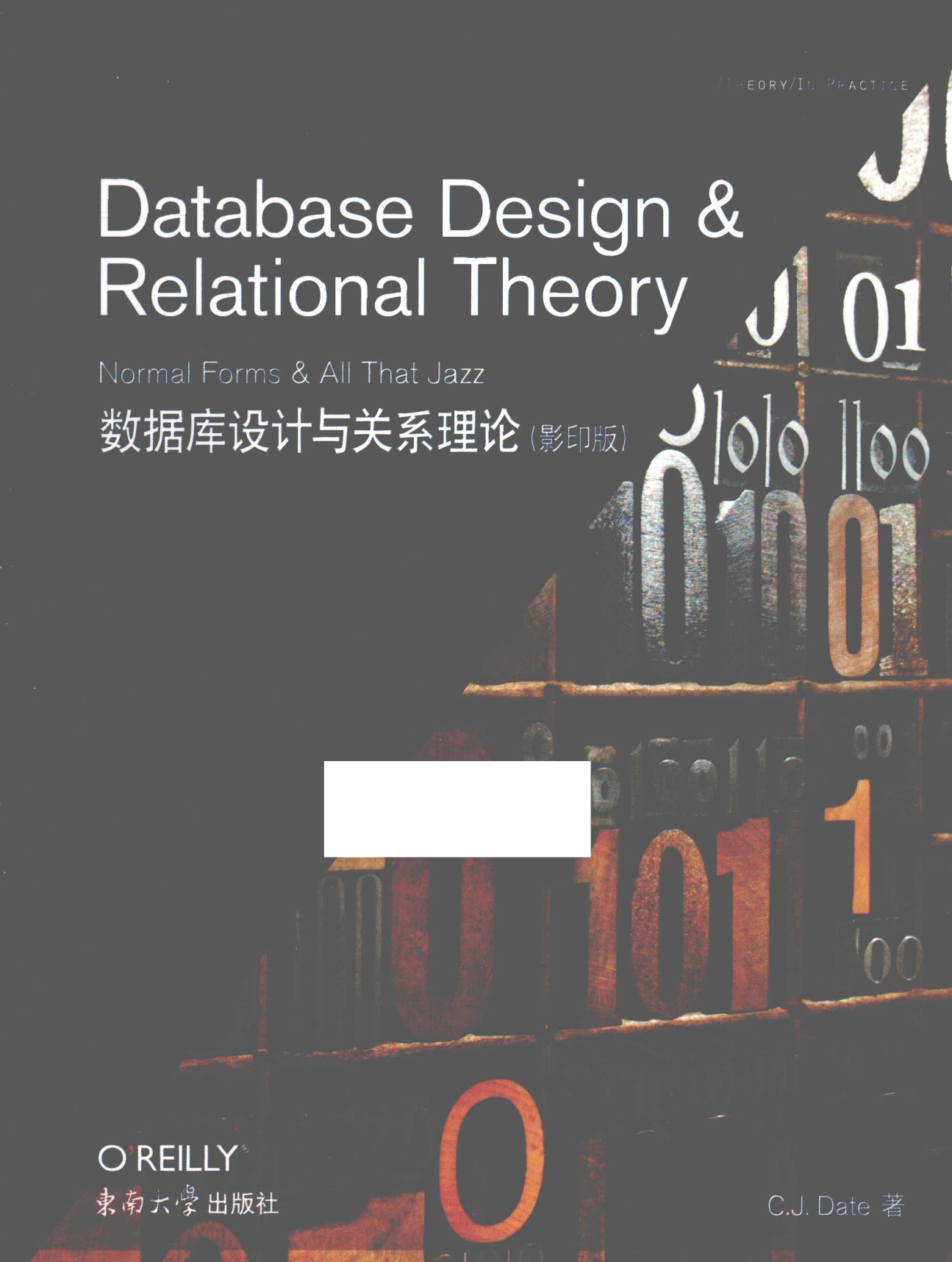
THEORY/IN PRACTICE

Database Design & Relational Theory

Normal Forms & All That Jazz

数据库设计与关系理论（影印版）



O'REILLY®
東南大學出版社

C.J. Date 著

数据库设计与关系理论（影印版）

Database Design and Relational Theory

C.J. Date

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

数据库设计与关系理论：英文/（英）戴特（Data, C.J.）著.—影印本.—南京：东南大学出版社，2013.1

书名原文：Database Design and Relatinal Theory

ISBN 978-7-5641-3890-5

Ⅰ. ①数… Ⅱ. ①戴… Ⅲ. ①数据库系统－英文 Ⅳ. ①TP311.13

中国版本图书馆CIP数据核字（2012）第273589号

江苏省版权局著作权合同登记

图字：10-2012-158号

数据库设计与关系理论（影印版）

出版发行：东南大学出版社
地　　址：南京四牌楼2号　　邮编：210096
出 版 人：江建中
网　　址：http://www.seupress.com
电子邮件：press@seupress.com
印　　刷：扬中市印刷有限公司
开　　本：787毫米×980毫米　16开本
印　　张：17.25
字　　数：338千字
版　　次：2013年1月第1版
印　　次：2013年1月第1次印刷
书　　号：ISBN 978-7-5641-3890-5
定　　价：49.00元（册）

本社图书若有印装质量问题，请直接与营销部联系。电话（传真）：025-83791830

Contents

Preface xi

PART I SETTING THE SCENE 1

Chapter 1 Preliminaries 3

Some quotes from the literature 3
A note on terminology 5
The running example 6
Keys 7
The place of design theory 8
Aims of this book 11
Concluding remarks 12
Exercises 12

Chapter 2 Prerequisites 15

Overview 15
Relations and relvars 16
Predicates and propositions 18
More on suppliers and parts 20
Exercises 22

PART II FUNCTIONAL DEPENDENCIES, BOYCE/CODD NORMAL FORM, AND RELATED MATTERS 25

Chapter 3 Normalization: Some Generalities 27

Normalization serves two purposes 29
Update anomalies 31
The normal form hierarchy 32
Normalization and constraints 34
Concluding remarks 35
Exercises 36

Chapter 4 FDs and BCNF (Informal) 37

First normal form 37
Functional dependencies 40
Keys revisited 42
Second normal form 43
Third normal form 45
Boyce/Codd normal form 45
Exercises 47

Chapter 5 **FDs and BCNF (Formal) 49**

Preliminary definitions 49
Functional dependencies 50
Boyce/Codd normal form 52
Heath's Theorem 54
Exercises 56

Chapter 6 **Preserving FDs 59**

An unfortunate conflict 60
Another example 63
... And another 64
... And still another 66
A procedure that works 67
Identity decompositions 71
More on the conflict 72
Independent projections 73
Exercises 74

Chapter 7 **FD Axiomatization 75**

Armstrong's axioms 75
Additional rules 76
Proving the additional rules 78
Another kind of closure 79
Exercises 80

Chapter 8 **Denormalization 83**

"Denormalize for performance"? 83
What does denormalization mean? 84
What denormalization isn't (I) 86
What denormalization isn't (II) 88
Denormalization considered harmful (I) 90
Denormalization considered harmful (II) 91
A final remark 92
Exercises 92

PART III **JOIN DEPENDENCIES, FIFTH NORMAL FORM, AND RELATED MATTERS 95**

Chapter 9 **JDs and 5NF (Informal) 97**

Join dependencies—the basic idea 98
A relvar in BCNF and not 5NF 100
Cyclic rules 103
Concluding remarks 104
Exercises 105

Chapter 10 **JDs and 5NF (Formal) 107**

Join dependencies 107
Fifth normal form 109
JDs implied by keys 110
A useful theorem 113
FDs aren't JDs 114
Update anomalies revisited 114
Exercises 116

Chapter 11 **Implicit Dependencies 117**

Irrelevant components 117
Combining components 118
Irreducible JDs 119
Summary so far 121
The chase algorithm 123
Concluding remarks 127
Exercises 127

Chapter 12 **MVDs and 4NF 129**

An introductory example 129
Multivalued dependencies (informal) 131
Multivalued dependencies (formal) 132
Fourth normal form 133
Axiomatization 134
Embedded dependencies 135
Exercises 136

Chapter 13 **Additional Normal Forms 139**

Equality dependencies 139
Sixth normal form 141
Superkey normal form 143
Redundancy free normal form 144
Domain-key normal form 149
Concluding remarks 150
Exercises 152

PART IV **ORTHOGONALITY 155**

Chapter 14 ***The Principle of Orthogonal Design*** **157**

Two cheers for normalization 157
A motivating example 159
A simpler example 160
Tuples vs. propositions 163
The first example revisited 166
The second example revisited 168
The final version 168
A clarification 168
Concluding remarks 170
Exercises 171

PART V **REDUNDANCY 173**

Chapter 15 **We Need More Science 175**

A little history 177
Database design is predicate design 178
Example 1 180
Example 2 181
Example 3 181
Example 4 181
Example 5 182
Example 6 183
Example 7 185
Example 8 187
Example 9 188
Example 10 189
Example 11 190
Example 12 190

Managing redundancy 191
Refining the definition 193
Concluding remarks 200
Exercises 200

APPENDIXES 201

Appendix A **Primary Keys Are Nice but Not Essential 203**

Arguments in favor of the PK:AK distinction 204
Relvars with more than one key 206
The invoices and shipments example 208
One primary key per entity type? 211
The applicants and employees example 212
Concluding remarks 214

Appendix B **Redundancy Revisited 215**

Appendix C **Historical Notes 219**

Appendix D **Answers to Exercises 223**

Chapter 1 223
Chapter 2 224
Chapter 3 227
Chapter 4 227
Chapter 5 232
Chapter 6 235
Chapter 7 237
Chapter 8 240
Chapter 9 242
Chapter 10 244
Chapter 11 245
Chapter 12 247
Chapter 13 250
Chapter 14 253
Chapter 15 253

Index 255

Part I

SETTING THE SCENE

This part of the book consists of two 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 subtitle *Normal Forms and All That Jazz*. Clearly some explanation is needed! First of all, of course, I'm talking about design theory, and everybody knows normal forms are a major component of that theory; hence the first part of my subtitle. But there's more to the 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 is riddled with terms and concepts that seem to be difficult 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, or impression, 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 done in practice. Let me be clear: Nobody could, or should, claim that designing databases is easy. But a sound knowledge of 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 design theory. There's just no alternative: at least, not if you want to claim to be a 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 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 enlightening, perhaps, but entertaining (?)—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. (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:

CUSTNO	PHONENO1	PHONENO2	...

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 one:

- 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. And while it's true that 1NF does mean, loosely, "no repeating groups," a repeating group is *not* "when you repeat the same basic attribute over and over again." (What it really is I'll explain in Chapter 4, when I explain what 1NF really is.)

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 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 tend to 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*:[1]

> I'm generally sympathetic to the idea of using more user friendly terms, if they can help make the ideas more palatable. In the case at hand, however, it seems to me that, regrettably, they don't make the ideas more palatable; instead, they distort them, and in fact do the cause of genuine understanding a grave disservice. The truth is, a relation is *not* a table, a tuple is *not* a row, and an attribute is *not* a column. And while it might be acceptable to pretend otherwise in informal contexts—indeed, I often do exactly that myself—I would argue that it's acceptable only if we all understand that the more user friendly terms are just an approximation to the truth and fail overall to capture the essence of what's really going on. To put it another way, if you do understand the true state of affairs, then judicious use of the user friendly terms can be a good idea; but in order to learn and appreciate that true state of affairs in the first place, you really do need to come to grips with the 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.[2] The first and more fundamental one is this:

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

This is the meaning we have in mind when we talk about the relational model in particular: The data structures in the relational model are relations, of course, and the data operators are the relational operators projection, join, and

---

[1] 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).

[2] 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.

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

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

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

- A data model in the first sense is like a programming language, whose constructs can be used to solve many specific problems but in and of themselves have no direct connection with any such specific problem.

- A data model in the second sense is like a specific program written in that language—it uses the facilities provided by the model, in the first sense of that term, to solve some specific problem.

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 probably know, most writings on database design, especially if their focus is on pragma rather than the underlying theory, use the term "model," or "data model," exclusively in the second sense. But—*please note 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 believe that using essentially the same example in a variety of different books and publications can help, not hinder, learning.) Sample values are shown in Fig. 1.1.[3] To elaborate:

- *Suppliers:* Relvar S denotes suppliers.[4] Each supplier has one supplier number (SNO), unique to that supplier; one name (SNAME), not necessarily unique (though the SNAME values in Fig. 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).

---

[3] For reasons that will become clear later, the values shown in Fig. 1.1 differ in two small respects from those in other books of mine: The status for supplier S2 is shown as 30 instead of 10, and the city for part P3 is shown as Paris instead of Oslo.

[4] 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.

- *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 one 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

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

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

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 Fig. 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 one 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. *Note:* The useful term *commalist* can be defined as follows: Let *xyz* be some syntactic construct (for example, "attribute name"). Then the term *xyz commalist* denotes a sequence of zero

or more *xyz*'s in which each pair of adjacent *xyz*'s is separated by a comma (as well as, optionally, one or more spaces before or after the comma or both).

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 percent necessary. If there's just one candidate key, then there's no choice and no problem; but if there are two or more, then having to choose one and make it primary smacks a little bit of arbitrariness, at least to me. (Certainly there are situations where there don't seem to be any good reasons for making such a choice. There might even be good reasons for not doing so. Appendix A elaborates on such matters.) For reasons of familiarity, I'll usually follow the primary key discipline myself in this book—and in pictures like Fig. 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 A). 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).[5] With reference to Fig. 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

To repeat something 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.[6] 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, barring explicit statements to the contrary.

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, it could certainly be argued that design theory is a *logical consequence* of the

---

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

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

relational model, at least in part. In other words, 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 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 should never be driven by 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 have to know how to prove those theorems (though in fact the proofs are often quite simple); what I mean is, you have to know what the theorems say—i.e., you have to know the results—and you have to be prepared to apply those results. That's the nice thing about theorems: Once somebody's proved them, 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 Fig. 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; suppose we move the STATUS attribute out of relvar S, for example, and into relvar SP (intuitively the wrong place for it, since status is a property of suppliers, not shipments). Fig. 1.2 below shows a sample value for this revised shipments relvar, which I'll call STP to avoid confusion:[7]

STP

SNO	STATUS	PNO	QTY
S1	20	P1	300
S1	20	P2	200
S1	20	P3	400
S1	20	P4	200
S1	20	P5	100
S1	20	P6	100
S2	30	P1	300
S2	30	P2	400
S3	30	P2	200
S4	20	P2	200
S4	20	P4	300
S4	20	P5	400

Fig. 1.2: Relvar STP--sample value

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