CHAPTER 5

The Relational Data Model

Abstract

This chapter presents the theory of the relational data model. It discusses characteristics of the columns and rows in a table, the importance of choosing good primary keys, using concatenated primary keys, using foreign keys to represent data relationships, referential integrity, views, and the use of data dictionary tables to store a database design.

Keywords

relational data model
database tables
relations
primary keys
foreign keys
database views
data dictionaries
referential integrity

Once you have a completed ER diagram, you can translate that conceptual logical schema into the formal data model required by your DBMS. Today, most new database installations are based on the relational data model. We call databases that adhere to that model *relational databases*.

Note: The older data models that are described in Appendix A are still in use, in many legacy database systems. However, it is extremely rare to find a business creating a new one. On the other hand, the object-oriented data model is still current and although it has not replaced the relational data model and does not appear to be doing so, some new installations use either object-oriented or a combination of relational and object-oriented (see Chapter 27). The existence of extremely large data stores has also led to the development of post-relational DBMSs known as NoSQL. You will read about them in Chapter 28.

A relational database is a database whose logical structure is made up of nothing but a collection of *relations*. Although you may have read somewhere that a relational database has "relationships between files," nothing could be further from the truth. In this chapter, you will learn exactly what a relational database is, and how relations provide representations of data relationships.

Note: Remember from Chapter 4 that we said that a DBMS isolates database users from physical storage. A logical data model, therefore, has absolutely nothing to do with how the data are stored in files on disk.

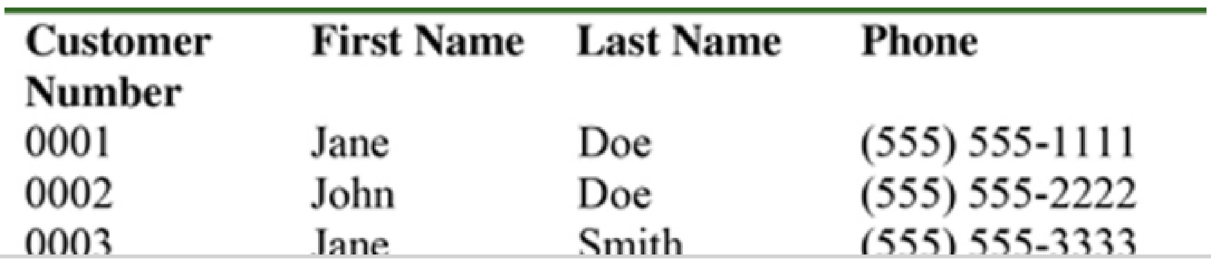
The relational data model is the result of the work of one man—Edgar (E. F.) Codd. During the 1960s, Dr. Codd, although trained as a mathematician, was working with existing data models. His experience led him to believe that they were clumsy and unnatural ways of representing data relationships. He therefore went back to mathematical set theory and focused on the construct known as a relation. He extended that concept to produce the relational database model, which he introduced in a paper in 1970.

Note: You will find the citations for Codd's original paper and his other writings on the relational data model in the For Further Reading section at the end of this chapter.

Note: E. F. Codd was born in England in 1923 and later migrated to the United States, where he did most of his work on the relational data model at IBM's Watson Research Center. He died in 2003.

Understanding Relations

In mathematical set theory, a *relation* is the definition of a table with columns (*attributes*) and rows (*tuples*). (The word "table" is used synonymously with "relation" in the relational data model, although, to be strictly correct, not every table is a relation.) The definition specifies what will be contained in each column of the table, but does not include data. When you include rows of data, you have an *instance* of a relation, such as the small *customer* relation in Figure 5.1.



Customer Number	First Name	Last Name	Phone
0001	Jane	Doe	(555) 555-1111
0002	John	Doe	(555) 555-2222
0003	Jane	Smith	(555) 555-3333

0003	Jane	Smith	(555) 555-3333
0004	John	Smith	(555) 555-4444

FIGURE 5.1 **A simple *customer* relation.**

At first glance, a relation looks much like a flat file or a rectangular portion of a spreadsheet. However, because it has its underpinnings in mathematical set theory, a relation has some very specific characteristics that distinguish it from other rectangular ways of looking at data. Each of these characteristics forms the basis of a constraint that will be enforced by the DBMS.

Columns and Column Characteristics

A column in a relation has the following properties:

- A name that is unique within the table: Two or more tables within the same relational database schema may have columns with the same names —in fact, as you will see shortly, in some circumstances this is highly desirable—but a single table must have unique column names. When the same column name appears in more than one table, and tables that contain that column are used in the same data manipulation operation, you qualify the name of the column by preceding it with the name of its table and a period, as in:

```
customer.customer_number
```

- A domain: The values in a column are drawn from one and only one domain. As a result, relations are said to be *column homogeneous*. In addition, every column in a table is subject to a domain constraint. Depending on your DBMS, the domain constraint may be as simple as a data type, such as integers or dates. Alternatively, your DBMS may allow you to create your own, very specific, domains that can be attached to columns.
- There are no "positional concepts." In other words, the columns can be viewed in any order without affecting the meaning of the data.

Rows and Row Characteristics

In relational design theory, a row in a relation has the following properties:

- Only one value at the intersection of a column and row: A relation does not allow multivalued attributes.
- Uniqueness: There are no duplicate rows in a relation.

Note: for the most part, DBMSs do not enforce the unique row constraint automatically. However, as you will see in the next bullet, there is another way to obtain the same effect.

- A primary key: A *primary key* is a column or combination of columns with a value that uniquely identifies each row. As long as you have unique primary keys, you will ensure that you also have unique rows. We will look at the issue of what makes a good primary key in great depth in the next major section of this chapter.
- There are no positional concepts. The rows can be viewed in any order without affecting the meaning of the data.

Note: You can't necessarily move both columns and rows around at the same time and maintain the integrity of a relation. When you change the order of the columns, the rows must remain in the same order; when you change the order of the rows, you must move each entire row as a unit.

Types of Tables

A relational database works with two types of tables. *Base tables* are relations that are actually stored in the database. These are the tables that are described by your schema.

However, relational operations on tables produce additional tables as their result. Such tables, which exist only in main memory, are known as *virtual tables*. Virtual tables may not be legal relations—in particular, they may have no primary key—but because virtual tables contain copies of data in the base tables and are never stored in the database, this presents no problem in terms of the overall design of the database.

The use of virtual tables benefits a DBMS in several ways. First, it allows the DBMS to keep intermediate query tables in main memory, rather than storing them on disk, enhancing query performance. Second, it allows tables that violate the rules of the relational data model to exist in main memory without affecting the integrity of the database. Finally, it helps avoid fragmentation of database files and disk surfaces by avoiding repeated write, read, and delete operations on temporary tables.

Note: SQL, the language used to manage most relational databases, also supports "temporary base tables." Although called base tables, temporary tables are actually virtual tables in the sense that they exist only in main memory for a short time and are never stored in the physical database.

A Notation for Relations

You will see instances of relations throughout this book, used as examples. However, we do not usually include data in a relation, when documenting that relation. One common way to express a relation is as follows:

```
relation_name (primary_key, non_primary_key_column …)
```

For example, the *customer* relation that you saw in Figure 5.1 would be written as:

```
customer (customer_numb, first_name last_name, phone)
```

The preceding expression is a true relation, an expression of the structure of a relation. It correctly does not contain any data. (As mentioned earlier, when data are included, you have an instance of a relation.)

Primary Keys

As you have just read, a unique primary key makes it possible to uniquely identify every row in a table. Why is this so important? The issue is the same as with entity identifiers: You want to be able to retrieve every single piece of data you put into a database.

As far as a relational database is concerned, you should need only three pieces of information to retrieve any specific bit of data: the name of the table, the name of the column, and the primary key of the row. If primary keys are unique for every row, then we can be sure that we are retrieving exactly the row we want. If they are not unique, then we are retrieving only *some* row with the primary key value, which may not be the row containing the data for which we are searching.

Along with being unique, a primary key must not contain the value *null*. Null is a special database value meaning "unknown." It is not the same as a zero or a blank. If you have one row with a null primary key, then you are actually alright. However, the minute you introduce a second one, you have lost the property of uniqueness. We therefore forbid the presence of nulls in any primary key columns. This constraint, known as *entity integrity*, will be enforced by a DBMS whenever data are entered or modified.

Selecting a good primary key can be a challenge. As you may remember from Chapter 4, some entities have natural primary keys, such as purchase order numbers. These are arbitrary, meaningless unique identifiers that a company attaches to the orders it sends to vendors and are therefore ideal primary keys.

Occasionally, you will run across a relation that has two or more attributes (or combinations of attributes) that can serve as a primary key. Each of these possible primary keys is known as a *candidate key*. You will therefore need to choose from among the candidate keys: A relation can have only one primary key.

Primary Keys to Identify People

What about a primary key to identify people? The first thing that pops into your mind might be a social security number (or, for those outside the United States, a national identification number). Every person in the United States is supposed to have a social security number (SSN); parents apply for them in the hospital when a baby is born, right? And the US government assigns them, so they are unique, right? Unfortunately, the answer to both questions is "no."

The Social Security Administration has been known to give everyone in an entire town the same SSN; over time, SSNs are reused. However, these are minor problems, compared to the issue of the social security number being null.

Consider what happens at a college that uses social security numbers as student numbers when international students enroll. Upon entry into the country, the international students do not have SSNs. Because primary keys cannot be null, the international students cannot sign up for classes or even be enrolled in the college, until they have some sort of SSN.

The college's solution is to give them "fake" numbers in the format 999-99-XXXX, where XXXX is some number currently not in use. Then, when the student receives a "real" SSN from the government, the college supposedly replaces the fake value with the real one. Sometimes, however, the process doesn't work. A graduate student ended up with his first semester's grades being stored under the fake SSN, but the rest of his grades under his real number. (Rather than changing the original data, someone created an entire new transcript for the student.) When the time came to audit his transcript to see if he had satisfied all his graduation requirements, he was told that he was missing an entire semester's worth of courses.

This example leads us to two important desirable qualities of primary keys:

- A primary key should be some value that is highly unlikely ever to be null.
- A primary key value should never change. (It should be *immutable*.)

In addition, there is significant concern of security problems that can arise from the use of social security numbers as identifiers in a database. The danger of identity theft has made it risky to store a national identifier. Many US state governments, for example, have mandated that publicly-supported organizations use something other than the SSN as a customer/client/student ID to help protect individual privacy.

Although SSNs initially look like good natural identifiers, you will be much better off in the long run using arbitrary numbers for people—such as student numbers or account numbers—rather than relying on government-issued identification numbers.

Avoiding Meaningful Identifiers

It can be very tempting to code meaning into a primary key. For example, assume that *Antique Opticals* wants to assign codes to its distributors, rather than giving them arbitrary distributor numbers. Someone might create codes such as TLC for *The Laser Club*, and JS for *Jones Services*. At first, this might seem like a good idea. The codes are short and by looking at them, you can figure out which distributor they represent.

But what happens if one of the companies changes its name? Perhaps *Jones Services* is renamed *Jones Distribution House*. Do you change the primary key value in the distributor table? Do you change the code so that it reads JDH? If the distributor table were all that we cared about, that would be the easy solution.

However, consider that the table that describes merchandise items contains the code for the distributor, so that *Antique Opticals* can know which distributor provides the item (you will read a great deal more about this concept in the next major section of this chapter.) If you change the distributor code value in the distributor table, you must change the value of the code for every merchandise item that comes from that distributor. Without the change, *Antique Opticals* will not be able to match the code to a distributor and get information about the distributor. It will appear that that the item comes from a nonexistent distributor!

Note: This is precisely the same problem about which you read in Chapter 3, concerning Antique Opticals' identifiers for its customers.

Meaningful primary keys tend to change and therefore introduce the potential for major data inconsistencies between tables. Resist the temptation to use them, at all costs. Here, then, is yet another property of a good primary key:

- A primary key should avoid using meaningful data. Use arbitrary identifiers or concatenations of arbitrary identifiers wherever possible.

Many of the Web sites on which you have accounts will use your e-mail address as your user ID. This is another example of why primary keys should be immutable. Our e-mail addresses change occasionally. (By this time, most of us dread having to change our primary e-mail address, but should be immutable. Our e-mail addresses change occasionally. (By this time, most of us dread having to change our primary e-mail address, but doing so can be forced on us by circumstances.) If a company is using a relational database with an e-mail address as the primary key for its customer table, a change in that key becomes a major event. Should the company keep a history of all the user's e-mail addresses and leave references from old orders intact? Should the all the user's orders (even those that have been filled) be changed to reflect the new e-mail address? If the company doesn't do one or the other, orders with an old e-mail address will be left hanging with no connection to a customer. No matter what the company decides to do, it will require some significant data modification and tight controls to ensure that the data remain consistent.

It is not always possible to use completely meaningless primary keys. You may find, for example, that you need to include dates or times in primary keys to distinguish between events. The suggestion that you should not use meaningful primary keys is therefore not a hard-and-fast rule, but a guideline to which you should try to adhere whenever it is realistic to do so.

Concatenated Primary Keys

Some tables have no single column in which the values never duplicate. As an example, look at the *order items* table in Figure 5.2. Because there is more than one item on an order, order numbers are repeated; because the same item can appear on more than one order, item numbers are repeated. Therefore, neither column by itself can serve as the table's primary key.

Order Number	Item Number	Quantity
10991	0022	1
10991	0209	2
10991	1001	1
10992	0022	1
10992	0486	1
10993	0209	1
10993	1001	2
10994	0621	1

FIGURE 5.2 A sample order items table.

However, the combination of an order number and an item number *is* unique. We can, therefore, concatenate the two columns to form the table's primary key.

It is true that you could also concatenate all three columns in the table, and still ensure a unique primary key. However, the quantity column is not necessary to ensure uniqueness and therefore should not be used. We now have some additional properties of a good primary key:

- A concatenated primary key should be made up of the smallest number of columns necessary to ensure the uniqueness of the primary key.
- Whenever possible, the columns used in a concatenated primary key should be meaningless identifiers.

All-Key Relations

It is possible to have a table in which every column is part of the primary key. As an example, consider a library book catalog. Each book title owned by a library has a natural unique primary key—the ISBN (International Standard Book Number). Each ISBN is assigned to one or more subject headings in the library's catalog; each subject heading is also assigned to one or more books. We therefore have a many-to-many relationship between books and subject headings.

A relation to represent this relationship might be:

```
subject_catalog (isbn, subject_heading)
```

All we need to do is pair a subject heading with a book identifier. No additional data are needed. Therefore, all columns in the table become part of the primary key.

There is absolutely no problem with having all-key relations in a database. In fact, they occur whenever a database design contains a composite entity that has no relationship data. They are not necessarily an error, and you can use them wherever needed.

Representing Data Relationships

Representing Data Relationships

In the preceding section we alluded to the use of identifiers in more than one relation. This is the one way in which relational databases represent relationships between entities. To make this concept clearer, take a look at the three tables in Figure 5.3.

Items

Item Number	Title	Distributor Number	Price
1001	Gone with the Wind	002	39.95
1002	Star Wars IV: Special Edition	002	59.95
1003	Die Hard	004	29.95
1004	Bambi	006	29.95

Orders

Order Number	Customer Number	Order Date
11100	0012	12/18/09
11101	0186	12/18/09
11102	0056	12/18/09

Order Item

Order Number	Item Number	Quantity	Shipped?
11100	1001	1	Y
11100	1002	1	Y
11101	1002	2	Y
11102	1002	1	N
11102	1003	1	N
11102	1001	1	N

FIGURE 5.3 **Three relations from the *Antique Opticals* database.**

Each table in the illustration is directly analogous to the entity by the same name in the *Antique Opticals* ER diagram. The *orders* table (the Order entity) is identified by an order number, an arbitrary unique primary key assigned by *Antique Opticals*. The *items* table (the Item entity) is identified by an item number, which could be another arbitrary unique identifier assigned by the company (often called an SKU, for stock keeping unit) or a UPC.

The third table—*order item* (the Order Item entity)—tells the company which items are part of which order. As you saw earlier in this chapter, this table requires a concatenated primary key because multiple items can appear on multiple orders. The columns in this primary key, however, have more significance than simply uniquely identifying each row. They also represent a relationship between the order items, the orders on which they appear, and the items being ordered.

The *item number* column in the *order item* relation is the same as the primary key of the *item* table. This indicates a one-to-many relationship between the two tables. By the same token, there is a one-to-many relationship between the *order* and *order item* tables because the *order number* column in the *order item* table is the same as the primary key of the *orders* table.

When a table contains a column (or concatenation of columns) that is the same as the primary key of some table in the database, the column is called a *foreign key*. The matching of foreign key values to primary key values represents data relationships in a relational database. As far as the user of a relational database is concerned, there are no structures that show relationships other than the matching column's values.

Note: This is why the idea that relational databases have "relationships between files" is so absurd. The relationships in a relational database are between logical constructs—tables—and nothing else. Such structures make absolutely no assumptions about physical storage.

are between logical constructs—tables—and nothing else. Such structures make absolutely no assumptions about physical storage.

Foreign keys may be part of a concatenated primary key, or they may not be part of their table's primary key at all. Consider, for example, a pair of simple *Antique Opticals customer* and *order* relations:

```
customer (customer_numb, first_name, last_name, phone)

order (order_numb, customer_numb, order_date)
```

The customer number column in the *order* table is a foreign key that matches the primary key of the *customer* table. It represents the one-to-many relationship between customers and the orders they place. However, the customer number is not part of the primary key of its table; it is a nonkey attribute that is nonetheless a foreign key.

Technically, foreign keys need not have values unless they are part of a concatenated primary key; they can be null. For example, there is no theoretical reason that the *orders* relation above must have a value for the customer number. It is not part of the primary key of the table. However, in this particular database, *Antique Opticals* would be in serious trouble if customer numbers were null: There would be no way to know which customer placed an order!

A relational DBMS uses the relationships indicated by matching data between primary and foreign keys. For example, assume that an *Antique Opticals* employee wanted to see what titles had been ordered on order number 11102. First, the DBMS identifies the rows in the *order item* table that contain an order number of 11102. Then, it takes the item numbers from those rows and matches them to the item numbers in the *item* table. In the rows where there are matches, the DBMS retrieves the associated data.

Referential Integrity

The procedure described in the preceding paragraph works very well—unless, for some reason, there is no order number in the *order* table to match a row in the *order item* table. This is a very undesirable condition, because there would be no way to ship the ordered items because there would be no way to find out which customer placed the order.

This relational data model therefore enforces a constraint called *referential integrity*, which states that *every nonnull foreign key value must match an existing primary key value*. Of all the constraints on a relational database, this is probably the most important, because it ensures the consistency of the cross-references among tables.

Referential integrity constraints are stored in the database, and enforced automatically by the DBMS. As with all other constraints, each time a user enters or modifies data, the DBMS checks the constraints, and verifies that they are met. If the constraints are violated, the data modification will not be allowed.

Concatenated Foreign Keys

A foreign key does not necessarily need to be made up of a single column. In some cases, you may have a concatenated foreign key that references a concatenated primary key. As an example, let us consider a very small accounting firm that uses the following relations:

```
accountant (acct_first_name, acct_last_name, date_hired,
     office_ext)

customer (customer_numb, first_name, last_name, street,
     city, state_province, zip_postcode, contact_phone)

job (tax_year, customer_numb, acct_first_name,
     acct_last_name)

form (tax_year, customer_numb, form_id, is_complete)
```

Because the firm is so small, the database designer decides that employee numbers are not necessary and, instead, uses the accountants' first and last names as the primary key of the *accountant* table. The *job* table, used to gather data about one accountant preparing one year's tax returns for one customer, uses the tax year and the customer number as its primary key. The *form* table that stores data about the forms that are part of a specific tax return uses the concatenation of the form's ID and the primary key of the project table for its primary key.

A foreign key is the same as the *complete* primary key of another table. Therefore, the *acct_first_name* attribute by itself in the *job* table is not a foreign key; neither is the *acct_last_name* attribute. If you concatenate them, however, then they are the same as the primary key of the *accountant* table and, in fact, this is the unit with which referential integrity should be enforced.

Assume that "Jane Johnson" is working on customer 10100's 2017 tax return. It is not enough to ensure that "Jane" appears somewhere in the first name column in the *accountant* table and "Johnson" appears anywhere in the last name column in the *accountant* table. There could be many people named "Jane" and many with the last name of "Johnson." What we need to ensure is that there is one person named "Jane Johnson" in the *accountant* table, the concatenation of the two attributes that make up the primary key.

The same holds true for the concatenated foreign key in the *form* table: the tax year and the customer number. A row with a matching pair must exist in the *job* table before referential integrity is satisfied.

Foreign Keys That Reference the Primary Key of Their Own Table

Foreign keys do not necessarily need to reference a primary key in a different table; they need only reference a primary key. As an example, consider the following *employee* relation:

```
employee (employee_ID, first_name, last_name, department,
     manager_ID)
```

A manager is also an employee. Therefore, the manager ID, although named differently from the employee ID, is actually a foreign key that references the primary key of its own table. The DBMS will, therefore, always ensure that whenever a user enters a manager ID, that manager already exists in the table as an employee.

Views

The people responsible for developing a database schema and those who write application programs for use by technologically unsophisticated users typically have knowledge of and access to the entire schema, including direct access to the database's base tables. However, it is usually undesirable to have end users working directly with base tables, primarily for security reasons.

The relational data model therefore includes a way to provide end users with their own window into the database, one that hides the details of the overall database design and prohibits direct access to the base tables: *views*.

The View Mechanism

A view is not stored with data. Instead, it is stored under a name in a database table, along with a database query that will retrieve its data. A view can therefore contain data from more than one table, selected rows, and selected columns.

Note: Although a view can be constructed in just about any way that you can query a relational database, many views can only be used for data display. As you will learn in Chapter 20, only views that meet a strict set of rules can be used to modify data.

The real beauty of storing views in this way, however, is that whenever the user includes the name of the view in a data manipulation language statement, the DBMS executes the query associated with the view name and recreates the view's table. This means that the data in a view will always be current.

A view table remains in main memory only for the duration of the data manipulation statement in which it was used. As soon as the user issues another query, the view table is removed from main memory to be replaced by the result of the most recent query. A view table is therefore a virtual table.

Note: Some end user DBMSs give the user the ability to save the contents of a view as a base table. This is a particularly undesirable feature, as there are no provisions for automatically updating the data in the saved table whenever the tables on which it was based change. The view table therefore quickly will become out of date and inaccurate.

Why Use Views?

There are three good reasons to include views in the design of a database:

- As mentioned earlier, views provide a significant security mechanism by restricting users from viewing portions of a schema to which they should not have access.
- Views can simplify the design of a database for technologically unsophisticated users.
- Because views are stored as named queries, they can be used to store frequently used, complex queries. The queries can then be executed by using the name of the view in a simple query statement.

Like other structural elements in a relational database, views can be created and destroyed at any time. However, because views do not contain stored data, but only specification of a query that will generate a virtual table, adding or removing view definitions has no impact on base tables or the data they contain. Removing a view will create problems when that view is used in an application program and the program is not modified to work with a different view or base table. A problem may also occur if a view used in the definition of another view is deleted.

The Data Dictionary

The structure of a relational database is stored in the database's *data dictionary*, or *catalog*. The data dictionary is made up of a set of relations, identical in properties to the relations used to hold data. They can be queried using the same tools used to query data-handling relations. No user can modify the data dictionary tables directly. However, data manipulation language commands that create, modify, and destroy database structural elements work by modifying rows in data dictionary tables.

You will typically find the following types of information in a data dictionary:

- Definitions of the columns that make up each table.
- Integrity constraints placed on relations.
- Security information (which user has the right to perform which operation on which table).
- Definitions of other database structure elements such as views and user-defined domains.

When a user attempts to access data in any way, a relational DBMS first goes to the data dictionary to determine whether the database elements the user has requested are actually part of the schema. In addition, the DBMS verifies that the user has the access right to whatever he or she is requesting.

When a user attempts to modify data, the DBMS also goes to the data dictionary to look for integrity constraints that may have been placed on the relation. If the data meet the constraints, then the modification is permitted. Otherwise, the DBMS returns an error message and does not make the change.

Because all access to a relational database is through the data dictionary, relational DBMSs are said to be *data dictionary driven*. The data in the data dictionary are known as *metadata*: data about data.

Sample Data Dictionary Tables

The precise tables that make up a data dictionary depend somewhat on the DBMS. In this section, you will see one example of a typical way in which a DBMS might organize its data dictionary.

The linchpin of the data dictionary is actually a table that documents all the data dictionary tables (often named *syscatalog*, the first few rows of which can be found in Figure 5.4). From the names of the data dictionary tables, you can probably guess that there are tables to store data about base tables, their columns, their indexes, and their foreign keys.

creator	tname	dbspace	tabletype	ncols	Primary_key
SYS	SYSTABLE	SYSTEM	TABLE	12	Y
SYS	SYSCOLUMN	SYSTEM	TABLE	14	Y
SYS	SYSINDEX	SYSTEM	TABLE	8	Y
SYS	SYSIXCOL	SYSTEM	TABLE	5	Y
SYS	SYSFOREIGNKEY	SYSTEM	TABLE	8	Y
SYS	SYSKCOL	SYSTEM	TABLE	4	Y
SYS	SYSFILE	SYSTEM	TABLE	3	Y
SYS	SYSDOMAIN	SYSTEM	TABLE	4	Y
SYS	SYSUSERPERM	SYSTEM	TABLE	10	Y
SYS	SYTSTABLEPERM	SYSTEM	TABLE	11	Y
SYS	SYSCOLPERM	SYSTEM	TABLE	6	Y

FIGURE 5.4 A portion of a *syscatalog* table.

The *syscolumn* table describes the columns in each table (including the data dictionary tables). In Figure 5.5, for example, you can see a portion of a *syscolumn* table that describes the *Antique Opticals* merchandise item table.

creator	cname	tname	coltype	nulls	length	Inprimarykey	Colno
DBA	item_numb	items	integer	N	4	Y	1
DBA	title	items	varchar	Y	60	N	2
DBA	distributor_numb	items	integer	Y	4	N	3
DBA	release_date	items	date	Y	6	N	4
DBA	retail_price	items	numeric	Y	8	N	5

FIGURE 5.5 Selected rows from a *syscolumn* table.

Keep in mind that these data dictionary tables have the same type of structure as all other tables in the database, and must adhere to the same rules as base tables. They must have non-null unique primary keys; they must also enforce referential integrity among themselves.

A Bit of History

When Codd published his paper describing the relational data model in 1970, software developers were bringing databases based on older data models to market. The software was becoming relatively mature and was being widely installed. Although many theorists recognized the benefits of the relational data model, it was some time before relational systems actually appeared.

IBM had a working prototype of its *System R* by 1976. This product, however, was never released. Instead, the first relational DBMS to feature SQL—an IBM development—was *Oracle*, released by the company of the same name in 1977. IBM didn't actually market a relational DBMS until 1981, when it released *SQL/DS*.

Oracle debuted on minicomputers running UNIX. *SQL/DS* ran under the VM operating environment (often specifically using CMS on top of VM) on IBM mainframes.[1] There was also a crop of early products that were designed specifically for PCs, the first of which was *dBase II*, from a company named Ashton-Tate. Released in 1981, the product ran on IBM PCs and Apple II + s.

Note: It is seriously questionable whether dBase was ever truly a relational DBMS. However, most consumers saw it as such and it is therefore considered the first relational product for PCs.

Oracle was joined by a large number of competing products in the UNIX market, including *Informix* and *Ingres*. *Oracle* has been the biggest winner in this group because it now runs on virtually every OS/hardware platform combination imaginable and has scaled well (down to PCs and up to mainframes). Prior to the widespread deployment of the open source DBMS *mySQL* as a database server for Web sites, it was safe to say that there were more copies of *Oracle* running on computers than any other DBMS.

The PC market for relational DBMSs has been flooded with products. As often happens with software, the best has not necessarily become the most successful. In 1983, Microrim released its *R:BASE* product, the first truly relational product for a PC. With its support for standard SQL, a powerful integrity rule facility, and a capable programming language, *R:BASE* was a robust product. It succumbed, however, to the market penetration of *dBASE*. The same can be said for *Paradox* (originally a Borland product and later purchased by Corel) and *FoxPro* (a *dBase*-compatible product originally developed by Fox Software).

dBase faded from prominence after being purchased by Borland in 1991. *FoxPro*, *dBase*'s major competitor, was purchased by Microsoft in 1992. It, too, has faded from the small computer DBMS market. Instead, the primary end user desktop DBMS for Windows today is *Access*, first released by Microsoft in 1993.

For Further Reading

If you want to follow the history of Codd's specifications for relational databases, consult the following:

Codd EF. A relational model of data for large shared databanks. *Commun. ACM*. 1970;13(6):377–387.
Codd EF. Extending the relational model to capture more meaning. *Trans. Database Sys*. 1979;4(4):397–434.

Codd EF. Extending the relational model to capture more meaning. *Trans. Database Sys.* 1979;4(4):397–434.

Codd EF. Relational database: a practical foundation for productivity. *Commun. ACM*. 1982;25(2):109–117.

Codd EF. *The Relational Data Model, Version 2*. Addison-Wesley; 1990:ISBN 978-0201141924.

There are also literally hundreds of books that discuss the details of specific relational DBMSs. After you finish reading this book, you may want to consult one or more books that deal with your specific product to help you learn to develop applications using that product's tools.

Other titles of interest:

Date CJ. *Relational Theory for Computer Professionals*. O'Reilly Media; 2013.

Lightstone SS, Teorey TJ, Nadeau T. *Physical Database Design: the database professional's guide to exploiting views, storage and more*. Morgan Kaufmann; 2007.

[1] VM (an acronym for "virtual machine") was an environment that ran on IBM mainframes. It was designed primarily to run other operating systems, such as CMS. Users rarely interacted with VM directly.