Database Design and Relational Theory

Normal Forms and All That Jazz

Second Edition

C. J. Date

Apress®

Database Design and Relational Theory

Normal Forms and All That Jazz

Second Edition

C. J. Date

Apress®

Database Design and Relational Theory: Normal Forms and All That Jazz

C. J. Date
Healdsburg, California, USA

ISBN-13 (pbk): 978-1-4842-5539-1 ISBN-13 (electronic): 978-1-4842-5540-7
https://doi.org/10.1007/978-1-4842-5540-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484255391. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

In computing, elegance is not a dispensable luxury
but a quality that decides between success and failure.

—Edsger W. Dijkstra

The ill design is most ill for the designer.

—Hesiod

It is to be noted that when any part of this paper is dull
there is design in it.

—Sir Richard Steele

The idea of a formal design discipline is often rejected on account of vague cultural / philosophical condemnations such as "stifling creativity"; this is more pronounced … where a romantic vision of "the humanities" in fact idealizes technical incompetence …

[We] know that for the sake of reliability and intellectual control we have to keep the design simple and disentangled.

—Edsger W. Dijkstra

My designs are strictly honorable.

—Anon.

——— ♦♦♦♦♦ ———

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

Table of Contents

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold some 900,000 copies at the time of writing and is used by several hundred colleges and universities worldwide. He is also the author of numerous other books on database management, including most recently:

- From Ventus: *Go Faster! The TransRelational™ Approach to DBMS Implementation* (2002, 2011)
- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, with Hugh Darwen, 2007)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007) and *Database Explorations: Essays on The Third Manifesto and Related Topics* (with Hugh Darwen, 2010)
- From Apress: *Date on Database: Writings 2000-2006* (2006)
- From Morgan Kaufmann: *Time and Relational Theory: Temporal Databases in the Relational Model and SQL* (with Hugh Darwen and Nikos A. Lorentzos, 2014)
- From O'Reilly: *Relational Theory for Computer Professionals: What Relational Databases Are Really All About* (2013); *View Updating and Relational Theory: Solving the View Update Problem* (2013); *SQL and Relational Theory: How to Write Accurate SQL Code* (3rd edition, 2015); *The **New** Relational Database Dictionary* (2016); and *Type Inheritance and Relational Theory: Subtypes, Supertypes, and Substitutability* (2016)
- From Lulu: *E. F. Codd and Relational Theory: A Detailed Review and Analysis of Codd's Major Database Writings* (2019)

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.

Preface to the First Edition

This book began life as a comparatively short chapter in a book called *Database in Depth: Relational Theory for Practitioners* (O'Reilly, 2005). That book was superseded by a greatly expanded version called *SQL and Relational Theory: How to Write Accurate SQL Code* (O'Reilly, 2009), where the design material, since it was somewhat tangential to the main theme of the book, ceased to be a chapter as such and became a (somewhat longer) appendix instead. I subsequently began work on a second edition of this latter book.[1] During the course of that work, I found there was so much that needed to be said on the subject of database design in general that the appendix threatened to grow out of all proportion to the rest of the book. Since the topic was, as I've indicated, rather out of line with the major emphasis of that book anyway, I decided to cut the Gordian knot and separate the material out into a book of its own: the one you're looking at right now.

Three points arise immediately from the foregoing:

- First, the present book does assume you're familiar with material covered in the *SQL and Relational Theory* book (in particular, it assumes you know exactly what *relations*, *attributes*, and *tuples* are). I make no apology for this state of affairs, however, 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.
- Second, the previous point notwithstanding, there's unavoidably a small amount of overlap between this book and that earlier book. I've done my best to keep that overlap to a minimum, however.

[1] That second edition was published by O'Reilly in 2012. It was followed in 2015 by a third. Thus, all references to that book in what follows should be understood as referring to that third edition specifically (where it makes any difference).

- Third, there are, again unavoidably, many references in this book to that earlier one. Now, most references in this book to other publications are given in full, as in this example:

  > Ronald Fagin: "Normal Forms and Relational Database Operators," Proc. 1979 ACM SIGMOD International Conference on Management of Data, Boston, Mass. (May/June 1979)

  In the case of references to the *SQL and Relational Theory* book in particular, however, from this point forward I'll give them in the form of that abbreviated title alone.

Actually I've published several short pieces over the years, in one place or another, on various aspects of design theory, and the present book is intended among other things to preserve the good parts of those earlier writings. But it's not just a cobbling together of previously published material, and I sincerely hope it won't be seen as such. For one thing, it contains much new material. For another, it presents a more coherent, and I think much better, perspective on the subject as a whole (I've learned a lot myself over the years!). Indeed, even when a portion of the text is based on some earlier publication, the material in question has been totally rewritten and, I trust, improved.

Now, there's no shortage of books on database design—so what makes this one different? In fact I don't think there's a book on the market that's quite like this one. There are many books (of considerably varying quality, in my not unbiased opinion) on design practice, but those books (again, in my own opinion) usually don't do a very good job of explaining the underlying theory. And there are a few books on design theory, too, but they tend to be aimed at theoreticians, not practitioners, and to be rather academic in tone. What I want to do is bridge the gap; in other words, I want to explain the theory in a way that practitioners should be able to understand, and I want to show why that theory is of considerable practical importance. What I'm not trying to do is be exhaustive; I don't want to discuss the theory in every last detail, I want to concentrate on what seem to me the important parts (though, naturally, my treatment of the parts I do cover is meant to be precise and accurate, as far as it goes). Also, I'm aiming at a judicious blend of the formal and the informal; in other words, I'm trying to provide a gentle introduction to the theory, so that:

a. You can use important theoretical results to help you actually do design, and

b. You'll be able, if you're so inclined, to go to the more academic texts and understand them.

In the interest of readability, I've deliberately written a fairly short book, and I've deliberately made each chapter fairly short, too.[2] (I'm a great believer in doling out information in small and digestible chunks.) Also, every chapter includes a set of exercises (answers to most of which are given in Appendix D at the back of the book),[3] and I do recommend that you have a go at some of those exercises if not all. Some of them are intended to show how to apply the theoretical ideas in practice; others provide (in the answers if not in the exercises as such) additional information on the subject matter, over and above what's covered in the main body of the text; and still others are meant—for example, by asking you to prove some simple theoretical result—to get you to gain some understanding as to what's involved in "thinking like a theoretician." Overall, I've tried to give some insight into what design theory is and why it is the way it is.

Prerequisites

My target audience is database professionals: more specifically, database professionals with a more than passing interest in database design. In particular, therefore, I assume you're reasonably familiar with the relational model, or at least with certain aspects of that model (Chapter 2 goes into more detail on these matters). As already indicated, familiarity with the *SQL and Relational Theory* book would be helpful.

Logical vs. Physical Design

This book is about design *theory*; by definition, therefore, it's about logical design, not physical design. Of course, I'm not saying physical design is unimportant (of course not); but I am saying it's a distinct activity, separate from and subsequent to logical design. To spell the point out, the "right" way to design a database is as follows:

1. Do a clean logical design first. Then, as a separate and subsequent step:

[2]Sadly, the second edition is somewhat larger than its predecessor. That always happens with new editions, of course, though in the present case the increase is due in part to the fact that—in response to reader requests—I've increased the font size. In any case, at least the individual chapters are still fairly short. Mostly.

[3]In response to reader requests again, in this second edition I've moved the answers that are specific to a given chapter to the end of the chapter in question and deleted the old Appendix D.

2. Map that logical design into whatever physical structures the target DBMS happens to support.[4]

Note, therefore, that the physical design should be derived from the logical design and not the other way around. (Ideally, in fact, the system should be able to derive the physical design "automatically" from the logical design, without the need for human involvement in the process at all.)[5]

To repeat, the book is about design theory. So another thing it's not about is the various ad hoc design methodologies—entity / relationship modeling and the like—that have been proposed over the years, at one time or another. Of course, I realize that certain of those methodologies are fairly widely used in practice, but the fact remains that they enjoy comparatively little by way of a solid theoretical basis. As a result, they're mostly beyond the scope of a book like this one. However, I do have a few remarks here and there on such "nontheoretical" matters (especially in Chapters 8 and 17, also in Appendix C).

Acknowledgments

I'd like to thank Hugh Darwen, Ron Fagin, David McGoveran, and Andy Oram for their meticulous reviews of earlier drafts of this book. Each of these reviewers helped correct a number of misconceptions on my part (rather more such, in fact, than I like to think). Of course, it goes without saying that any remaining errors are my responsibility. I'd also like to thank Chris Adamson for help with certain technical questions, and my wife Lindy for her support throughout the production of this book, as well as all of its predecessors.

C. J. Date
Healdsburg, California
2012 (minor revisions 2019)

[4]DBMS = database management system. Note that there's a logical difference between a DBMS and a database! Unfortunately, the industry very commonly uses the term *database* when it means either some DBMS 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, then what do you call the database?

[5]This idea isn't as farfetched as it might seem. See my book *Go Faster! The TransRelational™ Approach to DBMS Implementation* (Ventus, 2002, 2011), available as a free download from `http://bookboon.com`.

Preface to the Second Edition

This edition differs from its predecessor in many ways. The overall objective remains the same, of course—I'm still trying to provide a gentle introduction to design theory—but the text has been revised throughout to reflect, among other things, experience gained from teaching live classes based on the first edition. Quite a lot of new material has been added (including new chapters on sixth normal form and the various normal forms between fourth and fifth, and a couple of new appendixes on database design in general). Examples, exercises, and answers have been expanded and improved in various respects, and the text has been subjected to a thorough overhaul throughout. Numerous cosmetic improvements and a variety of technical corrections—an embarrassingly large number of these, I'm sorry to have to report—have also been made. The net effect is to make the text rather more comprehensive (but, sadly, some 50% bigger) than its predecessor.

My thanks to O'Reilly Media Inc. (publisher of the first edition) for permission to place this second edition with a different publisher.

C. J. Date
Healdsburg, California
2019

PART I

Setting the Scene

This part of the book consists of two introductory chapters, the titles of which ("Preliminaries" and "Prerequisites," respectively) are more or less self-explanatory.

CHAPTER 1

Preliminaries

(On being asked what jazz is:)
Man, if you gotta ask, you'll never know.

—Louis Armstrong (attrib.)

This book has as its subtitle *Normal Forms and All That Jazz*. Clearly some explanation is needed! First of all, of course, I'm talking about design theory—database design theory, that is—and everybody knows that normal forms are a major component of that theory; hence the first part of the subtitle. But there's more to that theory than just normal forms, and that fact accounts for that subtitle's second part. Third, it's unfortunately the case that—from the practitioner's point of view, at any rate—design theory seems to be riddled with terms and concepts that are hard to understand and don't seem to have much to do with design as actually done in practice. That's why I framed the latter part of my subtitle in colloquial (not to say slangy) terms; I wanted to convey the idea that, although we'd necessarily be dealing with "difficult" material on occasion, the treatment of that material would be as undaunting and unintimidating as I could make it. But whether I've succeeded in that aim is for you to judge, of course.

I'd also like to say a little more on the question of whether design theory has anything to do with design as carried out in practice. Let me be clear: Nobody could, or should, claim that database design is easy. But a sound knowledge of the theory can only help. In fact, if you want to do design properly—if you want to build databases that are as robust, flexible, and accurate as they're supposed to be—then you simply have to come to grips with the theory. There's just no alternative: at least, not if you want to claim to be a design professional. Design theory is the scientific foundation for database design, just as the relational model is the scientific foundation for database technology in general. And just as anyone professionally involved in database technology in general needs to be familiar with the relational model, so anyone involved in database design in particular needs to be familiar with design theory. Proper design is so important! After all, the

C. J. Date, *Database Design and Relational Theory*, https://doi.org/10.1007/978-1-4842-5540-7_1

database lies at the heart of so much of what we do in the computing world; so if it's badly designed, the negative impacts can be extraordinarily widespread.

Some Quotes from the Literature

Since we're going to be talking quite a lot about normal forms, I thought it might be—well, not exactly enlightening, but entertaining, possibly (?)—to begin with a few quotes from the literature. The starting point for the whole concept of normal forms is, of course, *first* normal form (1NF), and so an obvious question is: *Do you know what 1NF is?* As the following quotes demonstrate (sources omitted to protect the guilty), a lot of people don't:

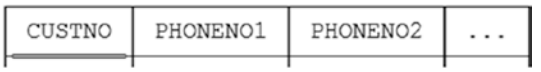
- To achieve first normal form, each field in a table must convey unique information.
- An entity is said to be in the first normal form (1NF) when all attributes are single valued.
- A relation is in 1NF if and only if all underlying domains contain atomic values only.
- If there are no repeating groups of attributes, then [the table] is in 1NF.

Now, it might be argued that some if not all of these quotes are at least vaguely correct—but they're all hopelessly sloppy, even when they're generally on the right lines. *Note:* In case you're wondering, I'll be giving a precise and accurate definition of 1NF in Chapter 4.

Let's take a closer look at what's going on here. Here again is the first of the foregoing quotes, now given in full:

- To achieve first normal form, each field in a table must convey unique information. For example, if you had a Customer table with two columns for the telephone number, your design would violate first normal form. First normal form is fairly easy to achieve, since few folks would see a need for duplicate information in a table.

  OK, so apparently we're talking about a design that looks something like this:



Now, I can't say whether this is a good design or not, but it certainly doesn't violate 1NF. (I can't say whether it's a good design because I don't know exactly what "two columns for the telephone number" means—the phrase "duplicate information in a table" suggests we're recording the same phone number twice, but such an interpretation is absurd on its face. But even if that interpretation is correct, it still wouldn't constitute a violation of 1NF as such.)

Here's another quote:

- First Normal Form ... means the table should have no "repeating groups" of fields ... A repeating group is when you repeat the same basic attribute (field) over and over again. A good example of this is when you wish to store the items you buy at a grocery store ... [*and the writer goes on to give an example, presumably meant to illustrate the concept of a repeating group, of a table called Item Table, with columns called Customer, Item1, Item2, Item3, and Item4*]:

CUSTOMER	ITEM1	ITEM2	ITEM3	ITEM4

Well, this design is almost certainly bad—what happens if the customer doesn't purchase exactly four items?—but the reason it's bad isn't that it violates 1NF; like the previous example, in fact, it's a 1NF design. So, while it might perhaps be claimed—indeed, it often is claimed—that 1NF does mean, loosely, "no repeating groups," a repeating group is *not* "when you repeat the same basic attribute over and over again."[1]

How about this one (a cry for help found on the Internet)? I'm quoting it absolutely verbatim, except that I've added some boldface:

- I have been trying to find the correct way of normalizing tables in Access. From what I understand, it goes from the 1st normal form to 2nd, then 3rd. Usually, that's as far as it goes, but sometimes to the 5th and 6th. Then, there's also the Cobb 3rd. This all makes sense to me. **I am supposed to teach a class in this starting next week,** and I just got the textbook. It says something entirely different. It says 2nd normal form is only for tables with a multiple-field primary key, 3rd normal form is only for tables with a single-field key. 4th normal form

[1]At the same time it's not as easy as you might think to say exactly what it is! See further discussion in Chapter 4.

can go from 1st to 4th, where there are no independent one-to-many relationships between primary key and non-key fields. Can someone clear this up for me please?

And one more (this time with a "helpful" response):

- *It's not clear to me what "normalized" means. Can you be specific about what normalization rules you are referring to? In what way is my schema not normalized?*

  Normalization: The process of replacing duplicate things with a reference to the original thing.

  For example, given "john is-a person" and "john obeys army," one observes that the "john" in the second sentence is a duplicate of "john" in the first sentence. Using the means provided by your system, the second sentence should be stored as "->john obeys army."

A Note on Terminology

As I'm sure you noticed, the quotes in the previous section were expressed for the most part in the familiar "user friendly" terminology of tables, rows, and columns (or fields). In this book, by contrast, I'll favor the more formal terms *relation*, *tuple* (usually pronounced to rhyme with *couple*), and *attribute*. I apologize if this decision on my part makes the text a little harder to follow, but I do have my reasons. As I said in *SQL and Relational Theory:*[2]

> I'm generally sympathetic to the idea of using more user friendly terms, if they can help make the ideas more palatable. In the case at hand, however, it seems to me that, regrettably, they don't make the ideas more palatable; instead, they distort them, and in fact do the cause of genuine understanding a grave disservice.

[2]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* (3rd edition, O'Reilly, 2015).

> The truth is, a relation is *not* a table, a tuple is *not* a row, and an attribute is *not* a column. And while it might be acceptable to pretend otherwise in informal contexts—indeed, I often do so myself—I would argue that it's acceptable only if all parties involved understand that those more user friendly terms are just an approximation to the truth and fail overall to capture the essence of what's really going on. To put it another way: If you do understand the true state of affairs, then judicious use of the user friendly terms can be a good idea; but in order to learn and appreciate that true state of affairs in the first place, you really do need to come to grips with the formal terms.

To the foregoing, let me add that (as I said in the preface) I do assume you know exactly what *relations*, *attributes*, and *tuples* are—though in fact formal definitions of these constructs can be found in Chapter 5.

There's another terminological matter I need to get out of the way, too. The relational model is, of course, a data model. Unfortunately, however, this latter term has two quite distinct meanings in the database world.[3] The first and more fundamental one is this:

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

This is the meaning we have in mind when we talk about the relational model in particular: The data structures in the relational model are relations, of course, and the data operators are the relational operators projection, join, and all the rest. (As for that "and so forth" in the definition, it covers such matters as keys, foreign keys, and various related concepts.)

The second meaning of the term *data model* is as follows:

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

[3]This observation is undeniably correct. However, one reviewer wanted me to add that the two meanings can be thought of as essentially the same concept at different levels of abstraction. I hope that helps!

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

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

- A data model in the first sense is like a programming language, whose constructs can be used to solve many specific problems but in and of themselves have no direct connection with any such specific problem.
- A data model in the second sense is like a specific program written in that language—it uses the facilities provided by the model, in the first sense of that term, to solve some specific problem.

It follows from all of the above that if we're talking about data models in the second sense, then we might reasonably speak of "relational models" in the plural, or "a" relational model, with an indefinite article. But if we're talking about data models in the first sense, then *there's only one relational model*, and it's *the* relational model, with the definite article.

Now, as you are probably aware, most writings on database design, especially if their focus is on pragma rather than the underlying theory, use the term "model," or the term "data model," exclusively in the second sense. But—*please note very carefully!*—I don't follow this practice in the present book; in fact, I don't use the term "model" at all, except occasionally to refer to the relational model as such.

The Running Example

Now let me introduce the example I'll be using as a basis for most of the discussions in the rest of the book: the familiar—not to say hackneyed—suppliers-and-parts database. (I apologize for dragging out this old warhorse yet one more time, but I do believe that using essentially the same example in a variety of different books and publications can help, not hinder, the learning process.) Sample values are shown in Figure 1-1 on the next page.[4] To elaborate:

[4]For reasons that might or might not become clear later, the values shown in Fig. 1.1 differ in two small respects from those in other books of mine: First, the status for supplier S2 is shown as 30 instead of 10; second, the city for part P3 is shown as Paris instead of Oslo.

- *Suppliers:* Relvar S denotes suppliers.[5] Each supplier has one supplier number (SNO), unique to that supplier; one name (SNAME), not necessarily unique (though the SNAME values in Figure 1-1 do happen to be unique); one status value (STATUS), representing some kind of ranking or preference level among suppliers; and one location (CITY).
- *Parts:* Relvar P denotes parts (more accurately, kinds of parts). Each kind of part has one part number (PNO), which is unique; one name (PNAME), not necessarily unique; one color (COLOR); one weight (WEIGHT); and one location where parts of that kind are stored (CITY).
- *Shipments:* Relvar SP denotes shipments—it shows which parts are supplied, or shipped, by which suppliers. Each shipment has one supplier number (SNO), one part number (PNO), and one quantity (QTY). Also, I assume for the sake of the example that there's at most one shipment at any given time for a given supplier and a given part, and so each shipment has a supplier-number / part-number combination that's unique.

S

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

SP

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

P

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

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

[5]If you don't know what a relvar is, for now you can just take it to be a table in the usual database sense. See Chapter 2 for further explanation.

Keys

Before going any further, I need to review the familiar concept of *keys*, in the relational sense of that term. First of all, as I'm sure you know, every relvar has at least one *candidate* key. A candidate key is basically just a unique identifier; in other words, it's a combination of attributes—often but not always a "combination" consisting of just a single attribute—such that every tuple in the relvar has a unique value for the combination in question. For example, with respect to the database of Figure 1-1:

- Every supplier has a unique supplier number and every part has a unique part number, so {SNO} is a candidate key for S and {PNO} is a candidate key for P.
- As for shipments, given the assumption that there's at most one shipment at any given time for a given supplier and a given part, {SNO,PNO} is a candidate key for SP.

Note the braces, by the way; to repeat, candidate keys are always combinations, or *sets*, of attributes (even when the set in question contains just one attribute), and the conventional representation of a set on paper is as a commalist of elements enclosed in braces.

This is the first time I've mentioned the term *commalist*, which I'll be using from time to time in the pages ahead. It can be defined as follows. Let *xyz* be some syntactic construct (for example, "attribute name"); then the term *xyz commalist* denotes a sequence of zero or more *xyz*'s in which each pair of adjacent *xyz*'s is separated by a comma (blank spaces appearing immediately before or after any comma are ignored). For example, if *A*, *B*, and *C* are attribute names, then the following are all attribute name commalists:

```
A , B , C

C , A , B

B

A , C
```

So too is the empty sequence of attribute names.

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

Next, as I'm sure you also know, a *primary* key is a candidate key that's been singled out in some way for some kind of special treatment. Now, if the relvar in question has just one candidate key, then it doesn't make any real difference if we call that key primary. But if the relvar has two or more candidate keys, then it's usual to choose one of them to be primary, meaning it's somehow "more equal than the others." Suppose, for example, that suppliers always have both a unique supplier number and a unique supplier name, so that {SNO} and {SNAME} are both candidate keys. Then we might choose {SNO}, say, to be the primary key.

Observe now that I said it's *usual* to choose a primary key. Indeed it is usual—but it's *not* 100% necessary. If there's just one candidate key, then there's no choice and no problem; but if there are two or more, then having to choose one and make it primary smacks a little bit of arbitrariness, at least to me. (Certainly there are situations where there don't seem to be any really good reasons for making such a choice. There might even be good reasons for not doing so. Appendix C elaborates on such matters.) For reasons of familiarity, I'll usually follow the primary key discipline myself in this book—and in pictures like Figure 1-1 I'll indicate primary key attributes by double underlining—but I want to stress the fact that it's really candidate keys, not primary keys, that are significant from a relational point of view, and indeed from a design theory point of view as well. Partly for such reasons, from this point forward I'll use the term *key*, unqualified, to mean any candidate key, regardless of whether the candidate key in question has additionally been designated as primary. (In case you were wondering, the special treatment enjoyed by primary keys over other candidate keys is mainly syntactic in nature, anyway; it isn't fundamental, and it isn't very important.)

More terminology: First, a key involving two or more attributes is said to be *composite* (and a noncomposite key is sometimes said to be *simple*). Second, if a given relvar has two or more keys and one is chosen as primary, then the others are sometimes said

to be *alternate* keys (see Appendix C). Third, a *foreign* key is a combination, or set, of attributes *FK* in some relvar *R2* such that each *FK* value is required to be equal to some value of some key *K* in some relvar *R1* (*R1* and *R2* not necessarily distinct).[6] With reference to Figure 1-1, for example, {SNO} and {PNO} are both foreign keys in relvar SP, corresponding to keys {SNO} and {PNO} in relvars S and P, respectively.

The Place of Design Theory

As I said in the preface, by the term *design* I mean logical design, not physical design. Logical design is concerned with what the database looks like to the user (which means, loosely, what relvars exist and what constraints apply to those relvars); physical design, by contrast, is concerned with how a given logical design maps to physical storage.[7] And the term *design theory* refers specifically to logical design, not physical design—the point being that physical design is necessarily dependent on aspects (performance aspects in particular) of the target DBMS, whereas logical design is, or should be, DBMS independent. Throughout this book, then, the unqualified term *design* should be understood to mean logical design specifically, unless the context demands otherwise.

Now, design theory as such isn't part of the relational model; rather, it's a separate theory that builds on top of that model. (It's appropriate to think of it as part of relational theory in general, but it's not, to repeat, part of the relational model per se.) Thus, design concepts such as further normalization are themselves based on more fundamental notions—e.g., the projection and join operators of the relational algebra—that *are* part of the relational model. (All of that being said, however, it could certainly be argued that design theory is a *logical consequence* of the relational model, in a sense. In other words, I think it would be inconsistent to agree with the relational model in general but not to agree with the design theory that's based on it.)

The overall objective of logical design is to achieve a design that's (a) hardware independent, for obvious reasons; (b) operating system and DBMS independent, again for obvious reasons; and finally, and perhaps a little controversially, (c) *application* independent (in other words, we're concerned primarily with what the data is, rather than with how it's going to be used). Application independence in this sense is desirable

[6]This definition is deliberately a little simplified (though it's good enough for present purposes). A better one can be found in Chapter 3, also in *SQL and Relational Theory*.

[7]Be aware, however, that other writers (a) use those terms *logical design* and *physical design* to mean something else and (b) use other terms to mean what I mean by those terms. *Caveat lector.*

for the very good reason that it's normally—perhaps always—the case that not all uses to which the data will be put are known at design time; thus, we want a design that'll be robust, in the sense that it won't be invalidated by the advent of application requirements that weren't foreseen at the time of the original design. Observe that one important consequence of this state of affairs is that we aren't (or at least shouldn't be) interested in making design compromises for physical performance reasons. Design theory in general, and individual database designs in particular, should never be driven by mere performance considerations.

Back to design theory as such. As we'll see, that theory includes a number of formal theorems, theorems that provide practical guidelines for designers to follow. So if you're a designer, you need to be familiar with those theorems. Let me quickly add that I don't mean you need to know how to prove the theorems in question (though in fact the proofs are often quite simple); what I mean is, you need to know what the theorems say—i.e., you need to know the results—and you need to be prepared to apply those results. That's the nice thing about theorems: Once somebody's proved them, then their results become available for anybody to use whenever they need to.

Now, it's sometimes claimed, not entirely unreasonably, that all design theory really does is *bolster up your intuition*. What do I mean by this remark? Well, consider the suppliers-and-parts database. The obvious design for that database is the one illustrated in Figure 1-1; I mean, it's "obvious" that three relvars are necessary, that attribute STATUS belongs in relvar S, that attribute COLOR belongs in relvar P, that attribute QTY belongs in relvar SP, and so on. But why exactly are these things obvious? Well, suppose we try a different design; for example, suppose we move the STATUS attribute out of relvar S and into relvar SP (intuitively the wrong place for it, of course, since status is a property of suppliers, not shipments). Figure 1-2 on the next page shows a sample value for this revised shipments relvar, which I'll call STP to avoid confusion:[8]

[8]For obvious reasons, throughout this book I use T, not S, as an abbreviation for STATUS.