes a tuple with three attributes called S, P, and SP, each of which is relation valued.[21] Within that tuple, the value of attribute S is the current value of relvar S; the value of attribute P is the current value of relvar P; and the value of attribute SP is the current value of relvar SP, minus those tuples for which the quantity is less than 150.

By way of another example, here again is the "double delete" I used earlier in this chapter to introduce the concept of multiple assignment:

```
DELETE ( S  WHERE SNO = 'S1' ) FROM S ,
DELETE ( SP WHERE SNO = 'S1' ) FROM SP ;
```

Observe now that this relational assignment—which is, of course, a multiple assignment—is logically equivalent to the following *single* database (and hence tuple) assignment:

```
SPDB := TUPLE { S  ( ( S FROM SPDB ) WHERE NOT ( SNO = 'S1' ) ) ,
                P  ( P FROM SPDB ) ,
                SP ( ( SP FROM SPDB ) WHERE NOT ( SNO = 'S1' ) ) } ;
```

In other words, a multiple assignment in which the target variables are all, specifically, relvars within the same database (or the same dbvar, rather) is merely a syntactic device that allows us to formulate what is logically a *database* assignment as a collection of separate *relational* assignments (see further discussion below). And a "single" relational assignment—i.e., a "multiple" relational assignment consisting of just one individual assignment, to one individual database relvar—is nothing but a special case, of course. ***Fundamentally, in fact, dbvars are the only true variables so far as database updating is concerned*;** that is, database updates are really *always* updates to some dbvar as such, and updates via a single database relvar are just a special case.

It follows from the all of the above that even a "single" relational assignment is really a database assignment, if the target relvar is a database relvar (i.e., if that target relvar is a relvar in some database). What's more, of course, that database assignment, like all assignments, is

---

[20] If we're to be able to write explicit database assignments as in this example, then dbvars will certainly have to have user visible names, which in **Tutorial D** (at least as currently defined) they don't.

[21] In fact the expression in question is a tuple selector invocation. Once again, refer to *SQL and Relational Theory* if you need further explanation of such matters.

subject to *The Assignment Principle* and—because it's a database assignment in particular—also to **The Golden Rule**.

The foregoing ideas are so important that it's worth going over them all again in slightly different words. First of all, then, a dbvar is a tuple variable, and a database (i.e., the value of some given dbvar at some given time) is a tuple in which all of the attributes are relation valued. Second, given a relational assignment of the form

```
R := rx
```

(where *R* is a relvar reference—i.e., a relvar name, syntactically speaking—that identifies some database relvar and *rx* is a relational expression, denoting a relation *r* of the same type as *R*), that relvar reference *R* is really a *pseudovariable* reference (see the remark immediately following this paragraph). In other words, the relational assignment is shorthand for an 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 as such at all; rather, they're a convenient fiction that gives the illusion that the database—or dbvar, rather—can be updated piecemeal, individual relvar by individual relvar. And it further follows that, in a sense, "relational assignment" (multiple or otherwise) is also a convenient fiction; to be specific, it's an operator that lets us pretend that updating a dbvar can be thought of as a collection of updates on individual relvars within that dbvar.[22]

> *A remark on pseudovariables:* Essentially, a pseudovariable reference is a construct—or a symbolic name for such a construct—that looks like an operational expression but, unlike such an expression, doesn't denote a value; instead, it appears in the target position within some assignment, where it denotes "part of a variable." For example, let X be a variable of type CHAR, and let the current value of X be 'antimony'. Then the assignment SUBSTR(X,5,3) := 'nom' has the effect of "zapping" the fifth through seventh character positions within X, replacing 'mon' by 'nom' and thereby replacing the "old" value of X overall ('antimony') by a "new" value ('antinomy'). The construct SUBSTR(X,5,3) on the left side of that assignment is a pseudovariable reference, and the assignment overall is shorthand for X := SUBSTRX(X,1,4) || 'nom' || SUBSTR(X,8,1), where the symbol "||" denotes string concatenation. *End of remark*.

All of that being said, I've decided, slightly against my own better judgment, not to use the terminology of database variables and "dbvars" in the remainder of this book—at least, not very much—but rather to stay with the familiar term "database" for both purposes, hoping that my intended meaning will always be clear from the context. *Caveat lector*.

---

[22] It follows from this discussion that multiple relational assignment in particular is extremely important (absent explicit support for database assignment as such); in fact, it's going to turn out to be absolutely crucial to certain of the view updating rules to be discussed in subsequent chapters. Given this state of affairs, therefore, I have to say that—forgive me for pointing out the obvious here—it's unfortunate to say the least that SQL doesn't support that operator.

Chapter 3

The View Concept:

A Closer Look

Distance lends enchantment to the view
—Thomas Campbell: *Pleasures of Hope* (1799)

Every scientific discipline has its share of unsolved problems. In mathematics, for example, there's the Riemann Hypothesis, which nobody has managed to prove or disprove in over 150 years; in computer science, there's the "P = NP?" question, which is still open after some 40 years; in physics and cosmology, there's the long search for a "theory of everything," which remains—in many people's opinion, though possibly not in everyone's—just that, a search; and in database theory, there's the problem of view updating. Now, I obviously don't mean to suggest that the problem of view updating is in the same league as the Riemann Hypothesis or the "P = NP?" question or some hypothetical "theory of everything";[1] however, I do claim it's of considerable practical importance and much theoretical interest. In this chapter, therefore, I want to lay some of the groundwork that's necessary for a systematic attack on that problem.

I begin with the trite observation that a view is a relvar—a virtual relvar, to be precise, or in other words a relvar that "looks and feels" just like a real, or base, relvar but (unlike a real or base relvar) doesn't exist independently of other relvars; rather, it's defined in terms of, or derived from, such other relvars. Here's a definition:

> **Definition:** A view is a derived, virtual relvar. The value of view *V* at time *T* is the result of evaluating a certain relational expression (the *view defining expression*) at that time *T*. That expression is specified when *V* is defined and must mention at least one relvar.[2]

---

[1] One of my reviewers claimed it most certainly is in the same league. Maybe it is. I don't want to argue the point.

[2] It must mention at least one relvar because otherwise the view wouldn't be a relvar at all but merely a relation constant (see further discussion in Chapter 13).

Here's an example (it's a **Tutorial D** definition of the "London suppliers" view LS from Chapter 1):

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

Now, it's well known that a view can be thought of as a kind of "window into" the relvar or relvars in terms of which it's defined, in the sense that operations on the view are "really" operations on those underlying relvars. In other words, the DBMS is supposed to support user operations on a view, be they retrievals or updates, by mapping them—the operations, that is—into suitable operations on the base relvars in terms of which the view in question is ultimately defined. (I say *ultimately defined* here because if views really do behave just like base relvars, then one thing we can certainly do is define further views on top of them, and so on to any depth.) Thus, for example, the following expression—

```
LS WHERE STATUS > 10
```

(which can be thought of as representing a retrieval operation on view LS)—maps to:

```
S WHERE CITY = 'London' AND STATUS > 10
```

And the following update operation on LS (actually a delete)—

```
DELETE ( LS WHERE STATUS > 10 ) FROM LS ;
```

—maps to:

```
DELETE ( S WHERE CITY = 'London' AND STATUS > 10 ) FROM S ;
```

From all of the above, it's clear there's a logical difference between views and base relvars. By *The Principle of Interchangeability*, however, that difference isn't one that should affect the user—the aim, to repeat, is to make views "look and feel" just like base relvars as far as the user is concerned, even with respect to updating. (The distinction between base relvars and views is relevant to the DBA and DBMS but not, to repeat, to the user. At least, that's the intent.) In other words, the user of a given view should ideally be unaware that his or her operations on that view are being mapped to operations on the underlying relvar(s); ideally, in fact, that user shouldn't even have to be aware that the view is indeed a view, nor indeed that the underlying relvars even exist.

Incidentally, it follows from the foregoing that the terminology of "real" vs. "virtual" relvars is somewhat misleading—views are actually no less (and no more!) "real" than base relvars are—but I'll stay with it for now, in part because it's actually reflected in the **Tutorial D** syntax for defining a view, as we've already seen. However, I'll have a little more to say about this perhaps unfortunate choice of terminology, and syntax, later in the chapter.

The View Update Problem

I can now state for the record exactly what the view update problem consists of: Given an arbitrary update operation—perhaps I should say an arbitrary update *request*—on an arbitrary view, the problem is to determine what operations need to be performed on the relvars in terms of which the view in question is defined. (Note the recursive nature of this problem. I'll have more to say on this particular point in Chapter 14.) Of course, I'm assuming here, tacitly, that such operations on the underlying relvars (a) do in fact exist and moreover (b) are well defined. I'll discuss these issues too in the chapters to come.

Note: Perhaps I should say too that of course I'm assuming the system does in fact support views. Personally I do think views should be supported, for reasons I'll explain in the section "Data Independence" on page 34; further, it's my position that since (a) views are relvars, and (b) relvars are variables, and (c) variables are updatable, then (d) views must be updatable. But I suppose you could take the position that there's no absolute requirement that views be supported in the first place! If that's your position, you might as well stop reading right now—if you've even got this far, I suppose I should add.

VIEWS ARE PSEUDOVARIABLES

Now I want to focus on the point that views are indeed relvars; in other words, as I've just said, they're variables,[3] which means by definition that they must be updatable. (To be a variable is to be updatable; equally, to be updatable is to be a variable.) In other words, it must in general be possible to make a view the target in a relational assignment. By way of example, given view LS as defined in the previous section, the following DELETE should be legal:

```
DELETE ( LS WHERE SNO = 'S1' ) FROM LS ;
```

Of course, this DELETE is really "shorthand" for the following relational assignment:

---

[3] At least inasmuch as the underlying base relvars are! See the remarks on this point in the final section of Chapter 2.

```
LS := LS WHERE NOT ( SNO = 'S1' ) ;
```

And this assignment is shorthand in turn for the following expanded version (which is, however, not legal syntax in **Tutorial D** as currently defined):

```
( S WHERE CITY = 'London' ) :=
                ( S WHERE CITY = 'London' ) WHERE NOT ( SNO = 'S1' ) ;
```

It follows that from an updating point of view, views act as *pseudovariables*[4] (recall from the previous chapter that a pseudovariable reference is basically a construct that looks like an operational expression but appears in the target position within an assignment, and that's exactly the situation we're dealing with here). But pseudovariables are variables (loosely speaking); please understand, therefore, that the term *variable* throughout the remainder of this book should be understood to include pseudovariables in particular.

DATA INDEPENDENCE

Consider the following quote from the abstract to Codd's famous 1970 paper ("A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM 13,* No. 6, June 1970):

> Future users of large data banks [*sic*] must be protected from having to know how the data is organized in the machine (the internal representation) ... Activities of users at terminals and ... application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed.

The first sentence here says we want physical data independence; the second says we want logical data independence as well. Indeed, data independence was one of Codd's primary reasons for introducing the relational model in the first place.[5] Physical data independence means we can change the way the data is physically stored and accessed without having to make changes in the way the data is perceived by the user; analogously, logical data independence means we can change the way the data is logically stored and accessed, again without having to make changes in the way the data is perceived by the user. And both kinds of independence

---

[4] So do base relvars, of course. Again, see the remarks on this point in the final section of Chapter 2.

[5] I note in passing that today's mainstream DBMS products still don't do a very good job on it.

translate into *protecting customer investment*—investment, that is, in existing training, existing applications, and existing databases (even in existing DBMS products, to some extent).

Turning now to views specifically: To the extent that logical data independence can be achieved at all, views provide the means for doing so. Let me elaborate. As mentioned in the preface to this book, we can assume without loss of generality that the database as seen by the user consists entirely of views. So if changes occur to the underlying base relvars, logical data independence is achieved by making appropriate changes to the mapping from the views as seen by the user to those underlying base relvars. Here's an example (in outline only, for brevity). Suppose the user initially sees a view V that's identical to the suppliers base relvar S:

```
VAR V VIRTUAL ( S ) ;
```

Now suppose for some reason—the precise reason isn't important for present purposes—we decide to replace base relvar S by base relvars LS ("London suppliers") and NLS ("non London suppliers"), thus:

```
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 } ;
```

What we do now is redefine view V as follows:

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

Assuming the system does correctly support operations—update operations in particular—on views (unfortunately a rather large assumption, given the state of today's products), the user of view V will thus be immune to this particular change to the underlying base data.

So now we see why the ability to update views is so important. In a way, in fact, it's a little hard to understand why there's so much skepticism surrounding—not to say downright opposition to—the idea that views should be updatable. The fact is, 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 *mus*t solve this problem, for otherwise we have to give up on the goal of data independence. (Logical data independence, that is—but as this brief discussion should be

sufficient to suggest, logical and physical data independence are basically the same problem, too; they too differ only in that they show up at different points in the overall system architecture).

Views Serve a Variety of Purposes

In *SQL and Relational Theory*, I said views serve two rather different purposes. To be specific, I said the following:

- The user who actually defines a given view *V* is, obviously, aware of the expression *exp*, say, that defines that view (i.e., the corresponding view defining expression). 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 perhaps unlikely to request any updates on *V*—though if such updates are requested, they must behave 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 that view defining expression; to that user, in fact, *V* is supposed to look and feel just like a base relvar, as I've said (see the foregoing discussion of logical data independence).

Of these two purposes, it's the second that's the important one, and the one that's the primary focus of this book. Note, however, that it subdivides into two further possibilities: The views seen by a given user can effectively be equivalent either (a) to the underlying base data in its entirety—see the discussion of information equivalence later in this chapter—or (b) to some proper subset of that underlying base data. The latter case constitutes one form of *information hiding*. And as noted in Chapter 1 (see the section "There's No Magic" in that chapter), a user from whom information is hidden is seeing only part of the picture, as it were, and it's only to be expected that there'll be certain operations that such a user can't be allowed to perform. We'll see several examples of such situations in later chapters.

Note: One specific reason for using views to hide information is *security*. For example, a rule to the effect that some user isn't allowed to see supplier cities can be enforced by making that user use a view that's the projection of relvar S on all attributes but CITY. In my opinion, however, there are better ways to manage authorization than using views: better ways, that is, of controlling who is allowed to do what to what data (see, e.g., my book *An Introduction to Database Systems*, 8th edition, Addison-Wesley, 2004). In other words, I regard this use of views as a red herring, and I don't propose to say much about it in this book.

A Psychological Mistake?

Let me focus on SQL specifically for a moment. The fact that, in SQL terms, views are supposed to look and feel just like base tables as far as the user is concerned suggests rather strongly that the syntactic distinction between CREATE TABLE and CREATE VIEW was always at least a psychological mistake. First of all, that keyword TABLE in CREATE TABLE refers to a base table specifically, thereby (most unfortunately!) lending weight to the widespread perception in the SQL world that views are somehow different from tables. To elaborate on this point briefly: As is well known, SQL documents—textbooks, product manuals, and indeed the SQL standard itself—all talk quite typically (fairly ubiquitously, in fact) in terms of "tables and views." But clearly, anyone who talks this way is under the impression that tables and views are different things, and probably 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 relvar); hence, we should be able to perform the same kinds of operations on views as we can on regular relvars, because views *are* "regular relvars." Throughout this book, therefore, I'll use the term *relvar* to refer to relvars in general—and if I want to talk about base relvars specifically or views specifically, I'll use the appropriate more specific term.

The second part of the mistake is this: The syntax of CREATE VIEW in SQL is such that the mapping between the view in question and the table(s) in terms of which it's defined—in other words, the pertinent view defining expression—is explicitly stated as part of the view definition. That mapping is thus explicitly visible to users who are exposed to that CREATE VIEW statement, despite the fact that those users have no logical need to see it (the mapping, that is). Ideally, in fact, they shouldn't be aware of it at all. One consequence is that users typically do know among other things that the view in question is indeed a view; more generally, they typically do know which tables are base ones and which ones are views. The problem is compounded by the fact that, ironically enough, the *structure* of the view, which users do need to be aware of, isn't explicitly defined in CREATE VIEW but has to be inferred from the mapping! (By *structure* here, I mean the names of the columns of the view and their corresponding data types—in other words, the heading—but *heading* isn't an SQL term. In fact, SQL doesn't seem to have a term for the concept at all.)

Now, lest I be accused of attacking SQL unfairly here, let me make it clear that the foregoing criticisms all apply, mutatis mutandis, to **Tutorial D** also (at least in its present incarnation). Let me also make clear what I would regard as an adequate fix for these problems:

- First, relvar definitions as perceived by the user should (of course) specify the pertinent heading but should not distinguish between base and virtual relvars in any way.

- Second, (a) the mapping between a view and the relvars in terms of which it's defined should be specified separately and concealed from the user, just as (b) the mapping between a base relvar and physical storage already is specified separately and concealed from the user.

HOW NOT TO DO IT

Before getting into further details of how I believe views ought to work, I think it's worth taking a quick look at "the state of the art" (such as it is) in this area. In fact, that "state of the art" leaves a lot to be desired; in fact, if truth be told, it's ad hoc in the extreme. Certainly this is the case in the SQL standard and in current commercial products, and (sadly) it's also the case in the technical literature, at least to some extent. Indeed, in the case of SQL specifically, it's quite usual to find that a given SQL DBMS will:[6]

- Prohibit updates on logically updatable views, or
- Permit updates on "logically nonupdatable" views,[7] or
- Implement view updates in a logically incorrect way, or
- Do all of the above

(and I'm tempted to say this last is the state of affairs most likely to be encountered in practice).

Let me pursue this point just a moment longer; more specifically, let me say a little more about the SQL standard in particular. I've already said the standard is deficient in this area. In fact, not only is it deficient, it's also extremely hard to understand!—indeed, it's even more

---

[6] The approach to updating views found in today's SQL DBMSs has its origins in the paper "Views, Authorization, and Locking in a Relational Data Base System," by Donald D. Chamberlin, James N. Gray, and Irving L. Traiger, in the proceedings of the 1975 National Computer Conference, Anaheim, Calif. (AFIPS Press, May 1975). With hindsight, it's tempting to suggest—perhaps a little unfairly—that some of the confusions commonly found in practice in connection with the view concept are reflected in the very title of this paper.

[7] You might raise an eyebrow at this point, given my claim in Chapter 1 to the effect that *all* views are logically updatable. But what I meant, and still do mean, by that claim is more specifically that all *relational* views are updatable. I neither know nor care whether nonrelational views—by which I mean ones involving nonrelational concepts such as duplicate rows and nulls—are updatable (though I'm pretty sure there's no way to update such views that can be defended as logically correct).

impenetrable in this area than it usually is. The following extract (which is quoted verbatim from SQL:2003, the 2003 version of the standard) 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, observe that the extract in question represents just one of the many rules that have to be taken in combination in order to determine whether some given view is updatable in SQL.
- The rules in question aren't all given 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—e.g., updatable columns, leaf generally underlying tables, <query term>s—that are defined in turn in still further portions of the standard.

What's more, the rule as defined by this particular extract doesn't even seem to make sense. 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 doesn't even mention any such *QE2* (?).

I haven't finished with my criticisms. The picture is complicated still further by the fact that SQL identifies four distinct cases; to be specific, a view in SQL can be *updatable*, *potentially updatable*, *simply updatable*, or *insertable into*. Now, the standard defines these terms formally but gives no insight into their intuitive meaning or why they were given those names. However, I can at least say that "updatable" refers to UPDATE and DELETE and "insertable into" refers to INSERT, and a view can't be insertable into unless it's updatable. But note the suggestion that some views might permit some updates but not others (e.g., DELETEs but not INSERTs), and the implication 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 at least potentially as further violations of *The Principle of Interchangeability*.

> *Aside:* There's an asymmetry here that seems intuitively odd. 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 then 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" (?). *End of aside*.

For what I take to be obvious reasons, I won't even attempt a precise characterization here of just which views SQL does regard as updatable. Loosely speaking, however, they do at least include the following:

1. Views defined as a restriction and/or projection (key preserving) 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

Regarding Case 1, 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 (i.e., not a UNION, INTERSECT, or EXCEPT expression) or (b) an "explicit table"—i.e., an expression of the

form TABLE <table name>—that's logically equivalent to such a simple SELECT expression. *Note:* I'll assume for simplicity in what follows that Case (b) is automatically converted to Case (a).

- The SELECT clause in that SELECT expression specifies ALL, not DISTINCT.
- After expansion of any "asterisk style" items, every item in the SELECT item commalist is a simple column reference (i.e., a column name, possibly dot qualified, and possibly accompanied by an 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 that table *T*.
- The SELECT expression contains no explicit GROUP BY or HAVING clause.

But even these limited cases are treated incorrectly, thanks to SQL's lack of understanding of (a) **The Golden Rule,** (b) *The Assignment Principle,* and (c) constraint inference (see the section "Constraints and Predicates" immediately following), and thanks also to (d) the fact that SQL permits nulls and duplicate rows. Surely we can do better than this?

CONSTRAINTS AND PREDICATES

A view is a relvar and is thus constrained to be of some relation type; in fact, the type of any given view *V* is, precisely, the type of the defining expression for *V*.[8] By way of example, here again is the **Tutorial D** definition of the view LS ("London suppliers"):[9]

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

[8] See Appendix B for an explanation of the type of an arbitrary relational expression.

[9] I'll continue to use conventional **Tutorial D** syntax for view definitions, despite my misgivings regarding that syntax as articulated in the section before last.

This view is a "restriction view"—its defining expression calls for invocation of a certain restriction operation on relvar S—and its type is therefore the same as that of relvar S, viz., RELATION {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}.

More generally, however, a view is a relvar and is thus subject to what in Chapter 2 I called a total relvar constraint (for the relvar in question). But the total relvar constraint that applies to a view *V* is, at least in part, a *derived* constraint: It's derived—again, at least in part—from the constraints for the relvars in terms of which *V* is defined, in accordance with the semantics of the relational operations involved in the view defining expression. In the case of view LS, for example, the total relvar constraint is the logical AND of:

a. The total relvar constraint for relvar S, and

b. The restriction condition (CITY = 'London') specified in the view defining expression.[10]

In other words, view LS is subject to all constraints that apply to relvar S—e.g., the constraint that {SNO} is a key—***and*** the constraint that the city is London. Note, therefore, that **The Golden Rule** applies to views just as much as it does to base relvars.

Let's agree to use the term *view constraint* to refer to any constraint that applies to some view. Now, even if view constraints can be derived as in the foregoing example, it doesn't follow that there's no need to declare them explicitly (for one thing, the system might not be "intelligent" enough to determine those constraints for itself, automatically). Thus, it should certainly be possible to declare explicit constraints for views. In particular, it should be possible (a) to include explicit KEY and FOREIGN KEY specifications in view definitions and (b) to allow the target relvar in a FOREIGN KEY specification to be a view. The following example illustrates both of the possibilities mentioned under (a) here:

```
VAR LS VIRTUAL ( S WHERE CITY = 'London' )
    KEY { SNO }
    FOREIGN KEY { SNO } REFERENCES S ;
```

Next, a view is a relvar and therefore has a relvar predicate, in which the parameters correspond one to one to the attributes of the relvar—i.e., the view—in question. But 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

---

[10] In general, it should be clear that the relation denoted at any given time by the restriction expression *rx* WHERE *bx* must satisfy all constraints that apply to the relation denoted by the relational expression *rx* ***and*** the constraint denoted by the boolean expression *bx* (in other words, WHERE is AND, loosely speaking). *Note:* The constraint represented by *bx* here is a *restriction condition*. See Appendix B for a precise definition of this concept.

operations involved in the view defining expression. In fact, we already know this, because we know from Chapter 2 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. That view is a restriction of relvar S, and its predicate is therefore the logical AND of the predicate for S and the specified 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 CITY London.*

Note, however, that this latter form obscures the fact that CITY is a parameter. Indeed it is a parameter, but the corresponding argument is always the constant London. (As noted in Chapter 1, a more realistic version of view LS would probably project away the CITY attribute, precisely because of this fact. I choose not to do so here in order to keep the example simple. However, I'll have more to say about this possibility in Chapter 5.)

> *Aside:* If you prefer the somewhat more formal style for predicates described in Chapter 2, you might alternatively say the predicate for view LS looks something like this:
>
> ```
> is_entity  ( SNO ) AND
> has_SNAME  ( SNO , SNAME ) AND
> has_STATUS ( SNO , STATUS ) AND
> has_CITY   ( SNO , CITY ) AND
> CITY = 'London'
> ```
>
> *End of aside.*

Multivariable Constraints

There's another matter I need to discuss briefly in connection with constraints as opposed to predicates (though the matter in question really has to do with constraints in general, not just view constraints in particular). First of all, recall this simple example from Chapter 2:

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 ) ;
```

This example is expressed in **Tutorial D**, of course, but for reasons that will quickly become clear I want to show the same constraint (or at least the boolean expression portion of the constraint, which is to say the portion between the constraint name CX1 and the terminating semicolon) in relational calculus instead:[11]

```
NOT EXISTS x ∈ S ( x.STATUS < 1 )
```

Or equivalently:

```
FORALL x ∈ S ( x.STATUS ≥ 1 )
```

In these formulations, the subexpression "$x \in$ S" (which can be read as "*x* such that *x* is a tuple in the current value of S") defines the variable *x* to be a *range variable*, "ranging over" the current value of relvar S. In other words, the permitted values of *x* at any given time are exactly the tuples appearing in the relation that's the value of relvar S at the time in question.

As a basis for further examples, consider view LS once again ("London suppliers"), together with its companion view NLS ("non London suppliers"). Here are relational calculus formulations of a couple of obvious constraints that apply to these two views:

```
FORALL x ∈ LS ( x.CITY = 'London' )

FORALL x ∈ NLS ( x.CITY ≠ 'London' )
```

Now, all of the constraints we've seen in this section so far have been *single-variable* constraints, because each of them involves just one range variable. Here by contrast is an example of a *multivariable* constraint (it's basically constraint CX3 from Chapter 2):

```
FORALL x ∈ S ( FORALL y ∈ SP
     ( IF x.STATUS < 20 AND x.SNO = y.SNO THEN y.PNO ≠ 'P6' ) )
```

---

[11] Refer to *SQL and Relational Theory* if you need further explanation of relational calculus expressions like those discussed in this section.

Aside: By the way, note that there's a logical difference between multivariable constraints and what *SQL and Relational Theory* calls *multirelvar* constraints. A multivariable constraint is one that involves involves two or more range variables; a multirelvar constraint is one that involves two or more relvars. Now, it's certainly true that every multirelvar constraint is also a multivariable constraint, but the converse is false—the range variables in a multivariable constraint might all range over the same relvar (e.g., see the next example but one). *End of aside*.

Here are some more examples of multivariable constraints (all of them based on relvars S and/or LS and/or NLS):

- No supplier number appears in both LS and NLS:

```
FORALL x ∈ LS ( FORALL y ∈ NLS ( x.SNO ≠ y.SNO ) )
```

- {SNO} is a key for each of S, LS, and NLS (for simplicity, I show the constraint for relvar S only):

```
FORALL x ∈ S ( UNIQUE y ∈ S ( x.SNO = y.SNO ) )
```

  Note: In case you're unfamiliar with the UNIQUE quantifier, it can be read as "there exists exactly one … such that."

- {SNO} in LS is a foreign key, referencing the key {SNO} in S:

```
FORALL x ∈ LS ( UNIQUE y ∈ S ( x.SNO = y.SNO ) )
```

- S is equal to the (disjoint) union of LS and NLS:

```
FORALL x ∈ S ( UNIQUE y ∈ LS ( x = y ) OR UNIQUE y ∈ NLS ( x = y ) ) AND
FORALL y ∈ LS ( UNIQUE x ∈ S  ( x = y ) AND
FORALL y ∈ NLS ( UNIQUE x ∈ S  ( x = y )
```

- At any given time, LS is equal to that restriction of S where the CITY value is London:

```
FORALL x ∈ LS ( UNIQUE y ∈ S ( x = y ) AND
FORALL y ∈ S ( IF y.CITY = 'London' THEN UNIQUE x ∈ LS ( x = y ) )
```

Now (at last) we come to the point of this perhaps rather lengthy digression. In Chapter 1, I introduced the idea of compensatory actions, which are additional updates that are performed

automatically by the DBMS, over and above updates explicitly requested by the user, in order to prevent some integrity constraint violation from occurring. Well, it's precisely if the constraint in question is a multivariable constraint specifically that some compensatory action might be needed—as we'll see in subsequent chapters.

INFORMATION EQUIVALENCE

When I first mentioned information equivalence in Chapter 1, I said by way of example (paraphrasing slightly) that a database design involving just the suppliers relvar S was information equivalent to one involving relvars LS and NLS—"London suppliers" and "non London suppliers," respectively—taken in combination. I also said a design involving just relvar LS wasn't information equivalent to either of the ones just mentioned (nor is one involving just relvar NLS, of course). Now I can explain these ideas in more detail.

Now, when I said the design involving just relvar S was information equivalent to the one involving relvars LS and NLS taken together, what I meant from an intuitive point of view—and I'm sure you understood me to mean exactly this—was that any proposition that could be represented by either one could also be represented by the other. Intuitively speaking, in other words, two designs are information equivalent if and only if they're capable of representing the same set of propositions.[12] And that's essentially what the following definition says:

- **Definition:**[13] Let *DB1* and *DB2* be sets of relvars.[14] Then *DB1* and *DB2* are information equivalent if and only if the constraints that apply to *DB1* and *DB2* are such that every proposition that can be represented by *DB1* can be represented by *DB2* and vice versa.

By way of example, consider relvars S, LS, and NLS once again. Let *DB1* and *DB2* be, respectively, the set of relvars consisting of relvar S in isolation and the set of relvars consisting of relvars LS and NLS taken together. Now, it's clear that we can define the relvars in *DB2* in terms of the sole relvar in *DB1*, thus:

---

[12] It follows that another term that might be used for the concept is *expressive* equivalence.

[13] Please note that this definition is slightly simplified, in a sense. I'll have more to say about it in Chapter 14. Until then, however, the definition as given here will serve.

[14] I use the symbols *DB1* and *DB2* here to suggest that we might intuitively think of each of those sets of relvars, together with the pertinent constraints, as constituting a database—or a dbvar, rather, to be a little more precise about the matter. In subsequent chapters, in fact, when I'm talking about examples of information equivalence, for brevity I'll sometimes refer to the sets of relvars under discussion as databases, or sometimes as dbvars. *Note:* With regard to the symbol *DB2* in particular, please understand that no reference is intended to the commercial product that goes, somewhat inappropriately, by that name.

```
LS   ≝  S WHERE CITY = 'London'

NLS  ≝  S WHERE CITY ≠ 'London'
```

(the symbol "≝" means "is defined as"). It's also clear that we can define the sole relvar in *DB1* in terms of the relvars in *DB2*, thus:

```
S  ≝  LS UNION NLS
```

Moreover, I think it's clear without going into too much detail that for every constraint that applies to *DB1* there's an analogous constraint that applies to *DB2* and vice versa. I'll give just one loose example: For *DB1*, no two distinct tuples in relvar S have the same supplier number; for *DB2*, (a) the union of relvars LS and NLS is disjoint and (b) no two distinct tuples in that union have the same supplier number. Thus, I hope you agree it's at least intuitively reasonable to say that every proposition that can be represented by *DB1* can be represented by *DB2* and vice versa; in other words, *DB1* and *DB2* are indeed information equivalent.

Several points arise from the foregoing, the following among them. Let *DB1* and *DB2* be information equivalent sets of relvars, and let their current values be *db1* and *db2*, respectively. Assume also that the specific set of propositions represented by *DB1* at any given time is supposed to be identical to the set of propositions represented by *DB2* at that same time. Then:

- Those values *db1* and *db2* can be said to be information equivalent in turn (certainly it's true that every proposition represented by *db1* is represented by *db2* and vice versa). In other words, the notion of information equivalence can usefully be applied to sets of relations as such as well as to sets of relvars.

- If *db1* and *db2* are information equivalent, there must exist mappings *M12* and *M21* that transform *db1* into *db2* and *db2* into *db1*, respectively, where the mappings in question are, or can be, formulated in terms of operations of the relational algebra (and are inverses of each other, of course). Conversely, if such mappings exist, then *db1* and *db2* are information equivalent. And if such mappings exist for all possible pairs of current values *db1* and *db2* of *DB1* and *DB2*, respectively, then *DB1* and *DB2* per se are information equivalent. (Of course, these observations, taken together, are little more than a paraphrase of the original definition.)

- For every query *Q1* on *db1,* there must exist a query *Q2* on *db2* that yields the same result (and vice versa, obviously). *Note:* It follows from this observation that IS_EMPTY (*Q1*) and IS_EMPTY (*Q2*) are either both true or both false, and hence that, as claimed previously, for every constraint *C1* on *DB1* there must exist a corresponding constraint *C2* on *DB2* and vice versa.

- Let *U1* be an update on *DB1* that yields a "new" value *db1'* of *DB1*; then there must exist an update *U2* on *DB2* that yields a "new" value *db2'* of *DB2*, such that *db1'* and *db2'* are information equivalent in turn (and vice versa once again). Fig. 3.1 illustrates this point.

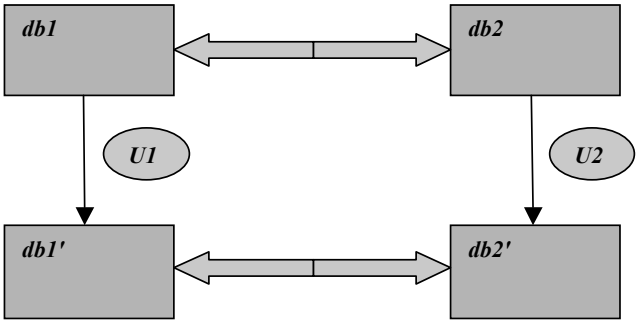


Fig. 3.1: Information equivalence and updating

In contrast to the foregoing, suppose *DB1* and *DB2* aren't information equivalent. Then there'll be operations—queries and updates—on *DB1* that have no counterpart on *DB2* or vice versa. (Note, incidentally, that if *DB1* and *DB2* are indeed not information equivalent, then their current values *db1* and *db2* might or might not themselves be information equivalent in turn; in general, however, they won't be.)

Observe now that all of the above applies in the particular case in which *DB2* consists exclusively of views of the relvars in *DB1*. In other words, if *DB2* is "views only" and that set of views is information equivalent to *DB1*, then every update on *DB2* has a counterpart update on *DB1* and vice versa. In the chapters to come, I'll consider this situation in detail; that is, I'll show in detail how updates on the views in *DB2* can be implemented in terms of updates on the relvars in *DB1*.

What about the case in which *DB2* consists only of views of the relvars in *DB1* but *DB2* and *DB1* are *not* information equivalent?[15] Well, notice first that, precisely because the relvars in *DB2* are derived from those in *DB1*, there can't be any propositions that can be represented by *DB2* and not by *DB1*—or, turning this statement around, everything that can be represented by *DB2* can certainly be represented by *DB1*, but the converse is false. In general, therefore, there might still be some updates on *DB2* that can be implemented in terms of updates on *DB1*. At the same time, there'll certainly be updates on *DB2* that can't be implemented in this way (because there's not enough information available), and those updates must therefore fail. The structure of the argument in the next several chapters is thus as follows:

1. When we do have information equivalence, it should always be possible to update *DB2* (the views) in a logically correct fashion, and I'll explain the rules for doing so as carefully as I can.

2. Even when we don't have information equivalence, sometimes it'll be possible to apply the view updating rules anyway (albeit only partially, perhaps), and I'll discuss such cases in detail too. ***But I must stress that updating in such cases can lead to results that, while of course always formally predictable and well defined, might sometimes be undesirable—possibly even unacceptable for some reason.*** Because of this state of affairs, I leave to others (perhaps the individual user, perhaps the DBA, perhaps even the DBMS) the choice as to whether updates should in fact be allowed in such situations.

3. When the rules don't apply at all—and I'll explain when they don't—then as I've said updates will necessarily fail. *Note:* I do have a few further remarks to make in connection with this case, however, but I'll defer those remarks to the very end of Chapter 5.

CONCLUDING REMARKS

To close this chapter, I have a few miscellaneous observations. First of all, I want to say something about terminology. This book is, of course, all about views, and I hope it goes without saying that when I use the term *view*, I do so in the original sense—the sense, that is, in

---

[15] In *Database Design and Relational Theory*, I argue that such a *DB2* is logically incorrect (or at least incomplete), in the sense that it's incapable of representing certain facts about the real world that we want to be able to represent. David McGoveran would say it's not *expressively complete*. Moreover, note that what we have here is exactly a case of what earlier in the chapter I referred to as *information hiding*: Users of *DB2* are seeing only part of the picture, as it were, and it's only to be expected that there'll be operations they can't be allowed to perform.

which that term 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. As we know, a view in the original sense is a derived relvar—it's derived in the obvious way from the relvars (base relvars and/or other views) in terms of which it's defined. Well, there's another kind of derived relvar too, called a *snapshot*.[16] As that name might tend to 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. *Note:* By *read-only* here, I mean no updates on the relvar are allowed, apart from the periodic "refresh"—see the bullet point 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 particular, REFRESH ON EVERY UPDATE would mean the snapshot is at all times in synch with the relvar(s) from which it's derived.

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 (typically the end of an accounting period), and snapshots allow such freezing to occur without locking out other applications.

---

[16] The term *snapshot* first appeared, so far as I'm aware, in the paper "Database Snapshots," by Michel E. Adiba and Bruce G. Lindsay (IBM Research Report RJ2772, March 7th, 1980). See also the extended version "Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Data," by Michel Adiba (Proc. 1981 International Conference on Very Large Data Bases, Cannes, France, September 1981).

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're not views! Views aren't supposed to be materialized at all; rather, operations on views are supposed to be implemented (as explained in the introduction to this chapter) by mapping them into suitable operations on the relvars in terms of which they're defined.[17] 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, as I've said, I do use the term *view* in its original sense, but be warned that it doesn't always have that meaning elsewhere.

My second point is this. As some readers might know, I've been thinking about the problem of view updating for some time; indeed, I've been trying to get view updating "right" for many years, and this book, which represents my most recent thinking on the subject, is far from being my first publication in this area. Thus, where there are discrepancies between this book and something I've said in an earlier publication, this book should be taken as superseding. Here for the record is a summary list of those earlier publications:

- "File Definition and Logical Data Independence" (coauthored with Paul Hopewell), Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, California (November 1971)

- "Updating Views," in *Relational Database: Selected Writings* (Addison-Wesley, 1986)

- "Updating Union, Intersection, and Difference Views" and "Updating Joins and Other Views" (both coauthored with David McGoveran), *Database Programming & Design 7,* Nos. 6 (June 1994) and 8 (August 1994); republished in *Relational Database Writings 1991-1994* (Addison-Wesley, 1995)

- "View Updates," Section 10.4 of *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004)

- "View Updating ," Appendix E of *Databases, Types, and the Relational Model: The Third Manifesto*, by Hugh Darwen and myself (3rd edition, Addison-Wesley, 2006)

---

[17] It's true that certain products do use materialization instead of operation mapping as an implementation technique—specifically, as a technique for implementing certain "complex" retrievals on certain "complex" views—but the fact that this is so isn't germane to the present discussion (in fact, it's a complete red herring).

- "The Logic of View Updating ," in *Logic and Databases: The Roots of Relational Theory* (Trafford, 2007)

- "How to Update Views ," in *Database Explorations: Essays on The Third Manifesto and Related Topics*, by Hugh Darwen and myself (Trafford, 2010)

The first two items in this list are rather embarrassingly bad, but the rest are (I believe) generally on the right lines, though they're all somewhat confused (and the farther back they go, chronologically speaking, the more confused they are; I guess that indicates progress, of a kind).

I should also mention David McGoveran's patents in this area:

- "Accessing and Updating Views and Relations in a Relational Database," U.S. Patent No. 7,263,512 (August 28th, 2007)

- "Computer-Implemented Method for Translating among Multiple Representations and Storage Structures," U.S. Patent No. 7,620,664 (November 17th, 2009)

The ideas to be described in the present book are somewhat similar to (and heavily influenced by) the ones discussed in these patents, though I must also make it clear that they differ considerably at the detail level.

Note: As should be obvious from the foregoing, I freely admit that I've changed my mind in this area quite a few times. In my defense here, I'd like to quote Bertrand Russell:

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

These wonderful remarks, which need no elaboration by me, are taken from Russell's own preface to *The Bertrand Russell Dictionary of Mind, Matter and Morals*, ed., Lester E. Denonn (Citadel Press, 1993).

To repeat, this book represents my most recent thinking. But I make no claim that what it has to say represents the last word on the subject. There are still some loose ends—indeed, there are one or two points on which the ideas could quite reasonably be challenged (and in the interest of full disclosure, I'll call out such points explicitly, when we get to them). But I'm an optimist; I believe those loose ends can and will be tied up in good time, and I don't believe there are any showstoppers lurking among them. Indeed, part of my reason for wanting to publish this book at this time and in its present form is to solicit constructive suggestions. Certainly I'm open to discussion in those areas where such suggestions might make sense. Though perhaps I should stress that qualifier *constructive* ... Perhaps I'm being a little defensive here, but the fact is that I've seen quite enough negative criticism in this area in the past, and I would prefer to keep the dialog positive.

One final point to close this chapter: The topic of view updating can unfortunately be quite confusing, even when it's not particularly controversial. Part of the difficulty lies in the fact that there's unavoidably a lot of detail to wade through, and it's easy to lose one's way in debates and discussions. In particular, it's all too easy to forget, especially in examples, which relvars are base ones and which ones are views. It's important to keep a clear head! Again I have to say: *Caveat lector*.

Chapter 4

Restriction Views

Updating restrictions
Leads to no contradictions

—Anon.: *Where Bugs Go*

The bulk of the remainder of this book consists of an investigation into what's involved in updating "through" the various relational operators (to use the conventional jargon). Thus, this chapter can be seen as the first in a series. In it, I propose to examine in detail the question of updating restriction views specifically (i.e., updating "through" a restriction operation). Note, however, that many of the issues to be raised in connection with that specific question will turn out to have more general significance; thus, this chapter is thus partly a continuation of Chapter 3, in a way. I'll begin by taking a closer look at the motivating example from Chapter 1 (with apologies for the small amount of repetition that closer look will necessarily entail).

THE MOTIVATING EXAMPLE REVISITED

Just to remind you, the motivating example from Chapter 1 involved base relvar S ("suppliers") from the suppliers-and-parts database and two restriction views of that base relvar, LS ("London suppliers") and NLS ("non London suppliers"). Here are **Tutorial D** definitions:

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

VAR LS VIRTUAL ( S WHERE CITY = 'London' )
    KEY { SNO } ;

VAR NLS VIRTUAL ( S WHERE CITY ≠ 'London' )
    KEY { SNO } ;
```

Note: As we saw in the section "Information Equivalence" in Chapter 3, we could clearly arrange matters, at least conceptually, such that relvar S is the only relvar in some database *DB1*,

relvars LS and NLS are the only relvars in another database *DB2*, and *DB1* and *DB2* are information equivalent. But first things first.

Now, as you can see from the definitions, {SNO} is a key (in fact, the sole key) for each of the three relvars; also, {SNO} is a foreign key in each of relvars LS and NLS, referencing the key {SNO} of relvar S, though I haven't bothered to declare those foreign keys explicitly.[1] *Note:* As mentioned a couple of times in previous chapters, in practice we would probably drop attribute CITY from relvar LS, precisely because its value is constant (it's always London); however, I won't do that here, in order to keep the example simple. Of course, if we did drop that attribute, then LS would no longer be just a restriction as such of S anyway, and in this chapter I want to limit my attention to restriction specifically.

Sample values are shown in Fig. 4.1.

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

LS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

Fig. 4.1: Relvars S, LS, and NLS—sample values

Now let me remind you of *The Principle of Interchangeability*, which says there shouldn't be any arbitrary and unnecessary distinctions between base relvars and views (i.e., views should "look and feel" just like base relvars so far as users are concerned). One consequence of that principle, in the case at hand, is that the behavior of relvars S, LS, and NLS shouldn't depend on which of them are base relvars and which of them are views. Until further notice, therefore, let's suppose they're all base relvars:

[1] Under the proposals of the paper "Inclusion Dependencies and Foreign Keys," in *Database Explorations: Essays on The Third Manifesto and Related Topics*, by Hugh Darwen and myself (Trafford, 2010), we could say {SNO} in relvar S is a foreign key as well, referencing the key {SNO} of the disjoint union of relvars LS and NLS.

```
VAR S   BASE RELATION { ... } KEY { SNO } ;
VAR LS  BASE RELATION { ... } KEY { SNO } ;
VAR NLS BASE RELATION { ... } KEY { SNO } ;
```

Here repeated from Chapter 1 are the predicates for these three relvars (and note that here and elsewhere I favor the informal prose style for such predicates):

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

LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is London).*

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*[2]

As I also pointed out in Chapter 1, these relvars are subject to a number of constraints, over and above the key constraints already mentioned. (I didn't state those constraints formally in Chapter 1, but now I will.) First of all, these two constraints obviously hold:[3]

```
CONSTRAINT ... LS  = ( S WHERE CITY = 'London' ) ;
CONSTRAINT ... NLS = ( S WHERE CITY ≠ 'London' ) ;
```

So do these two:

```
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
```

The foregoing constraints imply certain additional ones (including certain foreign key constraints in particular, but again I won't bother to show those):

```
CONSTRAINT ... S = UNION { LS , NLS } ;
CONSTRAINT ... DISJOINT { LS , NLS } ;
```

---

[2] In Chapter 2 I suggested as a rule of thumb that the predicates for relvars that are supposed to be disjoint should be so formulated as to preclude the possibility that any tuple might ever satisfy them both. Observe how I've applied that rule here to the predicates for LS and NLS in particular.

[3] The constraints are expressed in **Tutorial D**, of course, but for simplicity I've omitted the constraint names that **Tutorial D** would in fact require.

Note: In connection with the first of these additional constraints, I should explain that **Tutorial D**—at least, the version of **Tutorial D** I'm using in this book—supports both dyadic (infix) and *n*-adic (prefix) versions of certain relational operators, including union in particular (also join, as we'll see in the next chapter). Thus, for example, the union of LS and NLS can be expressed as either LS UNION NLS or (as above) UNION {LS,NLS}. In this book, I'll use both styles—whichever suits my purpose best at the time. As for the second of those "additional" constraints, here I should explain that the **Tutorial D** expression DISJOINT {*r1*,...,*rn*} returns TRUE if and only if no two of the argument relations *r1*, ..., *rn* have any tuples in common. Incidentally (as you might recall from Chapter 1), this particular constraint, along with the various key constraints, implies the following constraint as well:

```
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
```

("no supplier number appears in both LS and NLS").

Important: Observe how the foregoing constraints taken together ensure the information equivalence I mentioned earlier; in fact, they serve to show how each of the two designs, S by itself vs. the combination of LS and NLS, can be mapped into the other.

> *Aside:* Of course, a database that contains all three relvars obviously involves some redundancy. Indeed, the foregoing constraints—at least, the multivariable ones—serve to capture those redundancies in a formal way. As I've explained elsewhere, however (see *Database Design and Relational Theory*), such redundancy shouldn't cause any harm, just so long as it's properly controlled. I'll elaborate on this point briefly in the next section ("More on Compensatory Actions"). *End of aside.*

A note on terminology: Several of the foregoing CONSTRAINT statements—to be specific, all of the ones in which the boolean expression is of the form <*relation expression*> = <*relation expression*>—effectively require a certain *equality dependency* to hold (i.e., to be satisfied at all times). Here's a definition:

- **Definition:** An equality dependency (EQD for short) is a statement of the form *rx1* = *rx2*, where the expressions *rx1* and *rx2* denote relations *r1* and *r2*, respectively, and *r1* and *r2* are of the same type. The EQD is satisfied if and only if the bodies of *r1* and *r2* are equal.

Equality dependencies in turn are an important special case of a more general phenomenon known as *inclusion* dependencies:

- **Definition:** An inclusion dependency (IND for short) is a statement of the form *rx1* ⊆ *rx2*, where the expressions *rx1* and *rx2* denote relations *r1* and *r2*, respectively, and *r1* and *r2* are of the same type. The IND is satisfied if and only if the body of *r1* is a subset of the body of *r2*.

  Points arising from this latter definition:

- A foreign key constraint is a special case of an IND. In relvar LS, for example, {SNO} is a foreign key, referencing the key {SNO} in relvar S; thus, there's an IND from LS to S—the projection of LS on SNO is included in the projection of S on SNO (in symbols, LS{SNO} ⊆ S{SNO}).

- As already noted, an EQD is also a special case of an IND. In fact, the EQD *r1* = *r2* is equivalent to the pair of INDs *r1* ⊆ *r2* and *r2* ⊆ *r1*; in other words, an EQD is an IND that "goes both ways," as it were.

- Although it's not relevant to the topic of view updating as such, it would be remiss of me not to mention in passing that any constraint that can be expressed by means of a **Tutorial D** CONSTRAINT statement can in fact be expressed as an EQD (exercise for the reader!).

MORE ON COMPENSATORY ACTIONS

In Chapter 1, we saw how certain compensatory actions could be used in connection with the motivating example to ensure that all of the applicable constraints continue to hold after updates are done. The compensatory actions (or rules) in question were as follows:

```
ON DELETE d FROM LS : DELETE d FROM S ;
ON DELETE d FROM NLS : DELETE d FROM S ;

ON DELETE d FROM S : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;

ON INSERT i INTO LS : INSERT i INTO S ;
ON INSERT i INTO NLS : INSERT i INTO S ;

ON INSERT i INTO S : INSERT ( i WHERE CITY = 'London' ) INTO LS ,
                     INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;
```

These particular rules were explained in Chapter 1 (at least in outline), but there's quite a lot more to be said about such rules in general; hence the present section. To be specific:

- First of all, you might have already noticed that I've been a little sloppy in my use of terminology. To be specific, I've been using the term "compensatory action" to refer to both (a) the action, if any, that's performed after some particular update operation has been done and (b) the formal specification of such an action in concrete syntax (though I've also used the term "rule" to refer to this latter construct). What's more, I intend to continue this practice in the chapters ahead, and hope it won't lead to confusion.

- The rules in the example are expressed in a hypothetical and deliberately wordy extension to **Tutorial D**. Together, they serve to illustrate the kind of syntax I'll be using for compensatory actions in the pages to come.[4] Note that *d* and *i* are effectively parameters to those rules, denoting the pertinent delete set and insert set after the update request for which the rule is compensating has been mapped to one of the form DELETE *d* FROM *R*, INSERT *i* INTO *R*. Note too, therefore, that *d* and *i* are disjoint.

- It will often be the case that the compensatory actions can be stated in a variety of syntactically distinct but semantically equivalent forms. For example, the delete rule from S to LS and NLS—

```
ON DELETE d FROM S : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

  —could equally well be stated as follows:

```
ON DELETE d FROM S : DELETE ( LS MATCHING d ) FROM LS ,
                     DELETE ( NLS MATCHING d ) FROM NLS ;
```

  It could even be stated as follows—

```
ON DELETE d FROM S : DELETE d FROM LS ,
                     DELETE d FROM NLS ;
```

  —since an attempt to delete a tuple that doesn't exist isn't an error. As an extreme illustration of the point, it would always be possible to state the rules in terms of pure

[4] It goes without saying—I hope!—that I'm not irrevocably wedded to that syntax. It's sufficient for my purpose in this book, that's all.

relational assignment alone.[5] Partly for pedagogic reasons, I prefer to state them in terms of explicit INSERTs and DELETEs instead. But I must emphasize that my primary concern in this book is always just to get the formulations logically correct; I'm not very interested at this stage in trying to find the simplest or most efficient formulations.

- To elaborate on the previous point: I hope it's obvious in general that, as noted in Chapter 2, my overall goal is always to get the theory right first and not to worry about implementation matters (not yet, at any rate). Doubtless there'll be issues of optimization in any concrete implementation. *Note:* I'd like to mention one such issue here, however: Whenever the phrase DELETE *d* FROM *R* appears, that *d* can safely be replaced by *d* INTERSECT *R*; likewise, whenever the phrase INSERT *i* INTO *R* appears, that *i* can safely be replaced by *i* MINUS *R*. For simplicity, therefore, I'll assume throughout what follows, without loss of generality, that these replacements have always been made, and hence that *d* MINUS *R* and *i* INTERSECT *R* are always both empty; in other words—previous remarks on this point notwithstanding—I'll assume the rules as such never request insertion of a tuple that's already present or deletion of one that isn't.

- Consider a user *U* who sees all three relvars S, LS, and NLS and performs (say) a DELETE on relvar S. Thanks to the delete rule on that relvar, then, that DELETE will cause tuples to be deleted from both relvar LS and relvar NLS (in general). ***That rule must therefore be explicitly visible to user U***—for otherwise user *U* will perceive a violation of *The Assignment Principle* (again, in general). To be specific, if user *U* isn't aware of that rule, then that user will at least potentially see a change to the dbvar—i.e., the variable that's the entire database—that's not exactly what he or she originally requested. Equivalently, that user will see changes to variables LS and/or NLS that he or she didn't explicitly request at all. Hence, (a) compensatory actions can't be wholly "under the covers" but must be exposed to users (at least in some cases), and (b) users must be explicitly aware that the updates they request are really shorthand for some extended series of operations—in fact, for a certain multiple assignment (again, at least in some cases).[6]

---

[5] E,g., as follows: ON <*any update*> TO S : LS := S WHERE CITY = 'London', NLS := S WHERE CITY ≠ 'London'. Now, you might think this style for formulating rules is attractive, if only because it seems to be much simpler than the explicit INSERT and DELETE formulations that I prefer. However, my own opinion is that such simplification is mostly spurious. Certainly it makes it more difficult to examine certain pragmatically important issues, such as how to respond to a "pure INSERT" or "pure DELETE" request.

[6] Observe that this state of affairs exactly parallels that in which a user who sees both relvar S and relvar SP is allowed to perform (say) delete operations on relvar S, if those operations cascade to relvar SP.

- Following on from the previous point, an update explicitly requested by the user, together with any compensatory actions carried out in connection with that update, must be regarded in toto as a semantically atomic operation—a multiple assignment operation, to be precise. Thus, no integrity checking must be done until that multiple assignment has been executed in its entirety. *Note:* In general, as explained in Chapter 2, the individual assignments that make up a multiple assignment can be thought of as being executed simultaneously. In the case at hand, however, I'm going to assume for the purposes of this book, where it makes any difference, that the explicitly requested update is done first and the compensatory actions are done second (maybe AFTER would be a better keyword than ON). Please note, however, that this assumption is *not* logically required—it's just that it has the effect of simplifying the formulation of the pertinent rules (usually).

- A compensatory action, precisely because it is itself an update operation, might in general cause certain further compensatory actions to be performed, and so on. In the case of a DELETE on relvar S, for example, cascading the delete from S to LS will cause the cascade delete rule from LS to S to be invoked in turn. Of course, this second cascade won't have any effect in this particular example; however, it does raise the interesting question of when cascading has to stop in general. I believe this issue deserves more study; for the purposes of this book, I'll adopt the pragmatic position that it stops when a "fixpoint" is reached, meaning the database has reached a point when no further changes occur.[7] *Note:* If *U1* is the update requested by the user, and the compensatory action for *U1* causes another update request *U2*, which in turn causes another compensatory action to be performed (etc.), then *U2* is "the explicitly requested update" as far as this latter compensatory action is concerned—and so on, recursively. Of course, the complete set of updates *U1*, *U2*, ... must still behave as a semantically atomic operation.

- Eventually I want to examine the case in which LS and NLS are views and S is a base relvar—but recall that, so far, all three relvars are base ones. Given that such is the case, I now observe that compensatory actions such as the ones we've been discussing are precisely what's needed to *control the redundancy* among such relvars. In other words, I claim that compensatory actions would be needed anyway, in general, even if views as such weren't supported at all. *Note:* Redundancy is said to be *controlled* if it does exist (and the user is aware of it), but the task of "propagating updates" to ensure that it never

[7] I apologize for my use of the slightly illogical term *fixpoint* here. It's not my coinage. More important, note the tacit assumptions (a) that a fixpoint will indeed be reached, and moreover (b) that it's unique! To repeat, I believe this aspect requires more study.

leads to any inconsistencies is managed by the system, not the user. (Uncontrolled redundancy can be a problem, but controlled redundancy shouldn't be.) Controlling redundancy involves (a) explicitly declaring the redundancy to the DBMS and then (b) having the DBMS take responsibility for the necessary update propagation. And declaring the redundancy in the case at hand is, precisely, a matter of declaring the appropriate multivariable constraint(s).

- The preceding point is actually a special case of a more general one, already touched on in Chapter 3: namely, that compensatory actions are needed anyway in order to maintain certain multivariable constraints. Of course, I'm assuming here that we don't want users to be responsible for explicitly writing the necessary multiple assignment themselves whenever they do an update that might cause such a constraint to be violated. To put the point another way: By definition, multivariable constraints in general imply that certain updates must be, logically, multiple assignments. So we have a choice: Either (a) we require users always to write out those multiple assignments explicitly, or (b) we find a way of getting the system to do part of the work, such that users will often be able to get away with writing a single assignment instead. Compensatory actions are the means for achieving possibility (b).

- One reviewer of an early draft of this book asked why, in the example of relvars S, LS, and NLS, the compensatory actions all had to be cascades. To be specific, he—the reviewer was male—pointed out (a) that SQL permits a "referential action" [*sic*] called NO ACTION that, instead of cascading, simply causes updates to fail if they would otherwise cause a referential integrity violation, and he therefore asked (b) why we couldn't have a similar option in examples like the one under discussion. There are two responses to this question. First, specifying such an option would mean we would be requiring users always to write out those multiple assignments explicitly instead of getting the system to do part of the work (see the previous bullet item). Second, cascading *will* be what happens when we get to the real object of the exercise: namely, the situation in which (e.g.) S is a base relvar and LS and NLS are views.

- There's one final point I need to make in this section—but it's an important one. It follows on from the previous point. Recall this text from the last section of Chapter 2 ("Databases and Dbvars"):

  > "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 piecemeal, individual relvar by individual relvar. And it ... follows that "relational assignment," multiple or otherwise, is also a convenient fiction, in a sense; to be specific, it's an operator that lets us pretend that updating a dbvar can be thought of as a collection of updates on individual relvars within that dbvar.

What I need to say now, however, is this: *These fictions aren't always 100 percent sustainable*. That is, in the context of updating in general, and view updating in particular, it isn't always possible to use single assignment only (thereby updating the database individual relvar by individual relvar). Now, the compensatory actions to be described in detail in subsequent chapters do a pretty good job of maintaining those fictions, most of the time; but there are a few cases (which I'll point out explicitly when we encounter them) where they can produce effects that might be unacceptable for some reason. And in such cases (cases, that is, where the effects aren't fully acceptable), then I don't think there's any alternative to having the user write out an appropriate multiple assignment—which is to say, an appropriate *database* assignment, in effect—explicitly.

WHAT ABOUT TRIGGERS?

At this point I'd like to digress for a moment to head off at the pass, as it were, another objection that might have occurred to you—viz., isn't this whole idea of compensatory actions just SQL-style triggers by another name? In other words, is there really anything new in what I've been saying?

Well, obviously there are some points of similarity between the two concepts; it might even be possible to use triggers to implement compensatory actions, if (as is typically the case today) the system provides little or no direct support for such actions.[8] But there are lots of differences too. Here are some of them:

- Triggers can and often do involve procedural code. As a matter of fact, I checked several textbooks in this connection—textbooks both on commercial SQL products and on the SQL standard as such—and found the universal assumption to be that, in practice, triggers always do involve procedural code. By contrast, the compensatory actions I propose in this book are much more declarative in nature. (Of course, the terms *declarative* and *procedural* aren't formally defined, so this point isn't as precise as it might be. But what I mean by it is that the compensatory actions I propose are exactly as declarative as

[8] One reviewer commented at this point that to use triggers in this way would be (a) highly nontrivial in general and (b) actually impossible in some cases, if the system in question doesn't allow triggers to be invoked recursively.

Tutorial D is in general. Moreover, the fact that they're declarative in this sense means the DBMS can "understand" what those actions are meant to do and can reason about them in a formal way.)

- Following on from the previous point, triggers in general can do anything at all—that is, they can involve procedures of arbitrary complexity—whereas all I want my compensatory actions to do is, at their most complex, some combination of relational INSERTs and DELETEs.

- With triggers in general, there's no notion that the system should be able to determine for itself what actions are to be performed (indeed, if it could, then triggers wouldn't be necessary!). In other words, triggers are, by definition, user defined. With compensatory actions, by contrast, I've already said in Chapter 1 that the system should be able to work out for itself just what actions are required; in other words, such actions should be system defined, not user defined.

- Details of the operation, and possibly even the existence, of triggers in general are typically concealed from the user. As a consequence, therefore, it's highly likely from the user's perspective that triggers will lead to violations of *The Assignment Principle*. Again, contrast the situation with compensatory actions.

- In SQL in particular, triggers can and often do violate the set level nature of the relational model. As I said in Chapter 1, relational updates are set level by definition, and they mustn't be treated as a sequence of individual tuple level updates (row level updates, in SQL); in particular, compensatory actions mustn't be carried out until the update as requested by the user has been done in its entirety, and integrity constraints mustn't be checked until that update *and* any associated compensatory actions have all been done in their entirety. Yet SQL supports what it calls row level triggers, which clearly violate these relational prescriptions.

- Finally (again unlike triggers in general), the specific compensatory actions I propose in connection with view updating specifically are logically required,[9] thanks to *The Principle of Interchangeability*.

---

[9] At least, they're logically required so long as we have information equivalence.

WHAT ABOUT EXPLICIT UPDATE OPERATIONS?

Consider now the following explicit UPDATE operation on relvar S:

```
UPDATE S WHERE CITY = 'Paris' : { STATUS := 30 } ;
```

As explained in *SQL and Relational Theory*, this operation is defined to be shorthand for the following relational assignment:

```
S := ( S WHERE CITY ≠ 'Paris' )
       UNION
     ( EXTEND S WHERE CITY = 'Paris' : { STATUS := 30 } ) ;
```

In other words, the overall effect of the update (conceptually, at least) is to delete the suppliers in Paris, and then reinsert those same suppliers, as it were, with their status set to 30. But observe now that these two sets—suppliers in Paris and suppliers in Paris with their status set to 30—might not be disjoint, because some suppliers in Paris might already have had status 30 before the update was done (supplier S3 is a case in point, given our usual sample values). Thus, if we want to ensure that the delete set *d* and the insert set *i* are indeed disjoint—which we do[10]—we must revise the expansion of the original UPDATE as indicated here:

```
S := ( S WHERE CITY ≠ 'Paris' OR STATUS = 30 )
       UNION
     ( EXTEND S WHERE CITY = 'Paris' AND STATUS ≠ 30 ) :
                                        { STATUS := 30 } ) ;
```

From this expansion we have *d* = suppliers in Paris with status something other than 30 and *i* = those same suppliers with status set to 30, and these two sets are disjoint as required.

Now, in the foregoing example I deliberately ignored the compensatory actions, since they were irrelevant to the point I wanted to make. But now let's take a look at the effects of such actions in connection with explicit UPDATE operations specifically. Please keep in mind, however, that I'm still assuming that all relvars concerned are base relvars—no views yet. *Note:* The discussion that follows is partly (but only partly) a repeat of material from Chapter 1.

For a first example, consider the following UPDATE on relvar NLS:

```
UPDATE NLS WHERE SNO = 'S2' : { CITY := 'Oslo' } ;
```

---

[10] To spell the point out, we surely don't want the overhead of first doing some deletes and then doing some inserts that effectively undo those deletes again.

Let's assume for simplicity that we know the city for supplier S2 isn't Oslo already. Then what happens is this:

1. The existing tuple for supplier S2 is deleted from relvar NLS and a new tuple for that supplier, with CITY value Oslo, is inserted into that same relvar. *Note:* I assume for simplicity here (and in examples throughout the book, wherever it's helpful to do so) that we can talk in terms of deleting and inserting individual tuples as such, rather than relations. Technically, however, you should understand such talk as referring to *sets* of tuples, where the sets in question just happen to have cardinality one.

2. The existing tuple for supplier S2 is deleted from relvar S as well, thanks to the cascade delete rule from NLS to S, and the new tuple for that supplier, with CITY value Oslo, is inserted into relvar S as well, thanks to the cascade insert rule from NLS to S.

Note carefully that the foregoing DELETEs and INSERTs are all performed as part of the same atomic operation (a multiple assignment, of course); in particular, no integrity checking is done until all of those DELETEs and INSERTs have been done.

For a second example, consider this UPDATE on relvar S:

```
UPDATE S WHERE SNO = 'S2' : { CITY := 'London' } ;
```

Let's assume for simplicity that we know the city for supplier S2 isn't already London. Then what happens is this:

1. The existing tuple for supplier S2 is deleted from relvar S and a new tuple for that supplier, with CITY value London, is inserted into that same relvar.

2. The existing tuple for supplier S2 is deleted from relvar NLS as well, thanks to the cascade delete rule from S to NLS, and the new tuple for that supplier, with CITY value London, is inserted into relvar LS as well, thanks to the cascade insert rule from S to LS. In other words, the tuple for supplier S2 has "migrated" from relvar NLS to relvar LS!—here speaking *very* loosely, of course.

For a final example, suppose the preceding UPDATE had been directed at relvar NLS rather than relvar S:

```
UPDATE NLS WHERE SNO = 'S2' : { CITY := 'London' } ;
```

Now what happens is this:

1. The existing tuple for supplier S2 is deleted from relvar NLS.

2. An attempt is made to insert a new tuple for supplier S2, with CITY value London, into relvar NLS. That attempt fails, however, because it violates the constraint on relvar NLS that the CITY value in that relvar can never be London. So the update fails overall; the previous step (viz., deleting the original tple for supplier S2 from NLS) is undone, and the net effect is that the database remains unchanged.

SUPPLIERS AND SHIPMENTS

Note: This section is something of a digression and might well be skipped on a first reading. For one thing, the example it's based on is different in kind from the one I've been considering in this chapter so far (i.e., London vs. non London suppliers)—it has to do with relvars S and SP from the suppliers-and-parts database, and of course there's no question of one of those relvars being a restriction of the other, and certainly not of one being a view of the other. For another, the matters to be discussed, though they do have to do with updating in general, don't seem to have much to do with view updating (or redundancy control) in particular. For such reasons, the discussion that follows doesn't really belong in this chapter at all; rather, it ought to have been included in either Chapter 2 or Chapter 3. But it does rely on certain material—especially the material on explicit UPDATE operations from the section immediately preceding—that wasn't covered (and couldn't sensibly have been covered) in either Chapter 2 or Chapter 3; hence my decision to include it here.

Consider the foreign key constraint on the suppliers-and-parts database from the shipments relvar SP to the suppliers relvar S. Assume for the sake of discussion that we want to specify a cascade delete rule in connection with that foreign key, thus:[11]

```
ON DELETE d FROM S : DELETE ( SP MATCHING d ) FROM SP ;
```

[11] In SQL, such a rule would be more simply (?) defined by means of a specification of the form ON DELETE CASCADE, included, a trifle illogically, as part of the CREATE TABLE statement for table SP. (I say "illogically" because the delete that's referred to in that specification is a delete on table S, not table SP.) I prefer a definition that (a) uses my own deliberately verbose syntax and (b) stands alone as a separate statement in its own right.

Aside: Since relvar SP isn't actually defined in terms of relvar S—in particular, SP isn't a view of S—there's no question of the DBMS determining, from the relvar definitions alone, just what rule(s) if any should apply in this situation. Rather, the DBA, or some suitably authorized user, will have to specify those rules explicitly, somehow. *End of aside.*

Now consider the following UPDATE operation on relvar S:

```
UPDATE S WHERE SNO = 'S1' : { STATUS := 10 } ;
```

Assume for simplicity that we know the status for supplier S1 isn't already 10. What happens, then, is this:

1. The supplier tuple for supplier S1 is deleted from relvar S and a new tuple for that supplier, with STATUS value 10, is inserted into that same relvar.

2. The shipment tuples for supplier S1 are deleted from relvar SP as well, thanks to the cascade delete rule from S to SP. However, since there's no cascade insert rule from S to SP, no shipment tuples are inserted into relvar SP corresponding to the new tuple in S for supplier S1. *Net effect:* The shipment tuples for supplier S1 are lost!

Note: It's important to understand in this example that not only is there no cascade insert rule from S to SP, but there *can't* be—because no such rule makes sense (right?).

But the foregoing state of affairs is surely unacceptable; I mean, we must surely be able to do things like changing the status of a supplier without losing all of that supplier's shipments. Clearly, therefore, we need to do something about UPDATEs like the one in the example. But what?

The key to this problem, I think, lies in the recognition that, in general, updates involve *both* a delete set and an insert set, and hence that, in general, compensatory actions need to apply not just to delete or insert operations taken separately, but rather to such operations taken in combination. Here for example is an appropriate combined rule for the case of suppliers and shipments in particular (note the introduced names *t1* and *t2*):

```
ON DELETE d FROM S , INSERT i INTO S :
   WITH ( t1 := SP MATCHING d , t2 := SP MATCHING i ) :
   DELETE t1 FROM SP , INSERT t2 INTO SP ;
```

Note: It would be tempting to formulate the compensatory actions without those introduced names, thus: DELETE (SP MATCHING *d*) FROM SP, INSERT (SP MATCHING *i*) INTO SP. The trouble with this formulation, however, is that, as explained in Appendix A, the symbol "SP" in the subexpression SP MATCHING *i* (in the INSERT portion of the formulation) denotes the value of relvar SP *after* the DELETE portion of the formulation has been applied.

By the way, let me point out in passing that no compensatory actions have been defined for relvar SP. Of course, this state of affairs doesn't mean we can't do updates on SP; it just means no compensatory actions will be performed if we do.

Now let's revisit the UPDATE example discussed above (I repeat it here for convenience):

```
UPDATE S WHERE SNO = 'S1' : { STATUS := 10 } ;
```

Given our usual sample values, we have the following (using a simplified notation for tuples once again, also for delete sets and insert sets):

d = the S tuple (S1,Smith,20,London)
i = the S tuple (S1,Smith,10,London)
t1 = existing SP tuples for supplier S1
t2 = existing SP tuples for supplier S1(same as *t1*)

As you can see, therefore, the net effect (loosely speaking) is to update the supplier tuple for S1, changing the status value from 20 to 10, while leaving relvar SP unchanged.

I'll leave it as an exercise to confirm that explicit INSERT and DELETE operations on relvar S all continue to work as intuitively expected under the foregoing revised rule. But what about this UPDATE example?—

```
UPDATE S WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

Note that this is an example of what I referred to in Chapter 2 as a "key UPDATE" operation. This time we have:

d = the S tuple (S1,Smith,20,London)
i = the S tuple (S9,Smith,20,London)
t1 = existing SP tuples for supplier S1
t2 = empty

Thus, I think you can see that, even given our revised rule, tuples are still lost here; to be specific, the shipment tuples for supplier S1 simply "vanish"—they're *not* replaced by tuples for supplier S9.

Now, examples like this one might be used, and indeed have been used (by myself among others, I hasten to add) to support arguments to the effect that what we really need is an explicit "cascade UPDATE rule" that, in the case under discussion, would cause an UPDATE on SNO in relvar S to cascade to update SNO in relvar SP (speaking *very* loosely, of course). But I've already given my reasons in Chapter 2 for rejecting such explicit UPDATE rules. To repeat what I said in that chapter, what we want is for compensatory actions to be driven by the applicable delete set and insert set, not by the arbitrary choice of syntax in which the pertinent update request happens to have been formulated. So I regard this approach as a nonstarter. Thus, it seems to me that if we want a change of key value in some tuple in relvar S to be replicated in corresponding tuples in relvar SP, then there's no alternative to writing out the necessary multiple assignment explicitly (thereby, as noted earlier in the chapter, effectively writing out the necessary *database* assignment explicitly), like this:

```
UPDATE S  WHERE SNO = 'S1' : { SNO := 'S9' } ,
UPDATE SP WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

Now what happens is this:

1. The supplier tuple for supplier S1 is deleted from relvar S and a new tuple, identical to its "old" counterpart except that the SNO value has been changed from S1 to S9, is inserted into that same relvar S. Likewise. all shipment tuples for supplier S1 are deleted from relvar SP and a set of new shipment tuples, identical to their "old" counterparts except that the SNO value has been changed from S1 to S9, is inserted into that same relvar SP.

2. The cascade delete rule from S to SP is invoked and requests deletion of all shipment tuples from relvar SP with SNO value S1. However, that request has no effect, since the pertinent tuples have already been deleted anyway by explicit user request. *Net effect:* The change of supplier number from S1 to S9 has been applied appropriately to both relvar S and relvar SP.

Note, therefore, that the cascade delete rule from S to SP serves no real purpose in this example. Of course, this fact doesn't mean the rule is pointless—it's still useful in connection with DELETE operations as such—but it doesn't help with explicit UPDATE operations, as we've seen.

Anyway, the most important message from the discussion overall is this: *In general, compensatory actions apply not just to delete or insert operations taken separately, but rather to such operations taken in combination*. Of course, for any specific update, either the delete set *d* or the insert set *i* might be empty (and if *d* is empty, the rule reduces to a simple insert rule; if *i* is empty, it degenerates to a simple delete rule).

THE MOTIVATING EXAMPLE CONTINUED

Let's get back to relvars S, LS, and NLS. Further, let's continue to assume those relvars are all base relvars (still no views yet). However, let me now remind you that—conceptually speaking, at any rate—we can regard relvar S by itself as constituting a database *DB1* and relvars LS and NLS together as constituting another database *DB2*, such that *DB1* and *DB2* are information equivalent. So let's consider a user who sees just database *DB2*. That user:

1. Knows the corresponding predicates:

   LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is London).*

   NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

2. Is aware of all pertinent type information, is aware of the fact that {SNO} is a key for each of these relvars, and is aware of the following constraints:

   ```
   CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
   CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
   CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
   ```

   Note that these constraints refer to relvars LS and NLS only (this user doesn't even know relvar S exists).

3. Is *not* aware of any compensatory actions (because all of the compensatory actions in this example include some reference to relvar S).

And I hope you can see that, from this user's perspective, all updates—all INSERTs, all DELETEs, all UPDATEs, and more generally all relational assignments—work exactly as expected.

Information Hiding

Now what about a user who sees only relvar NLS, say? Such a user knows the pertinent predicate (see above), knows the pertinent type information, knows also that {SNO} is a key for the relvar, and is aware of the following constraint:

```
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
```

Clearly, this user can't be allowed to insert tuples into relvar NLS, nor to update supplier numbers within that relvar, because such operations have the potential to violate certain constraints that (of necessity) are hidden from the user in question. In other words, the user is seeing only part of the picture, as it were. As I put it in Chapter 1, there's no magic! To state the matter a little more precisely, there's no information equivalence here—there are clearly propositions that can be represented by relvar S and not by relvar NLS, and so it's only to be expected that there'll be operations available on relvar S that have no counterpart on relvar NLS when considered in isolation.

That said, I do need to elaborate somewhat ... First of all, when I say some user "can't be allowed" to perform some operation on some relvar, what I mean, of course, is that the user in question shouldn't be granted the relevant authority. In other words, I'm assuming the DBA, or some other suitably authorized user, will use the authorization subsystem appropriately in order to ensure that operations that don't make much sense can't be attempted.

Second, to repeat something I said in Chapter 1, I suppose it *might* be possible to allow the user to perform such operations after all, just so long as he or she is prepared to accept occasional error messages to the effect that an operation is rejected simply "because the system says so," without further explanation. The point is this: Whether such an operation succeeds or fails will depend in general, not only on those hidden constraints as such, but also on the current state of the hidden part of the database. (As a concrete illustration of this point, given our usual sample values, inserting the tuple (S1,Smith,20,Paris) into NLS will fail, while inserting (S6,Smith,20,Paris) will succeed.) From the user's point of view, in other words, the behavior of such operations will be both unpredictable and (when they fail) inexplicable. Such a situation seems to me highly unattractive, and I therefore don't propose that such a possibility be supported.

Third, I've said in effect that this user is allowed to update (say) supplier status values in relvar NLS but not to insert tuples into that relvar. Now, you might be wondering how this state of affairs can make sense—doesn't it involve some kind of contradiction, given that we want to regard an UPDATE operation as shorthand for a certain DELETE / INSERT combination? But this latter fact is precisely the point: The implicit INSERT that's performed as part of a (successful) explicit UPDATE is always accompanied by an implicit DELETE that guarantees that the INSERT in question doesn't violate any constraints (including in particular ones that are hidden from this user). Thus, the fact that the user is allowed to perform certain explicit UPDATE operations certainly doesn't imply that he or she has carte blanche to perform explicit INSERT operations as well.

Note: Remarks similar to those in the foregoing three paragraphs apply at numerous points in the pages ahead—essentially wherever I refer to some user as "not being allowed" to perform some operation—and I won't bother to repeat them every time, instead letting those paragraphs do duty for all.

PUTTING IT ALL TOGETHER

This section brings together for purposes of reference all of the relvar definitions, CONSTRAINT statements, and compensatory actions for the motivating example. However, given my argument in the previous chapter that it was a mistake to make a syntactic distinction between view and base relvar definitions, I'll simplify the relvar definitions, just this once, in such a way as to elide that distinction:

```
VAR S ..... RELATION { ... } KEY { SNO } ;
VAR LS .... RELATION { ... } KEY { SNO } ;
VAR NLS ... RELATION { ... } KEY { SNO } ;

CONSTRAINT ... LS  = ( S WHERE CITY = 'London' ) ;
CONSTRAINT ... NLS = ( S WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... S = UNION { LS , NLS } ;
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;

ON DELETE d FROM S , INSERT i INTO S :
   DELETE ( d WHERE CITY = 'London' ) FROM LS ,
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   INSERT ( i WHERE CITY = 'London' ) INTO LS ,
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;
```

```
ON DELETE d FROM LS , INSERT i INTO LS :
   DELETE d FROM S , INSERT i INTO S ;

ON DELETE d FROM NLS , INSERT i INTO NLS :
   DELETE d FROM S , INSERT i INTO S ;
```

As a matter of fact, we might consider combining the rules for the various compensatory actions still further. For example, the last two might be combined thus:

```
ON DELETE d1 FROM LS , INSERT i1 INTO LS ,
   DELETE d2 FROM NLS , INSERT i2 INTO NLS :
   DELETE d1 FROM S , DELETE d2 FROM S ,
   INSERT i1 INTO S , INSERT i2 INTO S ;
```

After all, multiple assignment means, loosely, that we can combine several distinct updates into a logically atomic operation, so why not do the same kind of thing with compensatory actions? In fact, it would theoretically be possible, though probably unattractive from a human factors point of view, to talk in terms of a single combined rule for the entire database. Then—thanks to Hugh Darwen for this observation—we'd have just one variable (the entire database), one rule (the combination of all compensatory actions), and one constraint (the total database constraint).

THE POINT AT LAST

Finally I come to my main point: ***Everything I've said in the chapter 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. Then:

1. The constraints specified by the six CONSTRAINT statements in the previous section are enforced automatically, precisely because LS and NLS *are* views of S. That is, no update to any of the relvars can possibly cause any of those constraints to be violated.

2. The compensatory actions from S to LS and NLS also happen automatically, again because LS and NLS are views of S. That is, updates to S will automatically be reflected appropriately in LS or NLS or both. So the compensatory actions from S to LS and NLS are, in effect, *derived:* They're derived from the corresponding view definitions[12] (where

[12] More accurately, the constraints are derived from the view definitions, and the compensatory actions are then derived from those constraints.

by "view definitions" I really mean the definitions of the *mappings* from the underlying base relvar(s) to the views in question—in other words, the pertinent "view defining expressions").

Note: Observe in particular that, e.g., changing the supplier number for some supplier in S, say from S1 to S9, will cascade to either LS or NLS, whichever is applicable. By contrast, recall from the section "Suppliers and Shipments" that such a cascade won't happen with S and SP. But that's because the situation with S and SP is different: With S, LS, and NLS, we're talking about what from the user's point of view is just an example of controlled redundancy; with S and SP, such is not the case. (In fact, to pursue the point a moment longer, the whole point about the view updating scheme I'm proposing is to make the situation look to the user as if it *were* just a matter of controlled redundancy—assuming, that is, that the user in question sees both the views as such and the relvars in terms of which those views are defined. More particularly, telling the user about the compensatory actions means we don't have to tell the user which relvars if any are base ones and which ones if any are views.)

3. The compensatory actions from LS and NLS to S—these are the ones that are generally thought of as the view updating rules as such—also happen automatically, again precisely because LS and NLS are views of S. That is, updates to LS or NLS are "really" updates to the underlying relvar S, and so are automatically visible in S, as well as in LS or NLS or both.

4. Now consider a user who sees only views LS and NLS. Then I hope you can see that *the user in question can behave in all respects exactly as if those views were base relvars*—in fact, exactly as described in the case where they actually were base relvars, in the section "The Motivating Example Continued," earlier in this chapter. Which is, of course, the object of the exercise.

   Note in particular that, e.g., an attempt to change the supplier city for some supplier in LS won't cause the tuple in question to "migrate" from LS to NLS—instead, it will fail on a violation of **The Golden Rule**. By contrast, such migration would probably have been expected, and would probably have occurred, given traditional ad hoc approaches to view updating.[13]

5. By contrast, a user who sees, say, only view NLS will clearly be limited in the operations he or she is allowed to perform—again just like a user who sees only base relvar NLS,

[13] See, e.g., *Using the New DB2: IBM's Object-Relational Database System*, by Don Chamberlin (Morgan Kaufmann, 1996).

when all three relvars are base ones (again, see the section "The Motivating Example Continued"). Such a user will be allowed to delete tuples, and/or to update attributes other than attribute SNO, within relvar NLS, but that's all. Once again, there's no magic.

Note: In connection with this last point, it might be thought that the fact that a user who sees only view NLS is limited in the operations he or she is allowed to perform on that relvar constitutes a violation of *The Principle of Interchangeability*—but of course it doesn't; as we've seen, the very same limitations apply in the case where NLS is a base relvar.

OVERLAPPING RESTRICTIONS

Let's take a look at a slightly more complicated restriction example. Suppose we define two restriction views of the suppliers relvar S, NLS ("non London suppliers") and NPS ("non Paris suppliers"). Both of these views have key {SNO}. Sample values are shown in Fig. 4.2. Here are the predicates for the views:

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

NPS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't Paris).*

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

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

NPS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 4.2: Relvars S, NLS, and NPS—sample values

The following constraints (actually EQDs) clearly hold:[14]

```
CONSTRAINT ... NLS = ( S WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... NPS = ( S WHERE CITY ≠ 'Paris' ) ;
```

So does this one (another EQD):

```
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                        ( NPS WHERE CITY ≠ 'London' ) ;
```

As this latter constraint indicates, views NLS and NPS *overlap*, in the sense that, in general, certain tuples appear in both; in fact, the tuples in question must appear in both, in order to guarantee that the constraint is always satisfied. To spell the point out, a given tuple *t* appearing in relvar S will and must also appear in both NLS and NPS, if and only if the CITY value in that tuple *t* is neither London nor Paris. (I note in passing, therefore, that relvars NLS and NPS together violate *The Principle of Orthogonal Design*. See *Database Design and Relational Theory* for further explanation of this point.)

It follows that certain updates on either of NLS and NPS will necessarily cascade to the other one, albeit indirectly. For example, here are the insert rules for S, NLS, and NPS:[15]

```
ON INSERT i INTO S :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;

ON INSERT i INTO NLS : INSERT i INTO S ;

ON INSERT i INTO NPS : INSERT i INTO S ;
```

Now observe what happens if we insert, say, the tuple (S6,Lopez,30,Madrid) into NLS. First of all, the insert cascades from NLS to S and the same tuple is thus inserted into relvar S; then this latter insert cascades from S to NPS, and the same tuple is inserted into this latter relvar as well. In effect, therefore, the insert has cascaded from NLS to NPS.

Here now are the delete rules (and observe that here too there's potential for an operation on one of NLS and NPS to cascade to the other, indirectly):

---

[14] Observe that once again we're dealing with a situation in which information equivalence holds (in particular, relvar S is equal to the union of relvars NLS and NPS, though this time the union isn't disjoint). Note, incidentally that the predicates for NLS and NPS violate the discipline suggested in Chapter 2 (in a footnote) to the effect that the predicates for relvars *R1* and *R2* should preferably be such as to preclude the possibility that the same tuple might satisfy both.

[15] Although I've said that in practice insert and delete rules can be combined, I'll continue to show them separately for reasons of clarity, most of the time.

```
ON DELETE d FROM S :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;

ON DELETE d FROM NLS : DELETE d FROM S ;

ON DELETE d FROM NPS : DELETE d FROM S ;
```

So a user who sees just NLS and NPS and not S will sometimes observe cascades from one to the other. But of course we want that user to be able to think of NLS and NPS as base relvars. In other words, even if they really were base relvars, the existence of certain constraints implies that certain cascades logically ought to occur, even between base relvars, on occasion. Indeed, relvars NLS and NPS are subject to a certain redundancy, and I've already said that redundancy, when it exists, ought to be controlled—and that's exactly what the cascade updates are doing (viz., propagating updates in order to control the redundancy). Here are the pertinent rules:

```
ON INSERT i INTO NLS :
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;

ON INSERT i INTO NPS :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS;

ON DELETE d FROM NLS :
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;

ON DELETE d FROM NPS :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

These rules are derived by combining the ones already given for S, NLS, and NPS and replacing references to S by references to NLS and/or NPS, as appropriate.

And what about information hiding?—for example, the case of a user who sees (say) just view NLS? I'll leave detailed consideration of this possibility as an exercise.

CONCLUDING REMARKS

I have a few more points to make. The first is a small matter of terminology. I've been talking about "restriction views" and "updating through restrictions"; I've said, for example, that view LS is a restriction of base relvar S. But restriction is an operation of the relational algebra; by definition, therefore, it doesn't really apply to relvars, it applies to relations. So what does it mean to talk about restrictions, or projections, or joins (etc.) of relvars, as opposed to relations?

Well, as I've written elsewhere (see, e.g., *The Relational Database Dictionary, Extended Edition*, Apress, 2008):

> By definition, the operators projection, join, and so on apply to relation values specifically. In particular, of course, they apply to the values that happen to be the current values of relvars. It thus clearly makes sense to talk about, e.g., the projection of relvar S on attributes {CITY,STATUS}, meaning the relation that results from taking the projection on those attributes of the relation that's the current value of that relvar S. In some contexts, however (normalization, for example), it turns out to be convenient to use expressions like "the projection of relvar S on attributes {CITY,STATUS}" in a slightly different sense. To be specific, we might say, loosely but very conveniently, that some *relvar*, CT say, is the projection of relvar S on attributes {CITY,STATUS}—meaning, more precisely, that the value of relvar CT at all times is the projection on those attributes of the value of relvar S at the time in question. In a sense, therefore, we can talk in terms of projections of relvars per se, rather than just in terms of projections of current values of relvars. Analogous remarks apply to all of the relational operations.

In other words, we do often talk in terms of (e.g.) restricting relvars as such; in particular, we do so in the context at hand. Such a manner of speaking is somewhat inappropriate—not to say sloppy—but it is at least succinct.

My second point is this. Among other things, I've considered the case in which S is a base relvar and LS and NLS are (restriction) views of that base relvar. For completeness, I ought really also to consider the inverse situation, in which LS and NLS are base relvars and S is a view of those relvars. But then S would be a union view, and so I'll defer discussion of this case to Chapter 10, where I'll be discussing union views in general. For now, let me just note the point that, in a sense, updating through union and updating through restriction are two sides of the same coin.

I'd like to close by reminding you of the following remarks (slightly paraphrased here) from the end of Chapter 1. As I said in that chapter, the conclusions from the motivating example are all rather obvious;[16] however, what I'm suggesting is that thinking of views as base relvars "living alongside" the relvars in terms of which they're defined (as it were) is a fruitful way to think about the view updating problem in general—indeed, not just a fruitful way, but a way that I believe is logically correct. The basic idea is thus as follows: First, the view defining expressions imply certain constraints; second, such constraints in turn imply certain compensatory actions. To say it one more time, it's not my intent that those actions should have to be specified explicitly by the DBA.

---

[16] Obvious they might be, but (as we've seen) they do differ in certain respects from what might have been expected, and might in fact have occurred, given traditional approaches to view updating. For example, in SQL, an attempt to change the supplier city for some supplier in view LS will succeed (and will effectively cause the supplier in question to migrate to view NLS), unless view LS was explicitly defined WITH CHECK OPTION.

Chapter 5

Projection Views

Updating projections?
Just follow directions!

—Anon.: *Where Bugs Go*

Now I want to consider what's involved in updating projection views (i.e., "updating through" a projection operation). *Note:* I claimed in the previous chapter that restriction and union views are two sides of the same coin, so to speak; well, a similar remark applies to projection and join views, as we'll see in this chapter and the next three.

Let me give you some idea as to how the chapter is structured. The discussion overall centers round a set of examples—three of them, to be exact. The first two both have to do primarily with situations in which we have full information equivalence; the third shows what happens if we don't (i.e., if certain information is hidden). In all three cases, I'll appeal to *The Principle of Interchangeability* and begin by considering what happens if all of the relvars concerned are base ones; then I'll go on to see how the situation changes if some of them are in fact views.

EXAMPLE 1: A NONLOSS DECOMPOSITION

Let's agree for simplicity to drop attribute SNAME from the suppliers relvar, so that throughout this chapter the name "S" refers to the following reduced form of that relvar:

```
S { SNO , STATUS , CITY } KEY { SNO }
```

(As you can see, I haven't just dropped attribute SNAME, I've also abbreviated the relvar definition considerably. In particular, I've omitted the attribute type specifications, again in the interest of simplicity.)

Now let relvars ST and SC be defined as projections of this relvar. To be specific, let ST be the projection on SNO and STATUS and let SC be the projection on SNO and CITY:

```
ST { SNO , STATUS } KEY { SNO }
SC { SNO , CITY } KEY { SNO }
```

Observe that these two projections taken together correspond to what in normalization terms we'd call a nonloss decomposition of relvar S—at all times, (a) the current values of relvars ST and SC are equal to the pertinent projections of the current value of relvar S and (b) the current value of relvar S is equal to the join of the current values of relvars ST and SC. As a direct consequence, the two designs, S by itself and the combination of ST and SC, are information equivalent.[1] Note that {SNO} is a key for each of the three relvars, as already indicated. Sample values are shown in Fig. 5.1.

S

SNO	STATUS	CITY
S1	20	London
S2	10	Paris
S3	30	Paris
S4	20	London
S5	30	Athens

ST

SNO	STATUS
S1	20
S2	10
S3	30
S4	20
S5	30

SC

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

Fig. 5.1: Relvars S, ST, and SC—sample values

Here now are the predicates for ST and SC:

ST: *Supplier SNO is under contract and has status STATUS.*

SC: *Supplier SNO is under contract and is located in city CITY.*

Aside: In accordance with the discussions in Chapter 2, it might be more accurate to say the predicate for relvar ST is "Supplier SNO is under contract and has status STATUS *and is located somewhere*" (and similarly for relvar SC). In this chapter, however, I prefer to give predicates, most of the time, in slightly simplified or truncated form. The issue isn't crucial for present purposes. *End of aside.*

---

[1] The general statement is this: Consider two databases, *DB1* and *DB2*, such that *DB1* contains only a single relvar *R* and *DB2* contains only projections of *R*. Then, in order for *DB1* and *DB2* to be information equivalent, it's necessary and sufficient that (a) the projections in *DB2* be, precisely, the projections obtained by decomposing *R* in accordance with some *join dependency* (JD) that holds in *R* and (b) each relvar in *DB2* is constrained to be equal at all times to the pertinent projection of the current value of *R*. See *Database Design and Relational Theory* for more background on such matters.

As indicated in the introduction to the chapter, I'll begin by considering what happens if all three relvars are base ones, living alongside one another as it were. Of course, if they are indeed all base relvars, we're looking at a database that involves some redundancy once again, and so we're going to want to control that redundancy accordingly. Here then are the integrity constraints that capture that redundancy (observe that they're all EQDs—equality dependencies—and together they spell out the specifics of the redundancy and guarantee the information equivalence referred to above):

```
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... S  = JOIN { ST , SC } ;
```

Note: In connection with the third constraint here (which is, of course, implied by the other two together with the fact that {SNO} is a key for S), I remind you from Chapter 4 that **Tutorial D**—at least, the version of **Tutorial D** I'm using in this book—supports both dyadic (infix) and *n*-adic (prefix) versions of certain relational operators, including join in particular. Thus, for example, the join of ST and SC can be expressed as either ST JOIN SC or (as above) JOIN {ST,SC}. As noted in that previous chapter, I'll use both styles in this book, whichever best suits my purpose at the time.

The following constraint is also implied by the first two above:

```
CONSTRAINT ... IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;
```

And here I should explain that the **Tutorial D** expression IDENTICAL {*r1*,...,*rn*} returns TRUE if and only if the argument relations *r1*, ..., *rn* are all equal (IDENTICAL can be thought of as a kind of "*n*-adic equals" operator). Thus, this particular constraint is really the logical AND of three separate equality dependencies:

```
S  { SNO } = ST { SNO } AND
ST { SNO } = SC { SNO } AND
SC { SNO } = S  { SNO }
```

What compensatory actions apply? Well, these ones are fairly obvious:

```
ON DELETE d FROM S : DELETE d { SNO , STATUS } FROM ST ,
                     DELETE d { SNO , CITY } FROM SC ;

ON INSERT i INTO S : INSERT i { SNO , STATUS } INTO ST ,
                     INSERT i { SNO , CITY } INTO SC ;
```

These rules take care of updates on S. What about updates on ST and SC? Let's think about DELETE operations first. Now, the following "double DELETE" certainly seems reasonable:[2]

```
DELETE ( S1 , 20 ) FROM ST ,
DELETE ( S1 , London ) FROM SC ;
```

The effect is to delete the specified tuples from ST and SC and—assuming, not unreasonably, that appropriate cascade delete rules are in effect—also the tuple (S1,20,London) from relvar S. But what about this "single DELETE"?—

```
DELETE ( S1 , 20 ) FROM ST ;
```

Now, what can't be allowed to happen here is for the tuple (S1,20) to be deleted from relvar ST, exactly as requested, while everything else remains unchanged (because such an outcome would clearly violate certain of the constraints specified earlier). So one possibility would be simply to reject the specified DELETE on a **Golden Rule** violation. On the grounds that it's surely desirable in general for user requests to succeed if they sensibly can, however, it seems preferable to accept the DELETE and let it cascade appropriately.[3] In other words, I propose the following as an appropriate rule for deletes on ST:

```
ON DELETE d FROM ST : DELETE ( S MATCHING d ) FROM S ;
```

Note that cascading a delete on the projection relvar ST to relvar S will cause the rule previously defined for deletes on S to come into play; the net effect will thus be that the original DELETE cascades to the other projection relvar SC as well.

Naturally we need a similar rule for SC:

```
ON DELETE d FROM SC : DELETE ( S MATCHING d ) FROM S ;
```

Aside: Of course, if you don't agree with my position that it's better for single DELETEs like those above to succeed by having them cascade as indicated—i.e., if you'd rather have

---

[2] This example is the first we've seen to illustrate a kind of pseudocode syntax I'll be making frequent use of from this point forward. I hope it's self-explanatory. Just to spell the details out, however, the expression (S1,20) in the example ought strictly to be the relation selector invocation RELATION {TUPLE {SNO 'S1', STATUS 20}}, and the expression (S1,London) ought strictly to be the relation selector invocation RELATION {TUPLE {SNO 'S1', CITY 'London'}}.

[3] It's relevant to mention that such a cascade would certainly occur in today's DBMSs if ST and SC were in fact views of relvar S instead of separate base relvars.

such DELETEs simply fail—then I presume you should be able to achieve the effect you want by means of the DBMS's authorization subsystem. *End of aside.*

Now what about INSERT operations on those projection relvars? Well, the following "double INSERT" does seem reasonable:

```
INSERT ( S9 , 20 ) INTO ST ,
INSERT ( S9 , London ) INTO SC ;
```

The effect is to insert the specified tuples into ST and SC and—again assuming, not unreasonably, that an appropriate cascade rule is in effect (see below)—also the tuple (S9,20,London) into relvar S. But what about this "single INSERT"?—

```
INSERT ( S9 , 20 ) INTO ST ;
```

Now, what can't be allowed to happen here is for the tuple (S9,20) to be inserted into relvar ST, exactly as requested, while everything else remains unchanged, because such an outcome would clearly violate certain of the constraints specified earlier. What's more, no cascading makes sense here (contrast the situation with DELETE operations as discussed above)—we can't make the insertion of tuple (S9,20) into relvar ST cascade to insert some tuple (S9,20,*c*) into relvar S, precisely because the user who requested the original INSERT hasn't provided us with that value *c* that we need to "complete the tuple," as it were.[4] (By the same token, no cascading of the original INSERT to relvar SC makes sense, either.) So the original INSERT has to fail on a **Golden Rule** violation, and the only insert rule that seems to make any sense is as follows:

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   INSERT ( it JOIN ic ) INTO S ;
```

Note in connection with this rule that if the projections of *it* and *ic* on SNO aren't equal, the INSERT overall will fail on, again, a violation of **The Golden Rule**. As a consequence, the fact that INSERT operations must (among other things) be "double INSERTs" if they're to succeed is automatically taken care of.

In the interest of clarity, let me now restate the foregoing rule in the following expanded but logically equivalent form (note in particular the introduced names *t1* and *t2*):

---

[4] If anyone dares mention "nulls" at this point, I'm afraid I'll have to ask them to go and wash their mouth out with soap.

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ) :
   INSERT t1 INTO S , INSERT t2 INTO S ;
```

Note carefully here that (as explained in Chapter 4) the original INSERTs into ST and SC are done before the specified compensatory actions—the two INSERTs into S—come into play. Thus, the symbols "ST" and "SC" in those compensatory actions (in the expressions *ic* JOIN ST and *it* JOIN SC, respectively) refer to the values of those relvars *after* the original INSERTs into those relvars have been done.

> *Aside:* As you can see, line 3 of the expanded rule—INSERT *t1* INTO S, INSERT *t2* INTO S—effectively specifies a double INSERT in which the target in each of the two individual INSERTs is the same (it's relvar S in both cases). In other words, we have here an example—the first we've seen—of a multiple assignment in which two or more of the individual assignments specify the same target variable. What happens in such a situation, loosely speaking, is that the individual assignments are executed in sequence as written (see Appendix A for further explanation). In the particular case at hand, however, we could simplify matters, at least conceptually, by combining the two individual INSERTs into one, thus:
>
> ```
> INSERT UNION { t1 , t2 } INTO S ;
> ```
>
> Note, however, that (again in the particular case at hand) if $t1 \neq t2$ we have a **Golden Rule** violation on our hands anyway. *End of aside.*

For convenience, let me now summarize the rules as defined so far for INSERTs and DELETEs on ST and SC as a single combined rule (again I use introduced names for clarity):

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := S MATCHING dt , t4 := S MATCHING dc ) :
   INSERT t1 INTO S ,
   INSERT t2 INTO S ,
   DELETE t3 FROM S ,
   DELETE t4 FROM S ;
```

Let me also restate the rules for INSERTs and DELETEs on S as a single combined rule:

```
ON DELETE d FROM S , INSERT i INTO S :
   DELETE d { SNO , STATUS } FROM ST , DELETE d { SNO , CITY } FROM SC ,
   INSERT i { SNO , STATUS } INTO ST , INSERT i { SNO , CITY } INTO SC ;
```

Aside: Again we have a situation—actually it arises in connection with both of the foregoing combined rules—in which two or more of the individual assignments within some multiple assignment specify the same target variable. But it's easy to see that the sequence in which those individual assignments are executed makes no difference to the overall outcome anyway, in these particular cases. Again, see Appendix A for further explanation. *End of aside*.

Now let's think about explicit UPDATE operations. First, I'll leave it as an exercise to show that explicit UPDATEs on relvar S all work as you would intuitively expect. (To convince yourself of this fact, you might like to consider the following examples:

```
UPDATE S WHERE SNO = 'S1' : { CITY := 'Paris' } ;

UPDATE S WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

Note in particular that the second of these examples is a "key UPDATE.")

But what about UPDATEs on ST and SC? By way of example, consider the following UPDATE request on ST:

```
UPDATE ST WHERE SNO = 'S1' : { STATUS := 10 } ;
```

Given our usual sample values, what happens here is as follows:

1. First, the requested update is done—i.e., the tuple (S1,20) is deleted from relvar ST and the tuple (S1,10) is inserted into relvar ST.

2. Second, the compensatory actions are performed—i.e., the tuple (S1,20,London) is deleted from relvar S, thanks to the cascade delete rule from ST to S, and the tuple (S1,10,London) is inserted into relvar S, thanks to the cascade insert rule from ST to S. *Note:* Observe that *dc* and *ic* are both empty in this example. As a consequence, the symbol "SC" in the expression *it* JOIN SC, in the rule for INSERTs on ST and SC, effectively denotes the value of relvar SC as it was before any updating is done, and that expression therefore yields the desired S tuple (S1,10,London).

   So this UPDATE works just fine. But what about the following "key UPDATE"?—

```
UPDATE ST WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

Well, it's easy to see here that this UPDATE unfortunately fails (to be specific, it fails on a **Golden Rule** violation). As in the case of the suppliers and shipments example in the previous chapter, in fact, it seems to me that here there's no alternative to writing out the necessary multiple assignment explicitly, thus:

```
UPDATE ST WHERE SNO = 'S1' : { SNO := 'S9' } ,
UPDATE SC WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

This "double UPDATE" succeeds, as you can easily confirm.

So: Despite this latter minor annoyance, it's still the case that (a) the two designs, S by itself and the combination of ST and SC, are information equivalent, and hence that (b) for every update on one, there's a logically equivalent update on the other that has the same overall effect. So far, so good.

EXAMPLE 1 CONTINUED: THE PROJECTION RELVARS

Let's continue to assume that relvars S, ST, and SC are all base relvars (still no views yet). Now consider a user who sees just the projection relvars ST and SC. That user:

1. Knows the corresponding predicates:

   ST: *Supplier SNO is under contract and has status STATUS.*

   SC: *Supplier SNO is under contract and is located in city CITY.*

2. Is aware of all pertinent type information,[5] is aware of the fact that {SNO} is a key for each of these relvars, and is aware of the following constraint:

   ```
   CONSTRAINT ... ST { SNO } = SC { SNO } ;
   ```

   (This is the only constraint from the previous section that makes no mention of relvar S.)

[5] I won't bother to keep saying this—you can just take it as true, wherever applicable, for all subsequent examples in the book.

3. Is aware of the following compensatory actions:

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := SC MATCHING dt , t4 := ST MATCHING dc ) :
   INSERT t1 { SNO , STATUS } INTO ST ,
   INSERT t2 { SNO , CITY } INTO SC ,
   DELETE t3 FROM SC ,
   DELETE t4 FROM ST ;
```

(This rule might look a little complicated, but it's basically just an edited version of the rule for DELETEs and INSERTs on ST and SC from the previous section, revised primarily to replace references to S by references to ST and/or SC, as appropriate.)

Given all of the above, I'll leave it as an exercise to show that, from this user's perspective, all updates on ST and/or SC work exactly as described in the previous section.

Information Hiding

Now what about a user who sees only (say) relvar ST? Well, such a user knows the predicate (see above) and knows also that {SNO} is a key for that relvar, but of course isn't aware of any constraints that mention either SC or S. Perhaps more to the point, that user isn't aware of any compensatory actions, either. Clearly, then, that user can't be allowed to insert tuples into relvar ST, nor to update supplier numbers within that relvar, because such operations have the potential to violate constraints of which this user is, and must be, unaware. (Note too that updating ST to change a supplier's status implicitly requires relvar SC to exist, even if that relvar is hidden from the user. Thus, if ST is defined and SC is not—in particular, if ST is defined without SC purely for security reasons—then such updates will have to be prohibited also.)

EXAMPLE 1 CONTINUED: VIEWS

In Chapter 4, I showed it made essentially no difference if some or all of the relvars under discussion were views instead of base relvars, just so long as information equivalence was preserved. And as I'm sure you've been expecting, the message here is the same, viz.: Everything I've said in this chapter so far applies pretty much unchanged if some or all of the relvars concerned are views. To be specific, suppose now that S is a base relvar and ST and SC are views. Then:

1. The constraints specified by the various CONSTRAINT statements in the section before last will all be enforced automatically, precisely because ST and SC are views of S.

2. The compensatory actions from S to ST and SC will also happen automatically, again because ST and SC are views of S; that is, updates to S will automatically be reflected appropriately in ST or SC or both.

3. The compensatory actions from ST and SC to S—these are the ones that are generally thought of as the view updating rules as such—will also happen automatically, again precisely because ST and SC are views of S. That is, updates to ST or SC are "really" updates to the underlying relvar S, and so will automatically be visible in S, as well as in ST or SC or both.

4. A user who sees only views ST and SC can behave in all respects exactly as if those views were base relvars—in fact, exactly as described in the previous section for the case in which they actually were base relvars. To spell out the details, that user knows that (a) DELETEs on either relvar cascade to the other; (b) INSERTs and key UPDATEs must be "double updates"; (c) other UPDATEs behave normally; and (d) explicit assignments must be logically equivalent to some sensible combination of the foregoing.

5. By contrast, a user who sees, say, only view ST will clearly be limited in the operations he or she is allowed to perform—again just like a user who sees only base relvar ST, when all three relvars are base ones (again as described in the previous section). To be specific, such a user won't be allowed to insert tuples into that relvar or to update supplier numbers within that relvar (and possibly not to change STATUS values, either).

EXAMPLE 2: ANOTHER NONLOSS DECOMPOSITION

Now I want to consider a revised version—in a way, a generalized version—of Example 1. Again I'll start with relvar S. This time, however, I'll add a constraint to the effect that the *functional dependency* (FD)

```
{ CITY } → { STATUS }
```

holds in that relvar; in other words, whenever two suppliers have the same city, they also have the same status.[6] (In order to bring our usual sample values into line with this new constraint, let's temporarily change the status of supplier S2 from 10 to 30.) Now suppose we define two projection relvars—SC, the projection of S on SNO and CITY (as in Example 1), and CT, the projection of S on CITY and STATUS:

```
SC { SNO , CITY } KEY { SNO }
CT { CITY , STATUS } KEY { CITY }
```

Observe that, as with Example 1, these two projections taken together correspond to a nonloss decomposition of relvar S—at all times, (a) the values of SC and CT are equal to the pertinent projections of the value of S and (b) the value of S is equal to the value of the expression SC JOIN CT. Thus, the two designs, S by itself and the combination of SC and CT, are information equivalent.[7] The sole keys for these relvars are {SNO} for SC and {CITY} for CT. Sample values are shown in Fig. 5.2.

S

SNO	STATUS	CITY
S1	20	London
S2	30	Paris
S3	30	Paris
S4	20	London
S5	30	Athens

SC

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

CT

CITY	STATUS
Athens	30
London	20
Paris	30

Fig. 5.2: Relvars S, SC, and CT, with {CITY} → {STATUS}—sample values

Here now are the predicates for SC and CT:

SC: *Supplier SNO is under contract and is located in city CITY.*

---

[6] FDs are discussed in depth in *Database Design and Relational Theory*. Here I just note that if K is a key for relvar R, then the FD $K \rightarrow X$ holds in R for all subsets X of the heading of R, and Example 1 thus implicitly involved certain FDs already.

[7] I said earlier that database *DB1* (containing only a single relvar R) and database *DB2* (containing only projections of R) are information equivalent only if the projections in *DB2* are obtained by decomposing R in accordance with some JD that holds in R—but the example under discussion (Example 2) involves FDs, not JDs. However, there's a theorem (Heath's Theorem) to the effect that if the union of A, B, and C is equal to the heading of R and the FD $A \rightarrow B$ holds in R, then a JD with components AB (= the union of A and B) and AC (= the union of A and C) also holds in R.

CT: *City CITY has status STATUS.*[8]

As usual, first let's see what happens if all three relvars are base ones. The following constraints clearly hold:

```
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... CT = S { CITY , STATUS } ;
CONSTRAINT ... S  = JOIN { SC , CT } ;
CONSTRAINT ... IDENTICAL { S { SNO } , SC { SNO } } ;
CONSTRAINT ... IDENTICAL { S { CITY } , SC { CITY } , CT { CITY } } ;
```

As for updates, INSERTs and DELETEs on S certainly cascade to SC appropriately:

```
ON INSERT i INTO S : INSERT i { SNO , CITY } INTO SC ;
ON DELETE d FROM S : DELETE d { SNO , CITY } FROM SC ;
```

But what about cascades to CT? Well, INSERTs are straightforward:

```
ON INSERT i INTO S : INSERT i { CITY , STATUS } INTO CT ;
```

(I remind you once again that an attempt to insert a tuple that already exists is effectively a "no op.") However, DELETEs are a little more complicated:

```
ON DELETE d FROM S :
   DELETE ( ( CT MATCHING d ) NOT MATCHING S ) FROM CT ;
```

Explanation: Let *r* denote the relation represented by the subexpression CT MATCHING *d*. Then, assuming sample values as given in Fig. 5.2:

1. Suppose we delete (S5,30,Athens) from S; S now contains just the tuples for S1, S2, S3, and S4. Then (a) *r* contains just the single tuple (30,Athens); (b) the expression *r* NOT MATCHING S thus reduces to just *r* (since the only tuple that included (30,Athens) as a subtuple has just been deleted from S); hence, (c) the tuple (30,Athens) is deleted from CT.

   Aside: Note that to say that, e.g., (30,Athens) is a subtuple of the tuple (S5,30,Athens) is to say the former is a *projection* of the latter. More generally, if *t* is a supplier tuple, then the projection *t*{SNO,CITY} of *t* on attributes SNO and CITY is that subtuple of *t* that

[8] Cities thus become "entities in their own right" in this design.

contains just the SNO and CITY components from *t*. In other words, we can (and **Tutorial D** does) conveniently define a version of—i.e., we can "overload"—the well known relational projection operator to apply to tuples as well as to relations. Refer to *SQL and Relational Theory* for further discussion. *End of aside*.

2. By contrast, suppose we delete (S1,20,London) from S; S now contains just the tuples for S2, S3, S4, and S5. Then (a) *r* contains just the single tuple (20,London); (b) the expression *r* NOT MATCHING S thus evaluates to an empty relation (since S still has a tuple—viz., that for supplier S4—that includes (20,London) as a subtuple); hence, (c) nothing is deleted from CT (in particular, the tuple (20,London) is not deleted), and CT thus remains unchanged.

Also, I observe without detailed discussion that given the foregoing compensatory actions, explicit UPDATEs on S all work exactly as expected, as you might like to confirm for yourself. Note in particular that updates that attempt to infringe the FD {CITY} → {STATUS} will fail on a **Golden Rule** violation.

Now what about updates on SC and CT? Well, again let's consider some examples (all based, of course, on the sample values in Fig. 5.2). First relvar SC:

1. Suppose we delete (S1,London) from SC; then it seems reasonable (a) to delete (S1,20,London) from S and (b) to do nothing to CT (since there's another supplier, S4, with status 20 and city London). By contrast, suppose we delete (S5,Athens) from SC; then it seems reasonable (a) to delete (S5,30,Athens) from S and (b) to delete (30,Athens) from CT as well. *Note:* When I say these actions seem reasonable, what I mean, of course, is that they achieve the desired effect, and they do so without either violating **The Golden Rule** or requiring the user to write an explicit multiple assignment.

2. Suppose we insert (S9,London) into SC. This update succeeds (it has the effect of inserting (S9,20,London) into S). By contrast, suppose we try to insert (S9,Rome) into SC; this update fails, because there's no tuple with city Rome in CT.[9]

Turning now to relvar CT:

---

[9] You might feel that INSERTs on SC thus succeed or fail unpredictably depending on the current state of the database and should therefore be rejected out of hand. But when they do fail, at least they do so because they violate some constraint of which the user is explicitly aware. In fact, such failures aren't really different in kind from failures arising from, e.g., conventional key constraint violations (which also occur, when they do, on account of the current state of the database).

1. Suppose we try to insert (Rome,20) into CT. Assuming no "simultaneous" update has been requested on SC (and hence, effectively, on S—see 4. below), the result can only be a **Golden Rule** violation, and the attempt must therefore fail.

2. Suppose we try to insert (London,40) into CT. This attempt also fails on a **Golden Rule** violation (a key constraint violation, in fact).

3. Suppose we delete (London,20) from CT. Then it seems reasonable (a) to delete (S1,20,London) and (S4,20,London) from S and (b) to delete (S1,London) and (S4,London) from SC.[10]

4. Suppose we insert (S9,Rome) into SC and (Rome,20) into CT, "simultaneously." This update succeeds (it has the effect of inserting (S9,20,Rome) into S).

Considerations such as the foregoing suggest the following rules:

```
ON DELETE dc FROM SC , DELETE dt FROM CT :
   DELETE ( S MATCHING dc ) FROM S ,
   DELETE ( S MATCHING dt ) FROM S ;

ON INSERT ic INTO SC , INSERT it INTO CT :
   INSERT ( ic JOIN CT ) INTO S ;
```

Note: This latter rule could in fact be simplified to the following without loss (why, exactly?):

```
ON INSERT i INTO SC :
   INSERT ( i JOIN CT ) INTO S ;
```

Given these rules, the foregoing INSERT and DELETE examples all work as described, and as intuitively expected. Also, I observe without detailed discussion that explicit UPDATEs work as expected as well (exercise for the reader). As with Example 1, therefore, the situation is that (a) the two designs, S by itself and the combination of SC and CT, are information equivalent, and hence that (b) for every update on one, there's a logically equivalent update on the other that has the same overall effect.

---

[10] If you feel a trifle uncomfortable with the idea that deletes on CT cascade to S, recall that cities are now "entities in their own right." You might also care to reflect on the notion that deletes on S cascade to SP (the cases are essentially similar).

Aside: Now I can take care of a small piece of unfinished business from earlier chapters, having to do with view LS ("London suppliers"). In those chapters, I said that in practice we would probably drop attribute CITY from that relvar. In fact, we can and probably should decompose that relvar into its projections (a) SNT, on SNO, SNAME, and STATUS, and (b) C, on CITY. (Of course, this latter projection has just one tuple, containing the single value London.) As explained in *Database Design and Relational Theory*, the FD {} → {CITY} holds in relvar LS;[11] thus, Heath's Theorem, mentioned in a footnote earlier in this chapter, guarantees that the decomposition is nonloss. So relvar SNT can be updated in accordance with these rules:

```
ON INSERT i INTO SNT : INSERT ( i JOIN C ) INTO LS ;

ON DELETE d FROM SNT : DELETE ( d MATCHING C ) FROM LS ;
```

(assuming, of course, that relvar LS is updatable in turn). *Note:* I'll leave consideration of updates on relvar C here as an exercise for you. *End of aside*.

Now, for completeness I do plan to round out my discussion of Example 2, as I did with Example 1, by (a) considering relvars SC and CT independently of relvar S; (b) considering what happens if the user sees just one of those two relvars (i.e., if part of the total picture is hidden); and (c) considering what happens if SC and CT are actually views. However, I think it's only fair to say up front that there aren't going to be any surprises in those discussions; indeed, I think it would be surprising if there were any surprises. In fact, let me draw your attention to a more important point: namely that, as you might have already realized, the update rules I've given for Example 2 are essentially identical to their counterparts for Example 1—or, to state the matter a little more precisely, the rules for Example 1 are a degenerate special case of those for Example 2. What's more, this state of affairs is only to be expected, given that the situation illustrated by Example 1 is a degenerate special case of that illustrated by Example 2. (Recall that Example 1 also involved a nonloss decomposition of relvar S, albeit into different projections.)

The Projection Relvars

So let's continue to assume that relvars S, SC, and CT are all base relvars (still no views yet). Now consider a user who sees just the projection relvars SC and CT (and I observe again that the

---

[11] Implying, incidentally, that LS isn't even in second normal form (2NF).

design consisting of those two relvars together is information equivalent to the design consisting of relvar S by itself). That user:

1. Knows the corresponding predicates:

   SC: *Supplier SNO is under contract and is located in city CITY.*

   CT: *City CITY has status STATUS.*

2. Is aware that {SNO} is a key for SC, {CITY} is a key for CT, and is aware also of the following constraint:

```
CONSTRAINT ... IDENTICAL { SC { CITY } , CT { CITY } } ;
```

3. Is aware of the following compensatory actions:

```
ON DELETE dc FROM SC , DELETE dt FROM CT :
   DELETE ( SC MATCHING dt ) FROM SC ,
   DELETE ( CT MATCHING dc ) NOT MATCHING SC ) FROM CT ;
```

Given all of the above, I'll leave it as an exercise to show that from this user's perspective, all updates on SC and/or CT work as intuitively expected.

Information Hiding

What about a user who sees only (say) relvar SC? Well, such a user knows the predicate (see above) and knows also that {SNO} is a key for that relvar, but of course isn't aware of any constraints that mention either CT or S. Perhaps more to the point, that user isn't aware of any compensatory actions, either. As a consequence, that user can't be allowed to perform either INSERTs or UPDATEs if the CITY values in the new or updated tuples don't already exist in the relvar. As for a user who sees only relvar CT, that user can't be allowed to do INSERTs, nor UPDATEs on CITY values. And yet again the reason for these limitations is, of course, that the user is seeing only part of the picture (information equivalence no longer applies).

Views

Finally, everything I've said in this chapter so far applies effectively unchanged if some or all of the relvars concerned are views. To be specific, suppose now that S is a base relvar and SC and CT are views. Then:

1. The constraints specified by the various CONSTRAINT statements given earlier in the section will all be enforced automatically, precisely because SC and CT are views of S.

2. The compensatory actions from S to SC and CT will also happen automatically, again because SC and CT are views of S; that is, updates to S will automatically be reflected appropriately in SC or CT or both.

3. The compensatory actions from SC and CT to S—these are the ones that are generally thought of as the view updating rules as such—will also happen automatically, again precisely because SC and CT are views of S. That is, updates to SC or CT are "really" updates to the underlying relvar S, and so will automatically be visible in S, as well as in SC or CT or both.

4. A user who sees only views SC and CT can behave in all respects exactly as if those views were base relvars—in fact, exactly as described above for the case in which they actually were base relvars.

5. By contrast, a user who sees, say, only view SC will clearly be limited in the operations he or she is allowed to perform—just like a user who sees only base relvar SC, when all three relvars are base ones (again as described above).

EXAMPLE 3: A LOSSY DECOMPOSITION

Now I want to change the example one final time (this is the last example I'll be discussing in this chapter). As usual I'll start with relvar S, but now I'll assume (in contrast to Example 2) that the functional dependency {CITY} → {STATUS} does *not* hold. Now let's define two projection relvars—ST, the projection on SNO and STATUS, and TC, the projection on STATUS and CITY:

```
ST { SNO , STATUS } KEY { SNO }
TC { STATUS , CITY } KEY { STATUS , CITY }
```

Note: Relvar ST is exactly as it was in Example 1. Relvar TC, by contrast, though it has the same heading as relvar CT in Example 2, has different semantics (as is reflected by the fact that it has a different key); for that reason, I've given it a different name in order—I hope—to reduce confusion. Sample values are shown in Fig. 5.3. Observe that I've changed the status of supplier S2 back to 10, since the FD {CITY} → {STATUS} no longer holds.

S

SNO	STATUS	CITY
S1	20	London
S2	10	Paris
S3	30	Paris
S4	20	London
S5	30	Athens

ST

SNO	STATUS
S1	20
S2	10
S3	30
S4	20
S5	30

TC

STATUS	CITY
20	London
10	Paris
30	Paris
30	Athens

Fig. 5.3: Relvars S, ST, and TC—sample values

As indicated, the sole keys for these relvars are {SNO} for ST and {STATUS,CITY} (the entire heading) for TC. Note that, in contrast to Examples 1 and 2, relvars ST and TC together correspond to a *lossy*, not a nonloss, decomposition of relvar S. In other words, S isn't equal to the join of ST and TC,[12] and the two designs—S by itself and the combination of ST and TC—are thus not information equivalent (to be specific, information regarding which suppliers are in which cities is lost in the latter design).

Be that as it may, let's begin as usual by assuming that all three relvars are base ones. Here then are the predicates for ST and TC:

ST: *Supplier SNO is under contract and has status STATUS.*

TC: *Some supplier is under contract and has status STATUS and is located in city CITY.*

Aside: Here, mainly just to stress the fact that the decomposition of relvar S of which relvar TC is a part is lossy, I depart from my usual practice in this chapter and give the predicate for that relvar in quantified form ("Some supplier ..."). *End of aside.*

[12] That's because ST and TC aren't obtained from S in accordance with any JD that holds in S. Once again, see *Database Design and Relational Theory* for further explanation.

The following constraints clearly hold:

```
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... TC = S { STATUS , CITY } ;
CONSTRAINT ... S { SNO } = ST { SNO } ;
CONSTRAINT ... ST { STATUS } = TC { STATUS } ;
```

Next, an analysis essentially similar to the one we went through for Example 2—I'll leave the details to you—shows that the rules for updates on S are as follows:

```
ON INSERT i INTO S :
   INSERT i { SNO , STATUS } INTO ST ,
   INSERT i { STATUS , CITY } INTO TC ;

ON DELETE d FROM S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE ( ( TC MATCHING d ) NOT MATCHING S ) FROM TC ;
```

But what about updates on ST and TC? Well, again let's consider some examples:

1. Suppose we delete (S1,20) from ST; then it seems reasonable (a) to delete (S1,20,London) from S and (b) to do nothing to TC (since there's another supplier, S4, with status 20 and city London). By contrast, suppose we delete (S2,10) from ST; then it seems reasonable (a) to delete (S2,10,Paris) from S and (b) to delete (10,Paris) from TC as well.

2. Suppose we try to insert (S9,20) into ST. Given our usual sample values, this update succeeds (it has the effect of inserting (S9,20,London) into S). By contrast, suppose we try to insert (S9,30) into ST; then this update fails on a key constraint violation on relvar S (because it attempts to insert two distinct tuples for supplier S9 into that relvar). Likewise, if we try to insert (S9,40) into ST, the update fails, this time because there's no tuple with status 40 in TC.

3. Suppose we try to delete some existing tuple from, or insert some new tuple into, TC. Assuming no "simultaneous" update has been requested on ST (and hence, effectively, on S—see 4. below), the result can only be a **Golden Rule** violation, and the attempt must therefore fail.

4. Suppose we insert (S9,40) into ST and (40,Rome) into TC, "simultaneously." That update succeeds (it has the effect of inserting (S9,40,Rome) into S). Likewise, suppose we delete

(S2,10) from ST and (10,Paris) from TC "simultaneously." Again this update succeeds (it has the effect of deleting (S2,10,Paris) from S); however, it seems to reasonable in this case to say the DELETE on TC should happen anyway, regardless of whether we request it explicitly (see 1. above).

Considerations such as the foregoing suggest the following rules:

```
ON DELETE d FROM ST :
   DELETE ( S MATCHING d ) FROM S ;

ON INSERT i INTO ST :
   INSERT ( i JOIN TC ) INTO S ;
```

Given these rules, the foregoing INSERT and DELETE examples all work as described. Explicit UPDATE operations are another matter, however. In fact, it seems that most such operations, on either ST or TC, will either fail or have undesirable side effects. By way of example, you might like to consider what happens if we try to update ST, changing the status for supplier S2 to (a) 40, (b) 20, (c) 30, or changing the supplier number for supplier S1 to S9. (Analogs of all of these updates can certainly be done on relvar S, of course.) And the reason for this state of affairs is, of course, that the two designs—S by itself and the combination of ST and TC—aren't information equivalent; to be specific (and to repeat), the fact that a given supplier is in a given city can be represented in the former design but not, in general, in the latter. Thus, while such explicit UPDATE operations can certainly be requested, they probably shouldn't be.

Now, once again I'll complete my investigation into the example overall by (a) considering the projection relvars independently of relvar S, (b) considering what happens if the user sees just one of those projection relvars, and (c) considering what happens if those relvars are views—but again I don't think you'll find any surprises in those discussions. What you might find a little surprising, however, is that the update rules I've given for the example overall are extremely similar, if not quite identical, to the rules I gave earlier for Examples 1 and 2. (To repeat, they're similar, but—speaking very loosely—updates are more likely to fail in the case of Example 3 than they are in the case of Examples 1 and 2.) It seems to me, therefore, that examples like the ones discussed in this chapter should suffice as a basis on which to define a set of rules for updating projections in general.

The Projection Relvars

Consider a user who sees just the projection relvars ST and TC. That user:

1. Knows the corresponding predicates:

   ST: *Supplier SNO is under contract and has status STATUS.*

   TC: *Some supplier is under contract and has status STATUS and is located in city CITY.*

2. Is aware that {SNO} is a key for relvar ST and that TC is "all key," and is aware also of the following constraint:

```
CONSTRAINT ... ST { STATUS } = TC { STATUS } ;
```

3. Is aware that updates on TC always fail (absent an appropriate "simultaneous" update on relvar ST) and that DELETEs on ST are subject to the following rule:

```
ON DELETE d FROM ST :
   DELETE ( ( TC MATCHING d ) NOT MATCHING ST ) FROM TC ;
```

Note that there's no rule for INSERTs on ST. In fact, such operations will fail unless both of the following are true: (a) the SNO value in each new tuple doesn't currently occur in ST, and (b) the STATUS value in each new tuple does currently occur in TC, but only once. As for explicit UPDATEs, the remarks in the second to last paragraph in the previous subsection apply.

Information Hiding

Of course, Example 3 already involves information hiding, since we don't have information equivalence in the first place. But we can hide even more (as it were) by presenting a user with just one of the two projection relvars, either ST or TC, in isolation. What operations can the user perform in such a situation? Well, in the case of ST, DELETEs make sense, but nothing else does; in the case of TC, no updates make sense at all.

Views

Finally, everything I've said in this section so far applies essentially unchanged if some or all of the relvars concerned are views. To be specific, suppose now that S is a base relvar and ST and TC are views. Then:

1. The constraints specified by the various CONSTRAINT statements given earlier in this section will all be enforced automatically, precisely because ST and TC are views of S.

2. The compensatory actions from S to ST and TC will happen automatically, precisely because ST and TC are views of S; that is, updates to S will automatically be reflected appropriately in ST or TC or both.

3. The compensatory actions from ST and TC to S—these are the ones that are generally thought of as the view updating rules as such—will also happen automatically, again precisely because ST and TC are views of S.[13] That is, updates to ST and/or TC are "really" updates to the underlying relvar S, and so are automatically visible in S, as well as in ST or TC or both.

4. A user who sees only views ST and TC can behave exactly as if those views were base relvars (to the extent described previously, that is, no more and no less, for the case in which they actually were base relvars).

   Note: There's a point here you might find a trifle odd, however, regarding explicit UPDATE operations in particular. I've said that attempts to change a supplier's status in relvar ST will either fail or have undesirable side effects (in general), and hence that such operations probably shouldn't be requested in the first place. Under the conventional approach to view updating, however (such as it is), such operations don't seem to be problematic at all; that is, they seem to work as intuitively expected. *But that's because the conventional approach tacitly assumes information equivalence*. That is, in the case at hand, the conventional approach effectively assumes we're dealing with a design consisting of relvars ST and *SC* (as in Example 1, earlier in the chapter), not a design consisting of relvars ST and *TC* as in the present discussion. In Example 1, as you'll recall, attempts to change a supplier's status in relvar ST worked just fine.

5. By contrast, a user who sees just view ST or just view TC will clearly be limited in the operations he or she is allowed to perform—again exactly as previously described in the present section.

---

[13] Actually there aren't any compensatory actions for relvar TC, but of course that fact doesn't invalidate the general point being made here.

CONCLUDING REMARKS

There are two final remarks I want to make. First, I haven't said quite all I want to say about updating projection views as such—I'll come back to the topic briefly in Chapter 7, in the section "Projection Views Revisited." Second, I'd like to clarify something I ought perhaps to have clarified much earlier ... In Chapter 1, I claimed that all views are updatable; in this chapter, by contrast, I've said that certain updates on certain views don't work after all. As an extreme case in point, view TC—defined as the projection of relvar S on STATUS and CITY, in the case where the functional dependency {CITY} → {STATUS} doesn't hold—can't sensibly be updated at all. So am I talking out of both sides of my mouth here? What exactly is going on?

Well, I did also say in Chapter 1 that I would elaborate later on "this very strong claim" (the claim, that is, that all views are updatable). The point is this: It's *not* that certain views are intrinsically nonupdatable. Rather, it's that certain updates on certain views fail because, if they—the updates, that is—were accepted, they would cause some integrity constraint to be violated (and no compensatory actions are in place to prevent such violations from occurring.) Note that this state of affairs exactly parallels the situation with base relvars: Certain updates on certain base relvars fail too, not because the relvar in question is intrinsically nonupdatable, but because some integrity constraint would otherwise be violated. Inserting a tuple for a nonexistent supplier into the shipments relvar SP is a case in point.

Chapter 6

Join Views I:

One to One Joins

This phrase I have coined
For views to be joined—
Updating through V
Updates A, also B

—Anon.: *Where Bugs Go*

I'd like to begin this chapter by repeating something I said at the end of Chapter 3:

> The topic of view updating can unfortunately be quite confusing, even when it's not particularly controversial. Part of the difficulty lies in the fact that there's unavoidably a lot of detail to wade through, and it's easy to lose one's way in debates and discussions. In particular, it's all too easy to forget, especially in examples, which relvars are base ones and which ones are views. It's important to keep a clear head!

I repeat these remarks here because I think it's only fair to warn you that they seem to be especially applicable in the case of join views in particular.

Given the foregoing, I thought it might be helpful to give some idea right at the outset of where we're going to wind up in our investigations into this topic. Our target, of course, is a set of rules for updating through join. And what I claim we're going to finish up with is a set of rules—a single, uniform set of rules—that work for all joins.[1] In particular, it's *not* going to make any difference whether the join we're dealing with is one to one, one to many, or many to many. In fact, let me immediately show in outline what the rules in question look like. Assume we're trying to update a view *V* defined as *A* JOIN *B*. Then the rules can be stated as follows:

```
ON INSERT INTO V : INSERT A (sub)tuples if they don't already exist,
                   INSERT B (sub)tuples if they don't already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don't exist elsewhere,
                   DELETE B (sub)tuples if they don't exist elsewhere
```

[1] Some might say "work" should be in quotation marks here. What I mean by my claim is: The rules do at least produce well defined, predictable results in all circumstances. Whether those results are always acceptable is another question! See later for further discussion.

Of course, the rules as just stated are loose in the extreme; nevertheless, I think this rough and ready formulation should help you hang on to the big picture as we struggle through all of the detailed discussions to follow in this chapter and the next two.

EXAMPLE 1: INFORMATION EQUIVALENCE

Because join is so important, and because there are so many issues we need to discuss in connection with it, I've decided to split my treatment of the topic into three separate chapters. In this first one, I want to limit my attention to what might be regarded as the simplest possible case: viz., the case in which the join in question is a one to one join specifically. As usual, I'll base my discussions on some simple examples.

My first example is effectively the inverse of the first of the projection examples from the previous chapter. (As noted in that chapter, there's a tight connection between join views and projection views in general.) To be specific, suppose we're given base relvars ST and SC, looking like this (in outline):

```
ST { SNO , STATUS } KEY { SNO }
SC { SNO , CITY } KEY { SNO }
```

(As in Chapter 5, I ignore attribute SNAME for simplicity.) Now suppose we define the join of these two relvars, ST JOIN SC, as a view S:

```
S { SNO , STATUS , CITY } KEY { SNO }
```

Further, let's assume this join is strictly one to one, in the sense that every tuple in ST joins to exactly one tuple in SC and vice versa. In other words, we have information equivalence—the design consisting of ST and SC taken together and the design consisting of just view S are clearly information equivalent. Sample values are shown in Fig. 6.1 (of course, that figure is identical to Fig. 5.1 in Chapter 5, but now ST and SC are base relvars and S is a view).

S

SNO	STATUS	CITY
S1	20	London
S2	10	Paris
S3	30	Paris
S4	20	London
S5	30	Athens

ST

SNO	STATUS
S1	20
S2	10
S3	30
S4	20
S5	30

SC

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

Fig. 6.1: Relvars S, ST, and SC—sample values

Here's the predicate for the join (of course, it's basically the usual predicate for relvar S, but simplified to take account of the fact that we've dropped attribute SNAME):

S: *Supplier SNO is under contract, has status STATUS, and is located in city CITY.*

As usual, the first thing to do is consider what happens if all three relvars are base ones. But we've already done that, in the discussion of Example 1 in Chapter 5! So let me just repeat here the salient points from that discussion. First, as already indicated, each of the three relvars has {SNO} as its sole key. The following constraints also hold (as noted in Chapter 5, they're all equality dependencies (EQDs), and together they ensure that the two designs are, as I've already said, information equivalent):

```
CONSTRAINT ... S  = JOIN { ST , SC } ;
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;
```

Actually, if you check, you'll see I gave these constraints in a different order in the previous chapter. I've reordered them here to show that, first, S is indeed equal to the join of the other two relvars; next—in order to guarantee the desired information equivalence—ST and SC are indeed equal to the corresponding projections of S; and finally, every supplier number appearing in S also appears in both ST and SC and vice versa. *Note:* In fact, of course, this final constraint is a logical consequence of certain of the other three constraints taken together (which ones, exactly?).

The compensatory actions are the same as before, too, though again I've changed the sequence:

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
           t3 := S MATCHING dt , t4 := S MATCHING dc ) :
   INSERT t1 INTO S ,
   INSERT t2 INTO S ,
   DELETE t3 FROM S ,
   DELETE t4 FROM S ;

ON DELETE d FROM S , INSERT i INTO S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE d { SNO , CITY } FROM SC ,
   INSERT i { SNO , STATUS } INTO ST ,
   INSERT i { SNO , CITY } INTO SC ;
```

Let me remind you also that INSERT and "key UPDATE" operations on ST and SC must be "double updates," for otherwise they'll fail on a **Golden Rule** violation.

Now suppose as we originally did that S is in fact a view, the join of ST and SC. Then:

1. The fact that S is a view of ST and SC is *not* sufficient to ensure that the constraints specified by the various CONSTRAINT statements shown above (with the exception of the first one) will be enforced automatically. The compensatory actions aren't sufficient, either. For example, neither the fact that S is the join of ST and SC, nor the compensatory actions, are sufficient to prevent the insertion of a tuple for supplier S9 into ST without the simultaneous insertion of a tuple for S9 into SC. Thus, the pertinent constraints need to be explicitly stated (and explicitly enforced by the DBMS, of course).

2. However, the compensatory actions from ST and SC to S will happen automatically, precisely because S is a view; that is, updates on ST and/or SC will automatically be reflected appropriately in S.

3. The compensatory actions from S to ST and SC—these are the view updating rules as such—will also happen automatically, again precisely because S is a view. That is, updates on S are "really" updates on the underlying relvars ST and/or SC, and so are automatically visible in S, as well as in ST or SC or both.

Also, a user who sees only the join view S can behave in all respects exactly as if that view were a base relvar. Such a user will be aware of the fact that {SNO} is a key (but not of any other constraints, nor of any compensatory actions), and will be aware also of the corresponding predicate:

S: *Supplier SNO is under contract, has status STATUS, and is located in city CITY.*

And, to spell the point out, INSERTs, DELETEs, and explicit UPDATEs on that view S all work exactly as they would if S were a base relvar instead.

EXAMPLE 2: INFORMATION HIDING

So updating a one to one join view, in the case where we have information equivalence, is essentially straightforward. But what if we're dealing with an information hiding situation? By way of a concrete example, suppose a supplier can be represented in base relvar ST and not base relvar SC or the other way around; in other words, instead of requiring every supplier to have both a status and a city, as we did in Example 1, suppose now only that every supplier is required to have at least one of those two properties.[2] Then the join S of ST and SC loses information, inasmuch as it's no longer guaranteed that ST and SC are equal to the projections of that join on the appropriate attributes. In fact, of the constraints mentioned in the previous section, the only

[2] "At least one," because allowing a supplier to have neither would make no sense—such a supplier wouldn't appear in either ST or SC and therefore wouldn't appear in S (= ST JOIN SC) either.

ones that continue to hold are the usual key constraints ({SNO} is a key for each of S, ST, and SC), together with this one:[3]

```
CONSTRAINT ... S = JOIN { ST , SC } ;
```

Sample values satisfying the foregoing conditions can be obtained by modifying Fig. 6.1 to remove the tuple for supplier S5 from each of relvars S and ST but not from relvar SC (see Fig. 6.2). Of course, the predicate for S is still as it was for Example 1.

S

SNO	STATUS	CITY
S1	20	London
S2	10	Paris
S3	30	Paris
S4	20	London

ST

SNO	STATUS
S1	20
S2	10
S3	30
S4	20

SC

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

Fig. 6.2: A revised version of Fig. 6.1

A remark on terminology: The title of this chapter implies, or at least strongly suggests, that ST JOIN SC is still a one to one join, but of course it isn't—not really. The truth is, the term "one to one" is often used somewhat loosely to include both (a) what might be called the strict case, as in Example 1 in the previous section, and (b) cases like the one currently under discussion, in which it's possible for one participant (in whatever relationship it is that we happen to be talking about) to have an element with no counterpart in the other. Here's a lightly edited quote from *The Relational Database Dictionary, Extended Edition* (Apress, 2008):

> A one to one correspondence is, strictly, a rule pairing two sets *s1* and *s2* (not necessarily distinct) such that each element of *s1* corresponds to exactly one element of *s2* and each element of *s2* corresponds to exactly one element of *s1*. However, the term is often used somewhat loosely to mean a pairing such that (a) each element of *s1* corresponds to at most one element of *s2* and each element of *s2* corresponds to exactly one element of *s1*, or (b) each element of *s1* corresponds to exactly one element of *s2* and each element of *s2* corresponds to at most one element of *s1*, or (c) each element of *s1* corresponds to at most one element of *s2* and each element of *s2* corresponds to at most one element of *s1*. The term is probably best avoided unless the intended meaning is clear.

The final sentence here notwithstanding, I will in fact be using the term quite a lot in what follows, because I think my intended meaning will always be clear from the context. *End of remark.*

---

[3] Certain inclusion dependencies (INDs) hold too, but I won't bother to spell them out in detail.

Back to Example 2. Let me now stress the fact that, precisely because the join view S loses information, a database containing just that view isn't information equivalent to one containing base relvars ST and SC. (An example of a query on the latter that has no counterpart on the former is "Get supplier numbers for suppliers who have a city but no status.") More precisely, any information that can be represented by S alone can certainly be represented by the combination of ST and SC, but the converse is false. As a consequence, it's obvious that there'll be updates on ST and/or SC that have no counterpart on S. An example is "Insert a tuple into SC for a supplier with supplier number S9 and city London," without simultaneously inserting a tuple for that same supplier S9 into ST.

So what compensatory actions apply? Well, since there's no longer a strict one to one relationship between relvars ST and SC, it's now possible as just indicated to insert into, or delete from, just one of the two without simultaneously inserting into or deleting from the other. Thus, since S is required to be at all times equal to the join of ST and SC, we clearly have:

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   INSERT ( it JOIN SC ) INTO S ,
   INSERT ( ic JOIN ST ) INTO S ;

ON DELETE dt FROM ST , DELETE dc FROM SC :
   DELETE ( S MATCHING dt ) FROM S ,
   DELETE ( S MATCHING dc ) FROM S ;
```

Now, I've shown these rules separately for reasons of clarity, but of course they can be combined into one:

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   DELETE ( S MATCHING dt ) FROM S ,
   DELETE ( S MATCHING dc ) FROM S ,
   INSERT ( it JOIN SC ) INTO S ,
   INSERT ( ic JOIN ST ) INTO S ;
```

And if you compare this rule with the corresponding rule for Example 1 from the previous section—the strict one to one case—you'll see it's essentially identical (except that I used some introduced names in that previous section, for clarity). Here in order to make it easier to do the comparison is that previous rule:

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := S MATCHING dt , t4 := S MATCHING dc ) :
   INSERT t1 INTO S ,
   INSERT t2 INTO S ,
   DELETE t3 FROM S ,
   DELETE t4 FROM S ;
```

So much for updates on ST and/or SC. But what about the join view?—i.e., what happens if we do INSERTs or DELETEs on S, instead of on ST and/or SC? In fact INSERTs are reasonably straightforward:

```
ON INSERT i INTO S :
   INSERT i { SNO , STATUS } INTO ST ,
   INSERT i { SNO , CITY } INTO SC ;
```

Note that inserting a tuple *t* into S must cascade to both ST and SC. (It's true that one of those cascaded INSERTs might in fact be a "no op," but they can't both be.)[4] For (a) if S is a view—that's the case we're considering—and we were to cascade to (say) just ST, and no matching tuple previously existed in SC, then tuple *t* would never appear in S and we'd have an *Assignment Principle* violation on our hands; conversely, (b) if all three relvars were base ones and (again) we cascaded to just ST and no matching tuple previously existed in SC, then S would no longer be equal to ST JOIN SC and we'd have a **Golden Rule** violation on our hands instead. Indeed, the very fact that these two different scenarios give rise to two different exceptions suggests rather strongly (in accordance with *The Principle of Interchangeability*) that we need to have appropriate compensatory actions in place in order to hide the difference.

> *Aside:* There's still an issue here, though. E.g., given the sample values shown in Fig. 6.2, an attempt to insert a tuple into S for supplier S5 will violate the key constraint on SC if the city in that new tuple is anything other than Athens. But a user who sees only the join view S doesn't know about relvar SC. At the same time, we surely don't want to prohibit INSERTs on S entirely. Maybe this is a case where an operation has to be rejected simply "because the system says so," without further explanation (?). Similar remarks apply to (say) an attempt to change the supplier number in S for supplier S1 to S5. *End of aside.*

Observe now that the foregoing rule for INSERTs on S is identical to the corresponding rule for Example 1 from the previous section (the strict one to one case). Observe further that—as suggested in the introduction to this chapter—that rule can be loosely characterized as follows: "On inserting tuple *t* into *A* JOIN *B*, insert the *A* portion of *t* into *A* unless it already exists there, and insert the *B* portion of *t* into *B* unless it already exists there." *Note:* Of course, by *the A portion of t*, what I really mean is that subtuple of *t* that's the (tuple) projection of *t* on the attributes of *A* (and similarly for *the B portion of t*). And by *it already exists there* (where "it" refers to either the *A* portion or the *B* portion of *t*), I mean there's already a tuple in the pertinent relvar (*A* or *B*) that's equal to the pertinent subtuple.

So much for INSERTs on S. What about DELETEs? Well, first let me reiterate that we're talking here about the case in which a constraint to the effect that ST{SNO} = SC{SNO} has ***not*** been stated. Of course, the fact that such a constraint hasn't been stated doesn't mean it isn't supposed to hold!—it might or it might not (be supposed to hold, that is). To put the point

---

[4] They would both be if *t* already existed in S, but recall from Chapter 4 that we don't need to discuss that case.

another way, the fact that the DBMS doesn't know the constraint is supposed to hold doesn't mean it knows it isn't (if you follow me).[5] So the question becomes: What should the DBMS do—(a) assume the constraint holds anyway, or (b) assume it doesn't? (Of course, it goes without saying that it does have to operate on the basis of one or other of these two assumptions.) Now, if the DBMS operates under assumption (a), it will do its best to ensure that suppliers are represented in both ST and SC (if they're represented at all, that is), even though a supplier not so represented won't cause a violation of **The Golden Rule**. (It won't cause a violation of **The Golden Rule** precisely because the constraint hasn't been stated.) In fact, if the DBMS does operate under assumption (a), the updating rules—the delete rules in particular—become exactly the same as those for the strict one to one case, and that's the end of the discussion. So what happens if it operates under assumption (b)?

Before I try to answer that question, let me just state for the record that assumption (b) doesn't correspond to a constraint—it corresponds to the *absence* of a constraint. I mean, I hope it's obvious that there's no formal CONSTRAINT statement we can write that says it's legal for some supplier number to appear in ST and not SC or the other way around. (I mention this point because in fact I've encountered people who apparently believe the opposite. If you're such a person yourself, then I suggest you try writing such a CONSTRAINT statement. If you try this exercise, I think it'll quickly convince you that what I'm saying here is correct—if you weren't convinced already, that is.)

Be that as it may, let's take a closer look at assumption (b). Let's consider a concrete example. Suppose we try to delete the tuple (S1,20,London) from S. Clearly, the DBMS could achieve the desired effect by doing any of the following: 1. deleting the tuple (S1,20) from ST; 2. deleting the tuple (S1,London) from SC; or 3. doing both. In other words, we might consider any of the following as a possible delete rule for S:

1. Cascade the delete to just ST.

2. Cascade the delete to just SC.

3. Cascade the delete to both ST and SC.

Note: There's a fourth possibility too, of course: Reject the delete entirely, on the grounds that we don't know which of the other three options to choose. On the grounds that it's surely desirable in general for user requests to succeed if they sensibly can, however, it seems preferable to accept the delete and let it cascade appropriately—especially since cascading (more specifically, cascading to both ST and SC, option 3) is the logically correct thing to do if

---

[5] In symbols, NOT(*knows*(*p*)) doesn't imply *knows*(NOT(*p*)). Or as somebody or other once said: "Absence of evidence isn't evidence of absence." Mind you, if a certain constraint is supposed to hold but the DBMS hasn't been informed of that fact, then the design is certainly incomplete, and errors of various kinds are certain to ensue. But such violations of good design principles aren't exactly unknown in practice.

assumption (a) is in fact the right one (i.e., if ST{SNO} and SC{SNO} are supposed to be equal after all, even though no constraint to that effect has been explicitly stated).

Now, as I'm sure you've realized, we're dealing with a big issue here. Indeed, what we're talking about is the classic ambiguity issue (or what some people describe as an ambiguity issue, at any rate). To spell the point out, the fact that there doesn't seem to be a good way to choose among the various options is exactly what critics complain about; I mean, it's exactly why some writers believe view updating is, in the general case, impossible. So I want to present arguments in favor of my own position, which is that I think we should go with option 3 (i.e., cascade to both). Before I do that, however, let me just point out that no analogous problem arises in connection with the projection counterpart to the join example under discussion, even though (as I've said previously) join views and projection views are two sides of the same coin. The reason is this: If we start with relvar S and replace it by its projections ST and SC, then certainly any information that can be represented by the original design can also be represented by the revised design. But if we start with relvars ST and SC and replace them by their join S, then it's possible that some information can be represented by the original design and not by the revised design—in which case information equivalence is lost. And it's only if information equivalence is lost that the ambiguity issue arises.

Pragma

As I've said, I'm going to present arguments in favor of the position that we should go with option 3—the position, in other words, that we should treat Example 2 as far as possible just like Example 1. But first let me remind you of something I said in Chapter 3:

> Even when we don't have information equivalence, sometimes it'll be possible to apply the view updating rules anyway (albeit only partially, perhaps), and I'll discuss such cases in detail too. ***But I must stress that updating in such cases can lead to results that, while of course always formally predictable and well defined, might sometimes be undesirable—possibly even unacceptable for some reason).*** Because of this state of affairs, I leave to others (perhaps the individual user, perhaps the DBA, perhaps even the DBMS) the choice as to whether updates should in fact be allowed in such situations.

The case at hand is a perfect example of the foregoing; to go with option 3 means, precisely, applying the rules that work in the information equivalence case to the case where information is hidden, or lost. In other words, there's a clear element of pragma involved in my position here.[6] Now, I'm going to do my best to convince you that the pragma in question makes sense—in particular, as the extract from Chapter 3 says, it does at least produce results that are formally predictable and well defined—but, as that extract also makes clear, it's entirely possible that others will disagree with my position here and so might want to opt out (as it were) of my

[6] Here and elsewhere in this book I choose to overlook the fact that none of the dictionaries I consulted seem to recognize *pragma* as a legitimate English word.

proposals on this particular issue. I'll come back and revisit this possibility at the very end of the subsection immediately following. First, however, let me try to explain my own position.

Symmetry

I believe symmetry is a good design principle in general. To quote Polya:

- "If a problem is symmetric in some ways we may derive some profit from noticing its interchangeable parts and it often pays to treat those parts which play the same role in the same fashion ... Try to treat symmetrically what is symmetrical, and do not destroy wantonly any natural symmetry" (George Polya: *How To Solve It*, 2nd ed., Princeton University Press, 1971).

- "We expect that any symmetry found in the data and condition of the problem will be mirrored by the solution ... Symmetry should result from symmetry" (George Polya: *Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving*, 2nd ed., John Wiley & Sons, 1981).

  Aside: Polya also articulates in this latter book what he calls *The Principle of Nonsufficient Reason:* "No [solution] should be favored of eligible possibilities among which there is no sufficient reason to choose." Some might appeal to this principle as a basis for rejecting deletes on join views entirely: at least, join views like the one currently under discussion. To me, however, that position is too extreme; it throws the baby out with the bathwater, as it were. Let me continue to present what I regard as good reasons—well, fairly good reasons, anyway—for not taking such a drastic step but, rather, going with option 3. *End of aside.*

In my experience, asymmetry often means we've got something wrong. At the very least, it can lead to counterintuitive behavior, behavior that can seem capricious and, precisely for that reason, hard to understand, teach, learn, and remember. In the case at hand, an asymmetric solution would mean cascading deletes on S to ST but not SC (option 1) or the other way around (option 2). This arbitrariness has at least two unpleasant consequences. First, it means the DBA, or some other human agency, might to have to get involved in order to choose between the two options (without, I might add, any good guidelines to help in making that choice, in general). Second, it raises the possibility that a view defined as ST JOIN SC and one defined as SC JOIN ST might have different update behavior, a state of affairs that's surely undesirable. So I think considerations of symmetry alone are sufficient to make options 1 and 2 nonstarters.

I observe further that options 1 and 2 are actually the wrong choice if assumption (a) is in fact the right one. That is, options 1 and 2—and assumption (b)—are a safe way to go only if it's definitely the case that a supplier can have a status but no city or vice versa. But the DBMS doesn't *and can't* know that such is definitely the case if all it knows is that S = ST JOIN SC.

And if assumption (b) is in fact the right one after all, at least option 3 still works, and it produces a predictable result.

So let's consider option 3. Obviously, that option has the advantage that it avoids the foregoing problems. But there's another reason why that option is attractive—another appeal to symmetry, in fact—and that is that, under option 3, the delete rule is symmetric with respect to its insert rule counterpart. (We've already seen that inserts on S cascade to both ST and SC, so why shouldn't deletes on S be treated similarly?) In fact, option 3 implies, loosely speaking, that (a) deleting an existing tuple *t* from S and then inserting it again, and (b) inserting a new tuple *t* into S and then deleting it again, both preserve the status quo,[7] a state of affairs that seems intuitively both reasonable and desirable.

> *Aside:* I need to elaborate somewhat on the foregoing, however. When I say "preserve the status quo," I'm referring to the status quo with respect to relvar S (the join view) specifically, and my claim is 100 percent correct. The picture with respect to relvars ST and SC is unfortunately a little more complicated. To spell the matter out: The status quo with respect to those relvars is preserved if (a) an existing tuple is deleted from S and then inserted again, but not necessarily if (b) a new tuple is inserted into S and then deleted again. The reason is that inserting a new tuple into S might actually cause a tuple to be inserted into just one of ST and SC, whereas under option 3 deleting an existing tuple from S will always cause a tuple to be deleted from both.
>
> There's another (related) issue as well. Suppose we start off with just relvars ST and SC (i.e., without the join view). Let supplier S1 be represented in both ST and SC. Clearly, then, we can delete supplier S1 from relvar ST, say, without that delete cascading to relvar SC. But now suppose we introduce the join view S. Under option 3, then, that delete *will* now cascade to relvar SC—and it'll do so, moreover, even if the join view S isn't visible to the user issuing the delete on ST! I'll revisit this particular issue, and several similar ones, in Chapters 9–11, also in Chapter 15. *End of aside.*

Another argument in favor of option 3 is the following: If we can agree on that option, then we'll have a single, uniform rule that applies universally to the question of deleting through join, regardless of whether the join in question is one to one, one to many, or many to many. Of course, I haven't demonstrated this point yet, but I will, over the course of the next two chapters.

So my conclusion is this: I think it's better for the DBMS to operate under assumption (a), even if the corresponding constraint hasn't been stated explicitly. In other words, I think the view updating rules given in the previous section for the strict one to one case should be followed even if the join is only known to be one to one in the looser sense of that term. And I point out now that—as suggested in the introduction to this chapter— the option 3 delete rule can be loosely characterized as follows: "On deleting tuple *t* from *A* JOIN *B*, delete the *A* portion of *t* from *A* and the *B* portion of *t* from *B*" (and we could harmlessly add "unless it exists elsewhere

---

[7] The same would be true of options 1 and 2, though.

in the relvar" to each part of this rule, since we're talking about a one to one join and the *A* portion of *t* simply won't exist elsewhere in *A*, nor will the *B* portion of *t* exist elsewhere in *B*).

If we can agree on the foregoing position, then we'll have agreed on a universal set of rules for updating one to one join views that do at least always work and do guarantee that joins are strictly one to one when they're supposed to be. What's more, if those rules do sometimes give rise to consequences that are considered unpalatable for some reason, then there are always certain ad hoc fixes (such as using the DBMS's authorization subsystem to prohibit certain updates) that can be adopted to avoid those consequences.

Note: Please don't misunderstand me here. I'm not saying we *must* employ such fixes in order for the system to work properly. A system that relies for its correct operation on the user, or the DBA, always "doing the right thing"—e.g., using the authorization subsystem appropriately—is obviously not acceptable.[8] So we must always at least permit join view updates, even when the joins aren't strictly one to one, and we must have a set of rules that work even in that case. That's why I advocate the position I do.[9]

CONCLUDING REMARKS

There are a couple of final points I'd like to make in connection with updating one to one joins. The first is this. In terms of our running example, I've considered two possibilities: Every supplier has both a status and a city, or every supplier has at least one of those two properties. But there are at least two further possibilities:

- The properties are mutually exclusive (i.e., every supplier has exactly one of them).
- One property is mandatory but the other is optional (e.g., every supplier has a city, and some suppliers have a status as well).

A moment's reflection is sufficient to show that the first of these possibilities isn't very interesting. The reason is that such a supplier would always be represented in either ST or SC but not both; the join S would therefore always be empty, and deletes on that join would always

---

[8] And here is as good a place as any to make the more general point that whatever view updating rules we do adopt, they certainly can't rely on the database being well designed (e.g., they can't assume relvars are always properly normalized). In other words, we must allow the database designer the freedom to make a mess of things. *Note:* That said, I feel bound to say too that much of the complexity (such as it is) that arises with view updating in general arises precisely when the design we're dealing with is a bad one—in particular, when it's incomplete, in the sense that it's incapable of capturing all of the facts about the real world that it ought to be capable of capturing.

[9] The idea that if you don't like what happens when you do *X,* then you shouldn't do *X*—"If it hurts when you hit yourself over the head with a hammer, then don't hit yourself over the head with a hammer"—is sometimes referred to as *The Groucho Principle*. My suggestion here (viz., if you don't like what happens when you do certain updates, then don't do those updates) can be seen as an appeal to that principle. Now, I would certainly agree in general that if the only argument you can find in support of some position is an appeal to *The Groucho Principle*, then that position must be pretty weak. But, of course, I don't believe in the case at hand that an appeal to *The Groucho Principle* is the only argument I have to support my position.

be a "no op" and inserts on that join would always fail (unless they're "no ops" as well). As for the second possibility, I suppose it could be argued that this is a case where an option 1 or option 2 delete rule might make sense. Well, maybe; I don't find the argument very convincing myself; but I suppose I could be persuaded otherwise if it could be shown there was a really strong requirement here. For the time being, however, I propose to stay with my "one size fits all" rule as described earlier in the chapter (which does of course still work, even in the rather special case under discussion here).

My other point has to do with the fact that—as is of course well known—intersection is a special case of one to one join.[10] It follows that the rules for updating through a one to one join apply to intersection as well (more precisely, they reduce to the rules for updating through an intersection, in the special case where the join itself reduces to an intersection). To spell those rules out (albeit in simplified form),[11] let *V* be defined as *A* INTERSECT *B*. Then we have:

```
ON INSERT i INTO A :
   INSERT ( i INTERSECT B ) INTO V ;

ON INSERT i INTO B :
   INSERT ( i INTERSECT A ) INTO V ;

ON DELETE d FROM A :
   DELETE d FROM V ;

ON DELETE d FROM B :
   DELETE d FROM V ;

ON INSERT i INTO V :
   INSERT i INTO A ,
   INSERT i INTO B ;

ON DELETE d FROM V :
   DELETE d FROM A ,
   DELETE d FROM B ;
```

Now, I'll have a lot more to say about intersection as such in Chapter 9, but I wanted to mention it here because it allows me to spell out what earlier in this chapter I referred to as "the classic ambiguity issue" in logical terms. As I've said, the rule for deleting through a one to one join, in the case where the term "one to one" is being used only loosely (as in Example 2), does involve a certain degree of pragma. Well, now I can pin down exactly what that pragma consists of. Recall from Chapter 2 that every relvar, and more generally every relational expression, has an associated predicate (in the latter case, the predicate is derived from the predicates for the relvars involved in the expression, in accordance with the semantics of the relational operations involved in that expression). So let the predicates for *A* and *B* be *PA* and *PB*, respectively; then

---

[10] Strict or otherwise—though in fact "strict" makes little sense here (why?).

[11] I'll refine these rules slightly in Chapter 9.

the predicate for $V = A$ INTERSECT B is PA AND PB. That is, if tuple t appears in V, then $PA(t)$ AND $PB(t)$ is true (implying, of course, that $PA(t)$ and $PB(t)$ are both individually true).[12]

Now suppose we delete t from V. What we mean by that update is that the proposition $PA(t)$ AND $PB(t)$ is now no longer true; in other words, either $PA(t)$ is false OR $PB(t)$ is false (and possibly both). By contrast, deleting t from both A and B means $PA(t)$ is false AND $PB(t)$ is false; in other words, both are false. And so here we have the ambiguity issue in logical terms—namely, the proposed option 3 rule for deleting through a one to one join, in the case where the term "one to one" is being used only loosely, is tantamount to treating logical OR as logical AND.

Note: We might alternatively say that deleting t from V means NOT($PA(t)$ AND $PB(t)$) is true while deleting t from both A and B means NOT($PA(t)$) AND NOT ($PB(t)$) is true. But NOT($PA(t)$) AND NOT($PB(t)$) is equivalent to NOT($PA(t)$ OR $PB(t)$), and hence the option 3 delete rule means we're effectively treating logical AND as logical OR (the other way around, in other words). Of course, it makes no real difference to the objection either way.

I don't want to explore this issue any further at this juncture; suffice it to say that David McGoveran has a proposal for resolving it, which I'll describe in Chapter 15.

[12] I use the expression $PA(t)$ to denote the proposition obtained by instantiating predicate PA by using attribute values from tuple t as arguments to replace PA's parameters, and similarly for $PB(t)$.

Chapter 7

Join Views II:

Many to Many Joins

Recent researches at Harvard
And further researches at Yale
Show that joins can all be updated
By removing the spikes from their tail

—Anon.: *Where Bugs Go*

Now I want to examine the question of many to many joins (I'm deliberately leaving the one to many case till last). As in Chapter 6, I think it might help to state right at the outset where our investigations are going to take us. As you might intuitively expect, the many to many case is going to turn out to involve certain complications, complications that didn't arise in the one to one case; nevertheless, I claim we're still going to wind up with essentially the same rules as before. That is, given a view *V* defined as *A* JOIN *B*—where the join is now a many to many join specifically—the rules are still going to look like this (in outline):

```
ON INSERT INTO V : INSERT A (sub)tuples if they don't already exist,
                   INSERT B (sub)tuples if they don't already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don't exist elsewhere,
                   DELETE B (sub)tuples if they don't exist elsewhere
```

EXAMPLE 1: INFORMATION EQUIVALENCE

Consider a simplified version of our usual suppliers and parts relvars in which suppliers have no attributes except SNO and CITY and parts have no attributes except PNO and CITY, thus:

```
S { SNO , CITY } KEY { SNO }
P { PNO , CITY } KEY { PNO }
```

Suppose also for the sake of the example that every supplier city is required to be a part city and vice versa—in other words, there's a constraint in effect (actually an equality dependency once again) that looks like this:

```
CONSTRAINT ... S { CITY } = P { CITY } ;
```

In order to conform to this new requirement, let's agree until further notice to drop the tuple for supplier S5 (city Athens) from our usual suppliers relation and the tuple for part P3 (city Oslo) from our usual parts relation.

Now let's define the join of relvars S and P as a view SCP:

```
SCP { SNO , CITY , PNO } KEY { SNO , PNO }
```

Observe that the join here is indeed many to many, in the sense that (in general) every tuple in S joins to many tuples in P and vice versa. Nevertheless, we do have information equivalence, precisely because every tuple in either relvar does join to at least one tuple in the other. Sample values are shown in Fig. 7.1.

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. 7.1: Relvars S, P, and SCP—sample values

Here now are the predicates:[1]

S: *Supplier SNO has city CITY.*

P: *Part PNO has city CITY.*

[1] I've simplified these predicates slightly (a) to drop "is under contract" (for suppliers) and "is used in the enterprise" (for parts), and (b) to say that suppliers and parts both "have" a city (instead of saying suppliers are "located in," and parts are "stored in," a city). These changes are purely cosmetic, of course; I make the first merely for simplicity and the second in order to stress the parallel nature of the roles being played by suppliers and parts in this example.

SCP: *Supplier SNO and part PNO both have city CITY.*

As usual, I'll begin by considering what happens if all three relvars are base ones. If they are, what constraints hold? Well, first, the three relvars have {SNO}, {PNO}, and {SNO,PNO}, respectively, as their sole key (as already indicated). Second, we also clearly have:

```
CONSTRAINT ... SCP = S JOIN P ;
CONSTRAINT ... S   = SCP { SNO , CITY } ;
CONSTRAINT ... P   = SCP { PNO , CITY } ;
```

These constraints are, of course, all equality dependencies (EQDs), and together they ensure that the two designs—that consisting of just relvar SCP and that consisting of the combination of relvars S and P—are information equivalent, as previously claimed.

The following constraints—also EQDs—hold as well (actually they're just logical consequences of the three just stated):

```
CONSTRAINT ... IDENTICAL { S { CITY } , P { CITY } , SCP { CITY } } ;
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

I want to elaborate somewhat on this last constraint, however. The fact that it holds means relvar SCP isn't in fourth normal form (4NF), because the constraint in question is equivalent to the following join dependency (JD):

```
☼ { { SNO , CITY } , { PNO , CITY } }
```

And this JD in turn is equivalent to the following pair of multivalued dependencies (MVDs):

```
{ CITY } →→ { SNO }
{ CITY } →→ { PNO }
```

Or writing them as a "one liner":

```
{ CITY } →→ { SNO } | { PNO }
```

These MVDs are neither trivial nor implied by the sole key {SNO,PNO} of SCP, and SCP is thus not in 4NF.[2] *Note:* Informally, what these MVDs mean is this: If tuples (*c*,*s*,*p*) and (*c*,*s′*,*p′*) appear in the relvar, then tuples (*c*,*s*,*p′*) and (*c*,*s′*,*p*) must appear as well (where *c* is a city, *s* and *s′* are supplier numbers, and *p* and *p′* are part numbers). By way of illustration of this point (with

[2] An MVD is trivial if and only if either (a) the determinant (the set of attributes on the left side) includes the dependant (the set of attributes on the right side) or (b) the union of the determinant and the dependant is equal to the entire heading; a nontrivial MVD is implied by a key if and only if the determinant is a superkey. Refer to *Database Design and Relational Theory* if you need further explanation of these matters.

reference to Fig. 7.1), take *c* as Paris, *s* and *s′* as S2 and S3, respectively, and *p* and *p′* as P2 and P5, respectively.

Of course, the fact that the foregoing MVDs hold in relvar SCP isn't a fluke—it will always be the case with a many to many join that MVDs like those just shown hold. And the reason I mention all this is that relvars in which such MVDs hold (i.e., relvars not in 4NF) are of course subject to redundancy, and it's generally true that relvars that are subject to redundancy can be awkward to update—as we'll see later, in the case at hand.

While I'm on the subject of dependencies, let me also point out that the following functional dependencies (FDs) hold in relvar SCP as well:[3]

```
{ SNO } → { CITY }
{ PNO } → { CITY }
```

These FDs are inherited from relvars S and P, respectively (where they certainly hold, because the determinant—the set of attributes on the left side—is a key in each case).

Compensatory Actions

Consider the following INSERT on relvar S:

```
INSERT ( S9 , London ) INTO S ;
```

Given the sample values in Fig. 7.1, this insert can and will succeed, just so long as it has the additional effect of inserting the following tuples into relvar SCP:

```
( S9 , London , P1 )
( S9 , London , P4 )
( S9 , London , P6 )
```

By contrast, consider this INSERT:

```
INSERT ( S9 , Madrid ) INTO S ;
```

Given the sample values in Fig. 7.1, this insert must fail, because there aren't any parts in Madrid (and there's no possible compensatory action that makes sense here, either). On the other hand, the following double INSERT can and will succeed—

```
INSERT ( S9 , Madrid ) INTO S ,
INSERT ( P8 , Madrid ) INTO P ;
```

—so long as it has the additional effect of inserting the following tuple into relvar SCP:

---

[3] As a consequence, SCP isn't even in second normal form (2NF), let alone fourth. But it's the violation of 4NF as such that leads to the relvar's characteristic behavior as a many to many join.

```
( S9 , Madrid , P8 )
```

From these examples and others like them, I hope it's clear that the following is an appropriate rule for inserts on S and/or P:

```
ON INSERT is INTO S , INSERT ip INTO P :
   INSERT ( P JOIN is ) INTO SCP ,
   INSERT ( S JOIN ip ) INTO SCP ;
```

Turning now to deletes: Suppose just for the moment that we reinstate the tuples for supplier S5 and part P3, where the city for S5 is Athens as usual, but we make the city for P3 Athens as well instead of Oslo. Thus, relvar SCP additionally contains the tuple (S5,Athens,P3)—but note that this tuple is the only one for Athens in that relvar. Now the following double DELETE clearly makes sense:

```
DELETE ( S5 , Athens ) FROM S ,
DELETE ( P3 , Athens ) FROM P ;
```

The effect is to delete the specified tuples from S and P and—assuming an appropriate cascade delete rule is in effect—also the tuple (S5,Athens,P3) from relvar SCP. But what about this single DELETE?—

```
DELETE ( S5 , Athens ) FROM S ;
```

Now, what can't be allowed to happen here is for the tuple (S5,Athens) to be deleted from relvar S, exactly as requested, while everything else remains unchanged. So one possibility would be simply to reject the specified DELETE on a violation of **The Golden Rule**. On the grounds once again that it's surely desirable in general for user requests to succeed if they sensibly can, however, it seems preferable to accept the DELETE and let it cascade appropriately. In other words, I propose the following as a suitable rule for deletes on relvars S and/or P:

```
ON DELETE ds FROM S , DELETE dp FROM P :
   DELETE ( SCP JOIN ds ) FROM SCP ,
   DELETE ( SCP JOIN dp ) FROM SCP ;
```

Given the foregoing insert and delete rules, I'll leave it as an exercise to show that explicit UPDATEs on S and/or P all work as expected, too.

I turn now to updates on the join SCP. The insert rule 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. 7.1:

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 )      ( S2 , Paris , P7 )
                         ( S7 , Paris , P5 )      ( S3 , Paris , P7 )
```

And now let me point out, just in case you haven't realized it already, that the foregoing insert rule is essentially the same as its counterpart as described in Chapter 6 for the one to one case (though of course there was no question in this latter case of the original INSERT cascading to cause additional tuples to be inserted into the join).

Now, given that the foregoing insert rule can be characterized, loosely, as "Insert S subtuples unless they already exist and insert P subtuples unless they already exist," intuition and symmetry both suggest the corresponding delete rule should be "Delete S subtuples unless they exist elsewhere and delete P subtuples unless they exist elsewhere." 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. 7.1:

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 for supplier S1 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).

Now, if you compare the foregoing delete rule with its counterpart as described in Chapter 6 for the one to one case (repeated here for convenience)—

```
ON DELETE d FROM S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE d { SNO , CITY } FROM SC ;
```

—you might think the two are rather different. But they're not. To see why not, suppose just for the moment that no city has more than one supplier or more than one part (and so the many to many join reduces to a one to one join). Then, using *s*, *c*, and *p* to denote a supplier number, a city, and a part number, respectively, no (*s*,*c*) subtuple appears in SPC more than once and no (*p*,*c*) subtuple appears in SCP more than once either, and so (S MATCHING *d*) NOT MATCHING SCP and (P MATCHING *d*) NOT MATCHING SCP reduce to just S MATCHING *d* and P MATCHING *d*, respectively. Furthermore, S MATCHING *d* and P MATCHING *d* in turn are equivalent to just *d*{SNO,CITY} and *d*{PNO,CITY}, respectively. Thus, the delete rule in the one to one case can be regarded as a reduced or degenerate form of the delete rule in the many to many case.

Finally, I'll leave it as an exercise to show that, given the foregoing insert and delete rules, explicit UPDATEs all work as intuitively expected.

View Updating

Now suppose SCP is in fact a view, the join of S and P. Then:

1. Most, though not quite all, of the constraints spelled out near the beginning of this section will be enforced automatically. In particular, of course, the constraint to the effect that the projections S{CITY} and P{CITY} are supposed to be equal won't automatically be enforced.

2. The compensatory actions from S and P to SCP will happen automatically, precisely because SCP is a view; that is, updates on S and/or P will automatically be reflected appropriately in SCP.

3. The compensatory actions from SCP to S and SP—these are the view updating rules as such—will also happen automatically, again precisely because SCP is a view. That is, updates on SCP are "really" updates on the underlying relvars S and/or P, and so are automatically visible in SCP, as well as in S or P or both.

What about a user who sees only the join view SCP? Well, 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 isn't entirely straightforward, as I'll explain in a moment). Such a user will be aware that {SNO,PNO} is a key and aware also that the following FDs and MVDs hold:

```
{ SNO } → { CITY }
{ PNO } → { CITY }

{ CITY } →→ { SNO } | { PNO }
```

With regard to these MVDs, let me remind you that together they're equivalent to the following constraint:

```
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

The user will also be aware of the pertinent predicate:

SCP: *Supplier SNO and part PNO both have city CITY.*

Now, I said earlier that SCP isn't in 4NF, and I suggested that it might therefore be awkward to update (and let me stress that this remark applies regardless of whether SCP is a base relvar or a view). Now I can be more specific. First, updates in general must (of course) abide by those MVDs. Second, INSERTs in particular are subject to the following rule:[4]

```
ON INSERT i INTO SCP :
   INSERT ( SCP JOIN i { SNO , CITY } ) INTO SCP ,
   INSERT ( SCP JOIN i { PNO , CITY } ) INTO SCP ;
```

Note: 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. Observe the implication that such a rule ought to, and 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. Also, it's worth noting that the rule in question is an example—the first we've seen in this book so far—of a rule that has the effect of causing certain updates to cascade to the target relvar itself.

[4] Observe that I don't give an analogous delete rule. In fact, 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. 7.1, a request to delete just the tuple (S1,London,P1) will fail, while a request to delete all tuples for city Paris will succeed (as we saw earlier in both cases).

PROJECTION VIEWS REVISITED

I've discussed the case in which S and P are base relvars and SCP is a view. To complete my examination of Example 1, I ought really to discuss the inverse situation also, in which SCP is a base relvar and S and P are views of that relvar. Of course, S and P would then be projection views specifically, and such a discussion thus logically belongs in Chapter 5. Rightly or wrongly, however, I felt Chapter 5 was complicated enough already, and so I decided to defer that discussion to the present chapter.

Actually I don't think there's all that much to be said. Recall from Chapter 5 that if *DB1* contains just a single relvar *R* and *DB2* contains just projections of *R*, then *DB1* and *DB2* are information equivalent only if those projections are obtained by decomposing *R* in accordance with some join dependency (JD) that holds in *R*. Well, the following JD does hold in SCP:

```
☼ { { SNO , CITY } , { PNO , CITY } }
```

As a direct consequence, SCP can be nonloss decomposed into its projections on {SNO,CITY} and {PNO,CITY}—in other words, into relvars S and P—and those relvars are necessarily updatable (just so long as SCP itself is updatable in the first place, which of course it is). The various compensatory actions are as previously discussed:

```
ON INSERT i INTO SCP :
   INSERT i { SNO , CITY } INTO S ,
   INSERT i { PNO , CITY } INTO P ;

ON DELETE d FROM SCP :
   DELETE ( ( S MATCHING d ) NOT MATCHING SCP ) FROM S ,
   DELETE ( ( P MATCHING d ) NOT MATCHING SCP ) FROM P ;

ON INSERT is INTO S , INSERT ip INTO P :
   INSERT ( P JOIN is ) INTO SCP ,
   INSERT ( S JOIN ip ) INTO SCP ;

ON DELETE ds FROM S , DELETE dp FROM P :
   DELETE ( SCP JOIN ds ) FROM SCP ,
   DELETE ( SCP JOIN dp ) FROM SCP ;
```

Let me now add that (of course) the relationship between views S and P is many to many. And although I didn't discuss this specific example in Chapter 5, I did in fact discuss another example in that chapter involving projection views in a many to many relationship—namely, Example 3, which involved views ST (the projection of S on SNO and STATUS) and TC (the projection of S on STATUS and CITY). To spell the point out, the relationship in that example was indeed many to many, because the same STATUS value could be associated with many suppliers and many cities (STATUS value 30 is a case in point, given our usual sample values). However, that example involved a lossy decomposition, not a nonloss one. As a consequence, updates on the projection relvars didn't always work terribly well.

EXAMPLE 2: INFORMATION HIDING

Let's get back to the case in which S and P are both base relvars. Suppose now that (as in our original suppliers-and-parts database, in fact) a city can be represented in base relvar S and not base relvar P or the other way around; in other words, some cities have suppliers but no parts or vice versa). Then the join SCP of S and P loses information, inasmuch as it's no longer guaranteed that S and P are equal to the projections of that join on the corresponding attributes. In fact, of the constraints mentioned in the section before last, the only ones that still apply are the following (and this time I'll give them in prose form instead of formal syntax):

- The usual key constraints all hold—{SNO} is a key for S, {PNO} is a key for P, and {SNO,PNO} is a key for SCP.
- At any given time, the current value of SCP is equal to the join of the current values of S and P.
- The FDs {SNO} → {CITY} and {PNO} → {CITY} and the MVDs {CITY} →→ {SNO} and {CITY} →→ {PNO} hold in SCP.

(Certain inclusion dependencies hold too, but as with the second example in Chapter 6, I won't bother to spell out the details here.)

A concrete example of a database value satisfying the foregoing conditions can be obtained by extending Fig. 7.1 to reinstate the usual tuples for supplier S5 and part P3 (see Fig. 7.2, and observe in particular that the relation shown as the current value of relvar SCP in that figure is the same as it was in Fig. 7.1).

S

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

P

PNO	CITY
P1	London
P2	Paris
P3	Oslo
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. 7.2: A revised version of Fig. 7.1

A remark on terminology: The title of this chapter implies, or at least strongly suggests, that S JOIN P is still a many to many join, but it must be understood that "many" here includes the zero case. The truth is, the term "many to many" is often used somewhat loosely to include both (a) what might be called the strict case, as in Example 1 as discussed earlier, and (b) cases like the one currently under discussion, in which it's possible for one participant to have an element with no counterpart in the other. Here's a lightly edited quote from *The Relational Database Dictionary, Extended Edition* (Apress, 2008):

> A many to many correspondence is, strictly, a rule pairing two sets *s1* and *s2* (not necessarily distinct) such that each element of *s1* corresponds to at least one element of *s2* and each element of *s2* corresponds to at least one element of *s1*. However, the term is often used somewhat loosely to mean a pairing such that (a) each element of *s1* corresponds to any number of elements of *s2* (possibly none at all) and each element of *s2* corresponds to at least one element of *s1*, or (b) each element of *s1* corresponds to at least one element of *s2* and each element of *s2* corresponds to any number of elements of *s1* (possibly none at all), or (c) each element of *s1* corresponds to any number of elements of *s2* (possibly none at all) and each element of *s2* corresponds to any number of elements of *s1* (possibly none at all). The term is probably best avoided unless the intended meaning is clear.

The final sentence here notwithstanding, I will in fact be using the term occasionally in the little that's still to come of this chapter, but I think my intended meaning will always be clear from the context. *End of remark.*

Following on from the foregoing, let me now point out that, precisely because (as I've said) it loses information, the design consisting of relvar SCP alone is no longer information equivalent to the one consisting of relvars S and P taken together. (An example of a query on the latter that has no exact counterpart on the former is "Get supplier numbers for suppliers in a city in which there's no part."). More precisely, any information that can be represented by relvar SCP alone can certainly be represented by relvars S and P together, but the converse is false. As a consequence, it's obvious that there are going to be certain updates that can be done on S and/or P that have no exact counterpart on SCP. An example is "Insert a tuple into S for a supplier with city Madrid," without simultaneously inserting a tuple for that same city into P.

I don't propose to investigate this example in any further detail here. Instead, I'm simply going to appeal to the analysis of Example 2 in Chapter 6; to be more specific, I'm going to claim that the detailed analysis of that example in that previous chapter applies to the present example also, mutatis mutandis. In other words, it's my opinion that the rules developed earlier in the present chapter for updating a strict many to many join view can be taken, if desired, to apply to the present case as well. To paraphrase a remark from the discussions in Chapter 6, if we can agree on this position, then at least we'll have agreed on a universal set of rules for updating many to many join views that do always work and do guarantee that joins are strictly many to many when they're supposed to be—not to mention the point that the rules in question in fact work for all join views, regardless of whether the join in question is one to one, one to

many, or many to many. What's more, if those rules do sometimes give rise to consequences that are considered unpalatable for some reason, then there are always certain pragmatic fixes, such as using the DBMS's authorization subsystem to prohibit certain updates, that can be adopted to avoid those consequences.

CONCLUDING REMARKS

There's one more point I want to make in connection with many to many joins. It has to do with the fact that—as is of course well known—cartesian product is a special case of many to many join. It follows that the rules for updating through a many to many join apply to cartesian product in particular (more precisely, they reduce to the rules for updating through a cartesian product, in the case where the join itself reduces to a cartesian product). To spell those rules out, let *V* be defined as *A* TIMES *B* and let *HA* and *HB* be the attributes of *A* and *B*, respectively, and let *HA* and *HB* have no attribute names in common. Then we have:

```
ON INSERT i INTO V :
   INSERT i { HA } INTO A ,
   INSERT i { HB } INTO B ;

ON DELETE d FROM V :
   DELETE ( ( A MATCHING d ) NOT MATCHING V ) FROM A ,
   DELETE ( ( B MATCHING d ) NOT MATCHING V ) FROM B ;

ON INSERT ia INTO A , INSERT ib INTO B :
   INSERT ( B TIMES ia ) INTO V ,
   INSERT ( A TIMES ib ) INTO V ;

ON DELETE da FROM A , DELETE db FROM B :
   DELETE ( V MATCHING da ) FROM V ,
   DELETE ( V MATCHING db ) FROM V ;
```

Chapter 8

Join Views III:

One to Many Joins

What Codd hath joined
Update such without blunder

—Anon.: *Where Bugs Go*

Chapters 6 and 7 discussed one to one and many to many joins, respectively; this chapter is concerned with the sole remaining case, one to many joins. But of course you know by now where our investigations are going to take us—we're going to wind up with the same general rules as we did in the previous two chapters:

```
ON INSERT INTO V : INSERT A (sub)tuples if they don't already exist,
                   INSERT B (sub)tuples if they don't already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don't exist elsewhere,
                   DELETE B (sub)tuples if they don't exist elsewhere
```

EXAMPLE 1: INFORMATION EQUIVALENCE

Once again I'll start with our usual suppliers-and-parts database, but I want to focus in this chapter on relvars S and SP and ignore relvar P (I'll also continue to ignore attribute SNAME and, for simplicity, attribute STATUS as well). So we have two base relvars looking like this:

```
S  { SNO , CITY } KEY { SNO }
SP { SNO , PNO , QTY } KEY { SNO , PNO }
```

Moreover, suppose for the sake of this first example that every supplier has to supply at least one part. In other words, suppose there's a constraint in effect (actually an equality dependency once again) that looks like this:

```
CONSTRAINT ... S { SNO } = SP { SNO } ;
```

In order to conform to this requirement, let's also agree for the sake of the example to drop the tuple for supplier S5 from our usual suppliers relation.

Now let's define the join of relvars S and SP as a view SSP:

```
SSP { SNO , CITY , PNO , QTY } KEY { SNO , PNO }
```

Observe that the join here is indeed one to many, in the sense that every tuple in S joins to many tuples in SP (at least one) and every tuple in SP joins to exactly one tuple in S. The two designs—that consisting of just relvar SSP and that consisting of the combination of relvars S and SP—are clearly information equivalent. Sample values are shown in Fig. 8.1.

S

SNO	CITY
S1	London
S2	Paris
S3	Paris
S4	London

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

SSP

SNO	CITY	PNO	QTY
S1	London	P1	300
S1	London	P2	200
S1	London	P3	400
S1	London	P4	200
S1	London	P5	100
S1	London	P6	100
S2	Paris	P1	300
S2	Paris	P2	400
S3	Paris	P2	200
S4	London	P2	200
S4	London	P4	300
S4	London	P5	400

Fig. 8.1: Relvars S, SP, and SSP—sample values

Here now are the predicates:

S: *Supplier SNO has city CITY.*

SP: *Supplier SNO supplies part PNO in quantity QTY.*

SSP: *Supplier SNO has city CITY and supplies part PNO in quantity QTY.*

As usual, I'll begin by considering what happens if all three relvars are base ones. If they are, what constraints hold? Well, first of all, relvars S, SP, and SSP have {SNO}, {SNO,PNO}, and {SNO,PNO} again, respectively, as their sole key (as already indicated). Second, we also clearly have:

```
CONSTRAINT ... SSP = S JOIN SP ;
CONSTRAINT ... S   = SSP { SNO , CITY } ;
CONSTRAINT ... SP  = SSP { SNO , PNO , QTY } ;
```

These constraints are all equality dependencies (EQDs), and together they guarantee the information equivalence referred to above. In fact, we can regard the combination of S and SP as a nonloss decomposition of SSP, because the following functional dependency (FD)—

```
{ SNO } → { CITY }
```

—holds in that latter relvar (i.e., SSP).[1] It follows that the example overall is essentially similar to Example 2 in Chapter 5. For that reason, I won't go through a detailed analysis in order to come up with the pertinent compensatory actions. Instead, I'll simply present them here as a *fait accompli*, as it were:

```
ON INSERT is INTO S , INSERT isp INTO SP :
   INSERT ( is JOIN SP ) INTO SSP ,
   INSERT ( S JOIN isp ) INTO SSP ;

ON DELETE ds FROM S , DELETE dsp FROM SP :
   DELETE ( SSP MATCHING ds ) FROM SSP ,
   DELETE ( SSP MATCHING dsp ) FROM SSP ;

ON INSERT i INTO SSP :
   INSERT i { SNO , PNO , QTY } INTO SP ,
   INSERT i { SNO , CITY } INTO S ;

ON DELETE d FROM SSP :
   DELETE ( SP MATCHING d ) FROM SP ,
   DELETE ( ( S MATCHING d ) NOT MATCHING SSP ) FROM S ;
```

Let's consider some examples, using the sample values from Fig. 8.1. *Note:* Given that our ultimate objective is to study the case in which relvars S and SP are base ones and relvar SSP is a view, I'll concentrate first on updates on relvar SSP specifically.

1. Suppose we delete the tuple (S1,London,P1,300) from relvar SSP. The effect will be to delete the tuple (S1,P1,300) from relvar SP and (because relvar SSP still has some tuples for supplier S1 remaining after the original delete) to leave relvar S unchanged.

2. Suppose we delete the tuple (S3,Paris,P2,400) from relvar SSP. The effect will be to delete the tuple (S3,P2,400) from relvar SP and (because relvar SSP has no tuples for supplier S3 remaining after the original delete) also to delete the tuple (S3,Paris) from relvar S. *Note:* I remind you that we're assuming for the sake of the example that every supplier does supply at least one part.

[1] I'm appealing here to Heath's Theorem, which was mentioned a couple of times in Chapter 5. For further explanation, see *Database Design and Relational Theory*.

3. Suppose we insert the tuple (S2,Paris,P3,500) into relvar SSP. The effect will be to insert the tuple (S2,P3,500) into relvar SP and to leave relvar S unchanged.

4. Suppose we insert the tuple (S5,Athens,P3,500) into relvar SSP. The effect will be to insert the tuple (S5,P3,500) into relvar SP and also to insert the tuple (S5,Athens) into relvar S.

Without going into details, I claim also that explicit UPDATEs on SSP (a) to change the supplier number, part number, or quantity for some shipment, or (b) to change the city for some supplier, or even (c) to change the supplier number for some supplier, all work as intuitively expected.

Turning now to updates on relvars S and SP:

1. Suppose we delete the tuple (S1,P1,300) from relvar SP. The effect will be to delete the tuple (S1,London,P1,300) from relvar SSP and (because relvar SSP still has some tuples for supplier S1 remaining after the original delete) to leave relvar S unchanged.

2. Suppose we delete all tuples for supplier S1 from relvar SP. The effect will be to delete all tuples for supplier S1 from relvar SSP and also to delete the tuple (S1,20) from relvar S.

3. Suppose we insert the tuple (S2,P3,500) into relvar SP. The effect will be to insert the tuple (S2,Paris,P3,500) into relvar SSP.

4. Suppose we attempt to insert the tuple (S5,P3,500) into relvar SP. Absent an appropriate simultaneous insert into relvar S, this insert will fail.

5. Suppose we attempt to insert the tuple (S5,Athens) into relvar S. Absent an appropriate simultaneous insert into relvar SP, this insert will fail.

6. Suppose we delete the tuple (S1,20) from relvar S. The effect will be to delete all tuples for supplier S1 from relvar SSP. Of course, the rule governing deletes on SSP will now come into play, which will (not unreasonably) have the effect of deleting all tuples for supplier S1 from relvar SP as well.

As for explicit UPDATEs, I'll leave it as an exercise to show that (a) as certain discussions in Chapter 4 should have led us to expect, if we want to change the supplier number of some supplier, then there's no real alternative to writing out the necessary double UPDATE explicitly, as in this example—

```
UPDATE S  WHERE SNO = 'S1' : { SNO := 'S9' } ,
UPDATE SP WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

—but (b) other explicit UPDATEs all work satisfactorily.

Finally, let me state for the record that the foregoing rules, as they apply to updates on relvar SSP in particular, are essentially identical to the rules I gave for updating joins in Chapters 6 and 7. For that reason, I won't bother to go into details on what happens if (a) the user sees just that relvar SSP or (b) that relvar SSP is a view and S and SP are base relvars. Suffice it to say that, once again, everything works satisfactorily.

EXAMPLE 2: INFORMATION HIDING

Now let me revise the foregoing example and suppose that—as in our original suppliers-and-parts database, in fact—it's not the case that every supplier has to supply at least one part. Then information equivalence is lost; to be specific, the design consisting of relvars S and SP in combination is capable of representing suppliers (such as supplier S5 in our usual set of sample values for the suppliers-and-parts database) who currently supply no parts at all, while the design consisting of just the join SSP obviously can't represent such information. (Equivalently, it's no longer guaranteed that relvar S is equal to the projection of SSP, the join of S and SP, on SNO and CITY. On the other hand, it's at least still true that relvar SP is equal to the projection of that join on SNO, PNO, and QTY.)

Be that as it may, the following constraints among others apply to this revised form of the database:

- The usual key constraints all hold—{SNO} is a key for S and {SNO,PNO} is a key for each of SP and SSP.

- At any given time, the current value of SSP is equal to the join of the current values of S and SP.

- The FD {SNO} → {CITY} holds in SSP (also in S, of course).

A concrete example of a database value satisfying the foregoing conditions can be obtained by extending Fig. 8.1 to reinstate the usual tuple for supplier S5 (see Fig. 8.2 overleaf, and observe in particular that the relations shown as the current values of relvar SP and SSP in that figure are the same as they were in Fig. 8.1).

S

SNO	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	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

SSP

SNO	CITY	PNO	QTY
S1	London	P1	300
S1	London	P2	200
S1	London	P3	400
S1	London	P4	200
S1	London	P5	100
S1	London	P6	100
S2	Paris	P1	300
S2	Paris	P2	400
S3	Paris	P2	200
S4	London	P2	200
S4	London	P4	300
S4	London	P5	400

Fig. 8.2: A revised version of Fig. 8.1

A remark on terminology: The title of this chapter implies, or at least strongly suggests, that S JOIN SP is still a one to many join, but it needs to be clearly understood that "many" here includes the zero case. The truth is, the term "one to many" is often used somewhat loosely to include both (a) what might be called the strict case, as in Example 1 in the previous section, and (b) cases like the one currently under discussion, in which it's possible for one participant to have an element with no counterpart in the other. Here's a lightly edited quote from *The Relational Database Dictionary, Extended Edition* (Apress, 2008):

> A one to many correspondence is, strictly, a rule pairing two sets *s1* and *s2* (not necessarily distinct) such that each element of *s1* corresponds to at least one element of *s2* and each element of *s2* corresponds to exactly one element of *s1*. However, the term is often used somewhat loosely to mean a pairing such that (a) each element of *s1* corresponds to any number of elements of *s2* (possibly none at all) and each element of *s2* corresponds to exactly one element of *s1*, or (b) each element of *s1* corresponds to at least one element of *s2* and each element of *s2* corresponds to at most one element of *s1*, or (c) each element of *s1* corresponds to any number of elements of *s2* (possibly none at all) and each element of *s2* corresponds to at most one element of *s1*. The term is probably best avoided unless the intended meaning is clear.

The final sentence here notwithstanding, I will in fact be using the term occasionally in the rest of this chapter, but I think my intended meaning will always be clear from the context. *End of remark.*

To say it again, the join SSP of S and SP loses information, inasmuch as it's no longer guaranteed that relvar S in particular is equal to the projection of that join on the corresponding attributes. More precisely, any information that can be represented by relvar SSP alone can certainly be represented by the combination of relvars S and P, but the converse is false. (An example of a query on the latter that has no exact counterpart on the former is "Get supplier

numbers for suppliers who currently supply no parts.") As a consequence, it's obvious that there are going to be certain updates that can be done on S in particular that have no counterpart on SSP. An example is "Insert a tuple into S for supplier S9," without simultaneously inserting at least one tuple for that same supplier S9 into SP.

I don't propose to investigate this example in any further detail here. Instead, I'm simply going to appeal, as I did in Chapter 7, to the analysis of Example 2 in Chapter 6; to be more specific, I'm going to claim that the detailed analysis of that example in Chapter 6 applies to the present example also, mutatis mutandis. In other words, it's my opinion that the rules discussed earlier in the present chapter for updating a strict one to many join view can be taken, if desired, to apply to the present case as well. To paraphrase a remark from the discussions in Chapter 6, if we can agree on this position, then we'll have agreed on a universal set of rules for updating one to many join views that do at least always work and do guarantee that joins are strictly one to many when they're supposed to be—not to mention the point that the rules in question in fact work for all join views, regardless of whether the join in question is one to one, one to many, or many to many. What's more, if those rules do sometimes give rise to consequences that are considered unpalatable for some reason, then there are always certain pragmatic fixes, such as using the DBMS's authorization subsystem to prohibit certain updates, that can be adopted to avoid those consequences.

CONCLUDING REMARKS

This brings us to the end of these three chapters on updating joins. In closing, however, there are a few additional remarks I'd like to make (they're all somewhat tangential to our main topic, however, and you can ignore them if you want). The first concerns foreign keys. As you might have noticed, I made no explicit mention of foreign keys as such in any of these three chapters (nor in Chapter 5 either, on projections, come to that). In fact this omission was deliberate on my part. The point is, there seems to be a vague notion in the database community at large that updating joins has something to do with whether the join in a question is defined on the basis of some foreign key and its corresponding target key, and I wanted to get away from that notion if I could. That said, it's certainly true that several of my examples were indeed "foreign key joins" (if I might be permitted to use such a term)—but not all of them were; in particular, the ones in Chapter 7 weren't.

That said, I'd still like to say something about foreign keys in **Tutorial D** specifically. The following example, repeated from Chapter 2, illustrates the kind of syntax normally used to declare foreign keys:

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

However, I also said in Chapter 2 that FOREIGN KEY specifications like the ones shown here are essentially just shorthand for constraints that can also be expressed, possibly more longwindedly, using explicit CONSTRAINT syntax. More specifically, I showed in Chapter 3 how the fact that, e.g., {SNO} in relvar SP is a foreign key referencing the key {SNO} in relvar S could be expressed in relational calculus as a multivariable constraint, like this:

```
FORALL x ∈ SP ( UNIQUE y ∈ S ( x.SNO = y.SNO ) )
```

(I remind you that the UNIQUE quantifier can be read as "there exists exactly one … such that.") So you might be wondering what the **Tutorial D** "longhand" equivalent of this relational calculus expression would look like.

Well, we could certainly write the following in **Tutorial D**:

```
CONSTRAINT ... IS_EMPTY ( SP NOT MATCHING S ) ;
```

But this statement doesn't quite do the job (at least, not by itself). What it says is that every shipment has *at least* one corresponding supplier. What we want to say, however, is that each shipment has *exactly* one corresponding supplier.[2] Well, we can do that too by making use of another convenient shorthand called *image relations* (see *SQL and Relational Theory* for further discussion of this construct). To be specific, we can write:

```
CONSTRAINT ... IS_EMPTY ( SP WHERE COUNT ( ‼S ) ≠ 1 ) ;
```

Explanation: For a given SP tuple, the expression ‼S—pronounced "bang bang S" or "double bang S"—denotes the set of S tuples with the same supplier number as that SP tuple. Thus, the constraint overall requires every SP tuple to be such that the count of corresponding S tuples is exactly one.

Still on the subject of **Tutorial D** syntax, you might also be wondering how functional dependencies (FDs) can be formulated in **Tutorial D**. (You might have noticed that I didn't give any such formulations in connection with the examples in which I was making use of such FDs.) In fact there are several ways to do it, of which the most elegant is probably the one illustrated here:

```
CONSTRAINT ... SSP { SNO , CITY } KEY { SNO } ;
```

You can read this CONSTRAINT statement as saying "If we were to define a relvar consisting of the projection of SSP on SNO and CITY, then that relvar would have {SNO} as a key"—which effectively does what we want, since it states implicitly that the FD

---

[2] Actually the statement does do the job in a sense, thanks to the fact that there's an additional constraint in effect that requires supplier numbers in S to be unique. But having to deduce that some constraint—a rather important constraint at that—holds implicitly from the fact that two other constraints have been declared explicitly does seem, from the user's point of view at least. a trifle unfriendly.

```
{ SNO } → { CITY }
```

holds in relvar SSP.[3] (Recall from a footnote in Chapter 5 that if *K* is a key for relvar *R*, then the FD $K \rightarrow X$ holds in *R* for all subsets *X* of the heading of *R*.)

> *Aside:* Please take note of the idea, illustrated by the foregoing example, that we should be able to declare keys in connection with relational expressions of arbitrary complexity. I'll be appealing to it again in later chapters. *End of aside*.

Finally, you might be wondering whether we need any special syntax for declaring join dependencies (JDs) or multivalued dependencies (MVDs). In fact I don't think we do. For example, with reference to the first example from Chapter 7, specifying the following constraint—

```
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

—is precisely equivalent to stating that the following JD holds in relvar SCP:

```
☼ { { SNO , CITY } , { PNO , CITY } }
```

It's also precisely equivalent to stating that a certain pair of MVDs hold in relvar SCP. The situation is different with FDs; to say some relvar is equal to the join of certain of its projections is *not* equivalent to saying some FD holds, and that's why FDs in general need to be separately specified.

---

[3] Well, actually it states implicitly that the FD {SNO} → {CITY} holds in the *projection* of that relvar SSP on SNO and CITY. But it's easy to prove that a given FD holds in a given projection of a given relvar *R* if and only if it holds in that given relvar *R* itself.

Chapter 9

Intersection Views

Updating intersections
Can sometimes need corrections

—Anon.: *Where Bugs Go*

In this chapter and the next two, I turn my attention to intersection, union, and difference views. Of course, I've already said something in Chapter 6 about intersection views in particular (the subject of the present chapter)—recall that intersection is a special case of one to one join—but there's quite a lot more to be said on the subject.

First I need to say something about the structure of the chapter. Let *DB1* and *DB2* be a set of base relvars and a set of views, respectively. Now, in the last few chapters, on restriction, projection, and join views, I proceeded in general terms as follows: First, I considered a particular *DB1*; then I considered a particular *DB2* that was information equivalent to that *DB1*; finally, I considered another *DB2* that wasn't information equivalent to that *DB1* but involved some information hiding instead. But that approach doesn't work very well for intersection views.[1] The reason is this: Suppose *DB1* consists of base relvars *A* and *B* (only) and *DB2* consists of view *V* only, defined as the intersection of *A* and *B* (i.e., *V* = *A* INTERSECT *B*). Then *DB2* will be information equivalent to *DB1* only if *A* = *B*—in which case (as I hinted in a footnote near the end of Chapter 6) there's not much point in defining view *V* in the first place.

So there's not much point in considering the case in which *A* and *B* are equal. What's more, there's also not much point in considering the case in which they're totally disjoint, in the sense that no tuple ever appears in both. For if they're disjoint in this sense, then the intersection view *V* will always be empty; deletes on *V* will thus always be "no ops" and inserts on *V* will always fail (unless they're "no ops" as well). So the interesting case—the one I do want to examine—is the one in which *A* and *B* aren't equal but do at least overlap, in the sense that it's possible for the very same tuple to appear in both. Thus, I'll consider two principal examples, one in which the overlap is explicit (and I'll explain what I mean by that term), and one in which it's merely implicit. Also, I won't bother to examine in either case the situation in which all relvars concerned are base ones; instead, I'll start with two given relvars *A* and *B* and immediately define their intersection *A* INTERSECT *B* as a view per se, and see what happens.

---

[1] It also doesn't work very well (for different reasons) for union or difference views, the topics of the next two chapters.

EXAMPLE 1: EXPLICIT OVERLAP

My first example is based on one previously examined in the section "Overlapping Restrictions" in Chapter 4, on restriction views. It involves two relvars, NLS ("non London suppliers") and NPS ("non Paris suppliers"), that look like this:

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

Of course, given our usual suppliers-and-parts database, these relvars are probably views—views of relvar S, to be specific—but you can think of them as base relvars if you like. More to the point, observe that these relvars certainly do overlap; to be specific, tuples for suppliers who are in neither London nor Paris will appear in both. (Indeed, such tuples *must* appear in both, thanks to the corresponding predicates—see below—and *The Closed World Assumption.*)[2] And now perhaps you can see why I say the overlap is explicit in this example: The condition a tuple has to satisfy in order to appear in both relvars (a) can be checked by examining values explicitly appearing in the tuple in question, and (b) is in fact explicitly reflected in the corresponding constraints (which I'll be examining in a few moments). *Note:* Formally, in fact, the condition in question is a simple restriction condition (see Appendix B for a definition of this concept).

Here now are the relvar predicates:

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

NPS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't Paris).*

Now let's define the intersection of relvars NLS and NPS as a view XLP:

```
XLP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

The predicate for this relvar is:

XLP: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is neither London nor Paris).*

Sample values are shown in Fig. 9.1.

---

[2] To spell the point out, we have here (as we did in Chapter 4) a situation in which the predicates violate the discipline, suggested in a footnote in Chapter 2, to the effect that the predicates for relvars *R1* and *R2* should preferably be such as to preclude the possibility that the same tuple might satisfy both.

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

XLP

SNO	SNAME	STATUS	CITY
S5	Adams	30	Athens

NPS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 9.1: Relvars NLS, NPS, and XLP—sample values

What constraints hold in this example? Well, first, each of the three relvars has {SNO} as its sole key, as already indicated. Second, {SNO} in XLP is a foreign key, referencing both NLS and NPS. Third, we also clearly have:

```
CONSTRAINT ... XLP = NLS INTERSECT NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                          ( NPS WHERE CITY ≠ 'London' ) ;
```

Note the last of these in particular, which implies among other things that certain updates on each of NLS and NPS must cascade appropriately to the other:

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

Points arising:

- First of all, note that the rules just shown make no mention of relvar XLP—in fact, they would be needed even if relvar XLP didn't exist.

- In the delete rules, the restriction conditions CITY ≠ 'Paris' and CITY ≠ 'London' could be dropped (or, more precisely, replaced by just TRUE) without significant loss. I've included them for reasons of explicitness, also for symmetry with the insert rules.

- As was previously mentioned in Chapter 4, the last of the foregoing constraints—the one that implies the need for the cascade rules—also implies that relvars NLS and NPS together

violate *The Principle of Orthogonal Design*. However, the fact that they do so is unimportant for present purposes.

- Finally, we need a constraint to the effect that if some supplier is represented in both NLS and NPS, then the two tuples representing that supplier are in fact one and the same. The following constraint, which states that {SNO} is a key for the union of those two relvars, does the trick:

```
CONSTRAINT ... UNION { NLS , NPS } KEY { SNO } ;
```

Of course, this constraint will be enforced automatically if relvars NLS and NPS are in fact views of the suppliers relvar S.

Back to the rules as such. What about cascades from NLS and NPS to the intersection relvar XLP? In fact these rules are straightforward:

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO XLP ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO XLP ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM XLP ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM XLP ;
```

(Again the restriction conditions could be dropped from the delete rules without loss.)

Finally, here are the rules for updates on the intersection relvar as such:

```
ON INSERT i INTO XLP :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;

ON DELETE d FROM XLP :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
```

Points arising:

- Let's focus on the delete rule for a moment. As far as relvar XLP is concerned, it would of course be sufficient to cascade to just one of the relvars NLS and NPS. But cascading to either will cause an additional cascade to the other one anyway; and since there's no good reason to favor one over the other, at least in concrete syntax (assuming, of course, that both relvars are visible)—it's a difference that makes no difference, so to speak—I've specified cascades to both, for reasons of symmetry and explicitness.

- As a matter of fact, a similar remark applies to the insert rule also (in general, inserting into an intersection needs to cascade to both of the relvars involved; in the case at hand, however, cascading to either will cause an additional cascade to the other one anyway).

- Once again, the restriction conditions could be dropped without loss, this time from both the insert rule and the delete rule (why, exactly, in each case?).

Of course, as explained in the introduction to this chapter, we're talking about an intersection example, and thus we're dealing with a situation in which information equivalence is lost more or less by definition. To spell the point out, any information that can be represented by relvar XLP alone can certainly be represented by relvars NLS and NPS taken in combination, but the converse is false. (Here's an example of a query on the latter that has no exact counterpart on the former: "Get suppliers in London.") As a consequence, it should be obvious that there'll be certain updates that can be done on NLS and/or NPS that have no exact counterpart on XLP. An example of such an update is "Insert a tuple for supplier S9 into NPS without simultaneously inserting that same tuple into NLS" (i.e., update the database to say there's a "new" supplier S9 who is located in London). However, despite all of the foregoing—despite the lack of information equivalence in particular—it is at least true that updates on the intersection view XLP always work satisfactorily. And the reason for this satisfactory state of affairs is precisely that the overlap in this example is explicit.

Now let's briefly consider a user who sees just relvar XLP (an information hiding situation). Such a user:

a. Will know the relvar predicate;

b. Will know that supplier numbers are unique and CITY values are never London or Paris;

c. Won't be aware of any compensatory actions.

Such a user can't be allowed to insert tuples into the relvar, nor to update supplier numbers within the relvar, because such operations have the potential to violate constraints of which the user is unaware.

Finally, let me point out that if NLS and NPS are indeed themselves views (i.e., views of relvar S), then XLP could alternatively have been defined directly as a restriction of S, thus:

```
S WHERE CITY ≠ 'London' AND CITY ≠ 'Paris'
```

Let's call this restriction XLP′. Clearly, then, updates on XLP′ should have the same effect as updates on XLP—and so they do. To be specific:

- INSERT: Tuples to be inserted into either XLP or XLP′ must have CITY value neither London nor Paris and so must be inserted into the other as well.

- DELETE: Tuples to be deleted from either XLP or XLP′ certainly do have CITY value neither London nor Paris and so must be deleted from the other as well.

Note: I'll have quite a bit more to say about the general question of updates on relvars—more specifically, views—with different but equivalent definitions in Chapter 14.

EXAMPLE 2: IMPLICIT OVERLAP

For our second example, let's switch from suppliers to parts, just for a change. Suppose that, at any given time, some parts are on sale, some parts are in stock, and some are both. Suppose further that we represent this situation by means of two relvars, each with just a single attribute PNO—one relvar (PL) giving part numbers for parts on sale, the other (PK) giving part numbers for parts in stock. Of course, the intersection of these two relvars—call it XLK—gives part numbers for parts that are both on sale and in stock. The point is, however, we can't tell just by looking at a given tuple in the original parts relvar P whether that part ought to be represented in PL, PK, both, or neither; in other words, the overlap here is implicit, not explicit.[3] Thus, the predicates are very simple:

PL: *Part PNO is on sale.*

PK: *Part PNO is in stock.*

XLK: *Part PNO is on sale and in stock.*

Sample values are shown in Fig. 9.2. Note that (with reference to our usual set of PNO values) I'm assuming for the sake of the example that part P4 is neither on sale nor in stock.

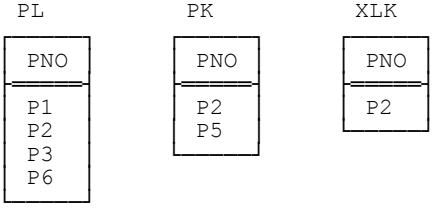


Fig. 9.2: Relvars PL, PK, and XLK—sample values

[3] Observe that relvars PL and PK resemble relvars NLS and NPS from the previous section in that they violate the suggested discipline that (in general) the pertinent predicates, q.v., should be such as to preclude the possibility that the same tuple might satisfy both. On the other hand, they differ from NLS and NPS in that they do *not* violate *The Principle of Orthogonal Design.* That's because there's no constraint in effect—nor can there be—to say that if a certain tuple appears in one of those relvars, it must appear in the other.

As for constraints, {PNO} is obviously the sole key for each of these three relvars, and of course XLK is equal to the intersection of the other two:

```
CONSTRAINT ... XLK = PL INTERSECT PK ;
```

Updates on PL and/or PK are noncontroversial:

```
ON INSERT i INTO PL : INSERT ( i INTERSECT PK ) INTO XLK ;
ON INSERT i INTO PK : INSERT ( i INTERSECT PL ) INTO XLK ;

ON DELETE d FROM PL : DELETE d FROM XLK ;
ON DELETE d FROM PK : DELETE d FROM XLK ;
```

Now what about updates on XLK? Well, the insert rule is obvious:

```
ON INSERT i INTO XLK :
   INSERT i INTO PL , INSERT i INTO PK ;
```

As for the delete rule, clearly it would be sufficient for deletes on XLK to cascade to just one of PL and PK (it must cascade to at least one, of course). However, there's no good reason for choosing either of these possibilities over the other; in fact, what we have here once again is the ambiguity (?) issue that I discussed in detail in Chapter 6—at least, it's a special case of that issue—and the discussions in that chapter apply here also, mutatis mutandis.[4] I therefore propose the following as an appropriate delete rule (and observe that, not incidentally, it has the same general form as its counterpart in Example 1 in the previous section):

```
ON DELETE d FROM XLK :
   DELETE d FROM PL , DELETE d FROM PK ;
```

Of course, once again we're dealing with a situation in which information equivalence is lost. To spell the point out, any information that can be represented by relvar XLK alone can certainly be represented by relvars PL and PK taken in combination, but the converse is false. (Here's an example of a query on the latter that has no exact counterpart on the former: "Get part numbers for parts that are on sale and not in stock.") As a consequence, it should be obvious that there'll be certain updates that can be done on PL and/or PK that have no exact counterpart on XLK. An example of such an update is "Insert a new tuple into PL without simultaneously inserting that same tuple into PK" (i.e., update the database to say some part is on sale but, if it wasn't previously in stock, still isn't).

I'll leave it as an exercise to determine the implications of all of the foregoing for a user who sees just relvar XLK. However, let me now point out that, sadly, there's another issue here (I touched on this issue in Chapter 6, but I didn't elaborate on it in that chapter). Suppose we

---

[4] I remind you that David McGoveran has a proposal for resolving the ambiguity, which I'll be discussing in Chapter 15. Do note carefully, however, that the ambiguity problem didn't arise with Example 1, where the overlap was explicit.

start off with just relvars PL and PK, without the intersection view, and suppose too that part P2 is represented in both relvars. Then the following update is clearly legitimate:

```
DELETE ( P2 ) FROM PL ;
```

Observe in particular that there's no question of this delete cascading to relvar PK. But now suppose we introduce the intersection view XLK. Given the rules defined above, then, this delete *will* now cascade to relvar PK!—and it'll do so, moreover, even if view XLK isn't visible to the user issuing the delete on relvar PL. Or to put the point another way: Introducing that view apparently requires the simultaneous introduction of a cascade delete rule from PL to PK and vice versa.[5]

Such a state of affairs doesn't seem very desirable, to say the least. Can we do anything about it? Well, one possible (though annoying) pragmatic fix, in the particular case at hand, is to execute an appropriate insert immediately after the delete, thus:

```
DELETE ( P2 ) FROM PL ;
INSERT ( P2 ) INTO PK ;
```

To my mind, however, the solution to be discussed in the subsection immediately following is greatly to be preferred.

A Better Design

Suppose we replace relvars PL and PK in their entirety by a single relvar—let's call it POI—with attributes PNO, ON_SALE, and IN_STOCK, where attributes ON_SALE and IN_STOCK are of type BOOLEAN (and have the obvious interpretations) and {PNO} is the sole key. A possible value for such a relvar is shown in Fig. 9.3 (note the tuple for part P4 in particular).

POI

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P4	FALSE	FALSE
P5	FALSE	TRUE
P6	TRUE	FALSE

Fig. 9.3: Relvar POI—sample value

[5] A related issue is the following. Under the stated rules for updates on XLK, deleting a tuple from XLK and then inserting it again will preserve the status quo, but inserting a tuple into XLK and then deleting it again might not. The reason is that inserting a tuple into XLK might actually cause a tuple to be inserted into just one of PL and PK, whereas deleting a tuple from XLK will always cause a tuple to be deleted from both. Of course, the status quo with respect to relvar XLK as such is always preserved.

Now we define restriction views PL′ and PK′ of relvar POI as indicated by the following constraints (I use the primed names PL′ and PK′ in order to avoid confusion with relvars PL and PK as previously defined, but of course they—i.e., relvars PL′ and PK′—serve essentially the same purpose as those earlier relvars did):[6]

```
CONSTRAINT ... PL' = POI WHERE ON_SALE ;
CONSTRAINT ... PK' = POI WHERE IN_STOCK ;
```

And now we can define XLK′ as the intersection of PL′ and PK′, and we have:

```
CONSTRAINT ... XLK' = PL' INTERSECT PK' ;
```

(Of course, we could also define XLK′ as a direct restriction of POI, thus: POI WHERE ON_SALE AND IN_STOCK. But let's ignore this possibility until further notice.) See Fig. 9.4 for some sample values, corresponding of course to the sample value of relvar POI as shown in Fig. 9.3.

PL′

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P6	TRUE	FALSE

XLK′

PNO	ON_SALE	IN_STOCK
P2	TRUE	TRUE

PK′

PNO	ON_SALE	IN_STOCK
P2	TRUE	TRUE
P5	FALSE	TRUE

Fig. 9.4: Relvars PL′, PK′, and XLK′—sample values

The predicates (deliberately spelled out in somewhat excruciating detail) are as follows:

PL′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and ON_SALE is TRUE).*

PK′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and IN_STOCK is TRUE).*

---

[6] Here's a question for you: Is the design consisting of relvars PL′ and PK′ information equivalent to the one consisting of relvars PL and PK?

XLK′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and ON_SALE is TRUE and IN_STOCK is TRUE).*

As for constraints, note first that each of the three relvars has {PNO} as sole key, and {PNO} in XLK′ is a foreign key, referencing both PL′ and PK′. The following constraints also obviously hold:

```
CONSTRAINT ... IS_EMPTY ( PL′ WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK′ WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( XLK′ WHERE NOT ( ON_SALE AND IN_STOCK ) ) ;
```

We also need a constraint to the effect that if some part is represented in both PL′ and PK′, then the two tuples representing that part are in fact one and the same:

```
CONSTRAINT ... UNION { PL′ , PK′ } KEY { PNO } ;
```

Of course, this constraint will automatically be enforced if relvars PL′ and PK′ are indeed, as stated, views of POI.

Aside: By the way, we also have the following:

```
CONSTRAINT ... ( PL′ WHERE IN_STOCK ) = ( PK′ WHERE ON_SALE ) ;
```

As a consequence (of the fact that this constraint holds, that is), relvars PL′ and PK′, like relvars NLS and NPS in the previous section, clearly violate *The Principle of Orthogonal Design*. In other words, I seem to be contravening one of my own design recommendations in this example! To be specific, in *Database Design and Relational Theory*, I suggested rather strongly that orthogonality should never be violated. But of course the violation here won't hurt, because the redundancy it causes will certainly be controlled. *End of aside*.

Be that as it may, what happens to the update rules? Well, first we have to consider how updates on relvar POI itself affect relvars PL′ and PK′:

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL′ ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′ ;

ON DELETE d FROM POI :
   DELETE ( d WHERE ON SALE ) FROM PL′ ,
   DELETE ( d WHERE IN_STOCK ) FROM PK′ ;
```

Next we have to consider how updates on relvars PL′ and PK′ affect relvar POI:

```
ON INSERT i INTO PL′ : INSERT i INTO POI ;
```

```
ON INSERT i INTO PK′ : INSERT i INTO POI ;

ON DELETE d FROM PL′ : DELETE d FROM POI ;

ON DELETE d FROM PK′ : DELETE d FROM POI ;
```

Note that the foregoing rules taken together imply that (a) inserting a tuple of the form (*p*,TRUE,TRUE) into either of PL′ and PK′ will effectively cascade to the other; likewise, (b) deleting a tuple of the form (*p*,TRUE,TRUE) from either of PL′ and PK′ will effectively cascade to the other. In fact, of course, such cascades would be logically necessary even if relvar POI—and in particular the rules that cascade inserts to, and deletes from, that relvar to relvars PL′ and PK′—didn't exist.

Now we can bring relvar XLK′ into the picture (to be specific, now we can consider how updates on relvars PL′ and PK′ affect relvar XLK′ and vice versa):

```
ON INSERT i INTO PL′ : INSERT ( i WHERE IN_STOCK ) INTO XLK′ ;
ON INSERT i INTO PK′ : INSERT ( i WHERE ON_SALE ) INTO XLK′ ;

ON DELETE d FROM PL′ : DELETE ( d WHERE IN_STOCK ) FROM XLK′ ;
ON DELETE d FROM PK′ : DELETE ( d WHERE ON_SALE ) FROM XLK′ ;

ON INSERT i INTO XLK′ :
   INSERT i INTO PL′ , INSERT i INTO PK′ ;

ON DELETE d FROM XLK′ :
   DELETE d FROM PL′ , DELETE d FROM PK′ ;
```

And these rules are all, as I hope you'll agree, completely noncontroversial[7] (though I think it's instructive to compare them with the rules I gave in connection with the previous version of the example). In effect, what I've done is redesign the database in such a way that information that was previously represented only implicitly, by the relvar names PL and PK, is now represented explicitly by values of the attributes ON_SALE and IN_STOCK instead. And the effect of that redesign is to convert the previous version of the example—which suffered from update behavior that was at least arguably unacceptable—into a version that behaves just like Example 1 as discussed earlier in the chapter. And, of course, this latter example didn't suffer from the ambiguities and consequent update problems that arose with the previous version of the example currently under discussion.

[7] In the case of the delete rule, it would of course be sufficient to cascade to just one of the relvars PL′ and PK′, but deleting from either will cascade to the other one anyway.

Aside: It's tempting to suggest there might be a general design principle here: Whenever two relvars have the same heading but different semantics, (a) introduce an attribute or attributes whose values serve to represent that difference and then (b) consider replacing those relvars by a single relvar. Or: Whenever some relvar *B* is such that its value is always a subset of the value of some other relvar *A*, but *B* can't be defined as a restriction as such of *A*, then introduce an attribute or attributes into both *A* and *B* whose values can be used to turn *B* into a restriction as such after all. Further consideration of such matters is beyond the scope of this book, however. *End of aside*.

I'll leave it as an exercise to determine the implications of all of the foregoing for a user who sees just relvar XLK′. As for defining XLK′ directly as a restriction of POI, I'll leave it as another exercise to show that the behavior of such a restriction with respect to updates would be essentially identical to that of the intersection version as described above.

Another Possible Design

Now, it might have occurred to you that, given that attribute ON_SALE necessarily has the value TRUE in every tuple of PL′ and attribute IN_STOCK necessarily has the value TRUE in every tuple of PK′, we could redefine those relvars—more specifically, those views—in such a way as to drop those attributes, as indicated by the following constraints:

```
CONSTRAINT ... PL'' = ( POI WHERE ON_SALE ) { PNO , IN_STOCK } ;
CONSTRAINT ... PK'' = ( POI WHERE IN_STOCK ) { PNO , ON_SALE } ;
```

(I've used double primes to distinguish these versions of the relvars from previous versions.) Here are the predicates:

PL″: *Part PNO is on sale and, if and only if IN_STOCK is TRUE, also in stock.*

PK″: *Part PNO is in stock and, if and only if ON_SALE is TRUE, also on sale.*

And now we could (re)define relvar XLK′—or XLK″, rather—as the intersection of the projections of PL″ and PK″ on PNO:

```
CONSTRAINT ... XLK'' = PL'' { PNO } INTERSECT PK'' { PNO } ;
```

The predicate is as for our original relvar XLK:

XLK″: *Part PNO is on sale and in stock.*

For some sample values for this redesigned database, see Fig. 9.5. You might find it instructive to compare this figure with Fig. 9.2.

PL''

PNO	IN_STOCK
P1	FALSE
P2	TRUE
P3	FALSE
P6	FALSE

PK''

PNO	ON_SALE
P2	TRUE
P5	FALSE

XLK''

PNO
P2

Fig. 9.5: Relvars PL'', PK'', and XLK''—sample values

Now let's focus for a moment on relvar PL″. That relvar is, of course, a projection relvar; to be specific, it's a projection of (a certain restriction of) relvar POI. As explained in Chapter 5, therefore, there's a loss of information equivalence here, as a consequence of which that relvar PL″ supports deletes but not inserts.[8] The pertinent delete rule is:

```
ON DELETE d FROM PL'' : DELETE ( POI MATCHING d ) FROM POI ;
```

Analogously, we also have:

```
ON DELETE d FROM PK'' : DELETE ( POI MATCHING d ) FROM POI ;
```

And the rule for deletes on XLK″ is as follows (note, however, that there can't be any insert rule):

```
ON DELETE d FROM XLK''  :
   DELETE d FROM PL'' , DELETE d FROM PK'' ;
```

In other words, while deletes on XLK″ work satisfactorily (or so it might be argued, at any rate), inserts on XLK″ can't be done at all. Overall, therefore, I think this "double prime" design is a nonstarter, at least if it's supposed to be capable of supporting update operations properly.

CONCLUDING REMARKS

Now, you might have found this chapter rather confusing—especially with respect to Example 2 in the previous section, where I kept changing the rules of the game, so to speak. So let me abstract from the discussions in that section and summarize the update rules for the case where there's an intersection involved. Let *V* be defined as *A* INTERSECT *B*. Further, assume for

---

[8] Inserting the tuple (*p*,*k*) into PL″, if it were allowed, would have to cause insertion of the tuple (*p*,TRUE,*k*) into POI. The trouble is, we know this, but the DBMS doesn't. *Note:* Actually, the example is essentially identical, or at least isomorphic, to one discussed in earlier chapters—viz., the one in which we considered dropping the CITY attribute from relvar LS ("London suppliers"). Refer to Chapter 5 for further discussion.

simplicity that tuples to be inserted into *A* and *B* are required to satisfy boolean expressions *ax* and *bx*, respectively, where *ax* and *bx* denote restriction conditions on the pertinent relvars (default simply TRUE). Then we have:

```
ON INSERT i INTO A : INSERT ( i WHERE bx ) INTO V ;
ON INSERT i INTO B : INSERT ( i WHERE ax ) INTO V ;

ON DELETE d FROM A : DELETE ( d WHERE bx ) FROM V ;
ON DELETE d FROM B : DELETE ( d WHERE ax ) FROM V ;

ON INSERT i INTO V : INSERT i INTO A , INSERT i INTO B ;

ON DELETE d FROM V : DELETE d FROM A , DELETE d FROM B ;
```

Note: I'll leave it as another exercise to show that the specific rules given in connection with the examples earlier in the chapter are all special cases—sometimes slightly "optimized" special cases—of the foregoing general rules.

We also saw that the rule for deleting through an intersection can sometimes lead to results that might be unacceptable, or at least undesirable, in practice. In particular, this criticism—which arises, as I pointed out in Chapter 6, because the rule effectively means treating logical OR as logical AND, or possibly the other way around—applied to the first version of the "parts on sale" vs. "parts in stock" example. But let's think about that example a moment longer ... The real problem with that example was that, given a particular tuple to be deleted from the intersection XLK, the DBMS was unable to tell whether that tuple still logically belonged in relvar PL or relvar PK or neither. In effect, the pertinent restriction condition for each of PL and PK in that example was simply TRUE (contrast the situation with Example 1, the NLS vs. NPS example, as discussed earlier in the chapter). And my proposed solution to this problem was, in effect, to redesign the database in such a way that the DBMS could tell which relvar(s) a given tuple logically belonged in after all—a solution that (I venture to suggest) will often work in practice, and indeed is likely to offer benefits in other areas as well, in addition to view updating as such.

One last point on the foregoing: Clearly, what we have in this example is a situation in which information is hidden once again. But the information hiding we're talking about here is different in kind from what we've seen earlier in this book. In earlier examples, information was hidden *from some user*, because (as I put it in earlier chapters) the user in question was seeing only part of the total picture—e.g., consider the example in Chapter 1, where the user saw the restriction LS ("London suppliers") and not the restriction NLS ("non London suppliers"). But in the present situation we're talking about information that's hidden not from the user, but *from the DBMS*. To be specific, what's being hidden in the case at hand is the criterion for deciding whether a given tuple logically belongs in PL or PK or neither. And as already indicated, my solution to this problem is to change the design in such a way as to make that information available to the DBMS after all.

Chapter 10

Union Views

Listen to me—go spread the news—
It's easy to update union views!

—Anon.: *Where Bugs Go*

Now let's move on to union. You won't be surprised to learn that the structure of this chapter is very similar to that of the previous one, on intersection. Just one preliminary remark: *A* UNION *B* resembles *A* INTERSECT *B* in that it's not very interesting if *A* = *B*; unlike *A* INTERSECT *B*, however, it certainly is interesting—well, somewhat interesting—if *A* and *B* are disjoint. So my first example will be exactly that, an example involving a disjoint union.

EXAMPLE 1: DISJOINT UNION

This example is essentially the inverse of the motivating example from Chapters 1 and 4. Thus, we're given two relvars, LS ("London suppliers") and NLS ("non London suppliers"), that look like this:

```
LS  { SNO , SNAME , STATUS , CITY } KEY { SNO }
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

Of course, in terms of our usual suppliers-and-parts database, these relvars are probably views—views of relvar S, to be specific—but you can think of them as base relvars if you like. More to the point, observe that these relvars are indeed disjoint, inasmuch as it's impossible for the very same tuple to appear in both. Here are the predicates:

LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is London).*

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

Now let's define the (disjoint) union of LS and NLS as a view ULN:

```
ULN { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

The predicate for this relvar is:

> ULN: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is either London or not London).*

(Of course, if LS and NLS are indeed views of relvar S, then ULN is identical to that relvar S.) Sample values are shown in Fig. 10.1 below.

LS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

ULN

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

Fig. 10.1: Relvars LS, NLS, and ULN—sample values

The following constraints, edited versions of the ones from the discussion of the corresponding example in Chapter 4, clearly apply:

```
CONSTRAINT ... ULN = LS UNION NLS ;
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... LS  = ( ULN WHERE CITY = 'London' ) ;
CONSTRAINT ... NLS = ( ULN WHERE CITY ≠ 'London' ) ;
```

Also, each of the three relvars has {SNO} as its sole key, as already indicated, and {SNO} is a foreign key, referencing ULN, in each of LS and NLS. Note that (in contrast to the intersection examples discussed in the previous chapter) here we do have information equivalence—the design consisting of ULN by itself is information equivalent to the design consisting of LS and NLS taken in combination.

The following compensatory actions are also taken, lightly edited, from Chapter 4:

```
ON DELETE d FROM LS : DELETE d FROM ULN ;
ON DELETE d FROM NLS : DELETE d FROM ULN ;

ON INSERT i INTO LS : INSERT i INTO ULN ;
ON INSERT i INTO NLS : INSERT i INTO ULN ;
```

```
ON DELETE d FROM ULN : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                       DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;

ON INSERT i INTO ULN : INSERT ( i WHERE CITY = 'London' ) INTO LS ,
                       INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;
```

Note that (again in contrast to the intersection examples in the previous chapter) there's no question here of inserts or deletes on either of LS and NLS cascading to the other.

As for a user who sees just relvar ULN, I hope it's obvious without going into detail that such a user can behave in all respects exactly as if that relvar is a base relvar.

Let me now point out that if LS and NLS are indeed views of relvar S, then ULN could alternatively have been defined directly as a restriction of S, thus:

```
S WHERE CITY = 'London' OR CITY ≠ 'London'
```

Let's call this restriction ULN′. Clearly, then, updates on ULN′ should have the same effect as updates on ULN—and so they do. To be specific:

- INSERT: Tuples to be inserted into either ULN or ULN′ must have CITY value either London or not London (!) and so must be inserted into the other as well.

- DELETE: Tuples to be deleted from either ULN or ULN′ certainly do have CITY value either London or not London and so must be deleted from the other as well.

(As noted in the previous chapter, I'll have a lot more to say about the general question of updates on relvars with different but equivalent definitions in Chapter 14.)

Well, as you can see, this first union example is essentially identical to the motivating example from Chapters 1 and 4, mutatis mutandis, and there's really not much more to say about it. So let's hurry on to another example, this one involving an overlapping union instead of a disjoint one.

EXAMPLE 2: EXPLICIT OVERLAP

My second example, like Example 1 in the previous chapter, is based on the one previously discussed in the section "Overlapping Restrictions" in Chapter 4. It involves two relvars, NLS ("non London suppliers") and NPS ("non Paris suppliers"), that look like this:

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

As pointed out in the previous chapter, these relvars do indeed overlap, in the sense that it's possible for the very same tuple to appear in both; moreover, the overlap is explicit, in the sense explained in that same chapter. Here once again are the predicates:

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

NPS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't Paris).*

Now let's define the union of these two relvars as a view ULP:

```
ULP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

The predicate for this relvar is:

ULP: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY either isn't London or isn't Paris).*

(Of course, if NLS and NPS are views of relvar S, as they well might be, then ULP, like ULN in the previous section, is identical to that relvar S.) Sample values are shown in Fig.10.2.

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

NPS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S5	Adams	30	Athens

ULP

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

Fig. 10.2: Relvars NLS, NPS, and ULP—sample values

The following constraints clearly apply:

```
CONSTRAINT ... ULP = NLS UNION NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                        ( NPS WHERE CITY ≠ 'London' ) ;
```

Also, each of the three relvars has {SNO} as its sole key, as already indicated, and {SNO} is a foreign key, referencing ULP, in each of NLS and NPS. Note that here again we have

information equivalence—the design consisting of ULP by itself is information equivalent to the design consisting of NLS and NPS in combination. *Note:* The fact that {SNO} is a key for ULP in particular guarantees that if some supplier is represented in both NLS and NPS, then the two tuples representing that supplier are in fact one and the same. Of course, this constraint will be enforced automatically if NLS and NPS are in fact views of the suppliers relvar S.

Here now are the compensatory actions—first, the rules for cascading updates on NLS to NPS and vice versa (which are the same as their counterparts in Chapter 9, of course):

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

And here are the rules for cascading updates on NLS and NPS to the union view ULP:

```
ON INSERT i INTO NLS : INSERT i INTO ULP ;
ON INSERT i INTO NPS : INSERT i INTO ULP ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM ULP ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM ULP ;
```

Finally, here are the rules for updates on the union relvar as such:

```
ON DELETE d FROM ULP :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;

ON INSERT i INTO ULP :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
```

Points arising:

- Let's focus on the insert rule for a moment. As far as relvar ULP is concerned, it would of course be sufficient to cascade to just one of the relvars NLS and NPS. But cascading to either NLS or NPS will (if logically necessary) cause an additional cascade to the other relvar anyway; and since there's no good reason to favor one over the other, at least in concrete syntax (assuming, of course, that both relvars are visible)—it's a difference that makes no difference, so to speak—I've specified cascades to both, for reasons of symmetry and explicitness.

- As a matter of fact, a similar remark applies to the delete rule also (in general, deleting from a union needs to cascade to both of the relvars involved; in the case at hand, however,

cascading to either will cause an additional cascade to the other one anyway, if logically necessary).

- In the case of the delete rule, the restriction conditions could be dropped without significant loss.

As for a user who sees just relvar ULP, I hope it's obvious without going into detail that such a user can behave in all respects exactly as if that relvar is a base relvar.

Let me now point out that if NLS and NPS are indeed views of relvar S, then ULP could alternatively have been defined directly as a restriction of S, thus:

```
S WHERE CITY ≠ 'London' OR CITY ≠ 'Paris'
```

Let's call this restriction ULP′. Clearly, then, updates on ULP′ should have the same effect as updates on ULP—and so they do. To be specific:

- INSERT: Tuples to be inserted into either ULP or ULP′ must have CITY value either not London or not Paris and so must be inserted into the other as well.

- DELETE: Tuples to be deleted from either ULP or ULP′ certainly do have CITY value either not London or not Paris and so must be deleted from the other as well.

(Once again, I'll have more to say about the general question of updates on relvars with different but equivalent definitions in Chapter 14.)

As you can see, then, this second example isn't seriously different in kind from the one discussed in the previous section (essentially because both preserve information equivalence). So let's move on to look at an example where information equivalence is lost.

EXAMPLE 3: IMPLICIT OVERLAP

This example is an edited version of Example 2 from the previous chapter. Just to remind you, the basic situation is as follows: At any given time, some parts are on sale, some parts are in stock, and some are both. Furthermore, this state of affairs is represented by means of two relvars, each with just a single attribute PNO—one relvar (PL) giving part numbers for parts on sale, the other (PK) giving part numbers for parts in stock. Of course, the union of these two relvars—call it ULK—gives part numbers for parts that are either on sale or in stock or both. The point is, however, we can't tell just by looking at a given tuple in the original parts relvar P whether that part ought to be represented in PL or PK or both or neither. So the predicates are as follows:

PL: *Part PNO is on sale.*

PK: *Part PNO is in stock.*

ULK: *Part PNO is on sale or in stock.*

Sample values are given in Fig. 10.3.

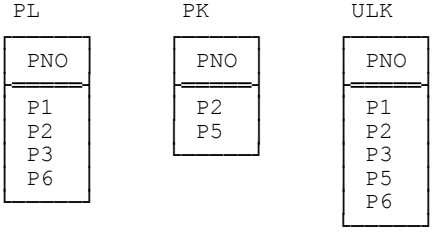


Fig. 10.3: Relvars PL, PK, and ULK—sample values

As for constraints, {PNO} is obviously the sole key for each of these three relvars, and of course ULK is equal to the union of the other two:

```
CONSTRAINT ... ULK = PL UNION PK ;
```

Updates on PL and/or PK are noncontroversial:

```
ON INSERT i INTO PL : INSERT i INTO ULK ;
ON INSERT i INTO PK : INSERT i INTO ULK ;

ON DELETE d FROM PL : DELETE ( d MINUS PK ) FROM ULK ;
ON DELETE d FROM PK : DELETE ( d MINUS PL ) FROM ULK ;
```

But what about updates on ULK? Well, the delete rule is obvious:

```
ON DELETE d FROM ULK :
   DELETE d FROM PL , DELETE d FROM PK ;
```

As for the insert rule, clearly it would be sufficient for inserts on ULK to cascade to just one of PL and PK (it must cascade to at least one, of course). However, there's no good reason for choosing either of these possibilities over the other; in fact, what we have here is the ambiguity (?) issue once again, albeit in a slightly different guise, and I'll have a little more to say about it in the final section of this chapter. For now, however, I propose the following as an appropriate insert rule (and observe that, not incidentally, it has the same general form as its counterparts in Examples 1 and 2 in the previous two sections):

```
ON INSERT i INTO ULK :
   INSERT i INTO PL , INSERT i INTO PK ;
```

Of course, as indicated earlier, we're dealing here once again with a situation in which information equivalence is lost. To be specific, any information that can be represented by relvar ULK alone can certainly be represented by relvars PL and PK taken in combination, but the converse is false. (Here's an example of a query on the latter that has no exact counterpart on the former: "Get part numbers for parts that are on sale and not in stock.") As a consequence, it should be obvious that there'll be certain updates that can be done on PL and/or PK that have no exact counterpart on ULK. An example of such an update is "Delete some existing tuple from PL without simultaneously deleting that same tuple from PK" (i.e., update the database to say some part is no longer on sale but, assuming it was previously in stock, still is).

I'll leave it as an exercise to determine the implications of all of the foregoing for a user who sees just relvar ULK. However, let me now point out that (as in the case of the intersection analog of this example in the previous chapter) there's unfortunately another issue here. Suppose we start off with just relvars PL and PK, without the union view. Suppose too, just to be definite, that part P4 is represented in neither PL nor PK. Then the following INSERT operation is clearly legitimate:

```
INSERT ( P4 ) INTO PL ;
```

Observe in particular that there's no question of this insert cascading to relvar PK. But now suppose we introduce the union view ULK. Given the rules defined above, then, this insert *will* now cascade to relvar PK!—and it'll do so, moreover, even if view ULK isn't visible to the user issuing the insert on relvar PL. Or to put the point another way: Introducing that view apparently requires the simultaneous introduction of a cascade insert rule from PL to PK and vice versa.[1]

Now, one possible fix for this problem, in the particular case at hand, is to execute an appropriate delete immediately after the insert, thus:

```
INSERT ( P4 ) INTO PL ;
DELETE ( P4 ) FROM PK ;
```

However, the solution to be discussed in the subsection immediately following is in my opinion greatly to be preferred (of course, it's essentially the same as the solution to the corresponding problem in the intersection case, as discussed in the previous chapter).

---

[1] A related issue is the following. Under the stated rules for updates on ULK, inserting a tuple into ULK and then deleting it again will preserve the status quo, but deleting a tuple from ULK and then inserting it again might not. The reason is that deleting a tuple from ULK might actually cause a tuple to be deleted from just one of PL and PK, whereas inserting a tuple into ULK will always cause a tuple to be inserted into both. Of course, the status quo with respect to relvar ULK as such is always preserved.

A Better Design

As you'll recall from the previous chapter, the solution I have in mind here involves replacing relvars PL and PK in their entirety by a single relvar, POI, with attributes PNO, ON_SALE, and IN_STOCK (where attributes ON_SALE and IN_STOCK are of type BOOLEAN and have the obvious interpretations, and {PNO} is the sole key). A possible value for such a relvar is shown in Fig. 10.4 (a repeat of Fig. 9.3 from the previous chapter).

POI

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P4	FALSE	FALSE
P5	FALSE	TRUE
P6	TRUE	FALSE

Fig. 10.4: Relvar POI—sample value

Now we define restriction views PL′ and PK′ and union view ULK′ as indicated by the following constraints:

```
CONSTRAINT ... PL'  = POI WHERE ON_SALE ;
CONSTRAINT ... PK'  = POI WHERE IN_STOCK ;
CONSTRAINT ... ULK' = PL' UNION PK' ;
```

Fig. 10.5 shows sample values corresponding to those in Fig. 10.4.

PL′

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P6	TRUE	FALSE

PK′

PNO	ON_SALE	IN_STOCK
P2	TRUE	TRUE
P5	FALSE	TRUE

ULK′

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P5	FALSE	TRUE
P6	TRUE	FALSE

Fig. 10.5: Relvars PL′, PK′, and ULK′—sample values

The predicates are as follows:

PL′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and ON_SALE is TRUE).*

PK′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and IN_STOCK is TRUE).*

ULK′: *Part PNO is on sale if and only if ON_SALE is TRUE and in stock if and only if IN_STOCK is TRUE (and ON_SALE is TRUE or IN_STOCK is TRUE).*

As for constraints, each of these relvars has {PNO} as sole key, and {PNO} is also a foreign key, referencing ULK′, in each of PL′ and PK′. The following constraints also hold:[2]

```
CONSTRAINT ... IS_EMPTY ( PL' WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK' WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( ULK' WHERE NOT ( ON_SALE OR IN_STOCK ) ) ;
```

Here now repeated from Chapter 9 are the update rules connecting relvar POI and relvars PL′ and PK′:

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL' ,
   INSERT ( i WHERE IN_STOCK ) INTO PK' ;

ON DELETE d FROM POI :
   DELETE ( d WHERE ON SALE ) FROM PL' ,
   DELETE ( d WHERE IN_STOCK ) FROM PK' ;

ON INSERT i INTO PL' : INSERT i INTO POI ;

ON INSERT i INTO PK' : INSERT i INTO POI ;

ON DELETE d FROM PL' : DELETE d FROM POI ;

ON DELETE d FROM PK' : DELETE d FROM POI ;
```

And here are the rules connecting relvars PL′ and PK′ and relvar ULK′:

```
ON INSERT i INTO PL' : INSERT i INTO ULK' ;
ON INSERT i INTO PK' : INSERT i INTO ULK' ;

ON DELETE d FROM PL' : DELETE ( d WHERE NOT ( IN_STOCK ) ) FROM ULK' ;
ON DELETE d FROM PK' : DELETE ( d WHERE NOT ( ON_SALE ) ) FROM ULK' ;
```

[2] As in Chapter 9, there's also a constraint to the effect that the restrictions PL′ WHERE IN_STOCK and PK′ WHERE ON_SALE must be equal, as a consequence of which relvars PL′ and PK′ violate *The Principle of Orthogonal Design*.

```
ON INSERT i INTO ULK′ :
   INSERT ( i WHERE ON_SALE ) INTO PL′ ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′ ;

ON DELETE d FROM ULK′ :
   DELETE d FROM PL′ , DELETE d FROM PK′ ;
```

And these rules are all, as I hope you'll agree, completely noncontroversial (though as in Chapter 9 I think it's instructive to compare them with the rules I gave in connection with the previous version of the example). To repeat from that previous chapter: In effect, what I've done here is redesign the database in such a way that information that was previously represented only implicitly, by the relvar names PL and PK, is now represented explicitly by values of the attributes ON_SALE and IN_STOCK instead. And the effect of that redesign is to convert the previous version of the example—which suffered from update behavior that was at least arguably unacceptable—into a version that behaves just like Example 1 as discussed earlier in the chapter. And, of course, this latter example didn't suffer from the ambiguities and consequent update problems that arose with the previous version of the example currently under discussion.

I'll leave it as an exercise to determine the implications of the foregoing for a user who sees just relvar ULK′. As for defining ULK′ directly as a restriction of POI—which we obviously could have done if we'd wanted to—I'll leave it as another exercise to show that the behavior of such a restriction with respect to updates would be essentially identical to that of the union version as described above. Finally, I'll also leave it as an exercise to show that (assuming it's supposed to be capable of supporting update operations properly) a "double prime" version of the example—analogous to the "double prime" version of the counterpart example, Example 2, in Chapter 9—is effectively a nonstarter.

A Remark on Disjoint Union

I'd like to consider, briefly, a slightly different example. Suppose some parts are manufactured abroad, others are manufactured domestically, and no parts are both. Suppose further that we represent this situation by means of two relvars, both with a single attribute PNO, one (PA) giving part numbers for parts manufactured abroad, the other (PD) giving part numbers for parts manufactured domestically. Note that the union of these two relvars—call it UAD—is a disjoint union and simply gives part numbers for all parts. But the point is, of course, we can't tell just by looking at a tuple in that union whether the tuple in question derives from PA or PD (contrast the situation with Example 1 earlier in this chapter, which also involved a disjoint union). The predicates are as follows:

PA: *Part PNO is manufactured abroad.*

PD: *Part PNO is manufactured domestically.*

UAD: *Part PNO is manufactured either abroad or domestically (and not both).*

Now, the significant point about this example, compared to the example involving relvars PL and PK as discussed earlier in this section (where the union wasn't disjoint), is that the following constraint clearly holds:

```
CONSTRAINT ... DISJOINT { PA , PD } ;
```

As a consequence of this constraint, first, deletes on UAD will succeed (in general), though deleting any given tuple will effectively cascade to just one of PA and PD; second, and perhaps more to the point, inserts on UAD will always fail on a **Golden Rule** violation (unless they're "no ops").

CONCLUDING REMARKS

In closing, let me abstract from the various examples discussed in earlier sections and summarize the update rules when there's a union involved. Let *V* be defined as *A* UNION *B*. Further, assume for simplicity that tuples to be inserted into *A* and *B* are required to satisfy boolean expressions *ax* and *bx*, respectively, where *ax* and *bx* denote restriction conditions on the pertinent relvars (default simply TRUE). Then we have:

```
ON INSERT i INTO A : INSERT i INTO V ;
ON INSERT i INTO B : INSERT i INTO V ;

ON DELETE d FROM A : DELETE ( d WHERE bx ) FROM V ;
ON DELETE d FROM B : DELETE ( d WHERE ax ) FROM V ;

ON INSERT i INTO V :
   INSERT ( i WHERE ax ) INTO A ,
   INSERT ( i WHERE bx ) INTO B ;

ON DELETE d FROM V :
   DELETE ( d WHERE ax ) FROM A ,
   DELETE ( d WHERE bx ) FROM B ;
```

Note: I'll leave it as another exercise to show that the specific rules given in connection with the examples earlier in the chapter are all special cases—sometimes slightly "optimized" special cases—of the foregoing general rules.

Now, we saw that the rule for inserting through a union can sometimes lead to results that might be unacceptable, or at least undesirable, in practice. Let me elaborate on this point for a moment. Let the predicates for *A* and *B* be *PA* and *PB*, respectively; then the predicate for *V* = *A* UNION *B* is *PA* OR *PB*. Thus, if we insert *t* into *V*, we mean the proposition *PA*(*t*) OR *PB*(*t*) is true. By contrast, inserting *t* into both *A* and *B* means *PA*(*t*) AND *PB*(*t*) is true. So if inserting *t*

into *V* causes *t* be inserted into both *A* and *B*, we're effectively treating logical OR as logical AND once again.[3]

We also saw that the foregoing criticism applied to the first version of the "parts on sale" vs. "parts in stock" example. However, the real problem with that example was that, given a particular tuple to be inserted into the union ULK, the DBMS was unable to tell whether that tuple logically belonged in relvar PL or relvar PK or both. In effect, the pertinent restriction condition for each of PL and PK in that example was simply TRUE (contrast the situation with Example 2, the NLS vs. NPS example, as discussed earlier in the chapter). And my proposed solution to this problem was, in effect, to redesign the database in such a way that the DBMS could tell which relvar(s) a given tuple logically belonged in after all—a solution that I venture to suggest will often work in practice, and indeed is likely to offer benefits in other areas as well, in addition to view updating as such.

[3] I remind you again that David McGoveran has a proposal for resolving this ambiguity, which I'll be discussing in Chapter 15.

Chapter 11

Difference Views

I'll teach you differences: away, away!
—William Shakespeare: *King Lear* (1605?)

In this chapter I'll consider the question of updating through the relational difference operator (MINUS, in **Tutorial D**). Now, I'm sure you won't be surprised to learn the chapter is fairly similar in structure to its immediate predecessors, on intersection and union. However, it also differs—how appropriate!—in certain important respects, as you'll quickly see. To elaborate briefly: Suppose we're given two relvars *A* and *B*. Clearly, if *A* = *B*, the difference *A* MINUS *B* and the difference *B* MINUS *A* are both empty, so that case isn't very interesting. Also, if *A* and *B* are disjoint, then *A* MINUS *B* is equal to *A* and *B* MINUS *A* is equal to *B*, so that case isn't very interesting either. So the interesting case is the one in which *A* and *B* aren't equal but do overlap. As in Chapter 9, therefore (on intersection), I'll consider two examples, one in which the overlap is explicit and one in which it's merely implicit. For reasons that will become clear later, however, this time I want to consider the implicit case first.

EXAMPLE 1: IMPLICIT OVERLAP

My first example is based once again on the "parts on sale" vs. "parts in stock" example from Chapters 9 and 10. Once again, then, we have two relvars PL and PK, each with just a single attribute PNO—relvar PL gives part numbers for parts on sale, and relvar PK gives part numbers for parts in stock. Now, there are obviously two possible differences we might consider; for definiteness, let's focus on the difference PL MINUS PK (DLK, say), which gives part numbers for parts that are on sale but not in stock. Thus, the predicates are as follows:

PL: *Part PNO is on sale.*

PK: *Part PNO is in stock.*

DLK: *Part PNO is on sale and not in stock.*

Sample values are shown in Fig. 11.1.

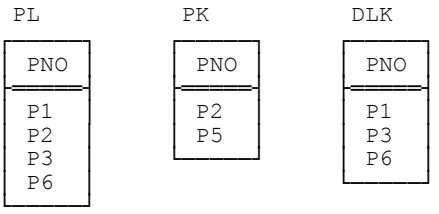


Fig. 11.1: Relvars PL, PK, and DLK—sample values

As for constraints, {PNO} is obviously the sole key for each of the three relvars, and of course DLK is equal to the difference—or, rather, one of the two possible differences—between the other two:

```
CONSTRAINT ... DLK = PL MINUS PK ;
```

Turning now to compensatory actions,[1] here first is the rule for deletes on relvar PL:

```
ON DELETE d FROM PL : DELETE d FROM DLK ;
```

Imagine deleting tuple *t* from PL. Now, *t* will appear in DLK only if it appears in PL and not in PK. And this state of affairs certainly won't exist after the delete on PL, regardless of whether it did so before; hence the specified rule.

Next, here's the rule for inserts on relvar PL:

```
ON INSERT i INTO PL : INSERT ( i MINUS PK ) INTO DLK ;
```

Imagine inserting tuple *t* into PL. Again, *t* will appear in DLK only if it appears in PL and not in PK. And this state of affairs will exist after the insert on PL only if *t* didn't appear in PK before the delete, and still doesn't do so afterward; hence the specified rule.

What about relvar PK? Well, first let me point out that PL and PK play asymmetric roles in the definition of DLK, and it's therefore only to be expected that the corresponding compensatory actions will be asymmetric too—and so indeed they turn out to be. Let's think about inserts first. Again imagine inserting tuple *t* (into PK this time). Now, either *t* already existed in PK or it did not. If it did,[2] then it certainly didn't appear in DLK before the insert, and it'll continue not to do so afterward. Conversely, if it didn't already exist in PK, then after the insert it certainly won't appear in DLK, even if it did so before. All of which leads to the following rule:

---

[1] I suggest using Venn diagrams to help in following the ensuing discussion.

[2] Actually we could ignore this possibility, since we saw in Chapter 4 that no rule ever requests insertion of a tuple that's already present, even if its formulation in concrete syntax appears to do so.

```
ON INSERT i INTO PK : DELETE i FROM DLK ;
```

Finally, imagine deleting tuple *t* from PK. Now, either *t* already existed in PL—the *other* relvar, observe—or it did not. If it did, then it will certainly appear in DLK after the delete, regardless of whether it did so before. Conversely, if it didn't already exist in PL, then it certainly didn't appear in DLK before the delete, and it'll continue not to do so afterward. Which leads to the following rule:

```
ON DELETE d FROM PK : INSERT ( d INTERSECT PL ) INTO DLK ;
```

I remark before continuing that the two compensatory actions just discussed are the first examples we've seen in this book so far in which an insert can cause a delete or vice versa (in other words, they're the first examples in which the compensatory action isn't basically just a simple cascade of some kind).

I turn now to updates on DLK. Here first is the insert rule:

```
ON INSERT i INTO DLK :
   INSERT i INTO PL , DELETE i FROM PK ;
```

Imagine inserting a tuple into DLK for part number *p*. What we mean by that update is that part *p* is on sale and not in stock. Thus, we must insert the tuple into PL if it isn't already there, and delete it from PK if it is already there. Note that neither of these two updates is sufficient in itself, in general—if we did the insert and not the delete, we might wind up effectively asserting that part *p* is both on sale and in stock; likewise, if we did the delete and not the insert, we might wind up effectively asserting that part *p* is neither on sale nor in stock. Both of these possibilities would cause the tuple for part *p* not to appear in DLK, thereby giving rise to an *Assignment Principle* violation.

> *Aside:* Actually there's another way to think about the foregoing (and this alternative perspective might help you understand the delete rule too, which is discussed next). As you can easily confirm by means of a Venn diagram, any given difference *A* MINUS *B* can be regarded as the *intersection A* INTERSECT *C*, where *C* is the (absolute) complement of *B*. In accordance with the discussions in Chapter 9, therefore, an insert on *A* MINUS *B* should cause an insert on *A* and an insert on *C*. But an insert on *C* is a *delete* on *B*—in effect, it causes tuples to be removed from *B* and added to *C*. *End of aside*.

So what about deletes on DLK? Well, symmetry suggests the following rule:

```
ON DELETE d FROM DLK :
   DELETE d FROM PL , INSERT d INTO PK ;
```

Does this rule make sense? Imagine deleting the tuple from DLK for part number *p*. What we mean by that update is that part *p* is either in stock or not on sale, and possibly both. Clearly,

then, it would be sufficient to perform just one of the specified actions—insert the tuple into PK (so the part is in stock) or delete it from PL (so the part is not on sale)—in order to achieve the desired effect; in fact, to do both as the foregoing rule requires is effectively to treat logical OR as logical AND once again. The trouble is, of course, there doesn't seem to be any good reason to choose either of these two actions over the other. For that reason, I propose to adopt the rule as I've stated it above, at least until further notice.[3]

Of course, we're dealing here with another situation in which information equivalence is lost. To be specific, any information that can be represented by relvar DLK alone can certainly be represented by relvars PL and PK taken in combination, but the converse is false. (Here's an example of a query on the latter that has no exact counterpart on the former: "Get part numbers for parts that are in stock and not on sale.") As a consequence, it should be obvious that there'll be certain updates that can be done on PL and/or PK that have no exact counterpart on DLK. An example of such an update is "Insert some new tuple into both PL and PK" (i.e., update the database to say some part is both on sale and in stock).

I'll leave it as an exercise to determine the implications of all of the foregoing for a user who sees just relvar DLK. However, let me now point out that, as in the case of the intersection and union analogs of this example in the previous two chapters, there's unfortunately another issue here. Suppose we start off with just relvars PL and PK, without the difference view. Suppose too, just to be definite, that part P1 is represented in PL but not PK. Then each of the following updates is clearly legitimate:

```
DELETE ( P1 ) FROM PL ;

INSERT ( P1 ) INTO PK ;
```

Observe in particular that there's no question of the foregoing delete causing an insert on relvar PK or the foregoing insert causing a delete on relvar PL. But now suppose we introduce the difference view DLK. Given the rules defined above, then, the delete on PL *will* now cause an insert on PK, and the insert on PK *will* now cause a delete on PL!—and they'll do so, moreover, even if view DLK isn't visible to the user issuing the delete on PL or the insert on PK. Or to put the point another way: Introducing that view apparently requires the simultaneous introduction of a rule that makes deletes on PL cause inserts on PK and vice versa.[4]

Now, one possible fix for this problem, in the particular case at hand, is to execute another delete or another insert immediately following the original one, as shown here:

---

[3] Once again I remind you that David McGoveran has a proposal for resolving such ambiguities, which I'll be discussing in Chapter 15.

[4] You might recall from the previous two chapters that there was a related issue that arose at this point: Under the update rules defined in those chapters, inserting a tuple into an intersection view and then deleting it again, or deleting a tuple from a union view and then inserting it again, might not preserve the status quo (the status quo, that is, with respect to the relvars in terms of which the view in question is defined—the status quo is always preserved with respect to the view as such). I'll leave it as another exercise to determine whether any analogous issues arise with difference views.

```
DELETE ( P1 ) FROM PL ;
DELETE ( P1 ) FROM PK ;

INSERT ( P1 ) INTO PK ;
INSERT ( P1 ) INTO PL ;
```

However, the solution to be discussed in the subsection immediately following is in my opinion greatly to be preferred. Of course, it's essentially the same as the solution to the corresponding problem in the intersection and union cases as discussed in the previous two chapters.

A Better Design

As you'll recall from those previous chapters, the solution I have in mind involves replacing relvars PL and PK in their entirety by a single relvar, POI, with attributes PNO, ON_SALE, and IN_STOCK (where attributes ON_SALE and IN_STOCK are of type BOOLEAN and have the obvious interpretations, and {PNO} is the sole key). A possible value for such a relvar is shown in Fig. 11.2 (a repeat of Fig. 9.3 from Chapter 9, also Fig. 10.4 from Chapter 10).

POI

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P4	FALSE	FALSE
P5	FALSE	TRUE
P6	TRUE	FALSE

Fig. 11.2: Relvar POI—sample value

Now we define restriction views PL′ and PK′ and difference view DLK′ as indicated by the following constraints:

```
CONSTRAINT ... PL'  = POI WHERE ON_SALE ;
CONSTRAINT ... PK'  = POI WHERE IN_STOCK ;
CONSTRAINT ... DLK' = PL' MINUS PK' ;
```

Fig. 11.3 shows sample values corresponding to those in Fig. 11.2.

PL′

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P2	TRUE	TRUE
P3	TRUE	FALSE
P6	TRUE	FALSE

DLK′

PNO	ON_SALE	IN_STOCK
P1	TRUE	FALSE
P3	TRUE	FALSE
P6	TRUE	FALSE

PK′

PNO	ON_SALE	IN_STOCK
P2	TRUE	TRUE
P5	FALSE	TRUE

Fig. 11.3: Relvars PL′, PK′, and DLK′—sample values

Each of these relvars has {PNO} as sole key, and {PNO} is also a foreign key, referencing PL′, in DLK′. The following constraints also obviously hold:[5]

```
CONSTRAINT ... IS_EMPTY ( PL′ WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK′ WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( DLK′ WHERE NOT ( ON_SALE ) OR IN_STOCK ) ) ;
```

As in Chapter 9 (though not Chapter 10), we need an additional constraint to the effect that if some part is represented in both PL′ and PK′, then the two tuples representing that part are in fact one and the same:

```
CONSTRAINT ... UNION { PL′ , PK′ } KEY { PNO } ;
```

Of course, this constraint will be enforced automatically if relvars PL′ and PK′ are indeed, as stated, views of POI.

Here now repeated from Chapters 9 and 10 are the update rules involving relvar POI and relvars PL′ and PK′:

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL′ ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′ ;

ON DELETE d FROM POI :
   DELETE ( d WHERE ON SALE ) FROM PL′ ,
   DELETE ( d WHERE IN_STOCK ) FROM PK′ ;
```

[5] As in Chapters 9 and 10, there's also a constraint to the effect that the restrictions PL′ WHERE IN_STOCK and PK′ WHERE ON_SALE must be equal, as a consequence of which relvars PL′ and PK′ violate *The Principle of Orthogonal Design*.

```
ON INSERT i INTO PL′ : INSERT i INTO POI ;

ON INSERT i INTO PK′ : INSERT i INTO POI ;

ON DELETE d FROM PL′ : DELETE d FROM POI ;

ON DELETE d FROM PK′ : DELETE d FROM POI ;
```

And here are the rules involving relvars PL′ and PK′ and relvar DLK′:

```
ON INSERT i INTO PL′ : INSERT ( i WHERE NOT ( IN_STOCK ) ) INTO DLK′ ;
ON INSERT i INTO PK′ : DELETE i FROM DLK′ ;

ON DELETE d FROM PL′ : DELETE d FROM DLK′ ;
ON DELETE d FROM PK′ : INSERT ( d WHERE ON_SALE ) INTO DLK′ ;

ON INSERT i INTO DLK′ : INSERT i INTO PL′ ;

ON DELETE d FROM DLK′ : DELETE d FROM PL′ ;
```

And these rules are all, as I hope you'll agree, completely noncontroversial (though as in Chapters 9 and 10 I think it's instructive to compare them with the rules I gave in connection with the previous version of the example). So, to repeat from those previous chapters: In effect, what I've done here is redesign the database in such a way that information that was previously represented only implicitly, by the relvar names PL and PK, is now represented explicitly by values of the attributes ON_SALE and IN_STOCK instead. And the effect of that redesign is to convert the previous version of the example—viz., the version with at least arguably unacceptable update behavior—into a version that behaves much more acceptably.

I'll leave it as an exercise to determine the implications of all of the foregoing for a user who sees just relvar DLK′. As for defining DLK′ directly as a restriction of POI—which we clearly could have done if we'd wanted to—I'll leave it as another exercise to show that the behavior of such a restriction with respect to updates would be essentially identical to that of the difference version as described above. Finally, I'll also leave it as an exercise to show that (assuming it's supposed to be capable of supporting update operations properly) a "double prime" version of the example—analogous to the "double prime" version of the counterpart example, Example 2, in Chapter 9—is effectively a nonstarter.

A Remark on Included Difference

I'd like to consider, briefly, a slightly different example. Suppose we're given two relvars, PL (giving, as above, part numbers for parts on sale) and PD (giving part numbers for parts manufactured domestically), and there's a business rule in effect that says that only parts manufactured domestically can be on sale—implying that, at all times, the relation that's the current value of PL is included in the relation that's the current value of PD. Of course, the difference DDL = PD MINUS PL gives part numbers for parts that are manufactured

domestically and aren't on sale. (Note that DDL is an example of an "included difference," because the second operand, relvar PL, is included in the first—meaning, more precisely, that the body of PL at any given time is a subset of the body of PD at the time in question.) The predicates are as follows:

PL: *Part PNO is on sale.*

PD: *Part PNO is manufactured domestically.*

DDL: *Part PNO is manufactured domestically and isn't on sale.*

Now, the significant point about this example, compared to the example involving relvars PL and PK discussed earlier in this section (where the difference wasn't an included difference), is that the following constraint clearly holds:

```
CONSTRAINT ... PL ⊆ PD ;
```

As a consequence of this constraint, given compensatory actions along the lines of those defined in the first part of this section, deletes on DDL will always fail on a **Golden Rule** violation.

EXAMPLE 2: EXPLICIT OVERLAP

Note: I include this section for completeness. It doesn't really add very much, and you can skip it if you like.

Now let's take a look at another example, based on the by now very familiar relvars NLS ("non London suppliers") and NPS ("non Paris suppliers"):

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

Here once again are the predicates:

NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't London).*

NPS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY isn't Paris).*

Now let's define the difference NLS MINUS NPS as a view DLP:

```
DLP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

The predicate for this relvar is:

> DLP: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is Paris).*[6]

Sample values are shown in Fig. 11.4.

NLS

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

DLP

SNO	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

NPS

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 11.4: Relvars NLS, NPS, and DLP—sample values

The following constraints clearly apply:

```
CONSTRAINT ... DLP = NLS MINUS NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... IS_EMPTY ( DLP WHERE CITY ≠ 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                          ( NPS WHERE CITY ≠ 'London' ) ;
```

Also, each of the three relvars has {SNO} as its sole key, as already indicated, and {SNO} in DLP is a foreign key referencing NLS. *Note:* As usual, we additionally need a constraint to the effect that {SNO} is a key for the union of NLS and NPS, in order to guarantee that if some supplier is represented in both of those relvars, then the two tuples representing that supplier are in fact one and the same:

---

[6] Strictly speaking, the portion of this predicate in parentheses should read "and CITY is not London and is not (not Paris)." But it's easy to see that this latter simplifies to just "and CITY is Paris." It follows that the update behavior of DLP and NPS with respect to each other should be identical to that of LS and NLS with respect to each other (see Chapter 4), except of course for the fact that references to London must all be replaced by references to Paris. *Note:* By contrast, the predicate for NPS MINUS NLS is *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is London).*

```
CONSTRAINT ... UNION { NLS , NPS } KEY { SNO } ;
```

Of course, this constraint will be enforced automatically if relvars NLS and NPS are in fact views of the suppliers relvar S.

Here now are the compensatory actions—first, the usual rules for cascading updates from NLS to NPS and vice versa:

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

Next, here's the rule for cascading inserts on NLS to DLP:

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY = 'Paris' ) INTO DLP ;
```

And the rule for cascading deletes on NLS to DLP:

```
ON DELETE d FROM NLS : DELETE ( d WHERE CITY = 'Paris' ) FROM DLP ;
```

(We could drop the restriction condition here without loss.)

As for inserts on NPS, the analysis in previous sections should lead you to expect something like the following:

```
ON INSERT i INTO NPS : DELETE i FROM DLP ;
```

But if tuple *t* is to be inserted into NPS, then tuple *t* must have CITY value something other than Paris—and since tuples in DLP all have CITY value Paris, the compensatory action here becomes a "no op."

Finally, what about deletes on NPS? Again, the analysis in previous sections should lead you to expect something like the following:

```
ON DELETE d FROM NPS : INSERT ( d INTERSECT NLS ) INTO DLP ;
```

But tuples in NPS all have CITY value something other than Paris; hence, so do all tuples in *d*, and so do all tuples in *d* INTERSECT NLS a fortiori.[7] And since tuples in DLP all have CITY value Paris, the compensatory action here becomes a "no op" again. *Note:* Actually, taking these two "no op" rules together, it should be intuitively obvious that—speaking pretty loosely, of course—inserts and deletes on NPS (= non Paris suppliers) can't possibly have any effect on DLP (= Paris suppliers).

---

[7] Again recall our assumption from Chapter 4, according to which every tuple in *d* also appears in NPS.

As for updates on the difference relvar as such, it's easy to see by an analysis very similar to the foregoing that the rules reduce to the following simple form:

```
ON INSERT i INTO DLP : INSERT i INTO NLS ;

ON DELETE d FROM DLP : DELETE d FROM NLS ;
```

This brings me to the end of Example 2. But perhaps you can now see why I wanted to discuss Example 1, the "parts on sale" vs. "parts in stock" example, first—Example 2, the NLS vs. NPS example, although in a sense better behaved than Example 1, isn't so good as an illustration of the underlying concepts, precisely because of the existence of certain constraints that cause the update rules to take a particularly simple form (a state of affairs that makes it hard to see the trees for the forest, one might say).

CONCLUDING REMARKS

Let me abstract from the examples discussed in previous sections and summarize the update rules when there's a difference involved. Let *V* be defined as *A* MINUS *B*. Further, assume for simplicity that tuples to be inserted into *A* and *B* are required to satisfy boolean expressions *ax* and *bx*, respectively, where *ax* and *bx* denote restriction conditions (default simply TRUE). Then we have:

```
ON INSERT i INTO A :
   INSERT ( i WHERE NOT ( bx ) ) INTO V ;
ON INSERT i INTO B :
   DELETE i FROM V ;

ON DELETE d FROM A :
   DELETE d FROM V ;
ON DELETE d FROM B :
   INSERT ( d WHERE ax ) INTO V ;

ON INSERT i INTO V :
   INSERT i INTO A , DELETE i FROM B ;

ON DELETE d FROM V :
   DELETE d FROM A , INSERT d INTO B ;
```

Once again I'll leave it as an exercise to show that the specific rules given in connection with the examples earlier in the chapter are all special cases—sometimes slightly "optimized" special cases—of the foregoing general rules.

Now, we saw, in connection with Example 1 (the implicit overlap example), that the rule for deleting through a difference can sometimes lead to results that in practice might be unacceptable, or at least undesirable. Let me elaborate on this point for a moment. Let the predicates for *A* and *B* be *PA* and *PB*, respectively; then the predicate for *V* = *A* MINUS *B* is *PA*

AND NOT (*PB*). Thus, if we delete *t* from *V*, we mean the proposition *PA*(*t*) AND NOT (*PB*(*t*)) is false, which is logically equivalent to saying NOT(*PA*(*t*)) OR *PB*(*t*) is true. By contrast, deleting *t* from *A* and inserting it into *B* means NOT(*PA*(*t*)) AND *PB*(*t*) is true. So if deleting *t* from *V* causes *t* be deleted from *A* and inserted into *B*, we're effectively treating logical OR as logical AND once again.[8]

To repeat, the foregoing criticism applied to the first version of the "parts on sale" vs. "parts in stock" example. However, the real problem with that example was that, given a particular tuple to be deleted from the difference DLK, the DBMS was unable to tell whether that tuple logically belonged in relvar PK or not. In effect, the pertinent restriction condition for relvar PK in that example was simply TRUE. And my proposed solution to this problem was, in effect, to redesign the database in such a way that the DBMS could tell which relvar(s) a given tuple logically belonged in after all—a solution that I venture to suggest will often work in practice, and indeed is likely to offer benefits in other areas as well, in addition to view updating as such.

[8] I remind you yet again that David McGoveran has a proposal for resolving such ambiguities, which I'll be discussing in Chapter 15.

Chapter 12

Group and Ungroup Views

Updating groups
Don't need no loops

—Anon.: *Where Bugs Go*

I turn now to the question of updating through the relational grouping and ungrouping operators (GROUP and UNGROUP, in **Tutorial D**). Since you might not be familiar with those operators, I'll begin with a brief tutorial.

THE GROUP AND UNGROUP OPERATORS

Consider the relations shown in Fig. 12.1, which we can take to be the current values of two relvars called SP and SPQ, respectively. Of course, relvar SP is just the shipments relvar from our usual suppliers-and-parts database.

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S2	P1	300

SPQ

SNO	PQ
S1	PNO: P1, P2 / QTY: 300, 200
S2	PNO: P1 / QTY: 300

Fig. 12.1: Relvars SP and SPQ—sample values

Now, I hope it's at least intuitively obvious that we have information equivalence here once again[1] (certainly the relations shown in the figure both represent the exact same information).

[1] But see the footnote at the very end of this section.

Now, with regard to those two relations, the one on the left of the figure is just a reduced version of our usual sample value for relvar SP, while the one on the right is what we get if we evaluate the following expression on that sample SP value:

```
SP GROUP ( { PNO , QTY } AS PQ )
```

Aside: In fact the GROUP invocation just shown is logically equivalent to a certain EXTEND invocation—to be specific, the invocation EXTEND SP{SNO}:{PQ := !!SP}, where !!SP denotes a certain *image relation*[2]—and there are reasons to prefer this EXTEND formulation over its GROUP equivalent. Detailed discussion of such matters is beyond the scope of this book, however. Refer to *SQL and Relational Theory* for further explanation. *End of aside*.

Now, given that the relation on the right of the figure is supposed to be the current value of a relvar called SPQ, if we evaluate the following expression on that current value—

```
SPQ UNGROUP ( PQ )
```

—then we get back to the relation on the left. *Terminology:* The relation on the right of the figure is a *grouped* version of the one on the left (grouped on PNO and QTY, to be precise), and the relation on the left is an *ungrouped* version of the one on the right (ungrouped on PQ, to be precise once again). Attribute PQ is a relation valued attribute (RVA for short).[3]

Let me get back for a moment to that relvar SPQ. Here's a **Tutorial D** definition for that relvar:

```
VAR SPQ BASE RELATION
  { SNO CHAR , PQ RELATION { PNO CHAR , QTY INTEGER } }
    KEY { SNO } ;
```

Observe in particular that attribute PQ is of a certain *relation* type, since as I've said that attribute is relation valued. *Note:* As usual, the relvar is subject to a variety of constraints, of course; I've specified just one here—viz., the constraint that {SNO} is the sole key for the relvar—but I don't want to get sidetracked into discussing any others at this juncture.

Now let me define the GROUP and UNGROUP operators more formally. In order to do so, however, it's convenient first to introduce two auxiliary operators called WRAP and UNWRAP. Basically, WRAP wraps up specified attributes of a given relation into a single attribute that's tuple valued, and UNWRAP does the opposite. For example, if the relation on the left of Fig. 12.1 is the current value of relvar SP, the expression

---

[2] Image relations were previously mentioned in Chapter 8.

[3] RVAs were previously mentioned in Chapter 2.

```
SP WRAP ( { PNO , QTY } AS PQ )
```

yields this relation:

SNO	PQ
S1	PNO QTY P1 300
S1	PNO QTY P2 200
S2	PNO QTY P1 300

Note that the PQ values in this relation are tuples, not relations—PQ here is a tuple valued attribute, not a relation valued attribute. Note also that (a) attribute PQ is double underlined (if this relation is understood as the current value of some relvar, then the sole key for that relvar is the entire heading); (b) attribute PNO in the three PQ values is *not* double underlined, because (to say it again) those PQ values are tuples, not relations, and keys apply to relations, not tuples.

As for UNWRAP, let's call the result of the foregoing WRAP operation *spw*. Then the expression

```
spw UNWRAP ( PQ )
```

returns the shipments relation we started with (the one with three tuples shown on the left of Fig. 12.1).

Here now are definitions of GROUP and UNGROUP. *Note:* I give these definitions here mainly for purposes of reference. Like many formal definitions, they might be a little difficult to understand, at least on a first reading. So I think it's important to say up front that it's not crucial to understand them 100 percent at this stage.

- **Definition:** Let the heading of relation *r* be partitioned into subsets $X = \{X1,X2,...,Xm\}$ and $Y = \{Y1,Y2 ...,Yn\}$; also, let *YR* be an attribute name not appearing in *X*. Then the expression *r* GROUP ({*Y1*,*Y2*,...,*Yn*} AS *YR*) returns a relation *s*. The heading of *s* is {*X1*,*X2*,...,*Xm*,*YR*}, where *YR* is of type RELATION {*Y1*,*Y2*,...,*Yn*}. The body of *s* is defined as follows. Let *z* be the result of *r* WRAP ({*Y1*,*Y2*,...,*Yn*} AS *YT*). For each distinct *X* value *x* in *z*, let *yr* be the relation whose tuples are all and only those *YT* values

from tuples in *z* in which the *X* value is *x*; let *t* be a tuple of type TUPLE {*X*,*YR*} with *X* value *x* and *YR* value *yr*; then, and only then, *t* is a tuple of *s*.

- **Definition:** Let relation *s* have an attribute *YR* of type RELATION {*Y1*,*Y2*,...,*Yn*}, and let *X* = {*X1*,*X2*,...,*Xm*} be all of the attributes of *s* other than *YR*; also, let no *Xi* have the same name as any *Yj* ($0 \le i \le m$, $0 \le j \le n$). Then the expression *s* UNGROUP (*YR*) returns a relation *r*. The heading of *r* is {*X1*,*X2*,...,*Xm*,*Y1*,*Y2*,...,*Yn*}. The body of *r* is defined as follows. Let *z* be a relation with heading {*X1*,*X2*,...,*Xm*,*YT*}, where *YT* is of type TUPLE {*Y1*,*Y2*,...,*Yn*}, and body defined thus: For each tuple of *s*, *z* contains a set of tuples of type {*X*,*YT*}, one (*t*, say) for each tuple in the *YR* value in that *s* tuple; each such tuple *t* contains an *X* value equal to the *X* value from the *s* tuple in question and a *YT* value equal to some tuple from the *YR* value in the *s* tuple in question. Let *z* contain no other tuples. Then *r* is the result of *z* UNWRAP (*YT*).

Note: Given a relation *r* and some grouping of *r,* there's always an inverse ungrouping that yields *r* again; however, the converse isn't necessarily so. The following example illustrates this point. Consider the relation—call it *spq*—shown on the left in Fig. 12.2; note in particular that the PQ value for supplier S2 in *spq* is an empty relation. The expression *spq* UNGROUP (PQ) yields the relation—call it *sp*—shown in the middle of the figure. And the expression *sp* GROUP ({PNO,QTY} AS PQ) then yields the relation shown on the right, which is obviously not equal to the original relation *spq*.[4]

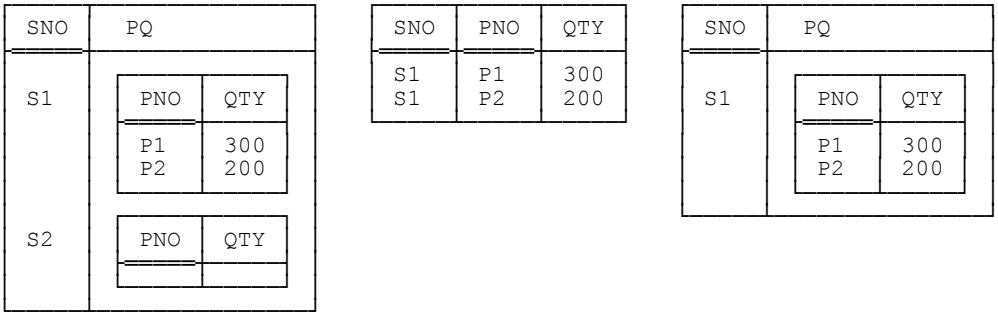


Fig. 12.2: UNGROUP and GROUP aren't necessarily inverses

[4] But do you think those two relations are information equivalent? (You might want to revisit this question after reading Chapter 14.)

A GROUP / UNGROUP EXAMPLE

In order to investigate the question of updating grouped and ungrouped relvars, let's start with the shipments relvar SP, and let's group that relvar on PNO and QTY to define another relvar SPQ in which the grouped attribute is named PQ (basically as in the previous section). Also, to be definite, let's assume until further notice that relvars SP and SPQ are both base ones—no views yet. The predicates are as follows:

SP: *Supplier SNO supplies part PNO in quantity QTY.*

SPQ: *Supplier SNO supplies part p in quantity q, if and only if the set of pairs PQ contains a pair (PNO,QTY) in which PNO is equal to p and QTY is equal to q.*[5]

As for constraints, {SNO,PNO} is the sole key for SP and {SNO} is the sole key for SPQ. Also, we obviously have:

```
CONSTRAINT ... SPQ = SP GROUP ( { PNO , QTY } AS PQ ) ;
CONSTRAINT ... SP  = SPQ UNGROUP ( PQ ) ;
```

Note: We do have reversibility here, because SPQ is defined in terms of SP instead of the other way around, implying among other things that no tuple in SPQ ever has an empty relation as its PQ value.

Now, it's also the case that no tuple in SPQ is ever such that two distinct tuples in the PQ value within that SPQ tuple have the same PNO value. In fact this constraint is a logical consequence of the fact that {SNO,PNO} is a key for SP, together with the fact that the grouping is done (in part) on attribute PNO—but we could state it explicitly if we wanted to:

```
CONSTRAINT ... ( SPQ UNGROUP ( PQ ) ) KEY { SNO , PNO } ;
```

Finally, we obviously have

```
CONSTRAINT ... SPQ { SNO } = SP { SNO } ;
```

(though this constraint too is in fact a logical consequence of the way SPQ is defined).

Now let's think about updates on relvar SP and the corresponding compensatory actions on relvar SPQ. For definiteness, let the current values of SP and SPQ be as shown in Fig. 12.1. Suppose, then, that we insert the shipment tuple (S3,P2,200) into SP. (The point about this first example is that relvar SP doesn't currently contain any tuples for supplier S3, and therefore

---

[5] As you can see, the RVA makes the predicate for relvar SPQ a little tricky. This state of affairs gives some indication as to why RVAs are usually—though not invariably—contraindicated, at least in base relvars. Detailed consideration of this point would take us much too far afield here, however; see *Database Design and Relational Theory* for further discussion.

relvar SPQ doesn't do so either.) Clearly, then, all we need to do to relvar SPQ in this case is just insert the following tuple:

SNO	PQ
S3	PNO QTY P2 200

By contrast, suppose we insert the shipment tuple (S1,P3,400) into SP. (The point about this second example is that relvar SP does currently contain some tuples for supplier S1, and therefore relvar SPQ does so too; in fact, of course, it contains exactly one such tuple.) Clearly, then, what we need to do to relvar SPQ in this case is replace the existing tuple for supplier S1 by the following one:

SNO	PQ
S1	PNO QTY P1 300 P2 200 P3 400

Considerations such as the foregoing lead to the following rule for inserts on SP:

```
ON INSERT i INTO SP :
   DELETE ( SPQ MATCHING i ) FROM SPQ ,
   INSERT ( ( SP MATCHING i { SNO } )
                          GROUP ( { PNO , QTY } AS PQ ) ) INTO SPQ ;
```

Note: I'm relying here on the fact that—thanks to the way multiple assignment is defined when two or more of the individual assignments involve the same target (see Appendix A)—the delete on SPQ will be done before the insert on that same relvar. However, we can avoid any such reliance by reformulating the rule as follows:

```
ON INSERT i INTO SP :
   WITH ( t1 := SPQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := t2 GROUP ( { PNO , QTY } AS PQ ) ) :
   DELETE t1 FROM SPQ ,
   INSERT t3 INTO SPQ ;
```

Now, one interesting thing about this example is that the corresponding delete rule is very similar to the insert rule as just defined. To be more specific, if a tuple is deleted from SP, the sole SPQ tuple for the pertinent supplier needs to be deleted too; then the remaining SP tuples (if any) for that supplier can be used to compute a new tuple to be inserted into SPQ. So here's the rule:

```
ON DELETE d FROM SP :
   WITH ( t1 := SPQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := t2 GROUP ( { PNO , QTY } AS PQ ) ) :
   DELETE t1 FROM SPQ ,
   INSERT t3 INTO SPQ ;
```

What about updates on SPQ? Well, deletes are easy:

```
ON DELETE d FROM SPQ :
   DELETE ( SP MATCHING d ) FROM SP ;
```

In fact I think inserts are easy too. By way of example, assume again that the current values of relvars SP and SPQ are the relations shown in Fig. 12.1. Now consider what happens if we try to insert the following tuple into relvar SPQ:

SNO	PQ
S4	PNO QTY P2 200 P4 300 P5 400

Well, I think it's clear that what we need to do here is insert the following three tuples into relvar SP:

SNO	PNO	QTY
S4	P2	200

SNO	PNO	QTY
S4	P4	300

SNO	PNO	QTY
S4	P5	400

So here's the insert rule:

```
ON INSERT i INTO SPQ :
   INSERT ( i UNGROUP ( PQ ) ) INTO SP ;
```

By the way, consider what happens according to this rule if we try to insert an SPQ tuple with an empty relation as its PQ value—the following tuple, say:

SNO	PQ
S4	PNO QTY

Well, *i* here is the relation containing just the foregoing tuple, and the expression *i* UNGROUP (PQ) thus evaluates to an empty relation. Thus, nothing is inserted into relvar SP. So the operation overall fails on a **Golden Rule** violation—to be specific, it violates the constraint to the effect that SPQ{SNO} = SP{SNO}.

Recall now that, so far, I've been assuming SP and SPQ are both base relvars. But I think you can see that if SPQ is in fact a (grouped) view of SP, it really makes no difference—all of the update rules as previously discussed apply unchanged. And the same is true if, conversely, SP is an (ungrouped) view of SPQ; what's more, it remains true even if it can't be guaranteed that no tuple in SPQ ever has an empty relation as its PQ value, despite the fact that information equivalence is clearly lost in this case (because the constraint SPQ = SP GROUP ({ PNO,QTY} AS PQ) no longer holds).

A SUMMARIZE EXAMPLE

The **Tutorial D** SUMMARIZE operator comes in two forms. I defer discussion of the more general form to the next chapter (Chapter 13); however, it's convenient to discuss the simpler form here. (The reason it's convenient is that the simpler version, although it's actually a special case of the more general version, can be thought of as a variation on the GROUP operator as discussed in the previous two sections.) Here's an example. Once again, let the current value of relvar SP be the relation shown on the left in Fig. 12.1. Then the following expression—

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

—yields the relation shown here:

SNO	TQY
S1	500
S2	300

You can think of this result being produced as follows (see later for a formal definition): First, the expression SP GROUP ({PNO,QTY} AS PQ) is evaluated, to produce an intermediate result *spq*, say; then, in each tuple in *spq*, the PQ value—which is a relation, of course—is replaced by a TQY value, that TQY value being obtained by computing the "summary" SUM(QTY) on that PQ relation.

> *Aside:* In fact the foregoing SUMMARIZE invocation is logically equivalent to a certain EXTEND invocation—to be specific, the invocation EXTEND SP{SNO}:{TQY := SUM(!!SP,QTY)}, where !!SP denotes a certain image relation—and there are reasons to prefer this EXTEND formulation over its SUMMARIZE equivalent. Again, however, detailed discussion of such matters is beyond the scope of this book. Refer to *SQL and Relational Theory* for further explanation. *End of aside*.

Here now is another example ("for each supplier, compute the sum of *distinct* shipment quantities"):

```
SUMMARIZE SP { SNO , QTY } BY { SNO } : { TQY  := SUM ( QTY ) }
```

This expression denotes a summarization of a certain projection of SP—the projection on SNO and QTY, to be specific—and that projection contains, for each supplier number, just the pertinent distinct shipment quantities, as required.[6] *Note:* Given the sample value of SP in Fig. 12.1, the expression overall happens to yield the same result as the previous example does, but it won't do so in general, of course.

Here now is a definition of this form of SUMMARIZE:[7]

- **Definition:** Let relation *r* have attributes called *A1*, *A2*, ..., *An* (and possibly others) and no attribute called *B*. Then the expression SUMMARIZE *r* BY {*A1*,*A2*,...,*An*} : {*B* := *summary*} returns a relation with heading {*A1*,*A2*,...,*An*,*B*} and body the set of all tuples *t* such that *t* is equal to the projection of some tuple of *r* on *A1*, *A2*, ..., *An*, extended with a value *b* for *B*. That value *b* is computed by evaluating *summary* over all tuples of *r* that have the same value for attributes *A1*, *A2*, ..., *An* as *t* does.

  > *Aside:* I assume for the purposes of this book that the notion of "evaluating a summary" can be defined precisely.[8] I certainly don't want to attempt such a definition here—there are certain complexities involved, complexities that (perhaps fortunately) have no bearing

---

[6] As this example should be sufficient to suggest, **Tutorial D** has no need for—nor does it support anything analogous to—SQL's ad hoc trick of allowing summaries to include the keyword DISTINCT.

[7] Observe that this definition makes use of the tuple version of projection, also a tuple version of the relational EXTEND operator. See *SQL and Relational Theory* for further discussion.

[8] In the example, evaluating the summary SUM(QTY) actually requires invoking the SUM *aggregate operator* on the QTY attribute of what I earlier referred to as "the PQ relation." For further explanation, see *SQL and Relational Theory*.

on the main subject of this book. So I'll just assume you have an adequate intuitive understanding of what's involved and leave it at that. I'll assume also that you're familiar with the kinds of summaries that can typically be requested—counts, sums, averages, and so forth. *End of aside.*

So now let's assume SP is a base relvar, and let's define a view STQ as indicated by the following constraint:

```
CONSTRAINT ... STQ = SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

The predicates are as follows:

SP: *Supplier SNO supplies part PNO in quantity QTY.*

STQ: *Supplier SNO supplies parts in total quantity TQY.*

Note: Actually, the predicate for STQ is inadequate as it stands—it could do with a tiny refinement. I'll come back to this point in the next chapter, but you might like to see if you can figure out for yourself just what it is I'm getting at here.

Of course, there's no information equivalence in this example. That's why defining STQ as a view of SP makes sense, but defining SP as a view of STQ certainly doesn't—in fact, it can't be done.

As for constraints, {SNO,PNO} is the sole key for SP and {SNO} is the sole key for STQ. Also, we obviously have the following—

```
CONSTRAINT ... STQ { SNO } = SP { SNO } ;
```

—though this constraint is in fact a logical consequence of the way STQ is defined.

Now let's think about updates on relvar SP and the corresponding compensatory actions. For definiteness, let the current values of relvars SP and STQ be as shown in Fig. 12.3.

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S2	P1	300

STQ

SNO	TQY
S1	500
S2	300

Fig. 12.3: Relvars SP and STQ—sample values

Suppose we insert the shipment tuple (S3,P2,200) into SP. Clearly, then, all we need to do to relvar STQ in this case is insert the tuple (S3,200). By contrast, suppose we insert the

shipment tuple (S1,P3,400) into SP; now what we need to do is replace the existing tuple for supplier S1 in relvar STQ by this one: (S1,900). So the rule for inserts on SP looks like this:

```
ON INSERT i INTO SP :
   WITH ( t1 := STQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := SUMMARIZE t2 BY { SNO } : { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

As you can see, therefore, the foregoing is very similar to its counterpart in the GROUP/UNGROUP example from the previous section. And it should be clear that an analogous remark applies to the delete rule as well. To be specific, if a tuple is deleted from SP, the sole STQ tuple for the pertinent supplier needs to be deleted too; then the remaining SP tuples (if any) for that supplier can be used to compute a new tuple to be inserted into STQ. So here's the rule:

```
ON DELETE d FROM SP :
   WITH ( t1 := STQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := SUMMARIZE t2 BY { SNO } : { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

What about updates on STQ? Well, once again deletes are easy:

```
ON DELETE d FROM STQ :
   DELETE ( SP MATCHING d ) FROM SP ;
```

However, inserts make no sense, in general; for example, what could it possibly mean to insert the tuple (S5,600) into STQ? So there's no rule for inserts on STQ.[9] Of course, the fact that there's no rule doesn't mean we can't do inserts (indeed, the update rules for SP explicitly call for such actions)—it only means that such operations are likely to fail on a **Golden Rule** violation, except possibly if they're part of some multiple assignment (as indeed they are, in the case of the rules for updates on SP).

[9] Actually it might sometimes be possible to perform inserts on a view such as STQ after all; for example, there might be just one summand. Alternatively, some constraint might hold that has the effect of making the view "insertable into"; an example might be a constraint to the effect that the summands must all be equal. (Thanks to David McGoveran for these observations.)

Chapter 13

Extension and Summarization Views

Flimmery flummery
Extension and summary
Both follow the rules
Taught in all the best schools

—Anon.: *Where Bugs Go*

The relational EXTEND and SUMMARIZE operators resemble each other inasmuch as they both produce a result containing a new "computed" attribute.[1] Informally, we might say that EXTEND performs computation "across tuples," while SUMMARIZE performs computation "down attributes." However, I hasten to add that this characterization is loose in the extreme, as is shown by—among other things—the fact that SUMMARIZE can actually be defined in terms of EXTEND, as is explained in *SQL and Relational Theory* (and as indeed was noted in passing in the previous chapter).

AN EXTEND EXAMPLE

I'll begin with a simplified version of an example from Chapter 2. First, let's agree for simplicity to ignore all attributes of relvar P other than PNO and WEIGHT. Second, suppose WEIGHT values in that relvar are given in pounds. Third, suppose 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 }
```

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

[1] I'm ignoring here what I referred to in Chapter 2 as the second form of EXTEND, where the attribute that's computed isn't a "new" one after all. Moreover, I'll continue to ignore that form of EXTEND throughout this chapter, because it can be defined straightforwardly in terms of the first form.

PNO	WEIGHT	GMWT
P1	12.0	5448.0
P2	17.0	7718.0
P3	17.0	7718.0
P4	14.0	6356.0
P5	12.0	5448.0
P6	19.0	8626.0

So let's assume relvar P is a base relvar (as usual), and let's define an extended version as above—let's call it PX—as a view. The predicates are as follows:

P: *Part PNO is used in the enterprise and has weight WEIGHT pounds.*

PX: *Part PNO is used in the enterprise and has weight WEIGHT pounds and GMWT grams (and GMWT = WEIGHT * 454).*

As for constraints, {PNO} is the sole key for both P and PX. Also, we obviously have:

```
CONSTRAINT ... PX = EXTEND P : { GMWT := WEIGHT * 454 } ;
CONSTRAINT ... IS_EMPTY ( PX WHERE GMWT ≠ WEIGHT * 454 ) ;
```

Now, updates on relvar P are straightforward. The compensatory actions are as follows:

```
ON DELETE d FROM P :
   DELETE ( PX MATCHING d ) FROM PX ;

ON INSERT i INTO P :
   INSERT ( EXTEND i : { GMWT := WEIGHT * 454 } ) INTO PX ;
```

Updates on PX are straightforward too:

```
ON DELETE d FROM PX :
   DELETE ( P MATCHING d ) FROM P ;

ON INSERT i INTO PX :
   INSERT i { PNO, WEIGHT ) INTO P ;
```

Observe in connection with this last rule that every tuple in *i* must have GMWT = WEIGHT * 454 (otherwise a **Golden Rule** violation will occur). To put the point another way, the two designs, one consisting of just relvar P and the other consisting of just relvar PX, are information equivalent; however, they are so only because this constraint—the constraint, that is, that every tuple in PX has GMWT = WEIGHT * 454—holds. Absent that constraint, we could insert a PX tuple with (say) WEIGHT = 12.0 and GMWT = 0.0, and that's an update on PX that has no counterpart on P.

Now consider a user who sees only relvar PX. That user:

1. Knows the predicate:

   PX: *Part PNO is used in the enterprise and has weight WEIGHT pounds and GMWT grams (and GMWT = WEIGHT * 454).*

2. Is aware of the fact that {PNO} is a key for this relvar, and is aware also of this constraint:

   ```
   CONSTRAINT ... IS_EMPTY ( PX WHERE GMWT ≠ WEIGHT * 454 ) ;
   ```

3. Is unaware of any compensatory actions (since this user is unaware of the fact that relvar P even exists).

And I think it's obvious that, from this user's perspective, all updates on PX work exactly as expected. Well, perhaps it isn't *completely* obvious; perhaps I need to say something about explicit UPDATE operations. The point is this: Such operations are (of course) required to comply with the constraint that GMWT = WEIGHT * 454. As a consequence, WEIGHT and GMWT must always be updated simultaneously, as in this example:

```
UPDATE PX WHERE PNO = 'P1' : { WEIGHT := 17.0 , GMWT := 7718.0 } ;
```

Aside: At this point we might perhaps get into a discussion of "virtual" or "computed" attributes. To be specific, we might say that attribute GMWT within relvar PX is such an attribute; and then we might say further that such attributes can't be directly updated and that it's the job of the DBMS to maintain them. Note, however, that (a) such attributes make at least as much sense in base relvars as they do in views, and—more important for present purposes—(b) the notion of computed attributes really has nothing to do with the topic of view updating anyway. (On the other hand, it certainly does have something to do with the various related notions of controlled redundancy, update propagation, multivariable constraints,[2] and compensatory actions. This state of affairs notwithstanding, however, for now I choose to invoke *The Principle of Cautious Design* and simply ignore the possibility of computed attributes.)[3] *End of aside.*

---

[2] Actually there aren't any multivariable constraints in the example at hand.

[3] *The Principle of Cautious Design* says: Given a design choice between options *A* and *B*, where *A* is upward compatible with *B* and the full implications of *B* aren't yet known, the cautious decision is to go with *A*. By way of example, consider SQL. The original designers of SQL had a choice between prohibiting duplicate rows (option *A*) and permitting them (option *B*). The cautious decision would have been to prohibit them (option *A*); they could then have been supported in the future, if a clear need for such support were ever demonstrated. Unfortunately, the designers chose to permit them (option *B*). Of course, this decision turned out to be a very bad one, but now there's no compatible way for SQL to fall back to option *A*.

Relation Constants

That's really all there is to say about EXTEND as such, but I'd like to use the foregoing EXTEND example to illustrate another general point. Suppose we define a *named relation constant* as follows:[4]

```
CONST WG INIT ( RELATION { TUPLE { WEIGHT 11.0 , GMWT 4994.0 } ,
                           TUPLE { WEIGHT 12.0 , GMWT 5448.0 } ,
                           TUPLE { WEIGHT 13.0 , GMWT 5902.0 } ,
                           TUPLE { WEIGHT 14.0 , GMWT 6356.0 } ,
                           TUPLE { WEIGHT 15.0 , GMWT 6810.0 } ,
                           TUPLE { WEIGHT 16.0 , GMWT 7264.0 } ,
                           TUPLE { WEIGHT 17.0 , GMWT 7718.0 } ,
                           TUPLE { WEIGHT 18.0 , GMWT 8172.0 } ,
                           TUPLE { WEIGHT 19.0 , GMWT 8626.0 } ,
                           TUPLE { WEIGHT 20.0 , GMWT 9080.0 } ;
```

The relation that's the value of this particular relation constant effectively acts as a conversion table from weights in pounds to weights in grams and vice versa, for all integer pound weights in the range 11 to 20 inclusive. Now suppose we compute the join JOIN {P,WG} of our usual parts relation and this relation constant. Then the result is exactly the relation I showed as the result of our original EXTEND example:

PNO	WEIGHT	GMWT
P1	12.0	5448.0
P2	17.0	7718.0
P3	17.0	7718.0
P4	14.0	6356.0
P5	12.0	5448.0
P6	19.0	8626.0

The example thus illustrates the point that, in a sense, JOIN and EXTEND are really the same operator. But that's not the issue I want to focus on here. Rather, I want to point out that the foregoing join is in fact—if you'll permit me to use the term—a one to many join (one tuple in WG joins to many tuples in P, loosely speaking),[5] and the rules I gave for updating such a join in Chapter 8 should therefore apply here. However, in that chapter, and indeed in the whole of this book prior to this point, I was tacitly assuming that views are always defined in terms of relation variables specifically, not relation constants, and hence that every relational operand to every relational operation was always at least potentially updatable. Clearly, to allow a constant to be updated would be absurd: a contradiction in terms, in fact. So the rules I've been proposing

[4] A named relation constant is like a view except that its value never changes (and so it's not actually a relvar as such). See *SQL and Relational Theory* for further explanation.

[5] Equivalently, of course, it's a many to one join—many tuples in P join to one tuple in WG.

for updating through various relational operations will need some refinement, or tweaking, in order to take relation constants properly into account. However, I don't expect any showstoppers here, and I don't intend to discuss the issue any further in this book. Suffice it for now to say that if you take the rules from Chapter 8 for updating a one to many join and ignore the cascades to the "one" side, what you're left with are essentially the rules I gave earlier in this section for updating through EXTEND.

ANOTHER SUMMARIZE EXAMPLE

Recall the following SUMMARIZE expression from Chapter 12:

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

Given our usual sample value for relvar SP as shown in, e.g., Fig. 1.1, this expression evaluates to the following:

SNO	TQY
S1	1300
S2	700
S3	200
S4	900

Now, one problem with this result is obvious: There's no tuple for supplier S5. Of course, there's no mystery here—the reason there's no tuple for S5 in the output is that there's no tuple for S5 in the input (viz., the current value of relvar SP). Intuitively, however, it would be nicer if there were a tuple for S5 in the output after all, as here:

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

This result is in fact what would be produced by the following expression, which makes use of what I referred to in Chapter 12 as the more general form of SUMMARIZE:

```
SUMMARIZE SP PER ( S { SNO } ) : { TQY  := SUM ( QTY ) }
```

As you can see, the BY specification in the first example has been replaced by a specification of the form PER (*r*), where *r* is a relation whose heading is (and in fact is required to be) some subset of that of the SUMMARIZE relation. (In the example, that SUMMARIZE relation is the relation that's the current value of relvar SP.) And relation *r* is the driver of the operation, in the sense that the output will contain exactly one tuple for each tuple in that relation *r*. In the example, therefore, there'll be five tuples in the output, because there are five tuples in the projection of relvar S on SNO; in particular, the output will contain a tuple for supplier S5—with, as you can see, a TQY value of zero—because there's a tuple for supplier S5 in that projection.[6] *Exercise:* Why exactly is the TQY value for supplier S5 here equal to zero?[7]

Here then is a definition (slightly simplified for present purposes) of this form of SUMMARIZE:

- **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 the expression SUMMARIZE *r1* PER (*r2*) : {*B* := *summary*} returns a relation with heading {*A1*,*A2*,...,*An*,*B*} and body the set 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 *summary* over all tuples of *r1* that have the same value for attributes *A1*, *A2*, ..., *An* as *t* does.

  Aside: Here repeated from Chapter 12 is a sample invocation of the simpler form of SUMMARIZE:

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

  This invocation is in fact defined to be equivalent to, and shorthand for, the following invocation of the more general form:

```
SUMMARIZE SP PER ( SP { SNO } ) : { TQY := SUM ( QTY ) } ;
```

  In other words, if the relation corresponding to *r2* in the definition is in fact a projection of that corresponding to *r1*—instead of merely having a heading that's equal to that of some projection of that latter relation, which the definition given above requires—we can replace the specification PER (*r2*) in the SUMMARIZE invocation by the specification BY {*A1*,*A2*,...,*An*}, where *A1*, *A2*,..., *An* are all of the attributes of *r2*. *End of aside*.

---

[6] As with the simpler form of SUMMARIZE discussed in Chapter 12, this SUMMARIZE invocation is in fact logically equivalent to a certain EXTEND invocation—to be specific, the invocation EXTEND S{SNO}:{TQY := SUM(!!SP,QTY)}, where !!SP denotes a certain image relation—and there are reasons to prefer this EXTEND formulation over its SUMMARIZE equivalent. Once again, however, detailed discussion of such matters is beyond the scope of this book; refer to *SQL and Relational Theory* for further explanation.

[7] *Answer:* Because, as *SQL and Relational Theory* explains, the sum of no summands is correctly zero (*pace* SQL, which thinks it should be null).

So now let's assume once again that S and SP are base relvars as usual, and let's define a view STQ as indicated by the following constraint:

```
CONSTRAINT ... STQ = SUMMARIZE SP PER ( S { SNO } ) :
                                      { TQY := SUM ( QTY ) } ;
```

The predicate for STQ is as follows:[8]

STQ: *Supplier SNO supplies parts in total quantity TQY.*

As with the SUMMARIZE example in the previous chapter, there's no information equivalence here. That's why defining STQ as a view of S and SP makes sense, but defining S and SP as views of STQ certainly doesn't (in fact, of course, it can't be done, at least not in general).

As for constraints, {SNO,PNO} is the sole key for SP and {SNO} is the sole key for S and STQ. Also, we obviously have:

```
CONSTRAINT ... STQ { SNO } = S { SNO } ;
```

Of course, this constraint is a logical consequence of the way STQ is defined. More important, note how it differs from the corresponding constraint in the SUMMARIZE example in the previous chapter: There, every supplier number in STQ also appeared in the shipments relvar SP; here, every supplier number in STQ also appears in the suppliers relvar S.

Now let's think about updates and compensatory actions. First let's consider relvar S. Suppose we insert a tuple for a new supplier, say supplier S6, into that relvar. Without loss of generality, we can assume that this update on relvar S is accompanied—i.e., as part of the same overall update operation—by an update on relvar SP that inserts a set of shipment tuples for that new supplier as well. Of course, if no such shipment tuples are in fact inserted, it just means the set of shipment tuples for the new supplier is an empty set. Either way, we'll want to compute the sum of the quantities in that set of shipment tuples so that we can insert an appropriate tuple into STQ (if that set is empty, the sum will simply be zero). So we have the following rule:

```
ON INSERT i INTO S :
   WITH ( t1 := SP MATCHING i ,
        ( t2 := SUMMARIZE t1 PER ( S { SNO } ) :
                                      { TQY := SUM ( QTY ) } ) :
   INSERT t2 INTO STQ ;
```

By contrast, deletes on S are more straightforward:

---

[8] This predicate is the same as the predicate in the SUMMARIZE example in the previous chapter (but the constraints are different, as we'll see in a moment). Strictly speaking, however, the predicate in the previous chapter ought to be extended as follows: *Supplier SNO* ***supplies at least one part and*** *supplies parts in total quantity TQY.*

```
ON DELETE d FROM S :
   DELETE ( STQ MATCHING d ) FROM STQ ;
```

(Of course, the delete will fail if it doesn't additionally cascade to SP and there are some tuples in SP matching the tuples being deleted from S. If it does cascade to SP, the rule governing deletes on SP will come into play—see below—but that rule will have no effect.)

Turning now to updates on relvar SP, the rules are very similar to their counterparts in the previous chapter, as I hope you'd be expecting:

```
ON INSERT i INTO SP :
   WITH ( t1 := STQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := SUMMARIZE t2 PER ( S { SNO } ) :
                                  { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;

ON DELETE d FROM SP :
   WITH ( t1 := STQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := SUMMARIZE t2 PER ( S { SNO } ) :
                                  { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

And now what about updates on STQ? Well, once again deletes are easy:[9]

```
ON DELETE d FROM STQ :
   DELETE ( S MATCHING d ) FROM S ;
```

However, inserts make no sense, in general;[10] for example, what could it possibly mean to insert the tuple (S6,800) into STQ? So there's no rule for inserts. Of course, the fact that there's no rule doesn't mean we can't do inserts on STQ (indeed, the update rules for SP explicitly call for such actions)—it only means that such operations are likely to fail on a **Golden Rule** violation, except possibly if they're part of some multiple assignment (as indeed they are, in the case of the rules for updates on SP).

[9] Except that once again the delete will fail if it doesn't additionally cascade to SP and there are some tuples in SP matching the ones being deleted from STQ.

[10] Except as noted in the footnote at the very end of the previous chapter.

Chapter 14

Updating through Expressions

When expressions conflict
And views contradict
The results that emerge
These rules will predict

—Anon.: *Where Bugs Go*

Throughout the book prior to this point, I've been concerned primarily with the question of updating through some individual relational operation (updating a restriction, updating a projection, and so on). What's more, I've been assuming, more or less tacitly, that the rules for updating a relvar defined by means of some more complicated expression can be determined by combining the rules for the operations involved in that expression—for example, updating a union of two joins can be done by first applying the rules for updating a union and then applying the rules for updating joins. Now it's time to take a closer look at that assumption.

I observe first of all that the assumption is surely correct in principle. The reason is that the alternative is just too horrible to contemplate! To spell it out, the alternative in question would mean treating every expression as a special case—i.e., defining one set of rules for updating the union of two joins, and another for updating the difference of two joins, and another for updating the join of a union and a difference, and so on and so forth ad infinitum.

To say it again, the assumption is surely correct. However, it does raise certain questions, questions to which the answers don't always seem entirely clear (at least, not to me, and not at this time). Thus, I have two aims in this chapter: First, I want to explain and discuss some of those questions; second, I want to draw, or attempt to draw, certain conclusions from those discussions. Do please note, however, that the chapter is unavoidably somewhat speculative in nature. I freely admit I don't have all the answers at this time.

SEMANTICS NOT SYNTAX (?)

Consider the following example ("Example 1"). Starting with our usual suppliers relvar S, suppose we define views V1 and V2 as follows:

```
VAR V1 VIRTUAL ( S WHERE CITY = 'London' OR CITY = 'Paris' ) ;

VAR V2 VIRTUAL ( ( S WHERE CITY = 'London' )
                   UNION
                 ( S WHERE CITY = 'Paris'  ) ) ;
```

Now, it's intuitively obvious that these two views are semantically equivalent, even though their definitions—more precisely, their defining expressions—are syntactically distinct. What I mean by this observation is that (to spell the point out) it's clearly the case that at any given time, those two expressions both denote the same relation. Thus, it's also intuitively obvious that (a) if *Q* is a query on either V1 or V2, then *Q* is defined for the other view as well (and it produces the same result on both), and (b) what's more, an analogous remark applies to updates also. But I want to take a closer look at part (b) of this claim in particular.

First, then, let's think about INSERT operations. Suppose we try to insert a tuple for supplier S9 with city London. Then:

- In the case of V1, the rules for inserting through a restriction come into play (see Chapter 4), and the net effect is that the new tuple is inserted into relvar S and thus into V1 as well.

- In the case of V2, the rules for inserting through a union come into play (see Chapter 10). Since the new tuple satisfies the restriction condition for one union operand—viz., the one denoted by the expression S WHERE CITY = 'London'—and not for the other, it's inserted into that operand and not into the other. The rules for inserting through a restriction then come into play again, and the net effect is that the new tuple is inserted into relvar S and thus into V2 as well.

Second, DELETE operations. Suppose we try to delete the tuple for supplier S1. Then:

- In the case of V1, the rules for deleting through a restriction come into play (see Chapter 4), and the net effect is that the specified tuple is deleted from relvar S and thus from V1 as well.

- In the case of V2, the rules for deleting through a union come into play (see Chapter 10). Since the specified tuple appears in one union operand—viz., the one denoted by the expression S WHERE CITY = 'London'—and not in the other, it's deleted from that operand. The rules for deleting through a restriction then come into play again, and the net effect is that the specified tuple is deleted from relvar S and thus from V2 as well.

So the claims I made earlier do seem to hold up, at least with respect to this first example. And it's tempting to conclude that, more generally, if views *V1* and *V2* are equivalent in the sense illustrated by that example—i.e., if their defining expressions are such that at any given time they both denote the same relation—then if *U* is an update on either *V1* or *V2*, it must be the

case that *U* is defined for the other one as well, and it has the same effect on both. Indeed, I'm on record[1] as making a series of rather dogmatic assertions on such matters, all of them along the following lines:

> ***The "Semantics Not Syntax Principle":*** The semantics of view updating should not depend on the particular syntactic form in which the view definition in question happens to be expressed.

Note: Before I try to elaborate on this "principle"—if principle it truly is—I need to clarify a couple of points. First, with respect to queries, what I said before, in essence, was this: If *Q* is a query on *V1*, then that same query *Q* is defined for *V2* as well, and it produces the same result on both. But *Q* obviously can't quite be "the same query" for both views, because of course the views have different names. Thus, for example, given views V1 and V2 as defined above for Example 1, if I claim that the query

```
V WHERE STATUS > 15
```

is defined for both V1 and V2, what I mean is that the symbol *V* can sensibly be replaced in that query by either V1 or V2. That's what I meant when I said the expression *Q* can be said to represent "the same query" for both views. And given our usual sample values, that query will of course return the same result in both cases:

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

And when I use the phrase "the same update," of course, I mean it to be understood in the same kind of way.

The second point I need to clarify has to do with what it means, precisely, for two views to be "equivalent." Of course, I've appealed to an intuitive understanding of this notion a couple of times already in this discussion, first in my investigation into Example 1, and then in my attempt to draw a general conclusion from that investigation. But the question is: Is that intuitive understanding sufficient? As you've probably guessed already, the answer to this question is *no*. In fact, much of the rest of this chapter consists of an attempt to pin down much more precisely just what such a notion—i.e., of equivalence of views—might really mean.

Back, then, to the "semantics not syntax principle." As I've indicated, I assumed for a long time that this "principle" obviously made sense. (After all, it certainly makes sense in another

[1] E.g., in "View Updating," Appendix E of *Databases, Types, and the Relational Model: The Third Manifesto*, by Hugh Darwen and myself (3rd edition, Addison-Wesley, 2006) and elsewhere.

context. I refer here to the point, explained in Chapter 2, that compensatory actions aren't driven by the arbitrary choice of syntax in which the pertinent update happens to have been formulated.) More recently, however, I began to wonder whether it might perhaps not be valid after all, or at best only partly valid. The reasons for this shift in attitude on my part are explained in the next couple of sections, but what they boil down to is this: It's all too easy to find examples where—on the face of it, at least—such a principle simply doesn't seem to hold up. What's more, the machinations that one has to go through in attempts to make it hold up after all get increasingly baroque (epicycles upon epicycles, as it were). All of which makes the following quote from Enrico Bombieri[2] (one of my favorite quotes, incidentally) increasingly germane: "When things get too complicated, it sometimes makes sense to stop and wonder: Have I asked the right question?" *Now read on ...*

SOME WELL KNOWN TAUTOLOGIES

In logic, a tautology is something that's unconditionally true; more precisely, it's a predicate whose every possible invocation is guaranteed to yield TRUE, regardless of what arguments are substituted for its parameters if any. For example, *p* OR NOT *p*, where *p* is an arbitrary proposition, is a tautology, and so is 1+1 = 2. (This latter is in fact a proposition, of course, and it doesn't actually have any parameters.)

Now, tautologies of the form *exp1* ≡ *exp* (where *exp1* and *exp2* are arbitrary expressions and the symbol "≡" denotes logical equivalence)[3] are particularly important, because they allow an expression containing an occurrence of *exp1* to be rewritten as one containing an occurrence of *exp2* instead. To spell the point out, let *X1* be an expression containing an occurrence of *exp1* as a subexpression; let *exp2* be logically equivalent to *exp1*; and let *X2* be the expression obtained from *X1* by substituting an occurrence of *exp2* for the occurrence of *exp1* in question. Then *X1* and *X2* are logically equivalent in turn; hence, *X1* can be rewritten as *X2*.

By way of illustration, I remind you of Example 1 from the previous section. In that example, I was effectively appealing to the fact that the expression

```
( S WHERE CITY = 'London' OR CITY = 'Paris' )
  ≡
( ( S WHERE CITY = 'London' ) UNION ( S WHERE CITY = 'Paris' ) )
```

is a tautology. *Note:* Of course, I'm sure you're familiar with all of these ideas, since they form the basis of one of the important things that relational optimizers are supposed to do: namely, *expression transformation*, sometimes known as "query rewrite." See *SQL and Relational Theory* for further discussion.

---

[2] Fields Medalist and IBM von Neumann professor of mathematics at the Princeton Institute for Advanced Study, New Jersey.

[3] Expressions *exp1* and *exp2* are logically equivalent if and only if each can be derived from the other in accordance with the rules of inference of the logical system in effect.

Now, set theory in general, and as a consequence relational algebra in particular, both include many interesting tautologies of the form discussed above (viz., *exp1* ≡ *exp*). Here are a few relational examples. Let *A* and *B* denote arbitrary relational expressions of the same relation type. (In set theory, they would denote arbitrary sets.) Then we have:

```
A  ≡  A INTERSECT ( A UNION B )

A  ≡  A UNION ( A INTERSECT B )

A INTERSECT B  ≡  A MINUS ( A MINUS B )

A INTERSECT B  ≡  B MINUS ( B MINUS A )

A MINUS B  ≡  A MINUS ( A INTERSECT B )

A UNION B  ≡  ( A MINUS B ) UNION ( A INTERSECT B ) UNION ( B MINUS A )

A MINUS A  ≡  B MINUS B
```

Now let's consider some implications of the foregoing ideas for the issue of view updating specifically. Consider the following example ("Example 2"), involving relvars PL and PK from Chapters 9, 10, and 11. Just to remind you, those relvars both have just a single attribute PNO (part number), and thus are certainly of the same relation type; relvar PL gives part numbers for parts on sale, and relvar PK gives part numbers for parts in stock. The predicates are thus both very simple:

PL: *Part PNO is on sale.*

PK: *Part PNO is in stock.*

Sample values are shown in Fig. 14.1 (which is extracted from various figures in Chapters 9, 10, and 11).

PL

PNO
P1
P2
P3
P6

PK

PNO
P2
P5

Fig. 14.1: Relvars PL and PK—sample values

Now consider two views, VPL1 and VPL2, whose defining expressions *exp1* and *exp2* are as indicated here (recall from Chapter 3 that the symbol "≝" means "is defined as"):

```
exp1  def=  PL                                  /* defining view VPL1 */

exp2  def=  PL INTERSECT ( PL UNION PK )        /* defining view VPL2 */
```

Observe that the defining expressions *exp1* and *exp2* here are logically equivalent (i.e., *exp1* ≡ *exp2* is a tautology—as a matter of fact, it's a specific illustration of the first tautology in the list I gave above). At any given time, therefore, VPL1 and VPL2 certainly both have the same value; in fact, they both have the same value as the underlying relvar PL does at the time in question. However, now consider the following delete operations:

```
DELETE ( P2 ) FROM VPL1 ;

DELETE ( P2 ) FROM VPL2 ;
```

Of course, these two deletes both have the same effect on the target view as such. As you can easily confirm, however, the first causes the P2 tuple to be deleted from relvar PL and has no effect on relvar PK; the second, by contrast, causes that same tuple to be deleted from relvar PL and—at least according to the rules given in Chapters 9 and 10 for deleting through intersections and unions—*also* causes that same tuple to be deleted from relvar PK. So here we apparently have a case where "the same" update on "equivalent" views certainly doesn't have the same effect (at least, not on the underlying relvars, although as I've said it does have the same effect on the views as such).

Now, it's true that to a user who sees view VPL1 or VPL2 (either one) and the underlying relvars PL and PK as well, the effect of the foregoing deletes on those underlying relvars is at least explicable—assuming the user in question is also aware of the pertinent compensatory actions, of course. But the fact remains that the two deletes do have different effects overall. And the obvious questions remain, too: viz., *why* do those deletes have different effects overall? And what are the implications of the fact that they do?

Before I try to answer these questions, consider these insert operations on VPL1 and VPL2:

```
INSERT ( P7 ) INTO VPL1 ;

INSERT ( P7 ) INTO VPL2 ;
```

Again the effect on the target view is the same in both cases. However, it's easy to see that the first insert causes a tuple for part P7 to be inserted into relvar PL and has no effect on relvar PK, while the second causes that same tuple to be inserted into relvar PL and—again, at least according to the rules given in Chapters 9 and 10 for inserting through intersections and unions—also causes that same tuple to be inserted into relvar PK. Again, then, it seems we have a case where "the same" update on "equivalent" views has different effects, at least on the underlying relvars.

To all of the above, let me add that the problems are compounded by the fact that *no possible update* on VPL1 has the same effect on the underlying relvars—on relvar PK in particular—as the foregoing delete or insert on VPL2 does.

Clearly, then, there's something wrong with the "semantics not syntax principle." Before I try to pin down what it might be, however, I want to consider another related issue.

"SEMANTIC TRANSFORMATIONS"

In *SQL and Relational Theory*, I briefly describe an implementation technique called *semantic optimization*. Here's a simple example, repeated from that earlier book. 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 pertinent target key in the referenced relvar, S. Thus, every SP tuple does join to some S tuple, and every SP tuple therefore does contribute a part number to the projection operation that produces the overall result. So there's no need to do the join!—the expression can be reduced for evaluation purposes 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 shown therefore won't be valid. But in the case at hand, every SP tuple necessarily does have a counterpart in S, because a constraint—actually a foreign key constraint—is in effect that says that every shipment must have a supplier, and so the transformation is valid after all. *Terminology:* A transformation that's valid only because a certain constraint is in effect is called a semantic transformation, and the resulting optimization is called a semantic optimization.

Given the foregoing, now consider the following example ("Example 3"). Suppose for simplicity that relvar S has just one attribute, SNO, and relvar SP has just two attributes, SNO and PNO (so both relvars are in fact "all key"). Fig. 14.2 shows some sample values.

S

SNO
S1
S2
S5

SP

SNO	PNO
S1	P1
S1	P2
S1	P3
S2	P2

Fig. 14.2: Relvars S and SP—sample values

Now consider two views, VSP1 and VSP2, whose defining expressions *exp1* and *exp2* are as indicated here:

```
exp1  def=  SP                /* defining view VSP1 */

exp2  def=  S JOIN SP         /* defining view VSP2 */
```

Observe here that (per the foregoing discussion) *exp2* can be "semantically transformed" into *exp1*.[4] It follows that, at any given time, VSP1 and VSP2 both have the same value; in fact, they both have the same value as the relvar SP does at the time in question. However, now consider the following deletes:

```
DELETE ( S2 , P2 ) FROM VSP1 ;

DELETE ( S2 , P2 ) FROM VSP2 ;
```

Once again the two deletes both have the same effect on the views as such. But it's easy to see that the first causes the tuple (S2,P2) to be deleted from relvar SP and has no effect on relvar S, while the second causes that same tuple to be deleted from relvar SP and—at least according to the rules given in Chapter 8 for deleting through a one to many join—also causes the tuple for supplier S2 to be deleted from relvar S.[5]

Next, consider the following inserts:

```
INSERT ( S4 , P1 ) INTO VSP1 ;

INSERT ( S4 , P1 ) INTO VSP2 ;
```

The first of these inserts fails on a **Golden Rule** violation (specifically, a violation of the foreign key constraint from SP to S). The second, however, causes the specified tuple to be inserted into relvar SP and—again, at least according to the rules given in Chapter 8 for inserting through a one to many join—also causes a tuple for supplier S4 to be inserted into relvar S. So this time, not only do the updates have different effects on the underlying relvars, they also have different effects on the views as such! In other words, to spell the point out, here we have a situation in which "the same" update doesn't even have the same effect on its target relvars, even though those target relvars do happen to be "equivalent."

To all of the above let me add that the problems are compounded by (a) the fact that no possible update on VSP1 has the same effect on the underlying suppliers relvar S as the

---

[4] Or the other way around, of course, a state of affairs that I think bolsters the argument that follows.

[5] It's true that the join of S and SP loses information (unless every supplier is required to supply at least one part), and you might therefore be thinking that the culprit here is the pragma involved in updating through such a join. But it's not. Rather, the problem seems to be intrinsic. In fact, I'd like to elaborate briefly on that pragma issue. We've seen that deleting the tuple (S2,P2) from views VSP1 and VSP2 has different effects on relvar S, depending on which view the delete is aimed at. But at least that delete has the desired effect on the view as such in both cases. By contrast, the position espoused by certain critics, according to which deletes on such views like VSP2 must be rejected entirely (see Chapter 6), would cause the delete on view VSP1 to succeed but the one on view VSP2 to fail! In my opinion, therefore, that position involves just as much pragma as, and has no more claim to "respectability" than, the approach described in Chapter 6 does.

foregoing delete on VSP2 does and (b) the fact that no possible update on VSP1 has the same effect on either S or VSP1 as the foregoing insert on VSP2 does. What's more, no possible update on VSP2 produces the same **Golden Rule** violation on VSP2 as the foregoing (attempted) insert does on VSP1.

Clearly there's more going on here than meets the eye. We need to take a closer look.

INFORMATION EQUIVALENCE REVISITED

To review briefly (and abstracting somewhat):

- In Example 2, we had a situation in which updates on a view whose definition took the form *A* INTERSECT (*A* UNION *B*) affected *B*, even though that defining expression is logically equivalent to just *A*.

- In Example 3, we had a situation in which updates on a view whose definition took the form *A* JOIN *B* again affected *B*, even though that defining expression (in that particular example) could be "semantically transformed" into just *A*.

So the obvious question in both cases is: Why is *B* affected at all?

These questions, and others like them, are very vexing. But now let me remind you of that quote from Enrico Bombieri: "When things get too complicated, it sometimes makes sense to stop and wonder: Have I asked the right question?" Maybe "Why is *B* affected at all?" *isn't* the right question to ask. Maybe the right question to ask is: What exactly does it mean for two views—or two view defining expressions, rather—to be equivalent, anyway?[6]

Well, I think the key to this latter question lies in a careful examination of a concept I've been appealing to throughout this book: viz., the notion of information equivalence. Here repeated from Chapter 3 is the definition I originally gave for that concept:

- **Definition:** Let *DB1* and *DB2* be sets of relvars. Then *DB1* and *DB2* are information equivalent if and only if the constraints that apply to *DB1* and *DB2* are such that every proposition that can be represented by *DB1* can be represented by *DB2* and vice versa.

Now, given the foregoing definition, it would appear with respect to Example 2—at least at first sight—that a design consisting of just VPL1 and one consisting of just VPL2 are indeed information equivalent, given that the values of those two relvars are certainly equal at any given time and thus apparently represent the same set of propositions at the time in question. Similarly, it would appear with respect to Example 3 that a design consisting of just VSP1 and

---

[6] Acknowledgments to David McGoveran once again for setting me on the right track in this section.

one consisting of just VSP2 are also information equivalent, given that the values of those relvars are also equal at any given time. But are these things really so?

For definiteness, let's focus until further notice on Example 2 and views VPL1 and VPL2. Here again are the view defining expressions:

```
exp1  def=  PL                                  /* defining view VPL1 */

exp2  def=  PL INTERSECT ( PL UNION PK )        /* defining view VPL2 */
```

Now, it's true that *exp1* and *exp2* here are logically equivalent (each can be logically derived from the other). But they aren't *informationally* equivalent! To be specific, *exp2* tells us something that *exp1* doesn't—it tells us that relvar PK exists, which *exp1* certainly doesn't.[7] By the same token, consider the tuple consisting of just the PNO value P1, which appears in both relvars. The appearance of that tuple in relvar VPL1 represents the following proposition:

Part P1 is on sale.

By contrast, the appearance of that same tuple in relvar VPL2 denotes the following proposition:

Part P1 is on sale and (part P1 is on sale or part P1 is in stock).

And just as the expressions *exp1* and *exp2* are logically but not informationally equivalent, so these two propositions too are logically but not informationally equivalent—i.e., they don't convey the same information. At the very least, the second proposition tells us it could possibly be the case that part P1 is in stock (a fact about the real world that isn't even mentioned in the first proposition); likewise, it also tells us that it must be the case that part P1 is either on sale or in stock or both, another fact about the real world that isn't mentioned in the first proposition.

With the foregoing by way of motivation, I propose the following definition for what it means for two relational expressions to be "informationally equivalent" (except that now I revert to our more usual phrase "*information* equivalent"):

- **Definition:** Relational expressions *exp1* and *exp2* are information equivalent if and only if (a) they're logically equivalent—i.e., each can be derived from the other by means of the system's rules of inference—and (b) every relvar mentioned in *exp1* is also mentioned in *exp2* and vice versa.

Clearly, the defining expressions for views VPL1 and VPL2, although they're logically equivalent, aren't information equivalent by this definition. Now, our original definition of information equivalence (i.e., for sets of relvars) talked in terms of constraints, not relational

[7] It's germane to point out here that relvar names can be thought of, conceptually, as nothing more than shorthand for the relevant predicates.

expressions—it said two sets of relvars are information equivalent if and only if the pertinent constraints are such that every proposition that can be represented by either set can also be represented by the other. But as we know from numerous discussions and examples earlier in the book, there's a one to one correspondence between constraints and relational expressions: For every relational expression *rx*, a constraint whose boolean expression component is IS_EMPTY(*rx*) can be defined; at the same time, every constraint can always be formulated in terms of such an IS_EMPTY invocation, and so for every constraint there's a corresponding relational expression. In particular, therefore, as we also know, every view defining expression implies a certain constraint. So we can surely say two such constraints meet the requirements of that original definition (i.e., of information equivalence) if and only if the applicable view defining expressions are information equivalent in the sense just defined.

> *A remark on terminology:* The relvar names PL and PK in the view defining expressions for VPL1 and VPL2 in Example 2 are serving as what logicians call *designators*—when the expressions containing them are evaluated, they effectively designate a specific value, viz., the value of the pertinent relvar at the time in question. By the way, note the logical difference between a designator and a parameter. A parameter can be replaced by any argument whatsoever, just so long as it's of the right type. A designator, by contrast, can't be (and therefore isn't) 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 (namely, the time when the containing expression is evaluated). *End of remark.*

Now let's extend these ideas to predicates and propositions. The predicate for VPL1 can be stated symbolically, and slightly more formally, as follows (I'll use the symbol "⇔" to mean "if and only if"):

```
t ∈ VPL1  ⇔  t ∈ PL
```

("tuple *t* appears in VPL1 if and only if it appears in PL"). Likewise, the predicate for VPL2 can be stated as follows:

```
t ∈ VPL2  ⇔  t ∈ PL AND ( t ∈ PL OR t ∈ PK )
```

("tuple *t* appears in VPL2 if and only if it appears in PL and also in either PL or PK"). Now, the point of these reformulations is merely to show that the predicates too can be thought of as containing references to the pertinent relvars—and given that they do, we can apply the notion of information equivalence to them also. In the case at hand, of course, we can see that the predicates we're talking about here are, again, not information equivalent by the proposed definition. What's more, it seems reasonable to say that if predicates *P1* and *P2* are information equivalent, then for every proposition that can be obtained by instantiation from *P1*, there's an

information equivalent proposition that can be obtained by instantiation from *P2* (and vice versa, of course).

Given all of the above, then, it seems to me to make sense to say, with respect to Example 2 specifically, that, appearances to the contrary notwithstanding, a design consisting of just VPL1 and one consisting of just VPL2 aren't information equivalent after all. And a similar remark applies to Example 3 also: The definition of VSP2 mentions relvar S, which the definition of VSP1 doesn't, and so a design consisting of just VSP1 and one consisting of just VSP2 aren't information equivalent. To be more specific, the definition of VSP2 tells us the suppliers relvar exists, which the definition of VSP1 doesn't.[8]

To get back to Example 2: As a consequence of the foregoing considerations, it isn't really even true to say the result of evaluating any given query *Q* on VPL1 at time *T* and the result of evaluating that same query *Q* on VPL2 at that same time *T* will always be identical. To be specific, the result will of course always be the same relation *r* in both cases—but the set of tuples in that relation *r* will represent different sets of propositions in the two cases. A fortiori, therefore, it seems to me entirely reasonable that a given update *U* on VPL1 at time *T* and the same update *U* on VPL2 at that same time *T* might sometimes have different effects. More particularly, I think it's perfectly reasonable to say updates on VPL1 never have any effect on relvar PK, while updates on VPL2, by contrast, sometimes do have an effect on that relvar. I even think it's reasonable in general that the same update might have different effects on two views (on the views as such, I mean), if the corresponding defining expressions are logically equivalent but not information equivalent. In fact, exactly such a situation arose in connection with Example 3, as you'll surely recall.

The net of all this is as follows: The definition of what it means for two sets of relvars to be information equivalent—see page 209—is adequate as it stands. Rather, what needs some refinement is our understanding of what it means for two propositions to be "the same proposition." I've proposed that they be considered the same if and only if they're information equivalent, and I've proposed a definition for this latter concept: viz., they need to be corresponding instantiations of predicates that are information equivalent in turn.

There is, however, still an open question. I've proposed that two views *V1* and *V2* not be regarded as interchangeable if their defining expressions aren't information equivalent. But the question remains: Are they necessarily interchangeable if their defining expressions *are* information equivalent? That is, is it possible for an update on *V1* and "the same" update on *V2* to have different effects, even if the defining expressions for *V1* and *V2* are information equivalent as defined above? This is a question that I believe deserves further study.

---

[8] By contrast, Example 1 was just fine—information equivalence did hold—and no analogous remark applies.

CONCLUDING REMARKS

I'd like to close this chapter by offering some observations that show that at least there's some precedent for the idea that just because expressions *exp1* and *exp2* appear to be equivalent in certain respects, it doesn't follow that they can be used completely interchangeably. Note, however, that these remarks are something of a digression from our main topic, and you can skip them if you like.

First of all, I want to make a general point: viz., I strongly suspect that what we're dealing with here—viz., the issue of updating through "equivalent" expressions—is one of those situation in which the logical difference between values and variables is significant. To be more specific, there's a logical difference between an expression and a pseudovariable reference:

> *An expression denotes a value, a pseudovariable reference doesn't.*

(See the very end of Chapter 2 if you need to refresh your memory regarding pseudovariables in general, and recall from Chapter 3 that views act as pseudovariables as far as updates are concerned.) In other words, an expression and a pseudovariable reference have very different semantics, even if they're syntactically identical.

Next, I can recall several occasions in the past where thinking carefully about the logical difference between values and variables led to an epiphany (or what I certainly thought of at the time as an epiphany, anyway). One case in point has to do with the model of type inheritance Hugh Darwen and I were developing, or trying to develop, as a logical consequence of the type theory aspects of *The Third Manifesto*. We were struggling with the concept of what's known in the world of object orientation (OO) as *substitutability*, which OO writings typically explain thus:

> If *S* is a subtype of *T*, then wherever the system expects an object of type *T*, an object of type *S* can be substituted.[9]

For example, if SQUARE is a subtype of RECTANGLE, then it should be possible to substitute squares for rectangles in the foregoing kind of way, because squares *are* rectangles. Taken at face value, however, this concept seems to give rise to all kinds of absurdities, including squares that aren't square, circles that aren't circular, and other such solecisms (contradictions in terms, really).[10] It was only when we realized that the OO term *object* sometimes meant a variable and sometimes a value that we also realized that what we needed to do was distinguish between variable substitutability and value substitutability—and it was making that distinction that enabled us to avoid those OO solecisms. To be more specific, it was making that distinction that led us to realize that certain expressions, though they might be

---

[9] This definition, such as it is, is paraphrased from Wikipedia (*en.wikipedia.org*).

[10] This criticism applies to SQL, incidentally.

interchanged with complete freedom in read-only contexts, could be interchanged only conditionally in update contexts.

My third point also has to do with our inheritance model but is perhaps more obviously related to the question of exactly what it means for two expressions to be equivalent. One of the things I mentioned in the section "Some Well Known Tautologies" was that the expressions *A* INTERSECT *B* and *A* MINUS (*A* MINUS *B*) are equivalent—and so indeed they are, inasmuch as they certainly always have the same value. However, it turns out that, first, all expressions have what the *Manifesto* calls a *declared type*; second, the declared type of *A* INTERSECT *B* isn't the same, in general, as the declared type of *A* MINUS (*A* MINUS *B*)! To be specific, the declared type of *A* MINUS (*A* MINUS *B*) is the same as that of *A*, but the declared type of *A* INTERSECT *B* is the "least specific common subtype" of the declared types of *A* and *B*, which is, in general, some proper subtype of that of *A*. Once again, therefore, it doesn't necessarily follow that just because two expressions always have the same value, they can be used interchangeably. In the case at hand, in fact, the two expressions, though they do always have the same value, aren't even interchangeable for read-only purposes!—at least, not 100 percent.[11]

[11] Incidentally, this state of affairs tends to suggest that (at least in a context where type inheritance is relevant) the definition of what it means for two relational expressions to be information equivalent might need to be extended to require the expressions in question to be of the same declared type. See *Database Explorations: Essays on The Third Manifesto and Related Topics,* by Hugh Darwen and myself (Trafford, 2010), for further explanation.

Chapter 15

Ambiguity Revisited

Seal up the mouth of outrage for a while,
Till we can clear these ambiguities
—William Shakespeare: *Romeo and Juliet* (1592)

As mentioned at several points in earlier chapters, David McGoveran has a proposal for resolving the ambiguities that can arise in connection with view updating (in the absence of information equivalence, that is). In this final chapter, I want to say something about that proposal. Before I do, however, there are some points I must make absolutely clear:

- To say only that his proposal resolves the ambiguities doesn't begin to do justice to David's scheme overall. Rather, that scheme is (of course) a scheme for addressing the database updating issue in its entirety. It begins by recognizing that, as I explained in Chapter 2, the only true variable—the only thing that can really be updated, logically speaking—is the entire database (or dbvar, rather, though it doesn't use that term). The database, and the relvars within that database, are characterized by predicates; however, those predicates, unlike the ones I've been talking about in this book, are completely formal in nature. In particular, they completely subsume what in Chapter 2 I called the total database constraint. Update requests too are expressed in terms of such predicates (they might refer to relvars by name for user convenience, but such references serve merely as shorthand for the pertinent relvar predicates). Implementing an update request then consists in combining the database predicates and update predicates appropriately and computing the relation or relations that together satisfy such combined predicates; those relations then effectively constitute the updated portion of the database.[1] For further details, I refer you to David's patents, which were cited in the "Concluding Remarks" section of Chapter 3.

- I should also make it clear that David wouldn't even agree that what I've been calling ambiguities are ambiguities anyway. In effect, that is, he believes a disambiguating mechanism along the lines of the one I'll be presenting in this chapter ought always to be employed. (Of course, if it were, then there would never be any need to appeal to the

[1] One consequence of this approach is that **Golden Rule** violations never occur! If *U* is an update request that would normally violate some constraint, the combined predicate for *U*, since it effectively includes all pertinent constraints, simply denotes an empty set, and so no updating is actually done. By way of example, consider an explicit UPDATE request that attempts to change the CITY value to London for some tuple in relvar NLS ("non London suppliers"). In effect, that UPDATE request is modified to say UPDATE NLS WHERE CITY = 'London'—and, of course, no tuple in NLS ever satisfies this WHERE clause.

various pieces of pragma I've described in earlier chapters.) In other words, not to use that disambiguating mechanism is simply to underspecify the update request in question—in which case it's rather hard to say just what an appropriate response should be on the part of the DBMS (all possible responses become equally "correct," in a sense).

How does the foregoing relate to the approach to updating I've been discussing in the present book? Well, I do believe that approach is generally in the spirit of David's proposal, at least to the extent that I understand that proposal. However, I've tried to present the approach in more familiar and more concrete terms—terms that I think make sense in the context of relational systems as generally understood (in other words, relational systems as I've tried to describe them in numerous previous books and other writings). I've also tried to present the approach in such a way as to make a path to possible implementation in such systems comparatively clear.

Without further ado, let me now explain the "disambiguating mechanism" (or one possible disambiguating mechanism, at any rate)—that is, a concrete scheme for resolving ambiguities that's based on the more abstract proposals described in David's patents.

PREDICATES AND CONSTRAINTS REVISITED

I'd like to begin with a quick review of predicates and constraints as I've been using those terms in this book. Recall from Chapter 2 that every relvar has an associated relvar predicate, which is, loosely, what the relvar in question means to the user. 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.

For brevity, let's refer to this predicate as *pred*(S); more generally, in fact, let's refer to the predicate for any given relvar *R* as *pred*(*R*).[2] Then each tuple in *R* at any given time *T* is supposed to be such that the attribute values from that tuple, when substituted for the parameters in *pred*(*R*), yield a proposition—more specifically, a proposition that's an instantiation of *pred*(*R*)—that we believe to be true at that time *T*. For example, the fact that the tuple (S1,Smith,20,London) appears in relvar S right now means we believe the following proposition to be true at this time:

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

---

[2] I adopt this notation, or naming convention, purely for expository purposes. In practice I hope users would be able to choose their own more "user friendly" predicate names if they want to (and I'll do exactly that myself in later examples).

Now, in order to be able to use relvar *R* in any meaningful way at all, the user must be aware of (and of course understand!) that predicate *pred*(*R*). It follows that the "owner" or "creator" of relvar *R*—or possibly the DBA—must somehow tell anyone who uses that relvar exactly what the corresponding predicate is. What's more, these remarks are true of virtual relvars (i.e., views) just as much as they are of base ones. For example, the predicate *pred*(NLS) for view NLS ("non London suppliers") from earlier chapters is:

> *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (and CITY is not London).*

Every user of view NLS, for whatever purpose, must be aware of and understand this predicate.

Now, as I've written elsewhere (in *SQL and Relational Theory*, for example), in a perfect world the predicate *pred*(*R*) would serve as the "criterion for acceptability of updates" on relvar *R*. But this goal is obviously unachievable as stated. In the case of relvar S, for example:

- First of all, the DBMS can't know what it means for a "supplier" to be "under contract" or to be "located" somewhere; 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 supplier S1 is located in London, but there's no way the DBMS can do anything analogous.

- What's more (and perhaps more important), even if the DBMS 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 (by means of some update operation) that there's a supplier S6 named Lopez with status 30 and city Madrid, then there's no way the DBMS can possibly know whether that assertion is true; all the DBMS can do is check that the assertion in question doesn't violate any integrity constraints. Assuming it doesn't, the DBMS will then insert the tuple (S6,Lopez,30,Madrid) into relvar S, and will effectively treat that tuple as representing a true proposition from this point forward.

In other words, the pragmatic criterion for acceptability of updates on a given relvar, in today's DBMSs, is, as also explained in Chapter 2, not the relvar predicate but rather the pertinent (total) relvar constraint. Thus, the relvar constraint for a given relvar might be regarded—with a rather large dose of wishful thinking—as the DBMS's approximation to the corresponding relvar predicate.[3] Of course, this state of affairs explains the emphasis on constraints throughout earlier chapters of this book.

---

[3] Wishful thinking is right! This remark is, unfortunately, more than a little charitable to today's DBMSs. The sad truth is, most DBMSs today (which are SQL systems, of course) don't even provide much by way of support for constraints in the first place, let alone support for the corresponding predicates.

AN INTERSECTION EXAMPLE

Now suppose there were a way to tell the DBMS the name of the predicate for any given relvar, so that the DBMS would at least know that, e.g., the predicate for the suppliers relvar S is *pred*(S). *Note:* Presumably the DBMS would keep those predicate names along with the relvar definitions in the catalog. Presumably too there would be a way of keeping text versions of those predicates in the catalog as well (for use in error messages and other ergonomic purposes), although this aspect of what I'm discussing here isn't relevant to the question of view updating as such. Of course, there's still no suggestion that the DBMS should be able to "understand" those predicates in any sense—that's not the point at issue.

Suppose further that we could refer to those predicates by name, somehow, in requests to the DBMS. By way of example, consider the following scenario. Imagine we have two relvars, *A* and *B,* with predicates *pred*(*A*) and *pred*(*B*), respectively. To make the example a little more concrete, let *A* and *B* be, respectively, relvar PL and relvar PK from Chapters 9, 10, and 11 (and 14). Just to remind you once again, these relvars both have just a single attribute PNO (part number); relvar PL gives part numbers for parts on sale, and relvar PK gives part numbers for parts in stock. So the predicates *pred*(PL) and *pred*(PK)are as follows:

PL: *Part PNO is on sale.*

PK: *Part PNO is in stock.*

Note that these relvars are of the same relation type. Sample values are shown in Fig. 15.1, along with sample values for the corresponding intersection, union, and difference relvars XLK, ULK, and DLK, respectively (see further discussion below).

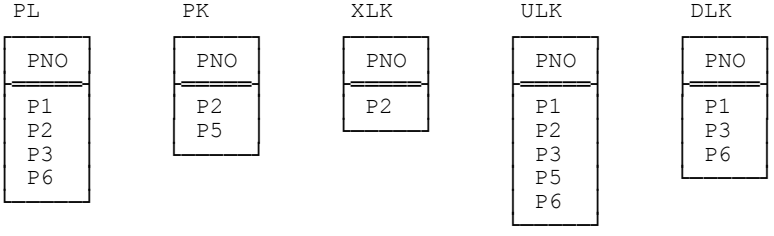


Fig. 15.1: Relvars PL, PK, XLK, ULK, and DLK—sample values

In order to make the discussions that follow a little more user friendly, let me introduce the names ON_SALE and IN_STOCK to be used in place of the slightly more formal names *pred*(PL) and *pred*(PK), respectively. (The fact that these names are the same as the attribute names I used in Chapters 9, 10, and 11 in connection with what in those chapters I called "a better design" is of course not a coincidence.) Now let relvars XLK, ULK, and DLK in fact be

views, defined as PL INTERSECT PK, PL UNION PK, and PL MINUS PK, respectively (sample values for these views are also shown in Fig. 15.1, as I've said). Clearly, then, the predicates for all of these relvars are as follows ("formal" or symbolic versions shown in parentheses):

PL: *Part PNO is on sale.* (ON_SALE)

PK: *Part PNO is in stock.* (IN_STOCK)

XLK: *Part PNO is both on sale and in stock.* (ON_SALE AND IN_STOCK)

ULK: *Part PNO is on sale or in stock.* (ON_SALE OR IN_STOCK)

DLK: *Part PNO is on sale and not in stock.* (ON_SALE AND NOT IN_STOCK)

What's more, given that the DBMS knows ON_SALE and IN_STOCK are the "formal" or symbolic predicates for relvars PL and PK, respectively, it can obviously figure out for itself the corresponding "formal" or symbolic predicates for relvars XLK, ULK, and DLK.

Now as we know from earlier chapters, views like those above can give rise to ambiguity. For example, consider the following update on the intersection view XLK:[4]

```
DELETE ( XLK WHERE PNO = 'P2' ) FROM XLK ;
```

According to the rules described in Chapter 9, the effect of this delete will be to cause the tuple for part P2 to be deleted from both PL and PK. But critics can and do object that the desired effect (removing the tuple from XLK) can be achieved by deleting that tuple just from PL or just from PK or from both, and there's no good reason for choosing any particular one of these options over the others, and so the delete should be rejected.

Now, I suggested in Chapter 9 that such deletes should be accepted anyway, despite the ambiguity—but of course I didn't mean to suggest by my arguments in that chapter or elsewhere that we shouldn't look for a way of resolving such ambiguities if we could. So let's take a closer look.

Consider the user, *U* say, who issues the foregoing delete on relvar XLK. Suppose for simplicity that XLK is the only relvar user *U* is aware of—i.e., user *U* doesn't even know relvars PL and PK exist. Despite this lack of awareness, user *U* must nevertheless be aware that the predicate for XLK is of the form ON_SALE AND IN_STOCK, where ON_SALE stands for *Part PNO is on sale* and IN_STOCK stands for *Part PNO is in stock*. So if user *U* wants to

[4] In this chapter I find it convenient to use a syntactic style for coding examples that more closely resembles genuine **Tutorial D** syntax, instead of the pseudocode style I've been using in previous chapters.

delete the tuple for part P2 from XLK, it must necessarily be the case that he or she wants to do so because he or she knows[5] that exactly one of the following three possibilities holds:

1. Part P2 is now not on sale (but is still in stock).

2. Part P2 is now not in stock (but is still on sale).

3. Part P2 is now neither on sale nor in stock.

What's more, of course, user *U* must also necessarily know exactly which one of these possibilities is in fact the case—for if not, then he or she has no business updating the database in the first place! Suppose for the sake of the example it's Possibility 2. Then (to invent some syntax on the fly) the user can and should issue the following delete (note the text in **boldface**):

```
DELETE ( XLK WHERE PNO = 'P2' ) PRED ( IN_STOCK ) FROM XLK ;
```

The purpose of the specification PRED (IN_STOCK) is to inform the DBMS that the tuple(s) to be deleted are all supposed to satisfy the predicate IN_STOCK (only), *and should therefore be deleted from relvar PK* (only). Of course, those tuples will still appear in relvar PL after the delete, but they certainly won't appear in the intersection view XLK.

Suppose now that Possibility 3 is in fact the case; then the appropriate PRED specification would be PRED (IN_STOCK AND ON_SALE).[6] And it seems reasonable to make this specification the default, so that if no explicit PRED specification appears at all, the delete behaves exactly as described in Chapter 9.

UNION AND DIFFERENCE EXAMPLES

Just to review, the situations where ambiguity at least potentially arises are as follows:[7]

1. Deleting through an intersection (see Chapter 9)

2. Inserting through a union (see Chapter 10)

3. Deleting through a difference (see Chapter 11)

---

[5] Actually "believes" might be better than "knows" here, but I'll stay with "knows" for simplicity.

[6] This specification is, of course, effectively just the predicate for XLK, but in a way that's a fluke, as we'll see in the next section.

[7] When I say "at least potentially" here, what I'm referring to is situations in which information equivalence is lost. Observe in particular that the situations in question include all of the various anomalous examples described in Chapter 14.

Of these, I've already discussed the first case in the previous section—except now let me point out that, of course, intersection is a special case of join, and so the ideas illustrated in that previous section can surely be generalized to deal with join views too. But let's consider examples of the other two cases.

Union

First union. Consider the following update on the union view ULK:

```
INSERT RELATION { TUPLE { PNO 'P4' } } INTO ULK ;
```

(The expression RELATION {TUPLE {PNO 'P4'}} is a relation selector invocation, denoting the relation containing just the specified tuple.) According to the rules described in Chapter 10, then, the effect of this insert is to cause the specified tuple to be inserted into both PL and PK, although the desired effect—inserting the tuple into ULK—could be achieved by inserting that tuple into either PL or PK, and there's no apparent need to insert it into both.

But consider the user once again (*U*, say) who issues the foregoing insert. Suppose for simplicity that ULK is the only relvar user *U* is aware of—i.e., user *U* doesn't even know relvars PL and PK exist. Despite this lack of awareness, user *U* must nevertheless be aware that the predicate for ULK is of the form ON_SALE OR IN_STOCK (where again ON_SALE stands for *Part PNO is on sale* and IN_STOCK stands for *Part PNO is in stock*). So if user *U* wants to insert the tuple for part P4 into ULK, it must necessarily be the case that he or she wants to do so because he or she knows that exactly one of the following possibilities holds:

1. Part P4 is now on sale (but not in stock).

2. Part P4 is now in stock (but not on sale).

3. Part P4 is now both on sale and in stock.

What's more, of course, user *U* must also know which one of these possibilities is in fact the case. Suppose for the sake of the example that it's Possibility 2. Then the user can and should issue the following insert:

```
INSERT RELATION { TUPLE { PNO 'P4' } } PRED ( IN_STOCK ) INTO ULK ;
```

Now the new tuple will be inserted into relvar PK (only).

Suppose now that Possibility 3 is in fact the case; then the appropriate PRED specification would be PRED (IN_STOCK AND ON_SALE).[8] And it seems reasonable to make this specification the default, so that if no explicit PRED specification appears at all, the delete behaves exactly as described in Chapter 10.

Difference

Turning now to difference, consider the following update on the difference view DLK:

```
DELETE ( DLK WHERE PNO = 'P1' ) FROM DLK ;
```

According to the rules described in Chapter 11, the effect of this delete is to cause the specified tuple to be deleted from PL and inserted into PK, although the desired effect—deleting the tuple from DLK—could be achieved by either one of these operations and there's no apparent need to do both.

Once again, however, consider the user *U* who issues the foregoing delete. Suppose DLK is the only relvar user *U* is aware of—i.e., user *U* doesn't even know relvars PL and PK exist. Despite this lack of awareness, user *U* must nevertheless be aware that the predicate for DLK is of the form ON_SALE AND NOT IN_STOCK (where once again ON_SALE stands for *Part PNO is on sale* and IN_STOCK stands for *Part PNO is in stock*). So if user *U* wants to delete the tuple for part P1 from DLK, it must necessarily be the case that he or she wants to do so because he or she knows exactly which one of the following possibilities holds:

1. Part P1 is now not on sale (and still not in stock).

2. Part P1 is now in stock (and still on sale).

3. Part P1 is now not on sale but is in stock.

Suppose for the sake of the example that Possibility 2 is the pertinent one. Then the user can and should issue the following delete:

```
DELETE ( DLK WHERE PNO = 'P1' ) PRED ( IN_STOCK ) FROM DLK ;
```

The PRED specification here is to be understood as follows: The tuple for part P1 now satisfies the predicate for relvar PK (it didn't do so before, of course, or it wouldn't have appeared in relvar DLK in the first place), and so it needs to be inserted into relvar PK, without being deleted from relvar PL. *Note:* For the record, the specification for Possibility 1 would be PRED (NOT ON_SALE), which would cause the tuple to be deleted from relvar PL, without

[8] Observe that AND is correct here, not OR, even though the predicate for ULK is IN_STOCK OR ON_SALE.

being inserted into any relvar PK; the specification for Possibility 3 would be PRED (NOT ON_SALE AND IN_STOCK), which would cause the tuple to be deleted from relvar PL and inserted into relvar PK. And it seems reasonable to make this last specification the default, so that if no explicit PRED specification appears at all, the delete behaves exactly as described in Chapter 11.

MORE ON PREDICATES

One reviewer of an early version of this chapter expressed some doubt as to whether the arguments in the last two sections were really valid. To be specific, he—the reviewer was male—suggested it might not always be true that (for example) in the case of a view defined as *A* UNION *B*, the user would know that the predicate was of the form *pred*(*A*) OR *pred*(*B*). And he went on to give an example of two relvars MOTHER and FATHER, each with a single attribute called NAME, and predicates *NAME is somebody's mother* and *NAME is somebody's father*, respectively, and then suggested that the predicate for MOTHER UNION FATHER might be, not *NAME is somebody's mother OR NAME is somebody's father*, but simply *NAME is somebody's parent*.

In response to this example, let me observe first that relvars MOTHER and FATHER are clearly disjoint, and the union is thus a disjoint union. So the true predicate for that union isn't quite as I gave it in the previous paragraph, but rather:

(NAME is somebody's mother OR NAME is somebody's father)
AND
NOT (NAME is somebody's mother AND NAME is somebody's father)

Thus, if a user of the union view really does know only the predicate *NAME is somebody's parent*, there won't be any way for that user to specify that INSERT operations on the view must effectively be targeted at just one of the two relvars MOTHER and FATHER. Instead, such operations will simply fail, on a **Golden Rule** violation.

That being said, I think I'd also have to question the wisdom of the DBA—or whoever it is who's responsible for informing the user of the predicate for the view—if that person decided, in the case at hand, *not* to tell the user what the proper predicate was. But you can't legislate wisdom, I suppose.

Be that as it may, what happens if the union view doesn't represent a disjoint union? Let's take a concrete example (a variation on an example discussed briefly in Chapter 10). Suppose some parts are manufactured abroad, others are manufactured domestically, and some are both. Suppose further that we represent this situation by means of two relvars, both with a single attribute PNO, one (PA) giving part numbers for parts manufactured abroad, the other (PD) giving part numbers for parts manufactured domestically. Then the union of these two relvars—

call it UAD—is an overlapping union, and it gives part numbers for all parts. The predicates are as follows:

PA: *Part PNO is used in the enterprise and is manufactured abroad.*

PD: *Part PNO is used in the enterprise and is manufactured domestically.*

UAD: *Part PNO is used in the enterprise and is manufactured either abroad or domestically (and possibly both).*

Well, if the user is told that the predicate for UAD is just *Part PNO is used in the enterprise*, then inserts will simply have to work as described in Chapter 10, and there's an end on't. Again, however, I think I'd have to question the wisdom of whoever it was who told the user that's what the predicate is. (By the way, note the relevance here of the discussions in the section "Information Equivalence Revisited" in the previous chapter.)

CONCLUDING REMARKS

I'd like to close this chapter, and the main part of this book, by repeating and stressing a point I touched on at the very beginning of the chapter (when I referred to ambiguities arising only "in the absence of information equivalence"). Indeed, the various ambiguities that can occur with the view updating approach as described in earlier chapters, and the various objections to that approach that have been voiced by certain critics, all have the same root cause: To be specific, they all have their origin in the fact that, sometimes, the DBMS simply hasn't been given enough information. For instance, suppose the database contains two distinct relvars with the same heading, as in the various examples involving relvars PL and PK discussed earlier in the present chapter and in several earlier chapters. Clearly those relvars represent different information (because if they didn't, there'd be no point in having both); in other words, they correspond to different predicates. But if those predicates aren't made known to the DBMS somehow—if they exist only external to the system, as it were (perhaps only in the mind of some user)—then there are bound to be occasions where it'll be impossible for the DBMS to do the right thing.

Now, my own preferred solution to such problems is as discussed in Chapters 9–11, under the general heading "A Better Design." Just to remind you, in the case of relvars PL and PK specifically, that solution involved the introduction of two attributes ON_SALE and IN_STOCK (each of type BOOLEAN, and having the obvious interpretations). In effect, what I did was redesign the database in such a manner as to give the DBMS the otherwise hidden information—in particular, to redesign it in such a manner that the DBMS could tell for itself which relvar(s) a given tuple logically belonged in. (To put the point another way, the introduction of those attributes allowed the pertinent predicates to be directly represented by means of constraints, thereby effectively making them—the predicates, that is—"known to the DBMS" as required.)

However, if the designer, for whatever reason, sees fit not to design the database in such a sensible manner, then I think a scheme along the lines of the one sketched in the present chapter is our only hope of getting the DBMS always to do the right thing.

Appendix A

Some Remarks on Relational Assignment

Change is not made without inconvenience
—Samuel Johnson: *A Dictionary of the English Language* (1755)

I claimed in Chapter 2 that the generic relational assignment—

```
R := rx
```

(where *R* is a relvar reference, or in other words a relvar name, syntactically speaking, and *rx* is a relational expression)—is logically equivalent to an assignment of the form:[1]

```
R := r - d ∪ i
```

Well, actually, in Chapter 2 I used the keywords MINUS and UNION in place of the set difference symbol "−" and the set union symbol "∪", respectively. In this appendix I find it more convenient to use the symbols. Other symbols I'll be using include "∩" (set intersection), "⊆" (set inclusion) and "Ø" (the empty set). What's more, in all of these contexts I ought really to be talking in terms of relations rather than general sets, but I propose to overlook this point of detail in what follows. *Note:* In Chapter 2 I also enclosed the subexpression *r* − *d* (or *r* MINUS *d*, rather) in parentheses; in fact, however, no parentheses are needed, as we'll soon see.

Be that as it may, let me now explain my notation in detail and elaborate on a few important points:

- *r* is the value of *R* before the assignment (the "old" value of *R*); *d* is a set of tuples to be deleted from *R* (the "delete set"); and *i* is a set of tuples to be inserted into *R* (the "insert set"). *Note:* More precisely, *d* and *i*, like *r*, are really relations, and the sets of tuples I'm talking about are really the bodies of those relations. I'll ignore this detail too in what follows.

[1] For obvious reasons, throughout this appendix I take the unqualified term *assignment* to mean a relational assignment specifically.

- $d \subseteq r$ (equivalently, $d - r = \emptyset$), because deleting a tuple that wasn't there in the first place has no logical effect. In other words, there's no point in making *d* any bigger than it need be.

- *r* and *i* are disjoint (i.e., $r \cap i = \emptyset$), because inserting a tuple that was already there has no logical effect. In other words, there's no point in making *i* any bigger than it need be.

- *d* and *i* are disjoint (i.e., $d \cap i = \emptyset$), because there's no point in deleting a tuple and then inserting it again (or the other way around), all as part of the same assignment. Of course, this fact (i.e., the fact that $d \cap i = \emptyset$) is a logical consequence of the two previous points, viz., that $d \subseteq r$ and $r \cap i = \emptyset$.

These points can be illustrated by means of a Venn diagram (see Fig. A.1, a repeat of Fig. 2.2 from Chapter 2). *Explanation:* In that diagram, *r*, *d*, and *i* are as above, and *u* is the universal relation of the pertinent type (i.e., the relation whose body consists of all tuples with the same heading as *R*—including, of course, those tuples that currently appear in *R*). Note that the difference $u - r$ is the (absolute) complement of *r*, i.e., the relation whose body consists of all tuples with the same heading as *R* that don't currently appear in *R*.

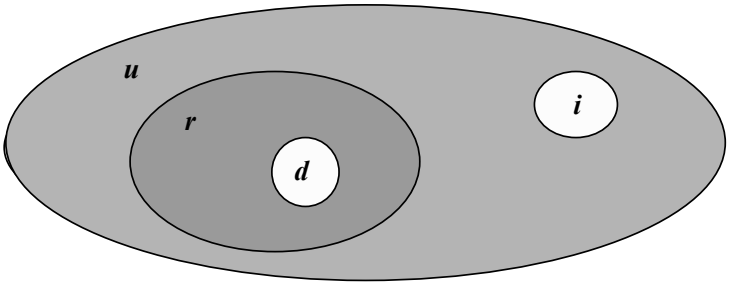


Fig. A.1: The delete set *d* and the insert set *i*

Note: Of course, the delete set *d* might be empty, in which case the original assignment is effectively an INSERT operation. Or the insert set *i* might be empty, in which case it's effectively a DELETE operation. Or they might both be empty, in which case the assignment overall degenerates to a "no op."

I now show that, given the assignment $R := r - d \cup i$ (where *r*, *d*, and *i* are as defined above), the delete set *d* and the insert set *i* are unique. Suppose, contrariwise, that

```
r - d1 ∪ i1  =  r - d2 ∪ i2
```

for some $d1 \neq d2$ and/or some $i1 \neq i2$, where in accordance with the conditions spelled out above $d1 - r = d2 - r = r \cap i1 = r \cap i2 = d1 \cap i1 = d2 \cap i2 = \emptyset$. Then:

- Let $t \in d1 - d2$. Since $d1 - r = \emptyset$, $t \in r - d2$ and $t \notin r - d1$. Hence $t \in r - d2 \cup i2$; therefore $t \in r - d1 \cup i1$. Since $t \notin r - d1$, $t \in i1$. But then $d1 \cap i1 \neq \emptyset$, which is a contradiction; hence no such t exists and $d1 - d2 = \emptyset$ (i.e., $d1 \subseteq d2$). A similar argument shows $d2 \subseteq d1$. Hence $d1 = d2$.

- We thus have $r - d1 = r - d2 = z$, say. Then $z \cup i1 = z \cup i2$. But $r \cap i1 = \emptyset$, so $(r - d1) \cap i1 = \emptyset$, so $z \cap i1 = \emptyset$. A similar argument shows $z \cap i2 = \emptyset$. Hence, since $z \cup i1 = z \cup i2$, $i1 = i2$.

It follows from the foregoing that if x is the relation to be assigned to R, then x can be expressed as either $r - d \cup i$ or $r \cup i - d$ (i.e., it makes no difference whether we "do the delete first" or "do the insert first"). It further follows that, given r and x, we have:

```
d  =  r - x
i  =  x - r
```

Now, to repeat a point I made earlier, an INSERT is basically an assignment for which d is empty, and a DELETE is basically an assignment for which i is empty. But now I'd like to say something about UPDATE specifically (by which I mean, of course, explicit UPDATE operations). Clearly, d and i must both be nonempty (in general) for such an operation. In fact, let's consider the generic form of such an operation:

```
UPDATE R WHERE bx : { A := ax }
```

Here R is a relvar reference; bx is a boolean expression (default simply TRUE); A is some attribute of relvar R; and ax is an expression denoting a value a of the same type as A. (For simplicity I limit my attention to the case of an UPDATE involving just the single "attribute assignment" $A := ax$. The discussion that follows can obviously be generalized to the case of an UPDATE involving two or more such assignments.)

Given this UPDATE, then it should be clear that the delete set d is defined as follows (I remind you that the symbol "$\stackrel{\text{def}}{=}$" means "is defined as"):

```
d  def=  { t : t ∈ R AND bx ( t ) AND t.A ≠ ax }
```

Explanation: What this definition means is that d consists of just those tuples t such that, first, t appears in R; second, bx evaluates to TRUE for t; and third, the A value within t isn't equal to a already.

The insert set i is then defined in terms of d, thus:

```
i  def=  { t' : EXISTS t ∈ d ( t' = EXTEND t : { A := ax } ) }
```

Explanation: As elsewhere in this book, the keyword EXISTS denotes the existential quantifier (it can be read as "there exists"). Thus, what the definition means is that *i* consists of just those tuples *t′* such that there exists a tuple *t* in *d* equal to that tuple *t′*, except that the *A* value in that tuple *t* has been replaced in that tuple *t′* by the value *a*. *Note:* I'm making use here of the "what if" version of EXTEND (see either Chapter 2 or Appendix B). Perhaps more to the point, I'm appealing also to the fact that—as mentioned in passing at various points earlier in this book—we can readily define analogs of several of the conventional relational operators, including EXTEND in particular, that apply to individual tuples instead of entire relations. See *SQL and Relational Theory* for further discussion of this point.

It follows from all of the above that explicit UPDATE operations do indeed correspond to a relational assignment for which *d* and *i* are (in general) both nonempty. However, it's worth pointing out that the converse is not true; that is, not all relational assignments for which *d* and *i* are both nonempty correspond to some explicit UPDATE operation. In fact, this is easy to see, for the following reason if no other:

- In the case of an explicit UPDATE operation, there is, loosely speaking, a one to one correspondence between the tuples of *d* and the tuples of *i*. To state the matter a little more precisely: First, every tuple of *d* corresponds to exactly one tuple of *i* (note, however, that two or more distinct tuples of *d* might correspond to the same tuple of *i*—but this can happen only if the UPDATE happens to be a "key UPDATE," and even then only rarely). Second, every tuple of *i* corresponds to at least one tuple of *d* (in fact, to exactly one tuple of *d*, except possibly if, again, the UPDATE happens to be a "key UPDATE").

- In the case of relational assignments in general, no such property holds. For example, a specific assignment might delete three tuples and insert five tuples.

The last issue I want to discuss in this appendix is the question of multiple assignment with repeated targets. Here again is what I had to say on the semantics of multiple assignment in Chapter 2:

> In outline, 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 are executed.
>
> 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 (you can even think of them as being executed in parallel, or "simultaneously," if you like).

However, I did also note that the foregoing explanation required some slight refinement in the case where two or more of the individual assignments specify the same target variable. Now I'd like to explain that refinement, briefly.

First of all, consider the following simple example:

```
DELETE ( S WHERE SNO = 'S1' ) FROM S ,
DELETE ( S WHERE SNO = 'S2' ) FROM S ;
```

Here's the corresponding expanded form:

```
S := S MINUS ( S WHERE SNO = 'S1' ) ,
S := S MINUS ( S WHERE SNO = 'S2' ) ;
```

Clearly, then, if both source expressions are evaluated before either of the individual assignments is executed, then one of these two assignments has no effect!—in other words, one of the DELETEs is "lost." (And to make matters worse, which of the two is lost is apparently not even defined.)

Because of situations like the foregoing, the definition of multiple assignment is extended to say, in effect, that if two or more of the individual assignments in that multiple assignment specify the same target variable, then those particular individual assignments are executed in sequence as written. A little more precisely, we can say of the foregoing double assignment that it's logically equivalent to the following single assignment:

```
S := WITH ( S := S MINUS ( S WHERE SNO = 'S1' ) ) :
                                    S MINUS ( S WHERE SNO = 'S2' ) ;
```

This assignment in turn is equivalent to the following:

```
WITH ( DELETE ( S WHERE SNO = 'S1' ) FROM S ) :
       DELETE ( S WHERE SNO = 'S2' ) FROM S ;
```

Note: Of course, the particular double assignment we're considering here is logically equivalent to the following single assignment:

```
S := ( S MINUS ( S WHERE SNO = 'S1' ) ) MINUS ( S WHERE SNO = 'S2' ) ;
```

More generally, it's intuitively obvious that a multiple assignment in which all of the individual assignments specify the same target relvar *R* is logically equivalent to some single assignment to that same relvar *R*. Why? Because any assignment, multiple or otherwise, to relvar *R* effectively assigns some "new" value $R - d \cup i$ to *R*, for some *d* and some *i*.

Of course, the foregoing observation, while obviously correct, is of no help in computing the actual values of *d* and *i* in the general case. But the following remarks might help a little in this regard. Consider the following double assignment (and note carefully that the relvar

reference *R* in the expression on the right side of the second assignment denotes the value of relvar *R* after the first assignment has been executed):

```
R := R - d1 ∪ i1 , R := R - d2 ∪ i2 ;
```

Now, the following pairs of sets aren't necessarily disjoint, in general: *d1* and *d2*; *i1* and *i2*; *d1* and *i2*; *d2* and *i1*. However, it's fairly easy to see (exercise for the reader!) that the foregoing double assignment is logically equivalent to a single assignment of the form—

```
R := r - d ∪ i ;
```

—where:

- *r* is the initial value of *R* (i.e., the value of *R* before the first assignment is done)
- `d = ( d1 - i2 ) ∪ d2`
- `i = ( i1 ∪ i2 ) - d2`

Note that *d* and *i* here are certainly disjoint; however, it's not the case, in general, that *d* is included in *r* or that *i* and *r* are disjoint. Now define:

- `d' = ( r ∩ d ) - i`
- `i' = i - ( r ∪ d )`

Then the original double assignment is logically equivalent to the following single assignment—

```
R := r - d' ∪ i' ;
```

—where $d' - r = r \cap i' = d' \cap i' = \emptyset$.

Note finally that it follows by induction from all of the above that a multiple assignment of the form

```
R := rx1 , R := rx2 , ... , R := rxn
```

for arbitrary $n > 0$ can always be reduced to a single assignment of the form

```
R := r - d ∪ i
```

where *r* is the initial value of *R* and $d - r = r \cap i = d \cap i = \emptyset$.

Appendix B

Relational Operators

Civilization advances by extending the number of important operations which we can perform without thinking about them
—Alfred North Whitehead: *Introduction to Mathematics* (1911)

In this appendix, I give for purposes of reference definitions of all of the relational operators discussed in the body of the book. The definitions are given in alphabetical order by name (i.e., the name by which the operator in question is known in **Tutorial D**); they're based on definitions in my book *The Relational Database Dictionary*, *Extended Edition* (Apress, 2008), but I've deliberately simplified them slightly here and there for present purposes. Observe in particular that the operators all return a result with a defined heading and therefore a defined relation type, which is thereby the type of any expression that represents an invocation of the operator in question. For further discussion, please refer to *SQL and Relational Theory*.

- **EXTEND:** 1. (*New attribute*) Let relation *r* not have an attribute called *A*. Then the expression EXTEND *r* : {*A* := *exp*} returns a 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 *exp* on that tuple of *r*. 2. (*Existing attribute*) Let relation *r* have an attribute called *A*. Then the expression EXTEND *r* : {*A* := *exp*} returns a 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*.

- **GROUP:** Let the heading of relation *r* be partitioned into subsets *X* = {*X1*,*X2*,...,*Xm*} and *Y* = {*Y1*,*Y2* ...,*Yn*}; also, let *YR* be an attribute name not appearing in *X*. Then the expression *r* GROUP ({*Y1*,*Y2*,...,*Yn*} AS *YR*) returns a relation *s*. The heading of *s* is {*X1*,*X2*,...,*Xm*,*YR*}, where *YR* is of type RELATION {*Y1*,*Y2*,...,*Yn*}. The body of *s* is defined as follows. Let *z* be the result of *r* WRAP ({*Y1*,*Y2*,...,*Yn*} AS *YT*). For each distinct *X* value *x* in *z*, let *yr* be the relation whose tuples are all and only those *YT* values from tuples in *z* in which the *X* value is *x*; let *t* be a tuple of type TUPLE {*X*,*YR*} with *X* value *x* and *YR* value *yr*; then, and only then, *t* is a tuple of *s*.

- **INTERSECT:** 1. (*Dyadic case*) Let relations *r1* and *r2* have the same heading *H*. Then the expression *r1* INTERSECT *r2* returns a relation with heading *H* and body the set of all

tuples *t* such that *t* appears in both *r1* and *r2*. 2. (*N-adic case*) Let relations *r1*, *r2*, ..., *rn* ($n \geq 0$) all have the same heading *H*. Then the expression INTERSECT {*r1*,*r2*,...,*rn*} returns a relation with heading *H* and body the set of all tuples *t* such that *t* appears in each of *r1*, *r2*, ..., *rn*. *Note:* If $n = 0$, (a) some syntactic mechanism, not shown here, is needed to specify the pertinent heading, and (b) the result is the universal relation of the pertinent type.

- **JOIN:** 1. (*Dyadic case*) Let relations *r1* and *r2* be such that attributes with the same name are of the same type. Then the expression *r1* JOIN *r2* returns a 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*. 2. (*N-adic case*) Let relations *r1*, *r2*, ..., *rn* ($n \geq 0$) be such that attributes with the same name are of the same type. Then the expression JOIN {*r1*,*r2*,...,*rn*} is defined as follows: If $n = 0$, the result is TABLE_DEE; if $n = 1$, the result is *r1*; otherwise, choose any two distinct relations from the set *r1*, *r2*, ..., *rn* and replace them by their dyadic join, and repeat this process until the set consists of just one relation *r*, which is the overall result.

- **MATCHING:** The expression *r1* MATCHING *r2* returns a relation equal to the relation returned by the expression *r1* JOIN *r2*, projected back on the attributes of *r1*.

- **MINUS:** Let relations *r1* and *r2* have the same heading *H*. Then the expression *r1* MINUS *r2* returns a relation with heading *H* and body the set of all tuples *t* such that *t* appears in *r1* and not in *r2*.

- **NOT MATCHING:** The expression *r1* NOT MATCHING *r2* returns a relation equal to the relation returned by the expression *r1* MINUS (*r1* MATCHING *r2*).

- **Project:** Let relation *r* have heading *H*, let *X* be a subset of *H*, and let *A1*, *A2*, ..., *An* be all of the attribute names in *X*. Then the projection *r*{*A1*,*A2*,...,*An*} of *r* on (or over) *X* is a relation with heading *X* and body consisting of all tuples *x* such that there exists a tuple *t* in *r* with *X* value *x*.

- **RENAME:** Let relation *r* have an attribute called *A* and no attribute called *B*. Then the expression *r* RENAME {*A* AS *B*} returns a 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 all references to *A* in that body—more precisely, in tuples in that body—are replaced by references to *B*. *Note:* The body of the book never discussed the question of updating through a renaming, but (of course) that was because the pertinent update rules are trivially obvious.

- **Restrict:** Let *r* be a relation and let *bx* be a boolean expression. Then the expression *r* WHERE *bx* returns a relation with heading the same as that of *r* and body consisting of all tuples of *r* for which *bx* evaluates to TRUE. *Note:* Technically, *bx* is supposed to be a restriction condition, which means that (a) all attribute references are to attributes of *r* and (b) there aren't any relvar references. However, **Tutorial D** (like SQL) permits boolean expressions in WHERE clauses that are more general than simple restriction conditions.

- **SUMMARIZE:** 1. (*BY*) Let relation *r* have attributes called *A1*, *A2*, ..., *An* (and possibly others) and no attribute called *B*. Then the expression SUMMARIZE *r* BY {*A1*,*A2*,...,*An*} : {*B* := *summary*} returns a relation with heading {*A1*,*A2*,...,*An*,*B*} and body the set of all tuples *t* such that *t* is equal to the projection of some tuple of *r* on *A1*, *A2*, ..., *An*, extended with a value *b* for *B*. That value *b* is computed by evaluating *summary* over all tuples of *r* that have the same value for attributes *A1*, *A2*, ..., *An* as *t* does. 2. (*PER*) 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 the expression SUMMARIZE *r1* PER (*r2*) : {*B* := *summary*} returns a relation with heading {*A1*,*A2*,...,*An*,*B*} and body the set 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 *summary* over all tuples of *r1* that have the same value for attributes *A1*, *A2*, ..., *An* as *t* does.

- **TIMES:** 1. (*Dyadic case*) Let relations *r1* and *r2* have no attribute names in common. Then the expression *r1* TIMES *r2* returns a 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*. 2. (*N-adic case*) Let relations *r1*, *r2*, ..., *rn* ($n \geq 0$) be such that no two of them have any attribute names in common. Then the expression TIMES {*r1*,*r2*,...,*rn*} returns a relation with heading the set theory union of the headings of *r1*, *r2*, ..., *rn* and body the set of all tuples *t* such that *t* is the set theory union of a tuple from *r1*, a tuple from *r2*, ..., and a tuple from *rn*. *Note:* TIMES is really just a special case of JOIN, q.v.

- **UNGROUP:** Let relation *s* have an attribute *YR* of type RELATION {*Y1*,*Y2*,...,*Yn*}, and let *X* = {*X1*,*X2*,...,*Xm*} be all of the attributes of *s* other than *YR*; also, let no *Xi* have the same name as any *Yj* ($0 \leq i \leq m$, $0 \leq j \leq n$). Then the expression *s* UNGROUP (*YR*) returns a relation *r*. The heading of *r* is {*X1*,*X2*,...,*Xm*,*Y1*,*Y2*,...,*Yn*}. The body of *r* is defined as follows. Let *z* be a relation with heading {*X1*,*X2*,...,*Xm*,*YT*}, where *YT* is of type TUPLE {*Y1*,*Y2*,...,*Yn*}, and body defined thus: For each tuple of *s*, *z* contains a set of tuples of type {*X*,*YT*}, one (*t*, say) for each tuple in the *YR* value in that *s* tuple; each such tuple *t* contains an *X* value equal to the *X* value from the *s* tuple in question and a *YT* value equal to some tuple from the *YR* value in the *s* tuple in question. Let *z* contain no other tuples. Then *r* is the result of *z* UNWRAP (*YT*).

- **UNION:** 1. (*Dyadic case*) Let relations *r1* and *r2* have the same heading *H*. Then the expression *r1* UNION *r2* returns a relation with heading *H* and body the set of all tuples *t* such that *t* appears in either or both of *r1* and *r2*. 2. (*N-adic case*) Let relations *r1*, *r2*, ..., *rn* ($n \geq 0$) all have the same heading *H*. Then the expression UNION {*r1*,*r2*,...,*rn*} returns a relation with heading *H* and body the set of all tuples *t* such that *t* appears in at least one of *r1*, *r2*, ..., *rn*. *Note:* If $n = 0$, (a) some syntactic mechanism, not shown here, is needed to specify the pertinent heading, and (b) the result is the empty relation of the pertinent type.

- **UNWRAP:** Let relation *s* have an attribute *YT* of type TUPLE {*Y1*,*Y2*,...,*Yn*}, and let *X* = {*X1*,*X2*,...,*Xm*}be all of the attributes of *s* other than *YT*; also, let no *Xi* have the same name as any *Yj* ($0 \leq i \leq m$, $0 \leq j \leq n$). Then the expression *s* UNWRAP (*YT*) returns a relation *r*. The heading of *r* is {*X1*,*X2*,...,*Xm*,*Y1*,*Y2*,...,*Yn*}. The body of *r* contains one tuple for each tuple in *s*, and no other tuples. Let tuple *t* of *s* have *X* value *x* and *YT* value a tuple with *Y1* value *y1*, *Y2* value *y2*, ..., and *Yn* value *yn*; then the corresponding tuple of *r* has *X* value *x*, *Y1* value *y1*, *Y2* value *y2*, ..., and *Yn* value *yn*.

- **WRAP:** Let the heading of relation *r* be partitioned into subsets *X* = {*X1*,*X2*,...,*Xm*} and *Y* = {*Y1*,*Y2* ...,*Yn*}; also, let *YT* be an attribute name not appearing in *X*. Then the expression *r* WRAP ({*Y1*,*Y2*,...,*Yn*} AS *YT*) returns a relation *s*. The heading of *s* is {*X1*,*X2*,...,*Xm*,*YT*}, where *YT* is of type TUPLE {*Y1*,*Y2*,...,*Yn*}. The body of *s* contains one tuple for each tuple in *r*, and no other tuples. Let tuple *t* of *r* have *X* value *x*, *Y1* value *y1*, *Y2* value *y2*, ..., and *Yn* value *yn*; then the corresponding tuple of *s* has *X* value *x* and *YT* value a tuple with *Y1* value *y1*, *Y2* value *y2*, ..., and *Yn* value *yn*.

Note: Of the foregoing operators, the ones for which tuple analogs are defined are these: EXTEND, project, RENAME, UNION, UNWRAP, and WRAP (only).

Finally, for completeness, here's a definition of the *image relation* construct, mentioned in passing in Chapters 8, 12, and 13:

- **Image relation:** Let relations *r1* and *r2* be such that attributes with the same name are of the same type); let *t1* be a tuple of *r1;* let *t2* be a tuple of *r2* that has the same values for those common attributes as tuple *t1* does; let relation *r3* be that restriction of *r2* that contains all and only such tuples *t2;* and let relation *r4* be the projection of *r3* on all but those common attributes. Then *r4* is the *image relation* (with respect to *r2*) corresponding to *t1*. *Note:* Image relation references are permitted only in contexts in which tuple *t1* is well defined. For further explanation, see *SQL and Relational Theory*.

Index

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.