CHAPTER 3

The Relational Model

SQL has a very practical exterior but a very theoretical interior. That interior is the relational model. The relational model came before SQL and created a need for SQL. The power of SQL lies not in the language itself but in the concepts set forth in the relational model. These concepts form the basis of SQL's design and operation.

The relational model is a powerful and elegant idea that has pervaded not only computer science but our daily lives as well, whether we know it or not. Like the automobile, or penicillin, it is one of the many great examples of human thought and discovery that has fundamentally impacted the way the world works. The relational model has spawned a billion-dollar industry, and has become an integral part of almost every other industry. You'll see the relational model at work nearly everywhere that deals with information of any kind—in Fortune 500 companies, universities, hospitals, grocery stores, websites, routers, MP3 players, cell phones, and even smart cards. It is truly pervasive.

Background

The relational model was born in 1969, inside of IBM. A researcher named E. F. Codd distributed an internal paper titled "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," which defined the basic theory of the relational model. This paper was circulated internally and not widely distributed. In 1970, Codd published a more refined version of this paper in *Communications of the ACM* called "A Relational Model for Large Shared Data Banks," which is more widely recognized as the seminal work on relational theory. This is the paper that changed the industry.

Despite the enormous influence of this paper, the relational model in its entirety did not appear overnight, or within the scope of a single historic paper. It evolved, grew, and expanded over time and was the product of many minds, not just Codd alone. For example, the term "relational model" wasn't coined until 1979, nine years after publication of the paper proposing it (see Date, 1999, in the References section). Codd's 12 rules, now famous as the working definition of relational, weren't posited until 1985, appearing in a two-part series published by *Computerworld* magazine. Thus, while there was a seminal paper proposing the general concept, much of the relational model as we know it today is actually the result of a series of papers, articles, and debates by many people over a 30-year period, and indeed continues to develop to this very day. That is also why in some of the various quotes included in this chapter you may see terms such as "data base," which may initially appear to be a typo. They aren't. At the time these statements were made, the terminology still had not congealed and all sorts of various terms were being thrown around.

The Three Components

By 1980, enough was known about the relational model for Codd to identify three principle components:

- **The structural component**: This component defines how information is structured, or represented. Specifically, all information is represented as *relations*, which are composed of *tuples*, which in turn are composed of *attribute* and *value* components. The relation is the sole data structure used to represent all information in the database. Relations as defined in the relational model are derived from relations in set theory, a branch of mathematics, and share many of their properties. They are formally defined in Codd's 1970 paper.
- **The integrity component**: This component defines methods that enforce relationships within and between relations (or tables) in the structural component. These methods are called *constraints*, and are expressed in the form of rules. There are three principle types of integrity: domain integrity, governing values in columns; entity integrity, governing rows in tables; and referential integrity, governing how tables relate to one another. Integrity has no analog in set theory, but rather is unique to relational theory. It was initially addressed in Codd's 1970 paper and greatly expanded upon in the 1980s by Codd and others.
- **The manipulative component**: This aspect defines the methods with which to operate on or manipulate information. Like relations, these operations also have their roots in mathematics. They are formalized in *relational algebra* and *relational calculus* as originally presented in Codd's 1972 paper "Relational Completeness of Data Base Sublanguages."

SQL, likewise, is structured similarly along these lines—so similarly, in fact, that it is hard to talk about SQL without addressing the relational model to some degree, directly or indirectly. It is true that SQL is for the most part a straightforward language, which to many people appears as an island unto itself. But in reality, it is the offspring of the relational model, and in many ways is clearly a reflection of it.

SQL and the Relational Model

This chapter presents the theoretical roots of SQL, and examines the power and elegance behind SQL. It prepares you to deal with SQL not in isolation but in the context of the relational model. If nothing else, it should give you an appreciation for how elegant, powerful, and complex a beast a relational database really is. You may be surprised by what even the most rudimentary relational databases are capable of.

You don't have to read this chapter in order to understand SQL; it merely provides a theoretical and historical backdrop. To this end, I present the theoretical aspects in terms of several influential papers and articles written by Codd between 1970 and 1985 that define the essence of the relational model, along with the ideas and work of other contributors to the field. Two of Codd's papers mentioned here are available online, and are listed in the References section at the end of the chapter. Again, others contributed to the development of the relational model, but this chapter will draw primarily from Codd's work.

This book was purchased by ybakos@mines.edu

This chapter is not an exhaustive treatment of the relational model, or a complete history of it. It is merely an appetizer. The SQL chapter that follows is the main course. This chapter presents only the minimal material needed to get an appreciation of the origin and theoretical underpinnings of SQL. We illustrate the relational model in terms of both its three components in general and in the context of Codd's 12 rules in particular. The relational model in its full splendor is far beyond the scope of this book, and I have neither the qualifications nor the stamina to fully describe it. See the References section at the end of this chapter if you want to learn more. The rules as they are presented here are taken directly from Codd's October 14, 1985, *Computerworld* article.

The Structural Component

The structural component of the relational model lays the foundation upon which the other components build. It defines the form in which information is represented. It is defined by the first of Codd's 12 rules, which is the cornerstone of the relational model.

The Information Principle

The first rule, called the *Information Rule*, is also known as the *Information Principle*. It is defined as follows:

> ***1. The Information Rule.*** *All information in a relational data base is represented explicitly at the logical level and in exactly one way—by values in tables.*

Date summarizes this rule as follows:

> *The entire information content of the database is represented in one and only one way, namely as explicit values in column positions in rows in tables.*

There are two import expressions here: "logical level" and "values in tables." The logic level, or logical representation, refers to the way that you, the user, see the database and information within it. It is a kind of ideal worldview for information. The logical level is a consistent, uniform depiction of data, which has two important properties:

- The view presented in the logical level consists of tables, made up of rows, which in turn are made up of values.
- The view is completely independent of the database system—the technology (software or hardware) that enables it.

The logical representation is a world unto itself, which is completely distinct from how the database is implemented, or how it stores data physically, or how it operates internally. These latter components—software, operating system, and hardware—are referred to as the *physical representation*—the technology of the system. If the database vendor decides to store database tables in a different way on disk, the Information Principle mandates that such a change in physical representation can in no way affect or change the logical representation of that data—the way in which you (or your programs) see that data. The logical representation is independent of physical representation. A result of the Information Principle is that it is possible to

represent the same information in the same way across multiple database implementations on multiple operating systems on different hardware, as shown in Figure 3-1.

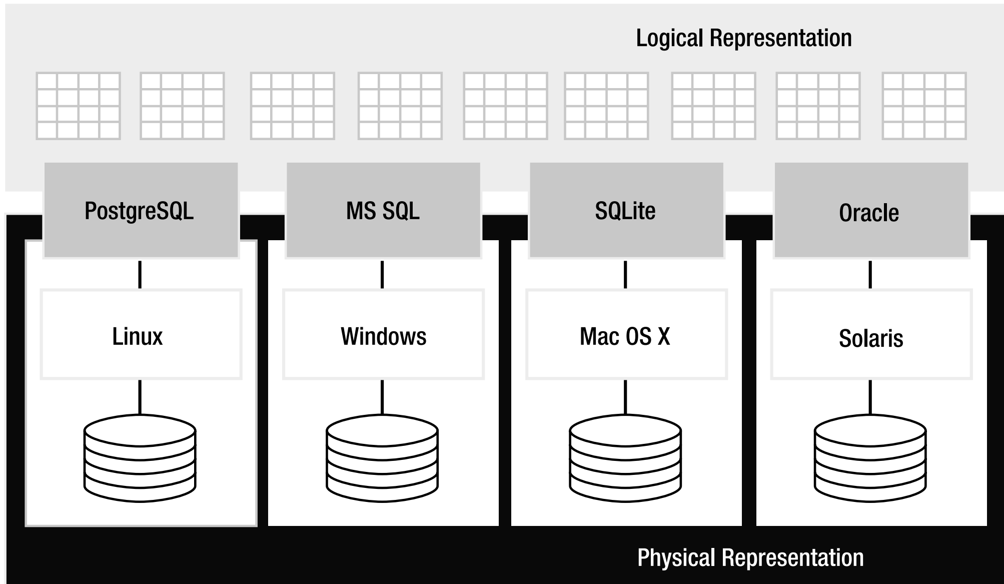


Figure 3-1. *Logical and physical representation*

You can create a relation (or table) in Oracle on Solaris that is represented in exactly the same way as a table in PostgreSQL on Linux, or SQLite on Mac OS X.[1] The Information Principle guarantees you a consistent logical view of information regardless of how the database software is implemented, or the operating system or hardware it runs on.

The Sanctity of the Logical Level

So important are these two constraints in the relational model that they are expanded upon and reinforced in several other rules (8, 9, 11, and 12) so as to eliminate any possible ambiguity. In short, Codd says:

> ***8. Physical Data Independence.*** *The logical view can in no way be impaired by the underlying software or hardware.*

> ***9. Logical Data Independence.*** *Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.*

1. In reality, this is not 100 percent true (it's more like 95 percent true). There are slight differences in database implementations so that relations in one database may contain features not present in other databases. Nevertheless, they still all adhere to the same general structure: relations made of tuples made of values.

> ***11. Distribution Independence.*** *Even if the database is spread across various locations, it cannot impact the logical view of data.*
>
> ***12. Nonsubversion Rule.*** *The database software may not provide any facility which can subvert the integrity constraints of the logical view.*

The separation of logical from physical was very important to Codd from the outset. In the opening of his original paper he had strong words concerning this separation:

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

The relational model was in part a reaction to the database systems of the day, which closely tied applications to both database implementation and data format on disk. Codd's relational model challenged this:

> *It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.*

So in the relational model, the Information Principle provides a level of abstraction through which information can be represented in a consistent way. The user sees and works with data exclusively in terms of this logical representation. This representation is completely insulated from the underlying technology. It cannot be undermined, influenced, or affected by it in any way. As stated before, the Information Principle is the foundation of the relational model.

The Anatomy of the Logical Level

The logical level is made up of more than relations. It is made up of *tables*, *rows*, *columns*, and *types*. In relational parlance, these are often referred to more formally as *relation variables*, *tuples*, *values*, and *domains*, respectively. SQL, for example, uses many of the former, and relational theory tends to use many of the latter. Throughout the literature, however, you will see these terms used almost interchangeably. Even in Codd's papers, both sets of terms are used. Interestingly, there is one term that both lexicons have in common: *relations*. (In case you're wondering, tables and relations are not the same thing; we'll address the difference later in this chapter.)

This big soup of terminology comprises the anatomy of the relational body, which will be explained in detail over the next several sections. The relationships between all of these terms are illustrated in Figure 3-2.

At the center of everything is the relation. It is the central object around which the structural, integrity, and manipulative components of the relational model are built. All of the fundamental operations in the relational model are expressed in terms of relations, and all integrity constraints are defined within relations. Your understanding of the relational model is only as good as your understanding of relations themselves. To have a good grasp of relations, however, you must first understand tuples.

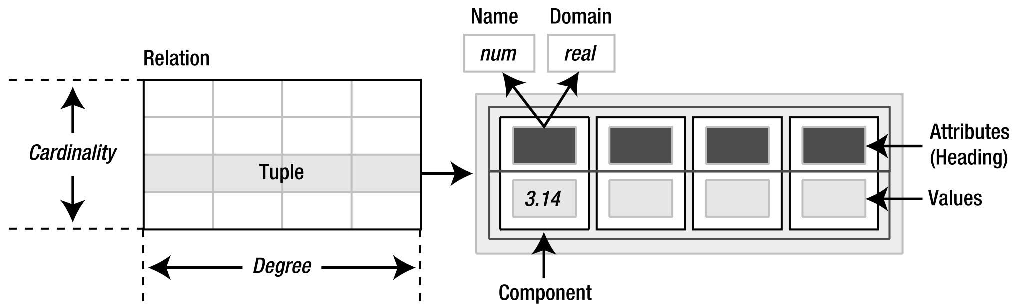


Figure 3-2. *The logical representation of data*

Tuples

A tuple is a set of *values*, each of which has an associated *attribute*. An attribute defines a value's *name* and *domain*. The attribute's name is used to identify the value in the tuple, and its domain defines the kind of information stored within it. The combination of attribute and value is called a *component*. In Figure 3-2, you can see that the first component is named *num*, and that it has a value of 3.14 and a domain of *real* (as in real numbers).

A component is kind of a tidy, self-contained unit of data. It has three essential ingredients: a name, a domain, and a value. Its name gives it identity, its domain a description of its content, and its value the content itself. Together, components aggregate into larger structures such as tables and relations, and their qualities propagate into those structures, imparting to them identity, description, and content as well.

The name and value parts of a component are easy enough to understand, but what exactly is a domain? The word *domain*, like the words *relation* and *tuple*, comes from mathematics. In mathematics, the domain of a function is the set of all values for which the function is defined. Some familiar domains in mathematics are the sets of all integers, rational numbers, real numbers, and complex numbers. Generally, the term *domain* corresponds to a set of permissible values. A domain is sometimes referred to as a *type*, and is synonymous to a data type in programming languages. A domain, especially in the relational sense, also implies a set of operators that can be used to operate on its associated values. For example, common operators associated with integers are addition, subtraction, multiplication, and division.

So the job of a domain is to define a finite or an infinite set of permissible values along with ways of operating on them. In a way, the domain controls or restricts the value of an attribute, but it does so only by providing information. The database actually restricts an attribute's value so that it conforms to its associated domain. As you will see later, this particular restriction is defined in the integrity component of the relational model, and is called *domain integrity*. The group of collective attributes in a tuple is called its *heading*. Just as an attribute defines the properties of its associated value, the heading defines the properties of its tuple.

Relations

A relation, simply enough, is a set of one or more tuples that share the same heading. Just as domains have analogs in programming languages, so do tuples and relations. And even if you are not a programmer, the analogy is quite helpful in illustrating the relationship between tuples, relations, and headings.

For example, consider the relation and its C equivalent shown in Figure 3-3. You could say that a tuple is similar to a C structure. They both are made up of attributes that have a name and a type. As shown in the figure, a C structure's attributes are defined in its declaration, just as a tuple's attributes are defined in its heading. The declaration and heading both provide information about the contents of their respective data structures.

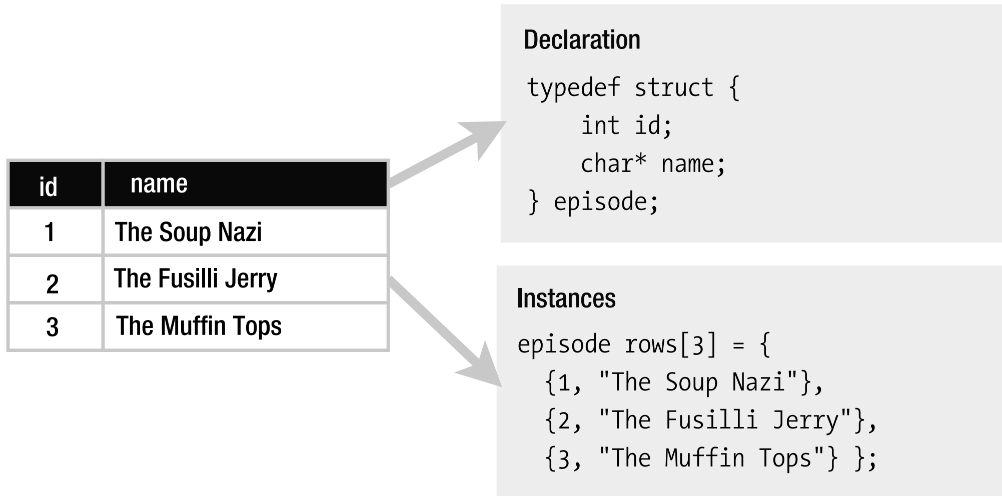


Figure 3-3. *Header as declaration; tuple as instance*

Similarly, a tuple's values are analogous to an *instance* of a C structure. Its values form a composite data structure that corresponds to the attributes defined in its heading. Each value in the structure is identifiable by an attribute name, and each value is restricted by an attribute domain.

Thus, tuples are more than just rows of amorphous values like what you might find in a spreadsheet. They are well-defined data structures with a high degree of specificity over the information within them. They have more in common with constructs in programming languages than with rows in a spreadsheet.

But the analogy doesn't stop there. As also shown in Figure 3-3, a relation is the C equivalent of an array of structures. Each structure in the array along with the array itself shares the same type, just as relations and their tuples share the same heading.

The bottom line is that relations and tuples are highly structured. Furthermore, this structure is defined by their common heading.

Degree and Cardinality

Associated with tuples and relations are the notions of *degree* and *cardinality*. These are just fancy words for width and height, respectively. You could say a relation's width is the number of attributes in its heading. This is called the degree. Its height is the number of tuples it contains; this is called cardinality. These terms are illustrated in Figure 3-2. A relation with four attributes and five tuples would be said to be of degree 4 and cardinality 5.

While tuples don't have cardinality, they nonetheless have their own fancy terms. A tuple of degree 1 is said to be a *unary* tuple. Tuples of degree 2, 3, and 4 are said to be *binary, ternary,* and *quaternary,* respectively. Generally, a tuple of degree *N* is said to be an *N*-ary tuple. In fact, the word *tuple* is taken from the *N*-ary form. In the strictest sense, a unary tuple is called a *monad,* a binary tuple a *pair,* a ternary tuple a *triple,* a quaternary tuple a *tetrad,* and so on, as shown in Table 3-1.

Table 3-1. *Tuple Terminology*

Degree	Qualification	Designation	Example
1	Unary	Monad	1
2	Binary	Pair, twin	(0, 1)
3	Ternary	Triple, triad	(0, 1, 2)
4	Quaternary	Quadruple, tetrad	(0, 1, 2, 3)
N	*N*-ary	N-tuple (tuple)	(0, 1, 2, … N)

Typically, the term *tuple* is used as a generic catchall for tuples of all degrees, and using terms like *monad* and *triad* is often more confusing than it is precise. As with tuples, a relation composed of binary tuples is a binary relation. A relation composed of *N*-ary tuples is an *N*-ary relation, and so forth.

Mathematical Relations

The notion of a relation in relational theory is taken directly from the relation in set theory, with a few modifications. To really understand relations, as defined in relational theory, you must understand their predecessors in mathematics.

Just as in relational theory, mathematical relations are sets of mathematical tuples. Like mathematical relations, mathematical tuples also differ somewhat from their relational namesakes. In mathematics, tuples are ordered sequences and relations are unordered sets. Sets by their very nature are ambivalent to order. That is, the order of elements in a set does not affect the fundamental identity of that set, whereas the order of items in a sequence does affect its identity.

To begin with, every value in a tuple has a specific domain associated with it. Suppose we have a tuple composed of the following two domains:

- The domain of all integers {...,-1,0,1,...}, denoted by **I**
- The domain of the first names of all Seinfeld characters {'Jerry', 'Cosmo', 'Newman', ...}, denoted by **F**

Now, suppose we have a relation composed of such tuples. The relation then is also composed of the domains **I** and **F** (in that order). The first column of the relation must consist of integer values, and the second column must consist of the first names of Seinfeld characters. Using this as an example, the formal definition of a relation can be expressed as follows:

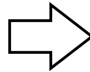
*A relation over **I** and **F** is any subset of the cross product of **I** and **F**, represented by **I×F**.*

This is a somewhat annoying definition because it defines a relation in terms of another perhaps unfamiliar concept: the *cross product*. The cross product, also called the *Cartesian product*, is quite simple, however. It is the combination of every value in every domain with every value in every other domain. To compute **I×F**, for example, you take each integer *i* in **I**, and pair it with each name *f* in **F**. This yields an infinitely large set of (*i,f*) tuples (binary tuples), as illustrated in Figure 3-4. Note that the figure is somewhat simplistic and depicts the set of all integers as the values to {-1, 0, 1}.

integers = {...,-1,0,1,2,3...}

first_names = {Jerry, Cosmo, Newman, ... }

integers × first_names = { (-1, Jerry),
(0, Jerry),
(1, Jerry),
(-1, Cosmo),
(0, Cosmo),
(1, Cosmo),
(-1, Newman),
(0, Newman),
(1, Newman) }



integer	first_name
-1	Jerry
0	Jerry
1	Jerry
-1	Cosmo
0	Cosmo
1	Cosmo
-1	Newman
0	Newman
1	Newman

Figure 3-4. *The cross product of integers and Seinfeld character first names*

What does the cross product do here? Why is it used? The cross product of domains **I** and **F** is a set containing every tuple that could ever exist by the combination of these two domains. That is, every tuple that has an integer as its first value and the first name of a Seinfeld character as its second value is contained in the cross product **I×F**. The cross product is itself just a set, albeit a very big one. It is a set defining the limit of every tuple that could ever be formed from the domains **I** and **F**. That being said, any subset of this cross product is a relation, specifically a "relation over **I** and **F**." That is a mathematical relation: any subset of a cross project.

Put more simply, a relation is defined in terms of its constituent domains—it is a portion (or subset) of their cross product. This cross product is a superset containing every possible tuple that could ever appear in the relation. (I know this seems circuitous.) The proper way of expressing a relation is to speak of a *relation R over the domains X, Y, and Z*. If we were to expand our relation to include the domain of all Seinfeld episodes, denoted by **E**, it would then be "a relation *R* over **I**, **F**, and **E**"—represented by **I×F×E**, which in turn is an even larger set than **I×F**.

Relational Relations

There is one fundamental difference between tuples in mathematics and tuples in relational theory. In mathematics, the order of values in a tuple is significant; in relational theory it isn't. This is because members (attributes) of relational tuples have names, which are used to identify them. Mathematical ones do not. Therefore, mathematical tuples in relations over **I×F** are always of the form (*i,f*) because this is the only way to identify them—by ordinal positions. This convention is not needed in relational tuples because they have attributes that have names to

identify them, rather than ordinal positions. Order, then, in a relational tuple offers no real advantage. The same applies to relational relations, as they share the same attributes through their common heading.

In his original paper, Codd differentiated mathematical relations (which he called domain-ordered relations) from his relational relations (called domain-unordered relations) by referring to the latter as "relationships":

> *Accordingly, we propose that users deal, not with relations which are domain-ordered, but with relationships which are their domain-unordered counterparts. To accomplish this, domains must be uniquely identifiable at least within any given relation, without using their position.*

Domains (or attributes) are uniquely identifiable through attribute names. Therefore, both column and row order do not matter in the relational model. This is the principal way in which the relations of set theory and the relations of relational theory differ. In all other respects, tuples and relations in the relational theory share all the same properties of their counterparts in set theory.

And that is a relation. It is the fundamental object upon which relational theory is built. If you understand it, you are well on your way to understanding the core mechanics of relational theory. Relational theory is fitted so as to mirror the true mathematical relation as closely as possible. The reason is that the closer the relational model fits the mathematical model, the more it benefits from other facets of mathematics already well established. A prime example is relational algebra, which is part of the manipulative component of the relational model. Many of the operations defined for relations in set theory carry over to relational algebra because relational relations are sufficiently similar. The same is true for relational calculus, which employs methods of formal logic. As Codd put it

> *Moreover, the (relational) approach has a close tie to first-order predicate logic—a logic on which most of mathematics is based, hence a logic which can be expected to have strength, endurance, and many applications.*

Codd recognized not only that the elegance of mathematics is beneficial as a basis to build upon, but that it also offers a vast reservoir of existing knowledge that can be harnessed as well.

Tables: Relation Variables

A relation, though it contains values, is itself just a value, just like an integer or a string. The value of a relation is given by the particular set of tuples it contains. Likewise, the value of each tuple in turn is given by the specific values within it. Thus, the value of a relation is determined by the sum of its parts. Figure 3-5 illustrates this. It depicts three relations, represented by *R1*, *R2*, and *R3*, taken over **I**×**F**. Each represents a different value, or relation.

R1, *R2*, and *R3* are not relations but rather *relation variables*. They *represent* relations. Codd referred to them as *named relations*. Date (2003) calls them *relvars*. SQL calls them *tables*. They are also known as *base tables*. The bottom line is that they are simply variables whose values are relations. I will simply refer to them here as tables, as that is what you will be dealing with in SQL.

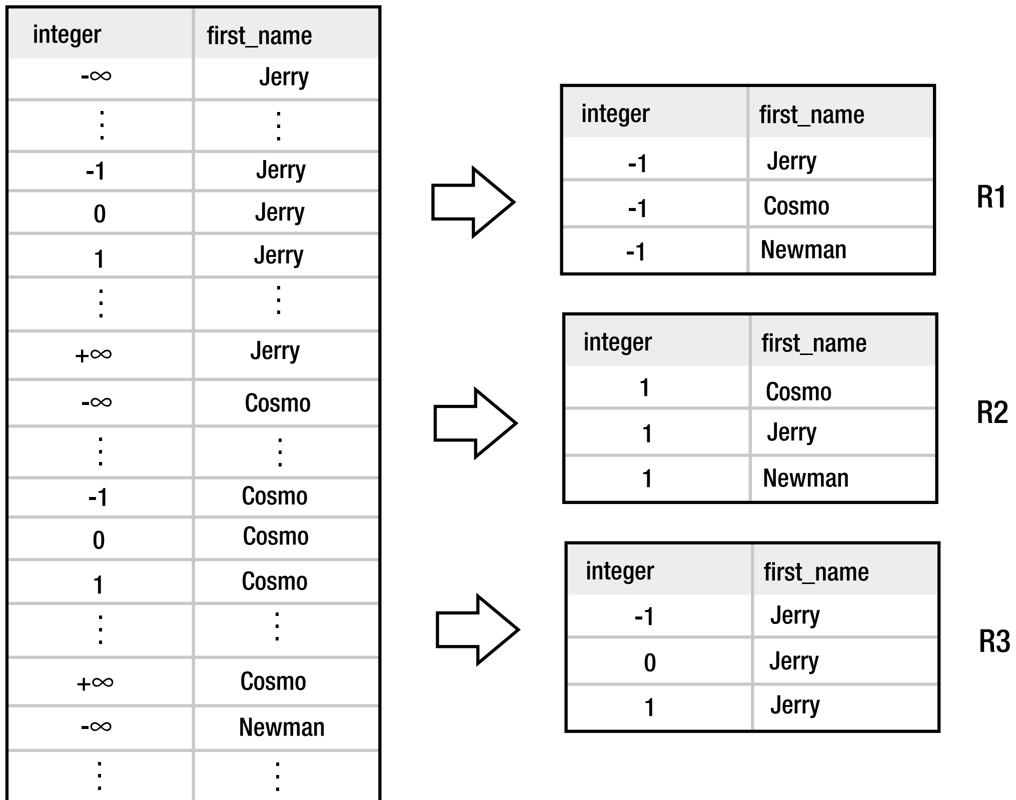


Figure 3-5. *Relations over I×F*

In practice, the precise meaning of "relation" can sometimes be a bit murky. Relations are often referred to in the same context as tables. However, they are not the same thing. One is a value; the other is a variable. A relation is a *value*, like an integer value is 1, 2, or 3. A table is a variable—to which relations are assigned. Tables, like variables, have both a name and a value. Their name is just a symbol. Their value is a relation. They are no different than variables in algebra, such as *x* and *y* in the equation of a line. Tables share all the properties of relations (heading, degree, cardinality, etc.) just as integer variables share the properties of integers.

With this in mind, what does it mean to update or modify a table? If you are familiar with SQL, you probably think of it in terms of modifying a single value, or a few rows. You see it the same as changing cells in a spreadsheet. You modify *parts* of the spreadsheet. You see those parts as individual values. The relational view is different. In the relational view, you don't modify parts of a table. A table is a variable holding an entire relation as its value. If you change anything, you change *the entire relation*—no matter how small the difference might be. You aren't changing a row or column; you are actually swapping one relation with an entirely new one, where the new relation contains the rows or columns you want changed.

I know that it may seem like frivolous semantics. But just as you don't modify part of an integer variable, you don't modify part of a relation variable either. There is nothing wrong

with thinking in terms of the SQL view—where you are changing values in a row. It is logically equivalent: in a SQL update, you are articulating the change you want to make in a statement, and that statement produces a new relation. But that is where you should make the distinction: the change produces a *new* relation, not a patched-up version of the old one. That new relation becomes a new value that the table holds.

Be aware that when you talk about tables in a database, you are referring to variables, specifically relation variables—not relations. If a database were composed strictly of relations, then its contents would be fixed values, not subject to change. But databases are dynamic. And the reason is because their contents—tables—are relation variables, which are subject to change. They change by *assigning* those relation variables new relation values, not by adding, subtracting, or changing tuples.

Views: Virtual Tables

Views are virtual tables. They look like tables and act like tables, but they're not. Views are relational expressions that yield relations. It is like saying that the algebraic expression $x+y$ is not a number per se, though it can yield a number if the expression is evaluated, provided we have specific values for x and y. The same is true for views. Codd (1980) described them as follows:

> *A view is a virtual relation (table) defined by means of an expression or sequence of commands. Although not directly supported by actual data, a view appears to a user as if were an additional base table kept up-to-date and in the state of integrity with other base tables. Views are useful for permitting application programs and users at terminals to interact with constant view structures, even when the base tables themselves are undergoing structural changes at the logical level...*

This book was purchased by ybakos@mines.edu

Logical Data Independence

In one sense, views are simply a matter of convenience. They let you assign a name to a set of operations and treat the result like a relation. They are a kind of shorthand. This is perhaps the most common use of views. Another use was for security. In some systems, views can be used to selectively present parts of tables, while excluding other sensitive or restricted parts. But the original intent of views was much more than these two applications. The main application for views was facilitating what is called *logical data independence*, which is defined as follows:

> ***9. Logical Data Independence.*** *Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.*

Logical data independence means that applications and users should theoretically be able to see the same logical structure (e.g., the same attributes in a relation), even if it is changed (e.g., an attribute in that relation is moved to another relation). An example is when a relation is decomposed into two relations for the sake of normalization, as described in the section "Normalization."

More precisely, logical data independence means that users and applications *should be able to be* insulated from changes at the logical level of the database. That doesn't mean the database is supposed to automatically shield users and programs from the database administrator (DBA) doing something really bone-headed like dropping a table and going home for the

day. Rather, it means that the relational model provides a means for the administrator to bridge the gap if she wants to make substantive changes at the logical level that would impact the logical view seen by users and applications. For instance, if the DBA decomposes a table into two smaller ones, she can create a view that looks like the original table, though it is in reality a relational expression that combines (joins) the two new tables. The users and applications know no differently.

Updatable Views

But logical data independence entails more than just viewing data. Many people who are familiar with views understand them as read-only. However, the relational model clearly states that you should be able to write to views as well, just as if they were ordinary tables. While logical data independence would seem to imply this, Rule 6 makes it explicit:

> ***6. View Updating Rule.*** *All views that are theoretically updatable are also updatable by the system.*

Rule 6 is essential for logical data independence. "Theoretically updatable" means the view is constructed in such a way that it is theoretically possible to map changes made on it to its respective base tables. That is, if it can be done in theory, the system should be able to do it.

That said, Rule 6 and updatable views (sometimes referred to as materialized views) are not fully supported in all relational databases, simply because they are not easy to implement. Programming a system to know what is "theoretically updatable" in relational algebra and/or calculus is not exactly a weekend project. On the other hand, read-only views are extremely common. In fact, it is almost hard to find relational databases today that don't support them.

The System Catalog

As a database is composed of tables and views, you might at some point wonder how you can find out exactly what is in a given database. As it turns out, a relational database contains tables and views describing its tables and views—information about the information. That is, all tables, views, constraints, and other database objects belonging in a database are registered in what is referred to as the *system catalog*. Per Rule 4:

> ***Rule 4. Dynamic On-Line Catalog Based on the Relational Model.*** *The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.*

The system catalog is subject to the Information Principle: it is required to be represented in a relational format that can be queried in the same way as other relations in the system. This means that even metadata (information about information) in a database has to be represented relationally. In many database products, many of the tables in the system catalog are often implemented as views. The beauty of views and the system catalog is that together they enable you to extend or enhance the catalog itself. You can create your own catalog views that contain information you find useful or informative. Those views in turn may even use catalog tables as a basis.

The Integrity Component

While the structural aspect of the model relates to the structure of information, the integrity aspect relates to the information within the structure. Information can be arranged in relations in a way that gives rise to various relationships, both within columns of a single relation and between columns of different relations. These relationships provide additional structure and indeed add even more information to what is already present.

The integrity aspect of the relational model provides a way to explicitly define and protect such relationships. Although their use in a database is entirely optional, the degree to which they are employed can ultimately determine the consistency of your data.

Primary Keys

Codd's second rule is the starting point of data integrity, and deals with the nature of data within relations. This rule is called the Guaranteed Access Rule and is defined as follows:

> ***2. Guaranteed Access Rule.*** *Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.*

The relational model requires that every relation have a primary key. The primary key is the set of attributes in a relation that uniquely identifies each tuple within it. This rule carries with it an important corollary: relations may not contain duplicate tuples.

The Guaranteed Access Rule states that every field (or value in a tuple) in a relational database must be addressable. To be addressable, it must be identifiable. To be identifiable, each tuple must be distinguishable in some way from all other tuples in the relation, which is to say unique (thus, duplicate tuples would violate this constraint).

Uniqueness is the business of the primary key. A key is a designated attribute (or group of attributes) in a relation such that

1. The value (or combined values) of that attribute (or attributes) is unique for every tuple in the relation.

2. If the key is composed of more than one attribute, all of the attributes that define the key must be necessary to ensure uniqueness. That is, every attribute in the key is sufficient to ensure uniqueness, but also necessary as well—if one were absent, then the uniqueness condition would not hold.

If both conditions 1 and 2 are met, then the resulting attribute or group of attributes is a key (also called a candidate key). If condition 1 is met but not condition 2, then the attribute (or group of attributes) is called a *superkey*. It is a key that could stand to lose some weight. That is, it has more attributes than necessary to ensure uniqueness: a smaller key containing fewer attributes could be defined that still guarantees uniqueness.

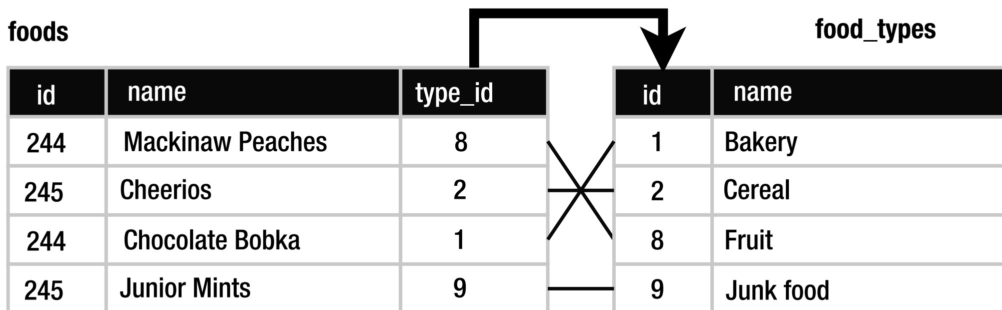
A relation may have one or more candidate keys. If it does, then which one of them that is defined as the primary key is arbitrary:

> *Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called the primary key of the relation.*

The primary key is just a rule that requires that every relation have at least one candidate key. The definition of a primary key serves more as an affirmation of the Guaranteed Access Rule than it does in defining a new relational concept.

Foreign Keys

With keys comes more than simply identification. While keys define relationships between tuples within a single relation (you might say vertically), they can also define relationships between tuples in different relations (horizontally). A key's identification property allows a tuple in one relation to identify (or reference) a specific tuple in another relation by way of a common key value. This is called a *foreign key relationship*. Specifically, a key in one table corresponds to, or references, the primary key in another table (the foreign key), thereby relating the tuples in each table. Take, for example, the `foods` and `food_types` tables, as shown in Figure 3-6. Each row in `foods` corresponds to a distinct food item (Junior Mints, Mackinaw Peaches, etc.), the name of which is stored in the `name` attribute. Each row in `food_types` stores various food classifications (e.g., Junk Food, Fruit, etc.). Each row in `foods` references a row in `food_types` by using a common key. `foods` contains a key called `type_id`, every value of which corresponds to a value in the primary key (`id`) of the `food_types`, as illustrated by the arrow in Figure 3-6.



foods

id	name	type_id
244	Mackinaw Peaches	8
245	Cheerios	2
244	Chocolate Bobka	1
245	Junior Mints	9

food_types

id	name
1	Bakery
2	Cereal
8	Fruit
9	Junk food

Figure 3-6. *Foreign key relationship between foods and food_types*

From a relational standpoint, this is merely a foreign key relationship. But in reality it is more than that. This relationship models a relationship in the real world. It has a basis in reality, has real meaning, and adds information above and beyond the information contained in the individual tables. Therefore, it is a critical part of the information itself.

It is easy to see that such relationships, if not properly maintained, can be precarious. If an application comes to assume this relationship, and its normal operation depends on the fact that a tuple in `foods` always has a corresponding tuple in `food_types`, what happens if someone deletes all the tuples from `food_types`? Where does that leave the `type_id` values in the `foods` tuples pointing to now? What has become of this relationship?

For that matter, what is to stop a user from just ignoring the primary key rule and jamming a thousand identical `food_types` tuples back into the database? If there is nothing in place to protect and ensure these relationships, they can be as destructive as they are beneficial. Such rules, like laws, are worthless without enforcement.

Constraints

Enter the *constraint*. Constraints are relational cops. They enforce database rules and relationships and preserve order in general. They bring consistency and uniformity to information within a database. With constraints, you can rest assured that tuples in the `foods` table will always reference legitimate tuples in `food_types`. Primary keys will always exist in tables, and their values will always be unique. This kind of uniformity and consistency is called *data integrity*. It is the integrity component of the relational model.

Constraints work by governing database operations. Like the precogs in *Minority Report*, they stop bad things before they happen. If a user or an application issues a request that would result in an inconsistent relationship or data, the database refuses to carry out the operation and issues an error, called a *constraint violation*. In the relational model, constraints fall into four general classes of integrity:

- **Domain integrity**: Domain integrity is the relationship between attribute values and their associated domains. In the relational model, domain integrity is instituted through *domain constraints*. A domain constraint requires that each attribute value in a tuple exist within its associated domain. For example, if the `type_id` attribute in the `foods` table is declared as an integer, then the corresponding values of `type_id` in all tuples in the `foods` table must be integer values—not floating-point numbers or strings. Domain integrity is also referred to as *attribute integrity*—it pertains to the attributes of a relation. Domain integrity is not limited to just checking that a given value resides in a given domain. It includes additional constraints as well, such as `CHECK` constraints. `CHECK` constraints (which are covered in Chapter 4) can define arbitrarily complex rules on what constitutes a permissible value for a given attribute.

- **Entity integrity**: This form of integrity is mandated by the Guaranteed Access Rule: each tuple in a relation must be uniquely identifiable. Whereas domain integrity is concerned with a relation's attribute values, entity integrity is concerned with its tuples. The term "entity" here is a rather loose term for table. It originates from database modeling (i.e., entity-relationship diagrams). In this particular context, an entity simply refers to anything in the real world that must be represented in a database.

- **Referential integrity**: This form of integrity pertains to relationships between tables, specifically the preservation of foreign key relationships. Whereas entity integrity pertains to tuples in a relation, referential integrity pertains to tuples between relations.

- **User-defined integrity**: User-defined integrity encompasses any form of integrity not defined in the other forms. Many relational databases offer various facilities that go beyond the normal constraint mechanisms. One such example is triggers, which are covered in Chapter 4.

The relational model requires that databases and query languages support integrity constraints. Furthermore, these constraints, like all other data and metadata in the database, must also be defined directly in the database, specifically in the system catalog:

> ***10. Integrity Independence.*** *Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs.*

Null Values

Closely associated with data integrity in the relational model is a special value (or lack thereof), which exists both inside and outside of every domain. This special value denotes the *absence* of a value and is called a *null value*, or *null* for short. Nulls are prescribed in Codd's third rule:

> ***3. Systematic Treatment of Null Values.*** *Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.*

There are multiple interpretations for what null values mean. The prevailing view of null seems to be "unknown." But there are others. For example, a tuple containing employee information may have an attribute for the employee's middle name. But not everyone has a middle name. A tuple for a person without a middle name might use a null value for that particular attribute. In this case, the null value doesn't necessarily mean "unknown" but rather "not applicable." Thus, a value may be null because it is either missing (a value exists but was not input) or uncertain (it is not known whether a value exists at all), or it is simply not applicable for the tuple (or employee) in question.

The inclusion of nulls in the relational model has been a source of controversy for many years, and there are people on both sides of the debate who feel very strongly for their positions. Codd, for example, felt the need for nulls:

> *In general, controversy still surrounds the problem of missing and inapplicable information in data bases. It seems to me that those who complain loudly about the complexities of manipulating nulls are overlooking the fact that handling missing information and inapplicable information is inherently complicated.*

Date, Codd's colleague and well-known authority on the relational model, is opposed to them:

> *…we should make it very clear that in our opinion (and in that of many other writers too, we hasten to add), nulls and 3VL are a serious mistake and have no place in a clean formal system like the relational model.*

The "3VL" acronym in the quote stands for "three-valued logic," which corresponds to how nulls are evaluated in logical expressions. This is addressed in Chapter 4.

Normalization

The implication of no duplicate tuples provided by the Guaranteed Access Rule gives rise to an important concept in database design called *normalization*. Normalization concerns itself with the organization of attributes within relations so as to minimize duplication of data. Data duplication, as you will see, has more deleterious effects than just taking up unnecessary space. It increases the opportunity for database inconsistencies to arise. Normalization is about designing your database to be more resistant to the ill effects of thoughtless users and buggy programs.

While based on principles of the relational model, normalization is somewhat a subject (some even say an art) in itself. It can become quite complicated, introducing a considerable

amount of new concepts and terminology. A proper treatment of the subject is beyond the scope of this book. What follows is a very brief introduction.

Normal Forms

As stated, the chief aim of normalization is to eradicate duplication. Relations that have duplication removed are said to be *normalized*. However, there are degrees of normalization. These degrees are called *normal forms*. The first degree is called *first normal form*, followed by *second normal form*, and so on. They are abbreviated 1NF, 2NF, 3NF, and so on. Each normal form defines specific conditions a relation must meet in order to be so classified. Thus a relation that meets the conditions of first normal form is said to be "in first normal form." Normal forms build on each other so that higher normal forms require all the conditions of lower forms as prerequisites. For data to be in 2NF, it must also be in 1NF. Essentially, the higher the normal form of a relation, the less duplication it has, and the more resistant it is to inconsistencies. The most common and perhaps most widely used normal form is 3NF, although there are even more advanced normal forms such as Boyce-Codd normal form (BCNF), 4NF, 5NF, and higher. The first three normal forms are easy enough to describe.

First Normal Form

First normal form simply states that all attributes in a relation use domains that are made up of atomic values. The working definition of "atomic" is the same as other disciplines, meaning simply "that which cannot be broken down further." For example, integer values would seem to be atomic, as you can't break them down further. But you could argue that integer values could be decomposed into their prime factors. Atomicity, then, is determined by the method of decomposition, which can be a subjective matter. For all practical purposes, it is the database management system that decides what is atomic. And the domains provided by the system can safely be considered as such.

That said, first normal form basically means that a single attribute cannot hold more than a single value. For example, take the `episodes` table shown in Table 3-2 in the next section. 1NF states that you couldn't store both the values 1992 and 1993 (two integer values) in a `year` attribute of a single tuple. It sounds so silly that it can be kind of hard to imagine. But it's that simple. It is so simple, in fact, that you have to work at violating 1NF; most databases won't even give you the means to do it.

Functional Dependencies

To understand second and third normal form, you have to first understand *functional dependencies*. The simple definition of a functional dependency is a correlation between columns in a table. If the values of one column (or set of columns) correlate to the values of another, then they have a functional dependency. More precisely, a functional dependency describes a relationship between two or more attributes in a relation such that the value of one attribute (or set of attributes) can be inferred from the value of another attribute (or attributes) for every tuple in the relation. This is more easily illustrated by example. Consider the table, called `episodes`, shown in Table 3-2.

There is a functional dependency between `season` and `year`. For any given value of `season`, you will find the same value of `year`. If you ever come across a tuple with a value of 4 for `season`, you know the value for `year` will be 1992. If you know the former, you can determine the latter.

Table 3-2. *The Unnormalized episodes Table*

season	week	year	name
4	1	1992	The Junior Mint
4	2	1992	The Smelly Car
5	1	1993	The Mango
5	2	1993	The Puffy Shirt
6	21	1994	The Fusilli Jerry
6	25	1994	The Understudy

Many times, functional dependencies are a warning sign that duplication lurks within a table. And already you can see in this example how inconsistencies can crop up. If there is in fact a correlation between `year` and `season`, what happens to that relationship if someone modifies the first row so that its value for `year` is 1999 (but fails to do so for the second row)? That relationship has been compromised. It is inconsistent. One tuple with `season=4` has `year=1992`, and another with `season=4` has `year=1999`. How can season 4 have happened in both 1992 and 1999? This is logically inconsistent, and the functional dependency (or lack of sufficient normalization) is what made it possible to introduce this inconsistency. What is even more interesting is that there is no standard integrity constraint designed to guard against this problem. It is purely the result of bad design.

Functional dependencies always involve exactly two sets of attributes: one set (the determinant) determines (or relates to) the value of another (the dependent). Call the attribute(s) making up the determinant *A* and the attribute(s) making up the dependent *B*. With this in place, you can express this relationship using the following two (equivalent) statements:

1. *B* is functionally dependent upon *A*.
2. *A* functionally determines *B*.

There is even a fancy notation for this as well: $A \rightarrow B$. The bottom line: for any value of *A*, the value of *B* has some kind of correlation.

Second Normal Form

Second normal form is defined in terms of functional dependencies. It requires that a relation be in first normal form *and* that all non-key attributes be functionally dependent upon *all* attributes of the primary key (not just part of them). Remember that normalization is about cutting out duplication, so while this sounds like an arbitrary rule, it is in fact aimed at weeding out duplication.

You have already seen what duplication looks like when 2NF is not followed, using `episodes` as an example. The primary key in `episodes` is composed of both `season` and `week`. You know that `year` is functionally dependent on `season`, which in turn is only part of the primary key (red-flag). Specifically, for every `season=`*x* (e.g., 4) you have the same `year=`*y* (e.g., 1992). There is no reason to include the `year` attribute in the relation if its value is correlated with `season`. That's duplication. It must go.

So what do you do? You decompose the table into two tables. Cut out `year`, and move it to its own table: call it the `seasons` table. Now `episodes` is decomposed into two tables with a foreign key relationship where `episodes.season` references `seasons.season` (see Tables 3-3 and 3-4).

Table 3-3. *The episodes Table in Second Normal Form*

season	week	name
4	1	The Junior Mint
4	2	The Smelly Car
5	1	The Mango
5	2	The Puffy Shirt
6	21	The Fusilli Jerry
6	25	The Understudy

Table 3-4. *The seasons Table from Normalizing episodes*

season	year
4	1992
5	1993
6	1994

This is like factoring out a common variable in an algebraic expression. Now `episodes` is in 2NF. Furthermore, no information is lost because `year` is functionally determined by `season`. But look what else you get: the new design allows referential integrity to back you up: you have a foreign key constraint guarding this relationship (the correlation between `season` and `year`), which you couldn't get in the previous form. What was previously only implied is now both explicit and enforceable.

Now consider the specific case mentioned earlier where someone modifies the first row. The logical inconsistency is no longer possible: it is impossible to mess up the year, because it is not in `episodes`. If someone changes `year`=1992 to 1999 in `seasons`, it may be inaccurate, but it is still logically consistent. That is, both rows that refer to it in `episodes` (`season`=4) will still be in agreement about the year in which `season` 4 took place. Normalization cannot make you get your facts right, but it can ensure that at least your data is consistent about them.

Second normal form works by tightening the specificity between the primary key and the non-key attributes. If the values of a non-key attribute correlate to only *part* of the primary key, then logically that attribute can introduce duplication. Why? Because only part of a primary key is not unique (*all* of the primary key is required for uniqueness, as mentioned earlier), and a correspondence between two columns, neither of which is unique, opens the possibility for duplication, and duplication invites inconsistency.

Third Normal Form

Third normal form shifts attention from functional dependencies on the primary key to dependencies on any other non-key attributes in a relation. It roots out a special class of functional dependencies called *transitive dependencies*. A transitive dependency is just a chain of two or more functional dependencies spanning two or more attribute groups (two or more correlations). For example, say you have a relation with three sets of attributes, *A*, *B*, and *C*, where *A* is the primary key, and *B* is a candidate key. If $A \rightarrow B$ and $B \rightarrow C$, then *C* is transitively dependent on *A*: it depends on *A* indirectly through its relationship with *B*. Transitive dependencies harbor duplication. Third normal form dictates that a relation must be in second normal form and also have no transitive dependencies.

To picture this, let's rig the original `episodes` table to have an integer primary key called `id`, and the candidate key is now the combination of `season` and `week`. This relation is shown in Table 3-5.

Table 3-5. *An Unormalized episodes Table with an Integer Primary Key*

id	season	week	year	name
1	4	1	1992	The Junior Mint
2	4	2	1992	The Smelly Car
3	5	1	1993	The Mango
4	5	2	1993	The Puffy Shirt
5	6	21	1994	The Fusilli Jerry
6	6	25	1994	The Understudy

In this version of `episodes`, `year` is functionally dependent on `season`, which in turn is functionally dependent on `id`. So we have `id` → `season` → `year`. How do you know this? Work backwards. Ask yourself if you can determine `season` from `id`. Yes. Then can you determine `year` from `season`? Yes. Therefore, you have a transitive dependency: `id` functionally determines `season` and `season` functionally determines `year`. To be in third normal form, `year`, which is functionally dependent on a non-key attribute, must go. And as with the previous example, you relegate `year` into its own table, again splitting `episodes` into two tables, as shown in Tables 3-6 and 3-7.

Table 3-6. *The episodes Table in Third Normal Form*

id	season	week	name
1	4	1	The Junior Mint
2	4	2	The Smelly Car
3	5	1	The Mango
4	5	2	The Puffy Shirt
5	6	21	The Fusilli Jerry
6	6	25	The Understudy

Just as in the 2NF example, decomposition made an implicit relationship explicit.

Table 3-7. *The seasons Table from Normalizing episodes*

season	year
4	1992
5	1993
6	1994

Since functional dependencies between non-key attributes are not allowed in a table, you may wonder why it is okay for functional dependencies to exist between keys. The answer is simple: uniqueness. Even if there is a correlation between one key and another, the fact that they are unique guarantees that their correlation is also unique; therefore, no duplication exists.

In the end, 2NF and 3NF aim not to introduce confusing rules, but to seek out and destroy duplication and the inconsistencies that can arise from it. It is essential to proper database design. The more important your data, the more attention you should pay to ensure that your database is properly designed and normalized.

This book was purchased by ybakos@mines.edu

The Manipulative Component

The manipulative component of the relational model defines the ways in which information can be manipulated and changed. It is the dynamic part of the model that connects the data in the logical view to the outside world.

Relational Algebra and Calculus

In his original paper, Codd described a language that could be used to operate on data:

> *The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus… Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented).*

This "universal data sublanguage" would have a sound mathematical basis. Codd defined this basis in the form of relational algebra and relational calculus, illustrated in his 1972 paper "Relational Completeness of Data Base Sublanguages." As described in the abstract of that paper:

> *In the near future, we can expect a great variety of languages to be proposed for interrogating and updating data bases. This paper attempts to provide a theoretical basis which may be used to determine how complete a selection capability is provided in a proposed data sublanguage independently of any host language in which the sublanguage may be embedded.*

> *A relational algebra and relational calculus are defined. Then, an algorithm is presented for reducing an arbitrary relation-defining expression (based on the calculus) into a semantically equivalent expression of the relational algebra.*
>
> *Finally, some opinions are stated regarding the relative merits of calculus-oriented versus algebra-oriented sublanguages from the standpoint of optimal search and highly discriminating authorization schemes.*

These two "pure" languages—relational algebra and calculus—focused on mathematical theory rather than a particular language syntax. The latter is the job of the "data sublanguage," or "query language," as we know it today. Like the structural part of the relational model, the manipulative part drew heavily from mathematics. Relational algebra has its basis in set theory. Likewise, relational calculus has its basis in predicate calculus. From a computer science perspective, relational algebra can be considered more of a *procedural language* while relational calculus is more of a *declarative language.* The meanings of these terms are explained in more detail in Chapter 4.

As stated in the quote, while the particular forms of expression in the two languages are different, they are nevertheless logically equivalent. That is, any operation in relational algebra can also be expressed in terms of calculus, and vice versa. Another way of saying this is that the two languages have the same *expressive power*—the same fundamental operations can be performed or expressed in either system.

Relational Query Language

Together, relational algebra and calculus serve as a guideline, or a yardstick as Codd describes it, for query languages implemented in relational databases. Any query language that can express all of the fundamental operations set forth in relational algebra and/or calculus is said to be *relationally complete.*

The query language must also address the other aspects (structural and integrity) of the relational model as well. This is summed up in Codd's fifth rule, defined as follows:

> ***5. Comprehensive Data Sublanguage Rule.*** *A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items:*
>
> *Data Definition, View Definition, Data Manipulation (Interactive and by program), Integrity Constraints, and Authorization, Transaction boundaries (begin, commit, and rollback).*

Additionally, Codd's seventh rule requires that the database (and by extension the query language) not only use relations for storage, manipulation, and retrieval, but also as operands for the purpose of modifying the database:

> ***7. High Level Insert, Update, and Delete.*** *The system must support set at a time insert, update, and delete operators.*

A user then should be able to insert tuples by providing a relation composed of the tuples to be inserted, and similarly for updating and deleting information within the system. While this rule keeps things consistent—using relations—its primary intent at the time was actually in optimizing database performance. The idea was that in some cases the database could make better optimizations by seeing modifications together as a set than it could by processing them individually.

So, first, a relational database must provide at least one query language that addresses the structural, integrity, and manipulative aspects of the relational model. And with respect to the manipulative aspect, it must be capable of expressing the mathematical concepts set forth in relational algebra and/or calculus. Furthermore, it must accept relations as a means for modifying data in the system. Note that Codd never mandated a particular query language. He only mandated that a relational database provide one and what it must do.

The Advent of SQL

Thus, over the years, there have been multiple competing query languages. However, the most popular and widely adopted of these languages today is undoubtedly SQL. SQL is a relationally complete query language that exhibits aspects of both relational algebra and relational calculus. That is, it has both declarative features (calculus) and procedural features (algebra). In fact, as you will see in the next chapter, SQL includes almost all of the operators defined in relational algebra.

SQL also reflects each aspect of the relational model. Part of its language is dedicated to working with the structural aspect of the model, specifically to creating, altering, and destroying tables. This part of the language is called data definition language (DDL). Within DDL lies also the integrity aspect, allowing the creation of keys and various database constraints. Likewise, part of the language is dedicated to the operational aspect, called data manipulation language (DML). And as stated, it includes ideas from both relational algebra and calculus.

Ironically, despite the clear influence of the relational model on SQL, the current SQL standard does not mention the relational model or use relational terminology.[2] And while SQL is relationally complete, in many ways it falls short of the true power of the relational model. Although it was primarily inspired by relational calculus, some have claimed that there are ways in which SQL also violates it. Furthermore, SQL lacks some relational operations that some consider important, such as relational assignment, and has a number of redundant features. Part of the reason for this was that the organization(s) responsible for creating the SQL standard felt that it was more important to release a standard as early as possible in order to establish a base that database implementations could build upon (Connolly, 2001). Thus, when the initial standard was released in 1987, though practical, it was found wanting in many ways by researchers involved with the relational model (Codd included). Among other things, it omitted some relational operations and included no mention of referential integrity constraints. Subsequent standards filled in some of the gaps here and there, but this seems to have done little to appease its detractors or deter database vendors from "extending" their SQL dialects to include proprietary features.

Entire books and websites are devoted to both SQL's inadequacies as well as what an ideal query language should be. Furthermore, alternative query languages have been proposed and implemented (both before and after SQL) that in the minds of their creators are more expressive and better reflect the principles and intent of the relational model.[3]

---

2. See `http://en.wikipedia.org/wiki/Relational_model`.
3. Tutorial D is one such example. See `www.thethirdmanifesto.com` for more information.

Despite its criticisms and shortcomings, however, SQL is what we have to work with. It is undoubtedly the most popular and widely adopted query language in the industry, and given its longstanding dominance in the marketplace, it is unlikely that its position will change any time in the near future.

The Meaning of Relational

Given all this information, what exactly is *relational*? It is a common misconception that relational databases derive the name "relational" from their ability to "relate" a column in one table to a column in another through a foreign key relationship. The true meaning of relational, however, stems from the central structural component of the relational model: the relation, which itself is based on the mathematical concept. First and foremost, a relational database is one that uses relations as the sole structural unit in which to represent information, as mandated by the Information Principle.

A more specific definition of relational, however, is provided by Codd's Rule Zero:

> ***Rule Zero.*** *For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.*

For a database to be called relational, it must provide all of the facilities required by the relational model. That is, it must conform to all of the other 12 rules. And believe it or not, you have covered every one of them. You therefore should have a good idea at this point what it means to be relational. Don't get too cocky, though, as Codd later expanded the 12 rules to over 300.

Summary

This was a nickel tour through the relational model. While our discussion isn't definitive, my goal was to give you enough history to make the topic enjoyable and enough theory for you to understand some of the thinking behind SQL—why it works the way it does. The relational model has clearly influenced its design and has provided it with a solid theoretical foundation.

The relational model has proven itself over its 30-year history. It has had an enormous impact not only on computing but on the way we do business. Today, it can be found in a wide array of electronic machinery, ranging from mainframes to cell phones.

The relational model was created to provide a logical, consistent representation of data that is independent of hardware and software. The model built on well-founded theory set forth in mathematics. This model provides database users with a consistent, unchanging view of information, powerful methods to operate on it, and mechanisms to protect and ensure its consistency and integrity.

There are many more aspects to the relational model, and much that builds on it. It is the subject of entire books, some of which I have included in the "References" section. While it is a large subject, its core concepts are logical and straightforward. These concepts ground, frame, and form the basis of the subject covered in the next chapter: SQL.

If you are new to SQL and you've patiently endured this chapter, you should have a much easier time grasping the concepts in the next chapter. You will see SQL not as an arbitrary language but as a gateway into a powerful database management system. You will find that its syntax is heavily geared to what you've learned here in this chapter.

References

Codd, E. F. 1970. "A relational model for large shared data banks." *Communications of ACM*, 13(6): 377–387.

Codd, E. F. 1972. Relational Completeness of Data Base Sublanguages in Data Base Systems. In Rustin, R. J. (ed.), *Data Base Systems*, Courant Computer Symposia Series, v. 6. Englewood Cliffs, NJ: Prentice-Hall.

Codd, E. F. 1979. "Extending the Database Relational Model to capture more meaning." *ACM Transactions on Database Systems.* 4(4): 397–434.

Codd, E. F. 1980. Data Models in Database Management. In *Proceedings of the 1980 Workshop on Data Abstraction, Databases and Conceptual Modeling* (Pingree Park, CO, June 23–26, 1980). New York: ACM Press, 112–114.

Codd, E. F. 1982. "Relational database: a practical foundation for productivity." *Communications of ACM* 25(2): 109–117.

Codd, E. F. 1985. "Is your DBMS really relational?," *Computerworld* (Part 1: October 14, 1985, Part 2: October 21, 1985).

Connolly, C. E. B. 2001. *Database Systems: A Practical Approach to Design, Implementation and Management.* Boston: Addison-Wesley.

Date, C. J. 1999. *Thirty Years of Relational: Relational* (series of 12 articles), Intelligent Enterprise 1, Nos. 1–3 and 2, Nos. 1–9 (October 1998 onward). Note: Most installments after the first publication in the online portion of the magazine at `www.intelligententerprise.com`, accessed on March 16, 2006.

Date, C. J. 2003. *An Introduction to Database Systems,* 8th ed. Boston: Addison-Wesley.

Grimaldi, R. P. 1998. *Discrete and Combinatorial Mathematics,* 4th ed. Boston: Addison-Wesley.

Reiter, R. 1978. On Closed World Data Bases. In Gallaire, H. and Minker, J. (eds.), *Logic and Data Bases.* New York: Plenum, 119–140.

Silberschatz, A., H. F. Korth, and S. Sidharshan. 2002. *Database System Concepts.* New York: McGraw Hill.