IN THIS CHAPTER

» Relating SQL to the relational model

» Figuring out functional dependencies

» Discovering keys, views, users, privileges, schemas, and catalogs

» Checking out connections, sessions, and transactions

» Understanding routines and paths

Chapter 1 SQL Foundations

This chapter offers a brief introduction to the (somewhat complicated) relationship between SQL and the relational database model. The chapter highlights how certain important terms and concepts may have slightly different meanings in the (practical) SQL world as opposed to the (theoretical) relational database world. The chapter also provides some general, all-inclusive definitions for good measure.

SQL and the Relational Model

SQL is a software tool designed to deal with relational database data. It does far more than just execute queries. Yes, of course, you can use it to retrieve the data you want from a database using a query. However, you can also use SQL to create and destroy databases, as well as modify their structure. In addition, you can add, modify, and delete data with SQL. Even with all that capability, SQL is still considered only a *data sublanguage,* which means that it does not have all the features of general-purpose programming languages such as C, C++, C#, or Java.

SQL is specifically designed for dealing with relational databases and thus does not include a number of features needed for creating useful application programs. As a result, to create a complete application — one that handles queries as well as provides access to a database — you must write the code in one of the general-purpose languages and embed SQL statements within the program whenever it communicates with the database.

The *relational database,* a type of data model, stores and provides access to data points that are related to one another, existed as a theoretical model for almost a decade before the first relational database product appeared on the market. Now, it turns out that the first commercial implementation of the relational model — a software program from the company that later became Oracle — did not even use SQL, which IBM had not yet released. In those early days, there were a number of competing data sublanguages. Gradually, SQL became a de facto standard, thanks in no small part to IBM's dominant position in the market, and the fact that Oracle started offering it as an alternative to its own language early on.

Although SQL was developed to work with a relational database management system, it's not entirely consistent with the relational model. However, it is close enough, and in many cases, it even offers capabilities not present in the relational model. Some of the most important aspects of SQL are direct analogs of some aspects of the relational model. Others are not.

Sets, Relations, Multisets, and Tables

The relational model is based on the mathematical discipline known as *set theory*. In set theory, a *set* is defined as a collection of unique objects — duplicates are not allowed. This carries over to the relational model. A *relation* is defined as a collection of unique objects called *tuples* — no duplicates are allowed among tuples.

In SQL, the equivalent of a relation is a table. However, tables are not exactly like relations in that a table can have duplicate rows. For that reason, tables in a relational database are not modeled on the sets of set theory but rather on *multisets*, which are similar to sets, except they allow duplicate objects.

Although a relation is not exactly the same thing as a table, the terms are often used interchangeably. Because relations were defined by theoreticians, they have a very precise definition. The word *table*, on the other hand, is in general use and is often much more loosely defined. This book uses the word *table*, in a more restricted sense, as being an alternate term for *relation*. The attributes and tuples of a relation are strictly equivalent to the columns and rows of a table.

So, what's an SQL relation? Formally, a relation is a two-dimensional table that has the following characteristics:

- Every cell in the table must contain a single value if it contains any value at all. Repeating groups and arrays are not allowed as values. (In this context, *groups* and *arrays* are examples of collections of values.)
- All the entries in any column must be the same. For example, if a column contains an employee name in one row, it must contain employee names in all rows that contain values.
- Each column has a unique name.
- The order of the columns doesn't matter.
- The order of the rows doesn't matter.
- No two rows may be identical.

If and only if a table meets all these criteria, it is a relation. You might have tables that fail to meet one or more of these criteria. For example, a table might have two identical rows. It is still a table in the loose sense, but it is not a relation.

Functional Dependencies

Functional dependencies are relationships between or among attributes. Consider the example of two attributes of the CUSTOMER relation, Zipcode and State. If you know the customer's zip code, the state can be obtained by a simple lookup because each zip code resides in one and only one state. This means that State is *functionally dependent* on Zipcode or that Zipcode *determines* state. Zipcode is called a *determinant* because it determines the value of another attribute. The reverse is not true. State does not determine Zipcode because states can contain multiple Zipcodes. You denote functional dependencies as follows:

```
Zipcode ➪ State
```

A group of attributes may act as a determinant. If one attribute depends on the values of multiple other attributes, that group of attributes, collectively, is a determinant of the first attribute.

Consider the relation INVOICE, made up as it is of the following attributes:

- **InvNo:** Invoice number.
- **CustID:** Customer ID.
- **WorR:** Wholesale or retail. I'm assuming that products have both a wholesale and a retail price, which is why I've added the WorR attribute to tell me whether this is a wholesale or a retail transaction.

- **ProdID:** Product ID.
- **Quantity:** Quantity.
- **Price:** You guessed it.
- **Extprice:** Extended price (which I get by multiplying Quantity and Price.)

With our definitions out of the way, check out what depends on what by following the handy determinant arrow:

```
(WorR, ProdID) ➪ Price
(Quantity, Price) ➪ Extprice,
```

W/R tells you whether you are charging the wholesale or the retail price. ProdID shows which product you are considering. Thus, the combination of WorR and ProdID determines Price. Similarly, the combination of Quantity and Price determines Extprice. Neither WorR nor ProdID by itself determines Price; they are both needed to determine Price. Both Quantity and Price are needed to determine Extprice.

Keys

A *key* is an attribute (or group of attributes) that uniquely identifies a tuple (a unique collection of attributes) in a relation. One of the characteristics of a relation is that no two rows (tuples) are identical. You can guarantee that no two rows are identical if at least one field (attribute) is guaranteed to have a unique value in every row, or if some combination of fields is guaranteed to be unique for each row.

Table 1-1 shows an example of the PROJECT relation. It lists researchers affiliated with the Gentoo Institute's Penguin Physiology Lab, the project that each participant is working on, and the location at which each participant is conducting his or her research.

In this table, each researcher is assigned to only one project. Is this a rule? Must a researcher be assigned to only one project, can a researcher be assigned to more than one? If a researcher can be assigned to only one project, ResearcherID is a key. It guarantees that every row in the PROJECT table is unique. What if there is no such rule? What if a researcher may work on multiple projects at the same time? Table 1-2 shows this situation.

TABLE 1-1 **PROJECT Relation**

ResearcherID	Project	Location
Pizarro	Why penguin feet don't freeze	Bahia Paraiso
Whitehead	Why penguins don't get the bends	Port Lockroy
Shelton	How penguin eggs stay warm in pebble nests	Peterman Island
Nansen	How penguin diet varies by season	Peterman Island

TABLE 1-2 **PROJECTS Relation**

ResearcherID	Project	Location
Pizarro	Why penguin feet don't freeze	Bahia Paraiso
Pizarro	How penguin eggs stay warm in pebble nests	Peterman Island
Whitehead	Why penguins don't get the bends	Port Lockroy
Shelton	How penguin eggs stay warm in pebble nests	Peterman Island
Shelton	How penguin diet varies by season	Peterman Island
Nansen	How penguin diet varies by season	Peterman Island

In this scenario, Dr. Pizarro works on the cold feet and the warm eggs projects, whereas Professor Shelton works on the warm eggs and the varied diet projects. Clearly, ResearcherID cannot be used as a key. However, the combination of ResearcherID and Project is unique and is thus a key.

You're probably wondering how you can reliably tell what is a key and what isn't. Looking at the relation in Table 1-1, it looks like ResearcherID is a key because every entry in that column is unique. However, this could be due to the fact that you are looking at a limited sample, and any minute now, someone could add a new row that duplicates the value of ResearcherID in one of the existing rows. How can you be sure that won't happen? Easy. Ask the users.

The relations you build are models of the mental images that the users have of the system they are dealing with. You want your relational model to correspond as closely as possible to the model the users have in their minds. If they tell you, for example, that in their organization, researchers never work on more than one project at a time, you can use ResearcherID as a key. On the other hand, if it is even remotely possible that a researcher might be assigned to two projects simultaneously, you have to revert to a composite key made up of both ResearcherID and Project.

REMEMBER

A question that might arise in your mind is, "Is it possible for a relation to exist that has no key?" By the definition of a relation, the answer is no. Every relation *must* have a key. One of the characteristics of a relation is that no two rows may be exactly the same. That means that you are always able to distinguish rows from each other, although you may have to include all the relation's attributes in the key to do it.

Views

Although the most fundamental constituent of a relational database is undoubtedly the table, another important concept is the virtual table or *view*. Unlike an ordinary table, a view has no physical existence until it is called upon in a query. There is no place on the disk where the rows in the view are stored. The view exists only in the metadata as a definition. The definition describes how to pull data from tables and present it to the user in the form of a view.

From the user's perspective, a view looks just like a table. You can do almost everything to a view that you can do to a table. The major exception is that you cannot always update a view the same way that you can update a table. The view may contain columns that are the result of some arithmetic operation on the data in columns from the tables upon which the view is based. You can't update a column that doesn't exist in your permanent storage device. Despite this limitation, views, after they're formulated, can save you considerable work: You don't need to code the same complex query every time you want to pull data from multiple tables. Create the view once, and then use it every time you need it.

Users

Although it may seem a little odd to include them, the users are an important part of any database system. After all, without the users, no data would be written into the system, no data would be manipulated, and no results would be displayed. When you think about it, the users are mighty important. Just as you want your hardware and software to be of the highest quality you can afford in order to produce the best results, and for the same reason, you want the highest-quality people. To ensure that only the people who meet your standards have access to the database system, you should have a robust security system that enables authorized users to do their job, and at the same time, prevents access to everyone else.

Privileges

A good security system not only keeps out unauthorized users but also provides authorized users with access privileges tailored to their needs. The night watchman has different database needs from those of the company CEO. One way of handling privileges is to assign every authorized user an authorization ID. When the person logs on with his authorization ID, the privileges associated with that authorization ID become available to him. This could include the ability to read the contents of certain columns of certain tables, the ability to add new rows to certain tables, delete rows, update rows, and so on.

A second way to assign privileges is with roles, which were introduced in SQL:1999. *Roles* are simply a way for you to assign the same privileges to multiple people, and they are particularly valuable in large organizations where a number of people have essentially the same job and, thus, the same needs for data.

For example, a security guard working the nightshift might have the same data needs as other security guards. You can grant a suite of privileges to the SECURITY_GUARD role. From then on, you can assign the SECURITY_GUARD role to any new guards, and all the privileges appropriate for that role are automatically assigned to them. When a person leaves, or changes jobs, revoking their role can be just as easy.

Schemas

Relational database applications typically use multiple tables. As a database grows to support multiple applications, it becomes more and more likely that an application developer will try to give one of her tables the same name as a table already in the database. This can cause problems and frustration. To get around this problem, SQL has a hierarchical namespace structure. A developer can define her tables as being members of a *schema*.

With this structure, one developer can have a table named CUSTOMER in her schema, whereas a second developer can also have an entirely different table, also named CUSTOMER, but in a different schema.

THE RELATIONAL DATABASE HIERARCHY

A relational database is organized in a hierarchical structure, where the highest level is the catalog. Generally, only the largest, most complex databases have multiple catalogs.

- **Catalogs:** A database catalog comes into play only in large, complex databases that have multiple schemas.
- **Schemas:** A database schema contains metadata. This metadata includes definitions of tables, views, value ranges, indexes, users, and user groups. It can also include stored procedures and triggers.
- **Tables:** A database table is a set of elements organized as a two-dimensional table with horizontal rows and vertical columns. The columns correspond to the attributes in an entity's entity relationship (ER) model. The rows hold the data about individual instances of the entity.
- **Columns:** A column is a component of a database table. Each column in the table corresponds to one of the attributes in the ER model of the entity being actualized by the table.

Catalogs

These days, organizations can be so big that if every developer had a schema for each of her applications, the number of schemas itself could be a problem. Someone might inadvertently give a new schema the same name as an existing schema. An additional level was added at the top of the namespace hierarchy to head off this possibility. A *catalog* can contain multiple schemas, which in turn can contain multiple tables. The smallest organizations don't have to worry about either catalogs or schemas, but those levels of the namespace hierarchy are there if they're needed. If your organization is big enough to worry about duplicate catalog names, it is big enough to figure out a way to deal with the problem.

Connections, Sessions, and Transactions

A database management system is typically divided into two main parts: a *client* side, which interfaces with the user, and a *server* side, which holds the data and operates on it. To operate on a database, a user must establish a *connection* between their client and the server that holds the data they want to access. Generally, the first thing you must do — if you want to work on a database at all — is to establish a connection to it. You can do this with a `CONNECT` statement that specifies your authorization ID and names the server you want to connect to. The exact implementation of this varies from one DBMS to another. (Most people today would use the DBMS's graphical user interface to connect to a server instead of using the SQL `CONNECT` statement.)

REMEMBER

A *session* is the context in which a single user executes a sequence of SQL statements, using a single connection. A *user* can either be a person entering SQL statements at the client console, or a program running on the client machine.

A *transaction* is a sequence of SQL statements that is atomic with respect to recovery. This means that if a failure occurs while a transaction is in progress, the effects of the transaction are erased so that the database is left in the state it was in before the transaction started. *Atomic* in this context means indivisible. Either the transaction runs to completion, or it aborts in such a way that any changes it made before the abort are undone.

Routines

Routines are procedures, functions, or methods that can be invoked either by an SQL `CALL` statement or by the host language program that the SQL code is operating with. *Methods* are a kind of function used in object-oriented programming.

Routines enable SQL code to take advantage of calculations performed by host language code and enable host language code to take advantage of data operations performed by SQL code.

Because either a host language program or SQL code can invoke a routine, and because the routine being invoked can be written either in SQL or in host language code, routines can cause confusion. A few definitions help to clarify the situation:

- **Externally invoked routine:** A procedure, written in SQL and residing in a module located on the client, which is invoked by the host language program
- **SQL-invoked routine:** Either a procedure or a function residing in a module located on the server, which could be written in either SQL or the host language that is invoked by SQL code
- **External routine:** Either a procedure or a function residing in a module located on the server, which is written in the host language, but is invoked by SQL
- **SQL routine:** Either a procedure or a function residing in a module located on either the server or the client, which is written in SQL and invoked by SQL

Paths

A *path* in SQL, similar to a path in operating systems, tells the system in what order to search locations to find a routine that has been invoked. For a system with several schemas (perhaps one for testing, one for QA, and one for production), the path tells the executing program where to look first, where to look next, and so on, to find an invoked routine.