/Theory/In/Practice

O'REILLY®

Database In Depth

Relational Theory for Practitioners

C. J. Date

CHAPTER ONE

Introduction

Professionals in any discipline need to know the foundations of their field. So if you're a database professional, you need to know the relational model, because that model is the foundation (or a huge part of the foundation, anyway) of the database field in particular. Now, every course in database management, be it academic or commercial, does at least pay lip service to the idea of teaching the relational model—but most of that teaching seems to be done very badly, if results are anything to go by. The relational model certainly isn't very well or very widely understood in the database community at large. Here are some possible reasons for this state of affairs:

- The model is taught in a vacuum. That is, for beginners at least, it's hard to see the relevance of the material, or it's hard to understand the problems it's meant to solve, or both.
- The instructors themselves don't fully understand or appreciate the significance of the material.
- (Most likely in practice.) The model as such isn't taught at all—the SQL language or some specific dialect of that language, such as the Oracle dialect, is taught instead.

So this book is aimed at database professionals, especially commercial database practitioners, who have had some exposure to the relational model but don't know as

much about it as they ought to. It's definitely *not* meant for beginners; however, it isn't a refresher course, either. To be more specific: I'm sure you know something about SQL, but—and I apologize if my tone is somewhat offensive here—if your knowledge of the relational model derives only from your knowledge of SQL, then I'm afraid you won't know the relational model as well as you should, and you'll probably know "some things that ain't so."

> **NOTE**
> **SQL ≠ the relational model!**

Here by way of illustration are some relational issues that SQL isn't too clear on (to put it mildly):

- What databases, relations, and tuples really are
- The difference between relations and types
- The difference between relation values and relation variables
- The relevance of predicates and propositions
- The legitimacy of relation-valued attributes
- The crucial role of integrity constraints

and so on (this isn't an exhaustive list). All of these issues, and of course many others, are addressed in this book.

I say again: if your knowledge of the relational model derives only from your knowledge of SQL, then you might know "some things that ain't so." One unfortunate consequence of this state of affairs is that you might find, in reading this book, that there are some things you have to unlearn—and unlearning something is notoriously hard to do. And a related point…I'd like to recommend, politely, that you not skip the discussion of some topic because you think you're thoroughly familiar with that topic already. For example, are you *sure* you know exactly what a key is, in relational terms? Or a join?

A Remark on Terminology

In that list of relational issues in the foregoing section, you probably noticed right away that I used the formal terms *relation*, *tuple*,* and *attribute*. SQL doesn't use these terms, of course—it uses the more "user-friendly" terms *table*, *row*, and *column* instead. And I'm generally sympathetic to the idea of 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

* Usually pronounced to rhyme with couple.

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've done so myself, in many of my books and other writings—I would argue that it's acceptable only if we all understand that the more user-friendly terms are just an approximation to the truth and fail overall to capture the essence of what's really going on. To put it another way: if you do understand the true state of affairs, then judicious use of the user-friendly terms can be a good idea; but in order to learn and appreciate that true state of affairs in the first place, you really do need to come to grips with the more formal terms. In this book, therefore, I'll use those more formal terms most of the time—and of course I'll give precise definitions for them when we need them (mostly not in this first chapter, though, where I'm just trying to lay a certain amount of elementary groundwork).

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

Talking of SQL, by the way, *please note that I use the term SQL to mean the standard version of that language exclusively,* not some product-specific dialect—barring explicit statements to the contrary, of course. (I did mention this point in the preface, but I know that few people actually read prefaces.) In particular, my criticisms of SQL apply to the standard version specifically. Thus, if some particular criticism happens not to apply to your own favorite product, well, good, I'm glad to hear it (and bully for you).

Principles, Not Products

It's worth taking a few moments to examine the question of why, as I claimed earlier, you as a database professional need to know the relational model. The reason is that the relational model isn't product-specific; rather, it is concerned with *principles*. What do I mean by *principles?* Well, here's a definition (from *Chambers Twentieth Century Dictionary*):

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

The point about principles is this: they *endure*. By contrast, products and technologies (and the SQL language, come to that) change all the time—but principles don't. For example, suppose you know Oracle; in fact, suppose you're an expert on Oracle. But if Oracle is *all* you know, then your knowledge is not necessarily transferable to, say, a DB2 or SQL Server environment (it might even get in the way of your making progress in that new environment). But if you know the underlying principles—in other words, if you know the relational model—then you have knowledge and skills that *will* be transferable: knowledge and skills that you'll be able to apply in *every* environment and that will never be obsolete.

In this book, therefore, we'll be concerned with principles, not products, and foundations, not fads. Of course, I realize that sometimes you do have to make compromises and trade-offs in the real world. For one example, sometimes you might have good pragmatic reasons for not designing the database in the theoretically optimal way (an issue I discuss in Chapter 7). For another, consider SQL once again. Although it's certainly possible to use SQL relationally (for the most part, at any rate), sometimes you'll find—because existing implementations are so far from perfect—that there are severe performance penalties for doing so…in which case you might more or less be forced into doing something not "truly relational" (like writing a query in some weird and unnatural way in order to get the implementation to use an index). However, I believe very firmly that you should always make such compromises and trade-offs from *a position of conceptual strength*. That is:

- You should understand what you're doing when you do have to make such a compromise.
- You should know what the theoretically correct situation is, and you should have very good reasons for departing from it.
- You should document those reasons, too, so that if they go away at some future time (for example, because a new release of the product you're using does a better job in some respect), then it might be possible to back off from the original compromise.

The following quote—which is attributed to Leonardo da Vinci (1452–1519) and is thus some 500 years old!—sums up the situation admirably:

> Those who are enamored of practice without theory are like a pilot who goes into a ship without rudder or compass and never has any certainty where he is going. *Practice should always be based on a sound knowledge of theory.*

(OK, I added the italics.)

A Review of the Original Model

You're a database professional, so you already have some familiarity with the relational model. The purpose of this section is to serve as a kickoff point for our subsequent discussions; it reviews some of the most basic aspects of that model as originally defined. Note the qualifier "as originally defined"! One widespread misconception about the relational model is that it's a totally static thing. It's not. It's like mathematics in that respect: mathematics too is not a static thing but changes over time. In fact, the relational model can itself be seen as a small branch of mathematics; as such, it evolves over time as new theorems are proved and new results discovered. What's more, those new contributions can be made by anyone who's competent to do so. Like mathematics again, the relational model, though originally invented by one man, has become a community effort and now belongs to the world.

By the way, in case you don't know, that one man was E. F. Codd, at the time a researcher at IBM.* It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into the field of database management, which was all too deficient in such qualities prior to that time. His original definition of the relational model appeared in an IBM Research Report in 1969, and I'll have a little more to say about that paper in Appendix B.

Structural Features

The original model had three major components—structure, integrity, and manipulation—and I'll briefly describe each in turn. Please note right away, however, that all of the "definitions" I'll be giving are very loose; I'll make them more precise as and when appropriate in later chapters.

First of all, then, structure. The principal structural feature is, of course, the relation itself, and as everybody knows it's common to picture relations as tables on paper (see Figure 1-1 for a self-explanatory example). Relations are defined over *types* (also known as *domains*); a type is basically a conceptual pool of values from which actual attributes in actual relations take their actual values. With reference to the simple departments-and-employees database illustrated in Figure 1-1, for example, there might be a type called DNO ("department numbers"), which is the set of all valid department numbers. The attribute called DNO in the DEPT relation and the attribute called DNO in the EMP relation then would each contain values that are taken from that conceptual pool. (By the way, it isn't necessary for attributes to have the same name as the corresponding type, and often they won't. We'll see plenty of counterexamples later.)

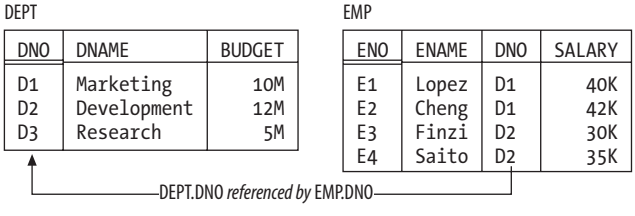


FIGURE 1-1. *The departments-and-employees database—sample values*

As I've said, tables like those in Figure 1-1 depict *relations: n*-ary relations, to be precise. An *n*-ary relation can be pictured as a table with *n* columns; the columns in the picture correspond to *attributes* of the relation and the rows correspond to *tuples*. Also, the value *n* can be any nonnegative integer. A 1-ary relation is said to be *unary;* a 2-ary relation, *binary;* a 3-ary relation, *ternary;* and so on.

* E for Edgar and F for Frank—but he always signed with his initials. To his friends, among whom I was proud to count myself, he was Ted.

The relational model also supports various kinds of *keys*. To begin with, every relation has at least one *candidate key*.* A candidate key is just a unique identifier; in other words, it's a combination of attributes—often, but not always, a "combination" involving just one attribute—such that every tuple in the relation has a unique value for the combination in question. In Figure 1-1, for example, every department has a unique department number and every employee has a unique employee number, so we can say that {DNO} is a candidate key for DEPT and {ENO} is a candidate key for EMP. Note the braces, by the way: candidate keys are always *combinations,* or *sets,* of attributes—even when the set contains just one attribute—and braces are conventionally used to enclose sets of things.

Next, a *primary key* is a candidate key that's been singled out for special treatment in some way. If a given relation has just one candidate key, then it obviously makes no real difference if we say it's the primary key. But if the relation has two or more candidate keys, then we're supposed to choose one of them as primary, meaning it's somehow "more equal than the others." Suppose, for example, that every employee has a unique employee number *and* a unique employee name, so that {ENO} and {ENAME} are both candidate keys for EMP. Then we might choose {ENO}, say, to be the primary key.

Notice I said we're *supposed* to choose a primary key. If there's just one candidate key, then there's no choice and no problem. But if there are two or more, then choosing one and making it primary smacks a little bit of arbitrariness (at least to me); certainly there are situations where there don't seem to be any good reasons for making such a choice. In this book, I usually *will* follow the primary key discipline—and in pictures like Figure 1-1 I'll mark primary key attributes by double underlining—but I stress the point that it's really candidate keys, not primary keys, that are significant from a relational point of view. Partly for this reason, from this point forward I'll use the term *key*, unqualified, to mean a candidate key specifically. (In case you were wondering, the "special treatment" enjoyed by primary keys over other candidate keys is mainly syntactic in nature, anyway; it isn't fundamental, and it isn't very important.)

Finally, a *foreign key* is a set of attributes in one relation whose values are required to match the values of some candidate key in some other relation (or possibly in the same relation). With reference to Figure 1-1, for example, {DNO} is a foreign key in EMP whose values are required to match values of the candidate key {DNO} in DEPT (as I've tried to suggest by means of a suitably labeled arrow in the figure). By *required to match*, I mean that if, for example, EMP includes a tuple in which DNO has the value D2, then DEPT had better also include a tuple in which DNO has the value D2; otherwise, EMP would show some employee as being in a nonexistent department, and the database wouldn't be "a faithful model of reality."

* Strictly speaking, this sentence should read "Every *relvar* has at least one candidate key" (see the section "Relations Versus Relvars," later in this chapter). Similar remarks apply at various places elsewhere in this chapter, too (see Exercise 1-1 at the end of the chapter).

Integrity Features

An *integrity constraint* (*constraint* for short) is basically just a boolean expression that must evaluate to TRUE. In the case of departments and employees, for example, we might have a constraint to the effect that SALARY values must be greater than zero. Now, any given database will be subject to numerous constraints, but those constraints will necessarily be expressed in terms of the relations in that particular database and will be specific to that database. By contrast, the relational model (at least as originally formulated) includes two *generic* integrity rules—generic in the sense that they apply to every database, loosely speaking. One has to do with primary keys and the other with foreign keys:

Entity integrity
: Primary key attributes don't permit nulls.

Referential integrity
: There mustn't be any unmatched foreign key values.

Let me explain the second first. By the term *unmatched foreign key value,* I mean a foreign key value for which there doesn't exist an equal value of the corresponding candidate key. Thus, for example, the departments-and-employees database would be in violation of the referential integrity rule if it included an EMP tuple with, say, a DNO value of D2 but no corresponding DEPT tuple. So the referential integrity rule simply spells out the semantics of foreign keys; the name *referential integrity* derives from the fact that any given foreign key value can be regarded as a *reference* to the tuple with that same value for the corresponding candidate key. In effect, therefore, the rule just says: "If *B* references *A,* then *A* must exist."

As for the entity integrity rule, well, here I have a problem. The fact is, I reject the concept of "nulls" entirely; that is, it is my very strong opinion that *nulls have no place in the relational model*. (Codd thought otherwise, obviously, but I have strong reasons for taking the position I do.) In order to explain the entity integrity rule, therefore, I need to suspend disbelief, as it were (at least for the time being), but please understand that I'll be revisiting the whole issue of nulls in Chapter 3.

In essence, then, a null is a "marker" that means *value unknown* (crucially, it's not itself a value; it is, to repeat, a *marker,* or *flag*). For example, suppose we don't know employee E2's salary. Then, instead of entering some real SALARY value in the tuple for that employee in relation EMP—we *can't* enter a real value, by definition, precisely because we don't know what that value should be—we *mark* the SALARY position within that tuple as null:

ENO	ENAME	DNO	SALARY
E2	Cheng	D1	

As you can see, this tuple contains *nothing at all* in the SALARY position. In this book I'll use shading as just shown to highlight such empty positions; you can think of that shading as constituting the null "marker" or flag.

In terms of relation EMP, then, the entity integrity rule says, loosely, that an employee might have an unknown name, department, or salary, but *not* an unknown employee number—because if the employee number were unknown, we wouldn't even know which employee (that is, which "entity") we were talking about.

That's all I want to say about nulls for now. Forget about them until Chapter 3.

Manipulative Features

The manipulative part of the model consists of:

- A set of relational operators, such as difference (or MINUS), collectively called the *relational algebra*, together with
- A *relational assignment* operator that allows the value of some relational expression, such as *r* MINUS *s* (where *r* and *s* are relations), to be assigned to some relation.

The relational assignment operator is fundamentally how updates are done in the relational model,* and I'll have more to say about it later, in the section "Relations Versus Relvars." As for the relational algebra, it consists of a set of operators that allow "new" relations to be derived from "old" ones (speaking very loosely). More precisely, each operator takes at least one relation as input and produces another relation as output; for example, difference (or MINUS) takes two relations as input and "subtracts" one from the other to derive another relation as output. And it's very important that the output is another relation: that's the well-known *closure* property of the relational algebra. The closure property is what lets us write *nested relational expressions*; since the output from every operation is the same kind of thing as the input, the output from one operation can become the input to another—meaning, for example, that we can take the difference between *r* and *s*, feed the result as input to a union with some relation *u*, feed that result as input to an intersection with some relation *v*, and so on.

Now, any number of operators can be defined that fit the simple definition of "at least one relation in, exactly one relation out." In the following list I'll briefly describe what are usually thought of as the original eight operators (essentially the ones Codd defined in his earliest papers); in Chapter 5 I'll introduce a number of additional operators and describe them in more detail. Figure 1-2 is a pictorial representation of the original eight operators. *Note:* If you're unfamiliar with any of these operators—especially divide!—and find the following brief descriptions hard to follow, don't worry about it; I'll be going into much more detail, with lots of examples, later (mostly in Chapter 5).

* I follow convention throughout this book in using the generic term "update" to refer to the INSERT, DELETE, and UPDATE (and assignment) operators considered collectively. When I want to refer to the UPDATE operator specifically, I'll set it in all caps as just shown.

Restrict

Returns a relation containing all tuples from a specified relation that satisfy a specified condition. For example, we might restrict relation EMP to just the tuples where the DNO value is D2.

Project

Returns a relation containing all (sub)tuples that remain in a specified relation after specified attributes have been removed. For example, we might project relation EMP on just the ENO and SALARY attributes.

Product

Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations. Product is also known variously as *cartesian product, cross product, cross join,* and *cartesian join* (in fact, it is just a special case of join, as we'll see in Chapter 5).

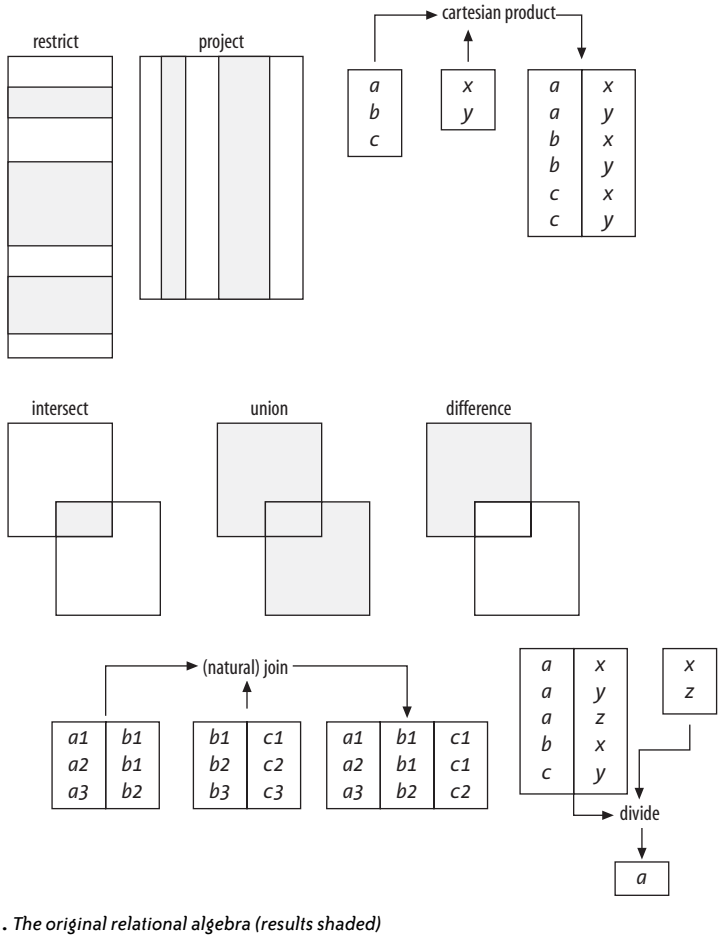


FIGURE 1-2. *The original relational algebra (results shaded)*

Intersect

Returns a relation containing all tuples that appear in both of two specified relations. (Actually, intersect also is a special case of join.)

Union

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

Difference

Returns a relation containing all tuples that appear in the first and not the second of two specified relations.

Join

Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations, such that the two tuples contributing to any given result tuple have a common value for the common attributes of the two relations (and that common value appears just once, not twice, in that result tuple).

> **NOTE**
>
> **This kind of join was originally called the *natural* join. Since natural join is far and away the most important kind, however, it's become standard practice to take the unqualified term *join* to mean the natural join specifically, and I'll follow that practice in this book.**

Divide

Takes two relations, one binary and one unary, and returns a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.

One last point to close this subsection: as you probably know, there's also something called the *relational calculus*. The relational calculus can be regarded as an alternative to the relational algebra; that is, instead of saying the manipulative part of the relational model consists of the relational algebra (plus relational assignment), we can equally well say it consists of the relational calculus (plus relational assignment). The two are equivalent and interchangeable, in the sense that for every algebraic expression there's a logically equivalent expression of the calculus and vice versa. I'll have a little more to say about the calculus in Appendix A.

The Running Example

I'll finish up this brief review by introducing the example I'll use as the basis for most if not all of the discussions in the rest of the book: the well-known suppliers-and-parts database (see Figure 1-3). To elaborate:

Suppliers

Relation S denotes suppliers (more accurately, suppliers under contract). Each supplier has one supplier number (SNO), which is unique to that supplier (and so {SNO} is the

primary key); one name (SNAME), not necessarily unique (though the SNAME values in Figure 1-3 do happen to be unique); one rating or status value (STATUS); and one location (CITY).

Parts

Relation P denotes parts (more accurately, kinds of parts). Each kind of part has one part number (PNO), which is unique (so {PNO} is the primary key); one name (PNAME); one color (COLOR); one weight (WEIGHT); and one location where parts of that kind are stored (CITY).

Shipments

Relation SP denotes shipments (it shows which parts are supplied by which suppliers). Each shipment has one supplier number (SNO), one part number (PNO), and one quantity (QTY). For the sake of the example, I assume there's at most one shipment at any given time for a given supplier and a given part (and so {SNO,PNO} is the primary key; also, {SNO} and {PNO} are both foreign keys, matching the primary keys of S and P, respectively). Note that the database shown in Figure 1-3 includes one supplier, supplier S5, with no shipments at all.

S

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

SP

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

P

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

FIGURE 1-3. *The suppliers-and-parts database—sample values*

Model Versus Implementation

Before going any further, there's one very important point I need to explain, because it underpins everything else in this book. The relational model is, of course, a *data* model. Unfortunately, however, this latter term has two quite distinct meanings in the database world. The first and more fundamental meaning is this:

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

This is the meaning we have in mind when we talk about the relational model in particular. And, armed with this definition, we can usefully (and importantly) go on to

distinguish a data model in this first sense from its *implementation*, which can be defined as follows:

> *Definition:* An *implementation* of a given data model is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

I'll illustrate these definitions in terms of the relational model specifically. First, and obviously enough, the concept of *relation* is itself part of the model: users have to know what relations are, they have to know they're made up of tuples and attributes, they have to know how to interpret them, and so on. All that is part of the model. But they don't have to know how relations are physically stored on the disk, or how individual data values are physically encoded, or what indexes or other access paths exist; all that is part of the implementation, not part of the model.

Or consider the concept *join:* users have to know what a join is, they have to know how to invoke a join, they have to know what the result of a join looks like, and so on. Again, all that is part of the model. But they don't have to know how joins are physically implemented, or what expression transformations take place under the covers, or what indexes or other access paths are used, or what physical I/O's occur;* all that is part of the implementation, not the model.

In a nutshell, then:

- The model (first meaning) is *what the user has to know*.
- The implementation is what the user *doesn't* have to know.

(Of course, I'm not saying users aren't *allowed* to know about the implementation; I'm just saying they don't have to. In other words, everything to do with the implementation should be, at least potentially, hidden from the user.)

Here are some important consequences of the foregoing definitions. First, note that performance is fundamentally an implementation issue, not a model issue—despite extremely common misconceptions to the contrary. We're often told, for example, that "joins are slow." But such remarks make no sense! Join is part of the model, and the model as such can't be said to be either fast or slow; only implementations can be said to possess any such quality. Thus, we might reasonably say that some specific product *X* has a faster or slower implementation of some specific join than some other specific product *Y*—but that's all.

* I/O = input/output operation.

NOTE

I don't want to give the wrong impression here. It's true that performance is basically an implementation issue; but that doesn't mean a good implementation will perform well if you use the model badly! Indeed, this is precisely one of the reasons why you need to know the model (I mean, so that you *don't* use it badly). If you write an expression such as S JOIN SP, you're within your rights to expect the implementation to do a good job on it; but if you insist on (in effect) hand-coding the join yourself, perhaps like this:

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

then there's no way you're going to get good performance. Relational systems should not be used like simple access methods.

Second, as you probably realize, it's precisely the fact that model and implementation are logically distinct that enables us to achieve *data independence*. Data independence (not a great term, by the way, but we're probably stuck with it) means we have the freedom to change the way the data is physically stored and accessed without having to make corresponding changes in the way the data is perceived by the user. The reason we might want to change those storage and access details is, of course, performance; and the fact that we can make such changes without having to change the way the data looks to the user means that existing application programs, queries, and so on can still work. Very importantly, therefore, data independence means *protecting your investment in user training and applications.*

As you can see from the foregoing definitions, the distinction between model and implementation is really just a special case (a very important special case) of the familiar distinction between *logical* and *physical*. Sadly, however, most of today's database systems, even those that claim to be relational, don't make those distinctions as clearly as they should. As a direct consequence, they deliver far less data independence than they should, and far less than relational systems are theoretically capable of. I'll come back to this issue in the next section, as well as in Chapter 7.

Now I want to turn to the second meaning of the term *data model,* which I dare say you're very familiar with. It can be defined thus:

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

In other words, a data model in the second sense is just a (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 now explained these two different meanings, I'd like to draw your attention to an analogy that I think nicely illuminates the relationship between them:

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

By the way, it follows from all of the above that if we're talking about data models in the second sense, 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* model (with the definite article). I'll have more to say on this issue in Chapter 8.

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

Properties of Relations

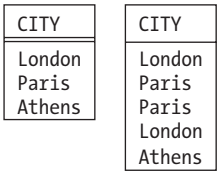
Now let's get back to an examination of basic relational concepts. In this section I want to focus on some specific properties of relations themselves. First of all, every relation has a *heading* and a *body:* the heading is a set of attributes (where an *attribute* is an attribute-name:type-name pair), and the body is a set of tuples that conform to that heading. In the case of the suppliers relation of Figure 1-3, for example, there are four attributes in the heading and five tuples in the body. Note, therefore, that a relation doesn't really contain tuples—it contains a body, and that body in turn contains the tuples—but we do usually talk as if relations contained tuples directly, for the sake of simplicity.

By the way, although it's strictly correct to say that the heading consists of attribute-name:type-name pairs, it's usual to omit the type names in pictures like Figure 1-3 and thereby pretend that the heading is a set of attribute names only. For example, the STATUS attribute does have a type (INTEGER, let's say), but I didn't show it in Figure 1-3. But you should never forget it's there!

Next, the number of attributes in the heading is the *degree* (sometimes the *arity*), and the number of tuples in the body is the *cardinality*. For example, relations S, P, and SP in Figure 1-3 have degree 4, 5, and 3, respectively, and cardinality 5, 6, and 12, respectively.

Next, relations *never* contain duplicate tuples. This property follows because a body is a *set* of tuples, and sets in mathematics do not contain duplicate elements. By the way, SQL fails here: SQL tables are allowed to contain duplicate rows, as I'm sure you know, and SQL tables are thus not relations, in general. Please understand, therefore, that in this book I *always* use the term "relation" to mean a relation—without duplicate tuples, by definition—and not an SQL table. Please understand also that relational operations *always* produce a result without duplicate tuples, again by definition. For example, projecting the

suppliers relation of Figure 1-3 on CITY produces the result shown on the left and *not* the one on the right:



CITY
London
Paris
Athens

CITY
London
Paris
Paris
London
Athens

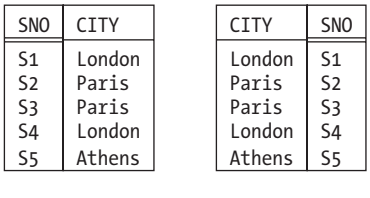
(The result on the left can be obtained via the SQL query SELECT DISTINCT S.CITY FROM S. Omitting DISTINCT leads to the nonrelational result on the right. Note in particular that the table on the right has no double underlining; that's because it has no key, and hence *a fortiori* no primary key.)

Next, the tuples of a relation are *unordered*, top to bottom. This property follows because, again, a body is a set, and sets in mathematics have no ordering to their elements. (Thus, for example, {*a*,*b*,*c*} and {*c*,*a*,*b*} are the same set in mathematics, and the same kind of thing is naturally true in the relational model.) Of course, when we draw a relation as a table on paper, we do have to show the rows in some top-to-bottom order, but that ordering doesn't correspond to anything relational. In the case of the suppliers relation of Figure 1-3, for example, I could have shown the rows in any order—say, supplier S3, then S1, then S5, then S4, then S2—and the picture would still represent the same relation.

NOTE

The fact that the tuples of a relation are unordered doesn't mean queries can't include an ORDER BY specification, but it does mean that such queries produce a result that's not a relation. ORDER BY is useful for the purpose of displaying results, but it isn't a relational operator as such.

In similar fashion, the attributes of a relation are also *unordered*, left to right, because a heading too is a mathematical set. Again, when we draw a relation as a table on paper, we have to show the columns in some left-to-right order, but that ordering doesn't correspond to anything relational. In the case of the suppliers relation of Figure 1-3, for example, I could have shown the columns in any left-to-right order—say STATUS, SNAME, CITY, SNO—and the picture would still represent the same relation in the relational model. Incidentally, SQL fails here too: SQL tables do have a left-to-right ordering to their columns (another reason why SQL tables aren't relations, in general). For example, the pictures below represent the same relation, but two different SQL tables:



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

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

(The corresponding SQL queries are, respectively, SELECT S.SNO, S.CITY FROM S and SELECT S.CITY, S.SNO FROM S. By the way, you might be thinking that the difference between these two tables is hardly very significant; in fact, however, it has some serious consequences, some of which I'll be touching on in later chapters.)

Next, relations are always *normalized* (equivalently, they're in *first normal form,* 1NF). Informally, this means that, in terms of the tabular picture of a relation, at every row-and-column intersection we always see just a single value. More formally, it means that every tuple in every relation contains just a single value, of the appropriate type, in every attribute position. I'll have quite a lot more to say on this particular issue in the next chapter.

Next, we draw a distinction between *base* and *derived* relations. As I explained earlier, the operators of the relational algebra allow us to start with some given relations—perhaps those of Figure 1-3—and obtain further relations from those given ones. The given relations are the base ones; the others are derived. Now, a relational system obviously has to provide a means for defining the base relations in the first place. In SQL, this task is performed by the CREATE TABLE statement (the SQL counterpart to a base relation being, of course, a base *table*). And base relations obviously have to be named—for example:

```
CREATE TABLE SP ... ;
```

But certain derived relations—including in particular what are called *views*—are also named. A view (also known as a *virtual relation*) is a named relation whose value at all times is the result of evaluating a certain relational expression at the time in question. Here's an SQL example:

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

You can operate on views as if they were base relations, but they *aren't* base relations; instead, you can think of views as being materialized—in effect, you can think of a base relation as being dynamically built—at the time they're referenced. (Though I should emphasize that thinking of views being materialized when they're referenced is *only* a way of thinking; it's not what's really supposed to happen. How views really work is explained in Chapter 4.)

There's an important point I need to make here. You'll often see people describe the difference between base relations and views like this:

- Base relations really exist—that is, they're physically stored in the database.
- Views, by contrast, don't "really exist"—they merely provide different ways of looking at the base relations.

But the relational model has nothing to say about what's physically stored! In particular, it does *not* say that base relations are physically stored. The only requirement is that there must be some mapping between whatever *is* physically stored and those base relations, so that those base relations can somehow be constructed when they're needed (conceptually speaking, at any rate). If the base relations can be constructed in this way, then so can everything else. For example, we might physically store the join of suppliers and shipments, instead of storing them separately; base relations S and SP then could be constructed, conceptually, by taking appropriate projections of that join. In other words, base relations are no more (and no less!) "physical" than views are, so far as the relational model is concerned.

The fact that the relational model says nothing about physical storage is deliberate, of course. The idea was to give implementers the freedom to implement the model in whatever way they chose—in particular, in whatever way seemed likely to yield good performance—without compromising on data independence. The sad fact is, however, that SQL vendors seem mostly not to have understood this point; instead, they map base tables fairly directly to physical storage,* and (as noted in the previous section) their products therefore provide far less data independence than relational systems are theoretically capable of. Indeed, this state of affairs is reflected in the SQL standard itself—as well as in most other SQL documentation—which typically (fairly ubiquitously, in fact) uses expressions such as "tables and views." Clearly, anyone who uses such an expression is under the impression that tables and views are different things, and probably under the impression too that "tables" are physical and views aren't. But the whole point about a view is that it *is* a table (or, as I would prefer to say, a relation); that is, we can perform the same kinds of operations on views as we can on regular relations (at least in the relational model), because views *are* "regular relations"! Throughout this book, therefore, I'll reserve the term *relation* to mean a relation (possibly a base relation, possibly a view, possibly a query result, and so on); if I mean (for example) a base relation specifically, then I'll *say* "base relation." And I suggest very strongly that you adopt the same discipline for yourself. Don't fall into the common trap of thinking that the term *relation* means a base relation specifically.

Relations Versus Relvars

Now, it's entirely possible that you already knew everything I've discussed in this chapter so far; in fact, I rather hope you did (though I also hope that doesn't mean you found the discussions boring). Anyway, now I come to something you might not know already. The fact is, historically there's been a lot of confusion between relations—I mean relations as such—on the one hand, and relation *variables* on the other.

* I say this knowing full well that today's SQL products provide a variety of options for hashing, partitioning, indexing, clustering, and otherwise organizing the data as stored on the disk. I still consider the mapping to physical storage in those products to be "fairly direct."

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

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

Then N here *is not an integer*. Rather, it's a *variable* whose *values* are integers as such (different integers at different times). Right? I'm sure we can agree on that. Well, in exactly the same way, if we say in SQL:

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

then T is not a table; it's a *table variable* or (as I would prefer to say, ignoring various SQL quirks such as nulls and duplicate rows) a *relation* variable, whose values are relations as such (different relations at different times).

Consider Figure 1-3 once again. That figure shows three relation values: namely, those that happen to appear in the database at some particular time. But if we were to look at the database at some different time, we would probably see three different relation values appearing in their place. In other words, S, P, and SP in that database are really variables: relation variables, to be precise. For example, suppose the relation variable S currently has the value—the relation value—shown in Figure 1-3, and suppose we delete the tuples (actually there's only one) for suppliers in Athens:

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

Here's the result:

S

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

Conceptually, what's happened is that the old value of S has been replaced in its entirety by a new value. Of course, the old value (with five tuples) and the new one (with four) are very similar, but they certainly are different values. In fact, the DELETE just shown is logically equivalent to, and indeed shorthand for, the following relational assignment:*

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

* I can't show this in SQL because SQL doesn't directly support relational assignment. Throughout this book, I'll show examples in SQL wherever possible—but when it's not possible for some reason, as here, I'll use a more or less self-explanatory (and truly relational) language called **Tutorial D** instead. **Tutorial D** is the language Hugh Darwen and I use to illustrate relational ideas in our book *Databases, Types, and the Relational Model: The Third Manifesto*, Third Edition (Addison-Wesley, 2006); you can regard it as a realization in concrete syntax of the abstract constructs of the relational model (which SQL, regrettably, is not).

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

In like fashion, of course, the familiar INSERT and UPDATE statements are also really shorthand for certain relational assignments. Thus, as I mentioned in the section "A Review of the Original Model," relational assignment is the fundamental update operator in the relational model, and indeed it's the only update operator that's really needed, logically speaking.

So we have these two different concepts, relation value and relation variable. The trouble is that the literature has historically used the same term, *relation,* to stand for both, and that practice has certainly led to confusion. In this book, therefore, I'll distinguish very carefully between the two from this point forward—I'll talk in terms of relation values when I mean relation values and relation variables when I mean relation variables. However, I'll also abbreviate *relation value* most of the time to just *relation* (exactly as we abbreviate *integer value* most of the time to just *integer*). And I'll abbreviate *relation variable* most of the time to just *relvar;* for example, I'll say the suppliers-and-parts database contains three *relvars*.

As an exercise, you might like to go back over the text of this chapter so far and see exactly where I used the term *relation* when I really ought to have used the term *relvar* instead (or as well).

Values Versus Variables

The difference between relations and relvars is actually a special case of the difference between values and variables in general, and I'd like to take a few moments to look at the more general case. (It's a bit of a digression, but I think it's worth taking the time because clear thinking here can be a tremendous help in so many areas.) Here then are some definitions:

> *Definition:* A *value* is an "individual constant": for example, the individual integer 3. A value has no location in time or space. However, values can be represented in memory by means of some encoding, and those representations do have locations in time and space. Indeed, distinct representations of the same value can appear at any number of distinct locations in time and space, meaning, loosely, that any number of different variables—see the definition below—can have the same value, at the same time or different times. Observe in particular that, by definition, *a value can't be updated*; for if it could, then after such an update it would no longer be that value.
>
> *Definition:* A *variable* is a holder for a representation of a value. A variable does have location in time and space. Also, of course, variables, unlike values, *can be updated;* that is, the current value of the variable can be replaced by another value.

Please note very carefully that it isn't just simple things like the integer 3 that are legitimate values. On the contrary, values can be arbitrarily complex; for example, a

value might be a geometric point, or a polygon, or an X ray, or an XML document, or a fingerprint, or an array, or a stack, or a list, or a relation (and on and on). Analogous remarks apply to variables too, of course. I'll have more to say on these matters in the next two chapters.

Now, it might be hard to imagine people getting confused over a distinction as obvious as that between values and variables, but in fact it's all too easy to fall into traps in this area. By way of illustration, consider the following extract from a tutorial on object-oriented databases (the italicized portions in brackets are comments by myself):

> We distinguish the declared type of a variable from…the type of the object that is the current value of the variable [*so an object is a value*]…We distinguish objects from values [*so an object isn't a value after all*]…A *mutator* [is an operator such that it's] possible to observe its effect on some object [*so in fact an object is a variable*].

Summary

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

- I explained why we'd be concerned in this book with principles, not products, and why I'd be using formal terminology such as *relations, tuples,* and *attributes* in place of their more "user-friendly" SQL counterparts.
- I claimed that SQL and the relational model aren't the same thing. We've seen a few differences already—for example, the fact that SQL permits duplicate rows—and we'll see many more in later chapters.
- I gave an overview of the original model, touching in particular on the following concepts: *type*, *n-ary relation*, *tuple*, *attribute*, *candidate key*, *primary key*, *foreign key*, *entity integrity*, *referential integrity*, *relational assignment*, and the *relational algebra*. With regard to the algebra, I mentioned *closure* and very briefly described the operators *restrict*, *project*, *product*, *intersection*, *union*, *difference*, *join*, and *divide*.
- I discussed various properties of relations, introducing the terms *heading, body, cardinality,* and *degree*. Relations have no duplicate tuples, no tuple ordering top to bottom, and no attribute ordering left to right. I also discussed *views*.
- I discussed the differences between *model* and *implementation, relations* and *relvars,* and *values* and *variables* in general. The model versus implementation discussion in particular led to a discussion of *data independence*.

One last point (I didn't mention this explicitly before, but I hope it's obvious from everything I did say): overall, the relational model is *declarative,* not *procedural,* in nature; that is, we favor declarative solutions over procedural ones, wherever such solutions are feasible. The reason is obvious: *declarative* means the system does the work, *procedural*

means the user does the work (so we're talking about productivity, among other things). That's why the relational model supports declarative queries, declarative updates, declarative view definitions, declarative integrity constraints, and so on.*

Exercises

As noted in the preface, you certainly don't have to do any of the exercises, but I think it's a good idea to try at least some of them. Answers, often giving more information about the subject at hand, can be found online at *http://oreilly.com/catalog/databaseid*.

Exercise 1-1. (Repeated from the body of the chapter, but slightly reworded here.) If you haven't done so already, go through the chapter again and identify all of the places where I used the term *relation* when I should by rights have used the term *relvar* instead (or as well).

Exercise 1-2. Who was E. F. Codd?

Exercise 1-3. What's a domain?

Exercise 1-4. What do you understand by the term *referential integrity?*

Exercise 1-5. The terms *heading, body, attribute, tuple, cardinality,* and *degree,* defined in the body of the chapter for relation values, can all be interpreted in the obvious way to apply to relvars as well. Make sure you understand this remark.

Exercise 1-6. Distinguish between the two meanings of the term *data model*.

Exercise 1-7. Explain the difference between model and implementation in your own words.

Exercise 1-8. In the body of the chapter, I said that tables like those in Figures 1-1 and 1-3 weren't relations as such but, rather, *pictures* of relations. What are some of the specific points of difference between such pictures and the corresponding relations?

Exercise 1-9. Explain *data independence* in your own words.

Exercise 1-10. (Try this exercise without looking back at the body of the chapter.) What relvars are contained in the suppliers-and-parts database? What attributes do they involve? What candidate and foreign keys exist? (The point of this exercise is that it's worth making yourself as familiar as possible with the structure, at least, of the running example. Of course, it's not so important to remember the actual data values in detail—though it wouldn't hurt if you did.)

Exercise 1-11. "There's only one relational model." Explain this remark.

* As this book was going to press, I was informed that at least one well-known SQL product apparently uses the term "declarative" to mean the system *doesn't* do the work! That is, it allows the user to state certain things declaratively (for example, the fact that a certain view has a certain key), but it doesn't enforce the constraint implied by that declaration—it simply assumes the user is going to enforce it instead. Such terminological abuses do little to help the cause of genuine understanding. *Caveat lector*.

Exercise 1-12. The following is an excerpt from a recent database textbook: "[It] is important to make a distinction between stored relations, which are *tables,* and virtual relations, which are *views*…[We] shall use *relation* only where a table or a view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term *base relation* or *base table*." This text betrays several confusions or misconceptions regarding the relational model. Identify as many as you can.

Exercise 1-13. The following is an excerpt from another recent database textbook: "[The relational] model…defines simple tables for each relation and many-to-many relationships. Cross-reference keys link the tables together, representing the relationships between entities. Primary and secondary indexes provide rapid access to data based upon qualifications." This text is intended as a *definition* of the relational model… what's wrong with it?

Exercise 1-14. Write SQL CREATE TABLE statements to define an SQL version of the suppliers-and-parts database.

Exercise 1-15. The following is a typical SQL INSERT statement against the suppliers-and-parts database:

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

Show an equivalent relational assignment operation. *Note:* I'm assuming here that attributes SNO, PNO, and QTY are of types SNO, PNO, and QTY, respectively, and the expressions SNO('S5'), PNO('P6'), and QTY(250) are literals of those types, with the obvious interpretation. I'll have more to say on such matters in the next two chapters. Also, I realize I haven't yet explained the syntax of relational assignment in detail, so don't worry too much about giving a syntactically correct answer—just do the best you can.

Exercise 1-16. (Harder.) The following is a typical SQL UPDATE statement against the suppliers-and-parts database:

```
UPDATE S
SET    STATUS = 25
WHERE  S.CITY = 'Paris' ;
```

Show an equivalent relational assignment operation. (The purpose of this exercise is to get you thinking about what's involved. I haven't told you enough in this chapter to allow you to answer it fully. See Chapter 5 for further discussion.)

Exercise 1-17. In the body of the chapter, I said that SQL doesn't directly support relational assignment. Does it support it indirectly? If so, how?

Exercise 1-18. From a *practical* point of view, why do you think duplicate tuples, top-to-bottom tuple ordering, and left-to-right attribute ordering are all very bad ideas? (These questions deliberately weren't answered in the body of the chapter, and this exercise might best serve as a basis for group discussion. We'll take a closer look at such matters later in the book.)