CHAPTER 11

Using SQL to Implement a Relational Design

Abstract

This chapter covers the elements of the SQL database manipulation language that create, modify, and delete database structural elements: schemas, domains, and tables. There is emphasis on data integrity using column constraints (domain checking, NOT NULL, primary keys, and so on). The chapter also discusses data integrity constraints between tables (referential integrity for foreign keys).

Keywords

SQL
CREATE TABLE
creating relations
CREATE SCHEMA
database constraints
primary keys
foreign keys
referential integrity
entity integrity
modifying relations
database default values

As a complete data manipulation language, SQL contains statements that allow you to insert, modify, delete, and retrieve data. However, to a database designer, the portions of SQL that support the creation of database structural elements are of utmost importance. In this chapter, you will be introduced to the SQL commands that you will use to create and maintain the tables that make up a relational database.

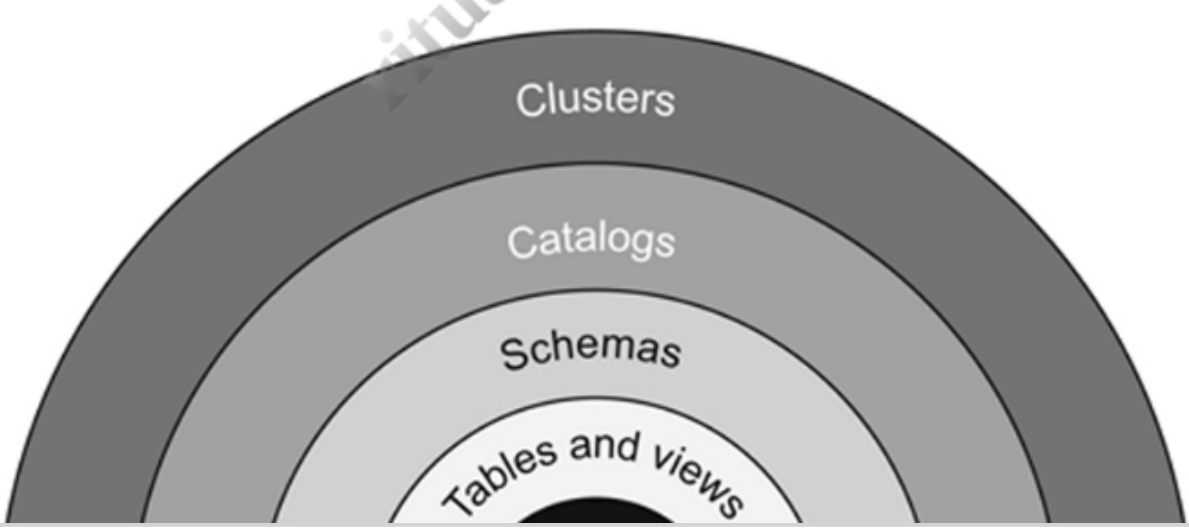
Some of the structural elements in a relational database are created with SQL retrieval commands embedded in them. We will therefore defer a discussion of those elements until Chapter 21.

The actual file structure of a database is implementation dependent, as is the procedure needed to create database files. Therefore, the discussion in this chapter assumes that the necessary database files are already in place.

Note: You will see extensive examples of the use of the syntax presented in this chapter at the end of each of the three case studies that follow in this book.

Database Structure Hierarchy

The elements that describe the structure of a SQL:2011-compliant database are arranged in a hierarchy, diagrammed in Figure 11.1. The smallest units with which the DBMS works—columns and rows—appear in the center. These, in turn, are grouped into tables and views.



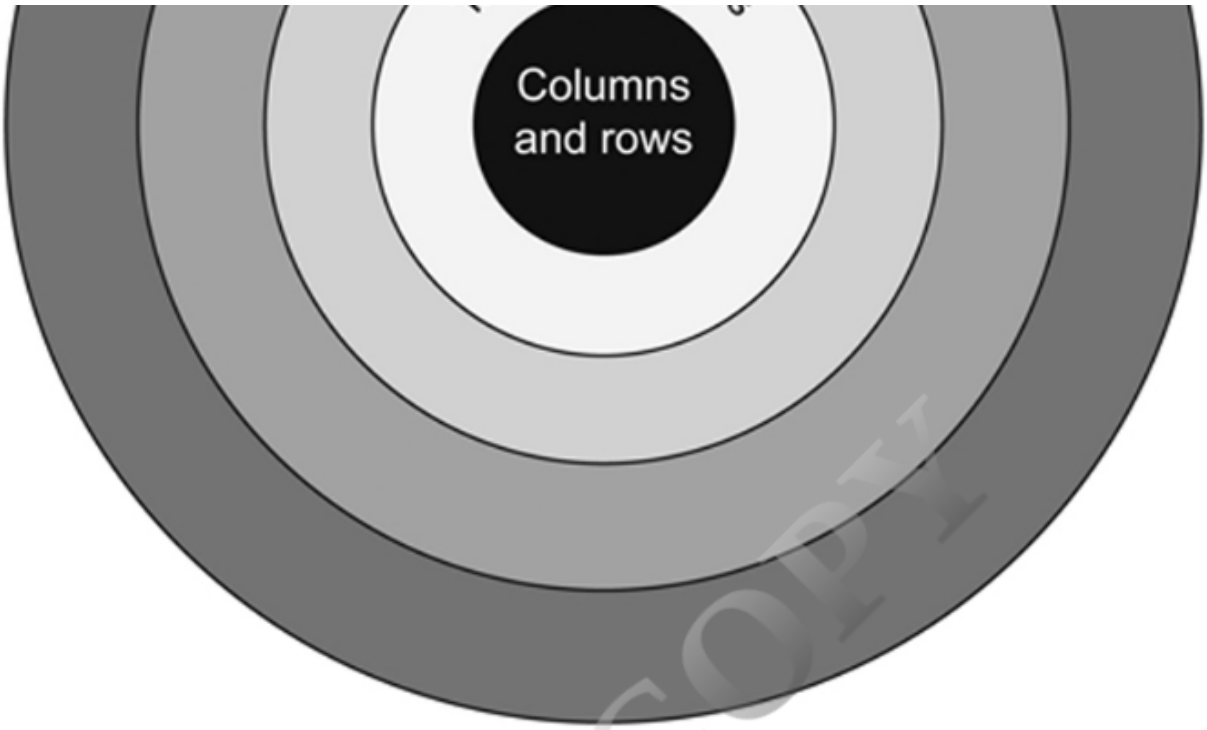


FIGURE 11.1 **The SQL:2011 database structure hierarchy.**

The tables and views that comprise a single logical database are collected into a schema. Multiple schemas are grouped into catalogs, which can then be grouped into clusters. A catalog usually contains information describing all the schemas handled by one DBMS. Catalog creation is implementation dependent and, therefore, not part of the SQL standard.

Prior to SQL-92, clusters often represented database files, and the clustering of database elements into files was a way to increase database performance by placing data accessed together in the same physical file. The SQL-92 and beyond concept of a cluster, however, is a group of catalogs that are accessible using the same connection to a database server.

Note: Don't forget that SQL is a dynamic language. The standard has been updated in 1999, 2003, 2006, and 2011.

In current versions of SQL, none of the groupings of database elements are related to physical storage structures. If you are working with a centralized mainframe DBMS, you may find multiple catalogs stored in the same database file. However, on smaller or distributed systems, you are likely to find one catalog or schema per database file or to find a catalog or schema split between multiple files.

Clusters, catalogs, and schemas are not required elements of a database environment. In a small installation where there is one collection of tables serving a single purpose, for example, it may not even be necessary to create a schema to hold them.

Naming and Identifying Structural Elements

The way in which you name and identify database structural elements is, in some measure, dictated by the structure hierarchy:

- Column names must be unique within the table.
- Table names must be unique within the schema.
- Schema names must be unique within their catalog.
- Catalog names must be unique within their cluster.

When a column name appears in more than one table in a SQL statement—as it often does, when there are primary key–foreign key references—the user must specify the table from which a column should be taken (even if it makes no difference which table is used). The general form for qualifying duplicate column names is:

```
table_name.column_name
```

If an installation has more than one schema, then a user must also indicate the schema in which a table resides:

```
schema_name.table_name.column_name
```

This naming convention means that two different schemas can include tables with the same name.

By the same token, if an installation has multiple catalogs, a user will need to indicate the catalog from which a database element comes:

```
catalog_name.schema_name.table_name.column_name
```

The names that you assign to database elements can include the following:

- Letters
- Numbers
- Underscores (_)

SQL names can be up to 128 characters long. According to the SQL standard, they should not be case sensitive. In fact, many SQL command processors convert names to all upper- or lowercase before submitting a SQL statement to a DBMS for processing. However, there are always exceptions, so you should check your DBMS's naming rules before creating any structural elements.

Note: Some DBMSs also allow pound signs (#) and dollar signs ($) in element names, but neither is recognized by SQL statements, so their use should be avoided.

Schemas

To a database designer, a schema represents the overall, logical design of a complete database. As far as SQL is concerned, however, a schema is nothing more than a container for tables, views, and other structural elements. It is up to the database designer to place a meaningful group of elements within each schema.

A schema is not required to create tables and views. In fact, if you are installing a database for an environment in which there is likely to be only one logical database, then you can just as easily do without one. However, if more than one database will be sharing the same DBMS and the same server, then organizing database elements into schemas can greatly simplify the maintenance of the individual databases.

Creating a Schema

To create a schema, you use the CREATE SCHEMA statement. In its simplest form, it has the syntax

```
CREATE SCHEMA schema_name
```

as in

```
CREATE SCHEMA antique_opticals;
```

Note: The majority of SQL command processors require a semicolon at the end of each statement. However, that end-of-statement marker is not a part of the SQL standard and you may encounter DBMSs that do not use it. We will use it in this book at the end of statements that could be submitted to a DBMS. It does not appear at the end of command "templates" that show the elements of a command.

By default, a schema belongs to the user who created it (the user ID under which the schema was created). The owner of the schema is the only user ID that can modify the schema, unless the owner grants that ability to other users.

To assign a different owner to a schema, you add an AUTHORIZATION clause:

```
CREATE SCHEMA schema_name AUTHORIZATION owner_user_ID
```

For example, to assign the *Antique Opticals* schema to the user ID DBA, someone could use:

```
CREATE SCHEMA antique_opticals AUTHORIZATION dba;
```

When creating a schema, you can also create additional elements at the same time. To do so, you use braces to group the CREATE statements for the other elements, as in:

```
CREATE SCHEMA schema_name AUTHORIZATION owner_user_id
{
// other CREATE statements go here
}
```

This automatically assigns the elements within the braces to the schema.

Identifying the Schema You Want to Use

One of the nicest things about a relational database is that you can add or delete database structural elements at any time. There must therefore be a way to specify a current schema for new database elements after the schema has been created initially with the CREATE SCHEMA statement.

One way to do this is with the SET SCHEMA statement:

```
SET SCHEMA schema_name
```

To use SET SCHEMA, the user ID under which you are working must have authorization to work with that schema.

Alternatively, you can qualify the name of a database element with the name of the schema. For example, if you are creating a table, then you would use something like

```
CREATE TABLE schema_name.table_name
```

For those DBMSs that do not support SET SCHEMA, this is the only way to attach new database elements to a schema after the schema has been created.

Domains

As you know, a domain is an expression of the permitted values for a column in a relation. When you define a table, you assign each column a data type (example, character, or integer) that provides a broad domain. A DBMS will not store data that violate that constraint.

The SQL-92 standard introduced the concept of user-defined domains, which can be viewed as user-defined data types that can be applied to columns in tables. (This means you have to create a domain before you can assign it to a column!)

Domains can be created as part of a CREATE SCHEMA statement, which has the following syntax:

```
CREATE DOMAIN domain_name data_type
CHECK (expression_to_validate_values)
```

The CHECK clause is actually a generic way of expressing a condition that data must meet. It can include a retrieval query to validate data against other data stored in the database, or it can include a simple logical expression. In that expression, the keyword VALUE represents the data being checked.

For example, if *Antique Opticals* wanted to validate the price of a disc, someone might create the following domain:

```
CREATE DOMAIN price numeric (6,2)
CHECK (VALUE >= 19.95);
```

After creating this domain, a column in a table can be given the data type of price. The DBMS will then check to be certain that the value in that column is always greater than, or equal to, 19.95. (We will leave a discussion of the data type used in the preceding SQL statement until we cover creating tables in the next section of this chapter.)

The domain mechanism is very flexible. Assume, for example, that you want to ensure that telephone numbers are always stored in the format XXX-XXX-XXXX. A domain to validate that format might be created as:[1]

```
CREATE DOMAIN telephone char (12)
CHECK (SUBSTRING (VALUE FROM 4 FOR 1 = '-') AND
     SUBSTRING (VALUE FROM 8 FOR 1 = '-'));[1]
```

When a user attempts to store a value in a column to which the *telephone* domain has been assigned, the DBMS performs two SUBSTRING functions. The first looks at the fourth character in the string to determine whether it is a -. The second examines the character at position 8. The data will be accepted only if both conditions are met.

You can use the CREATE DOMAIN statement to give a column a default value. For example, the following statement sets up a domain that holds either Y or N, and defaults to Y:[2]

```
CREATE DOMAIN boolean char (1)
DEFAULT = 'Y'
CHECK (UPPER(VALUE) = 'Y' OR UPPER(VALUE) = 'N');[2]
```

Tables

The most important structure within a relational database is the table. As you know, tables contain just about everything, including business data and the data dictionary.

SQL divides tables into three categories:

- *Permanent base tables*: Permanent base tables are tables whose contents are stored in the database, and remain permanently in the database, unless they are explicitly deleted.
- *Global temporary tables*: Global temporary tables are tables for working storage that are destroyed at the end of a SQL session. The definitions of the tables are stored in the data dictionary, but their data are not. The tables must be loaded with data each time they are going to be used. Global temporary tables can be used only by the current user, but they are visible to an entire SQL session (either an application program or a user working with an interactive query facility).
- *Local temporary tables*: Local temporary tables are similar to global temporary tables. However, they are visible only to the specific program module in which they are created.

Temporary base tables are subtly different from views, which assemble their data by executing a SQL query. You will read more about this difference and how temporary tables are created and used later in this chapter.

Most of the tables in a relational database are permanent base tables. You create them with the CREATE TABLE statement:

```
CREATE TABLE table_name
(   column1_name column1_data_type
    column1_constraints,
    column2_name column2_data_type
    column2_constraints, …
    table_constraints)
```

The constraints on a table include declarations of primary and foreign keys. The constraints on a column include whether values in the column are mandatory, as well as other constraints you may decide to include in a CHECK clause.

Column Data Types

Each column in a table must be given a data type (or a user-defined domain). Although data types are somewhat implementation dependent, most DBMSs that support SQL include the following predefined data types:

- INTEGER (abbreviated INT): a positive or negative whole number. The number of bits occupied by the value is implementation dependent. In most cases, integers are either 32 or 64 bits.
- SMALLINT: a positive or negative whole number. A small integer is usually half the size of a standard integer. Using small integers when you know you will need to store only small values can save space in the database.
- NUMERIC: a fixed-point positive or negative number. A numeric value has a whole number portion and a fractional portion. When you create it, you must specify the total length of the number (including the decimal point), and how many of those digits will be to the right of the decimal point (its *precision*). For example

```
NUMERIC (6, 2)
```

```
NUMERIC (6,2)
```

creates a number in the format XXX.XX. The DBMS will store exactly two digits to the right of the decimal point.

- DECIMAL: a fixed-point positive or negative number. A decimal number is similar to a numeric value. However, the DBMS may store more digits to the right of the decimal point than you specify. Although there is no guarantee that you will get the extra precision, its use can provide more accurate results in computations.
- REAL: a "single-precision" floating point value. A floating point number is expressed in the format:

```
XX,XXXXX * 10YY
```

where YY is the power to which 10 is raised. Because of the way in which computers store floating point numbers, a real number may not be an exact representation of a value, but only a close approximation. The range of values that can be stored is implementation dependent, as is the precision. You therefore cannot specify a size for a real number column.

- DOUBLE PRECISION (abbreviated DOUBLE): a "double-precision" floating point number. The range and precision of double precision values are implementation dependent, but generally both will be greater than single-precision real numbers.
- FLOAT: a floating point number for which you can specify the precision. The DBMS will maintain at least the precision that you specify. (It may be more.)
- BOOLEAN: a true/false value. The BOOLEAN data type was introduced in the SQL:1999 standard, but is not supported by all DBMSs. If your DBMS does not include BOOLEAN, the best alternative is usually a one-character text column with a CHECK clause restricting values to Y and N.
- BIT: storage for a fixed number of individual bits. You must indicate the number of bits, as in

```
BIT (n)
```

where *n* is the number of bits. (If you do not, you will have room for only one bit.)

- BIT VARYING: storage for a varying number of bits up to a specified maximum, as in

```
BIT VARYING (n)
```

where *n* is the maximum number of bits. In some DBMSs, columns of this type can be used to store graphic images.

- DATE: a date.
- TIME: a time.
- TIMESTAMP: the combination of a date and a time.
- CHARACTER (abbreviated CHAR): a fixed-length space to hold a string of characters. When declaring a CHAR column, you need to indicate the width of the column:

```
CHAR (n)
```

where *n* is the amount of space that will be allocated for the column in every row. Even if you store less than *n* characters, the column will always take up *n* bytes—or n × 2 bytes, if you are storing UNICODE characters—and the column will be padded with blanks to fill up empty space. The maximum number of characters allowed is implementation dependent (usually 256 or more).

- CHARACTER VARYING (abbreviated VARCHAR): a variable length space to hold a string of characters. You must indicate the maximum width of the column—

```
VARCHAR (n)
```

VARCHAR (n)

—but the DBMS stores only as many characters as you insert, up to the maximum *n*. The overall maximum number of characters allowed is implementation dependent (usually 256 or more).

- INTERVAL: a date or time interval. An interval data type is followed by a qualifier that specifies the size of the interval and, optionally, the number of digits. For example:

```
INTERVAL YEAR
INTERVAL YEAR (n)
INTERVAL MONTH
INTERVAL MONTH (n)
INTERVAL YEAR TO MONTH
INTERVAL YEAR (n) TO MONTH
INTERVAL DAY
INTERVAL DAY (n)
INTERVAL DAY TO HOUR
INTERVAL DAY (n) TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY (n) TO MINUTE
INTERVAL MINUTE
INTERVAL MINUTE (n)
```

In the preceding examples, *n* specifies the number of digits. When the interval covers more than one date and/or time unit, such as YEAR TO MONTH, you can specify a size for only the first unit. Year–month intervals can include days, hours, minutes, and/or seconds.

- BLOB (Binary Large Object): a block of binary code (often a graphic) stored as a unit and retrievable only as a unit. In many cases, the DBMS cannot interpret the contents of a BLOB (although the application that created the BLOB can do so). Because BLOB data are stored as undifferentiated binary, BLOB columns cannot be searched directly. Identifying information about the contents of a BLOB must be contained in other columns of the table, using data types that can be searched.

In Figure 11.2 you will find bare-bones CREATE TABLE statements for the *Antique Opticals* database. These statements include only column names and data types. SQL will create tables from statements in this format, but because the tables have no primary keys, many DBMSs will not allow you to enter data.

```
CREATE TABLE customer
```

```
CREATE TABLE customer
      (customer_numb int,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup_or_ship char (1));

CREATE TABLE order_line
      (order_numb int,
      item_numb int,
      quantity int,
```

```
      discount_percent int,
      selling_price numeric (6,2),
      line_cost numeric (7,2),
      shipped boolean,
      shipping_date date);
```

```
CREATE TABLE purchase
      (purchase_date date,
      customer_numb int,
      items_received boolean),
      customer_paid boolean);

CREATE TABLE purchase_item
      (purchase_date date,
      customer_numb int,
      item_numb int,
      condition char (15),
      price_paid numeric (6,2));

CREATE TABLE actor
      (actor_numb int,
      actor_name varchar (60));

CREATE TABLE performance
      (actor_numb int,
      item_numb int,
      role varchar (60));

CREATE TABLE producer
      (producer_name varchar (60),
      studio varchar (40));

CREATE TABLE production
      (producer_name varchar (60),
      item_numb int);
```


FIGURE 11.2 **Initial CREATE TABLE statements for the *Antique Opticals* database.**

Default Values

As you are defining columns, you can designate a default value for individual columns. To indicate a default value, you add a DEFAULT keyword to the column definition, followed by the default value. For example, in the *orders* relation the order date column defaults to the current system date. The column declaration is therefore written:

```
order_date date DEFAULT CURRENT_DATE;
```

```
order_date date DEFAULT CURRENT_DATE;
```

Notice that this particular declaration is using the SQL value CURRENT_DATE. However, you can place any value after DEFAULT that is a valid instance of the column's data type.

NOT NULL Constraints

The values in primary key columns must be unique and not null. In addition, there may be other columns for which you want to require a value. You can specify such columns by adding NOT NULL after the column declaration. Since *Antique Opticals* wants to ensure that an order date is always entered, the complete declaration for that column in the *orders* table is

```
order_date date NOT NULL DEFAULT CURRENT_DATE;
```

Primary Keys

There are two ways to specify a primary key:

- Add a PRIMARY KEY clause to a CREATE TABLE statement. The keywords PRIMARY KEY are followed by the names of the primary key column or columns, surrounded by parentheses.

- Add the keywords PRIMARY KEY to the declaration of each column that is part of the primary key. Use a CONSTRAINT clause if you want to name the primary key constraint.

In Figure 11.3, you will find the CREATE TABLE statement for the *Antique Opticals* database, including both PRIMARY KEY and CONSTRAINT clauses. Notice that in those tables that have concatenated primary keys, all the primary key columns have been included in a PRIMARY KEY clause.

```
CREATE TABLE customer
      (customer_numb int PRIMARY KEY,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int PRIMARY KEY,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int CONSTRAINT item_pk PRIMARY KEY,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup or ship char (1)
```

```
      pickup_or_ship char (1)
      PRIMARY KEY (order_numb));

CREATE TABLE order_line
      (order_numb int,
      item_numb int,
      quantity int,
      discount_percent int,
      selling_price numeric (6,2),
      line_cost numeric (7,2),
      shipped boolean,
      shipping_date date
      PRIMARY KEY (order_numb, item_numb));

CREATE TABLE purchase
      (purchase_date date,
      customer_numb int,
      items_received boolean,
      customer_paid boolean
      PRIMARY KEY (purchase_date, customer_numb));

CREATE TABLE purchase_item
      (purchase_date date,
      customer_numb int,
      item_numb int,
      condition char (15),
      price_paid numeric (6,2)
      PRIMARY KEY (purchase_date, customer_numb, item_numb));

CREATE TABLE actor
      (actor_numb int PRIMARY KEY,
      actor_name varchar (60));

CREATE TABLE performance
      (actor_numb int,
      item_numb int,
      role varchar (60)
      PRIMARY KEY (actor_numb, item_numb));

CREATE TABLE producer
      (producer_name varchar (60) CONSTRAINT producer_pk PRIMARY KEY,
      studio varchar (40));

CREATE TABLE production
      (producer_name varchar (60),
      item_numb int
      PRIMARY KEY (producer_name, item_numb));
```

FIGURE 11.3 **CREATE TABLE statements for the *Antique Opticals* database including primary key declarations.**

Foreign Keys

As you know, a foreign key is a column (or combination of columns) that is exactly the same as the primary of some table. When a foreign key value matches a primary key value, we know that there is a logical relationship between the database objects represented by the matching rows.

One of the major constraints on a relation is referential integrity, which states that every nonnull foreign key must reference an existing primary key value. To maintain the integrity of the database, it is vital that foreign key constraints be stored within the database's data dictionary so that the DBMS can be responsible for enforcing those constraints.

To specify a foreign key for a table, you add a FOREIGN KEY clause:

```
FOREIGN KEY foreign_key_name (foreign_key_columns)
REFERENCES primary_key_table (primary_key_columns)
    ON UPDATE update_option
    ON DELETE delete_option
```

Each foreign key–primary key reference is given a name. This makes it possible to identify the reference at a later time, in particular, so you can remove the reference if necessary.

Note: Some DBMSs, such as Oracle, do not support the naming of foreign keys, in which case you would use preceding syntax without the name.

The names of the foreign key columns follow the name of the foreign key. The REFERENCES clause contains the name of the primary key table being referenced. If the primary key columns are named in the PRIMARY KEY clause of their table, then you don't need to list the primary key columns. However, if the columns aren't part of a PRIMARY KEY clause, you must list the primary key columns in the REFERENCES clause.

The final part of the FOREIGN KEY specification indicates what should happen when a primary key value being referenced by a foreign key value is updated or deleted. There are three options that apply to both updates and deletions and one additional option for each:

- SET NULL: Replace the foreign key value with null. This isn't possible when the foreign key is part of the primary key of its table.
- SET DEFAULT: Replace the foreign key value with the column's default value.
- CASCADE: Delete or update all foreign key rows.
- NO ACTION: On update, make no modifications of foreign key values.
- RESTICT: Do not allow deletions of primary key rows.

The complete declarations for the *Antique Opticals* database tables, which include foreign key constraints, can be found in Figure 11.4. Notice that, although there are no restrictions on how to name foreign keys, the foreign keys in this database have been named to indicate the tables involved. This makes them easier to identify, if you need to delete or modify a foreign key at a later date.

```
CREATE TABLE customer
      (customer_numb int PRIMARY KEY,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int PRIMARY KEY,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
```

```
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int  CONSTRAINT item_pk PRIMARY KEY,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup_or_ship char (1)
      PRIMARY KEY (order_numb)
      FOREIGN KEY order2customer (customer_numb)
      REFERENCES customer
            ON UPDATE CASCADE
            ON DELETE RESTRICT);
```

```
CREATE TABLE order_line
     (order_numb int,
     item_numb int,
     quantity int,
     discount_percent int,
     selling_price numeric (6,2),
     line_cost numeric (7,2),
     shipped boolean,
     shipping_date date
     PRIMARY KEY (order_numb, item_numb)
     FOREIGN KEY order_line2item (item_numb)
     REFERENCES item
           ON UPDATE CASCADE
           ON DELETE RESTRICT
     FOREIGN KEY order_line2order (order_numb)
     REFERENCES order
           ON UPDATE CASCADE
           ON DELETE CASCADE);

CREATE TABLE purchase
```

```
CREATE TABLE purchase
      (purchase_date date,
      customer_numb int,
      items_received boolean,
      customer_paid boolean
      PRIMARY KEY (purchase_date, customer_numb)
      FOREIGN KEY purchase2customer (customer_numb)
      REFERENCES customer
            ON UPDATE CASCADE)
            ON DELETE RESTRICT);

CREATE TABLE purchase_item
      (purchase_date date,
      customer_numb int,
      item_numb int,
      condition char (15),
      price_paid numeric (6,2)
      PRIMARY KEY (purchase_date, customer_numb, item_numb)
      FOREIGN KEY purchase_item2purchase (purchase_date, customer_numb
            ON UPDATE CASCADE
            ON DELETE CASCADE
      FOREIGN KEY purchase_item2item (item_numb)
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE RESTRICT);

CREATE TABLE actor
      (actor_numb int PRIMARY KEY,
      actor_name varchar (60))
```

```
CREATE TABLE performance
      (actor_numb int,
      item_numb int,
      role varchar (60)
      PRIMARY KEY (actor_numb, item_numb)
      FOREIGN KEY performance2actor (actor_numb)
      REFERENCES actor
            ON UPDATE CASCADE
            ON DELETE CASCADE
      FOREIGN KEY performance2item (item_numb)
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE CASCADE);

CREATE TABLE producer
      (producer_name varchar (60) CONSTRAINT producer_pk PRIMARY KEY,
      studio varchar (40));

CREATE TABLE production
      (producer_name varchar (60),
      item_numb int
      PRIMARY KEY (producer_name, item_numb)
      FOREIGN KEY production2producer (producer_name)
```

```
        FOREIGN KEY production2producer (producer_name)
        REFERENCES producer
              ON UPDATE CASCADE
              ON DELETE CASCADE
        FOREIGN KEY production2item
        REFERENCES item
              ON UPDATE CASCADE
              ON DELETE CASCADE);
```

FIGURE 11.4 **The complete CREATE TABLE statements for the *Antique Opticals* database.**

Additional Column Constraints

There are additional constraints that you can place on columns in a table beyond primary and foreign key constraints. These include requiring unique values and predicates in CHECK clauses.

Requiring Unique Values

If you want to ensure that the values in a non-primary key column are unique, then you can use the UNIQUE keyword. UNIQUE verifies that all non-null values are unique. For example, if you were storing social security numbers in an employee table that used an employee ID as the primary key, you could also enforce unique social security numbers with

```
ssn char (11) UNIQUE
```

The UNIQUE clause can also be placed at the end of the CREATE TABLE statement, along with the primary key and foreign key specifications. In that case, it takes the form

```
UNIQUE (column_names)
```

Check Clauses

The CHECK clause to which you were introduced earlier in the chapter, in the "Domains" section, can also be used with individual columns to declare column-specific constraints. To add a constraint, you place a CHECK clause after the column declaration, using the keyword VALUE in a predicate to indicate the value being checked.

For example, to verify that a column used to hold true-false values is limited to T and F, you could write a CHECK clause as

```
CHECK (UPPER(VALUE) = 'T' OR UPPER(VALUE) = 'F')
```

Modifying Database Elements

With the exception of tables, database elements are largely unchangeable. When you want to modify them, you must delete them from the database and create them from scratch. In contrast, just about every characteristic of a table can be modified without deleting the table, using the ALTER TABLE statement.

Adding Columns

To add a new column to a table, use the ALTER TABLE statement with the following syntax:

```
ALTER TABLE table_name
ADD column_name column_data_type column_constraints
```

For example, if *Antique Opticals* wanted to add a telephone number column to the producer table, they would use

```
ALTER TABLE producer
ADD producer_phone char (12);
```

To add more than one column at the same time, simply separate the clauses with commas:

```
ALTER TABLE producer
ADD producer_phone char (12),
ADD studio_street char (30),
ADD studio_city char (15),
ADD studio_state char (2),
ADD studio_zip char (10);
```

Adding Table Constraints

You can add table constraints, such as foreign keys, at any time. To do so, include the new constraint in an ALTER TABLE statement:

```
ALTER TABLE table_name
ADD table_constraint
```

Assume, for example, that *Antique Opticals* created a new table named *states*, and included in it all the two-character US state abbreviations. The company would then need to add a foreign key reference to that table from the customer, distributor, and producer tables:

```
ALTER TABLE customer
ADD FOREIGN KEY customer2states (customer_state)
    REFERENCES states (state_name);

ALTER TABLE distributor
ADD FOREIGN KEY distributor2states (distributor_state)
    REFERENCES states (state_name);

ALTER TABLE producer
ADD FOREIGN KEY producer2states (studio_state)
```

```
ADD FOREIGN KEY producer2states (studio_state)
    REFERENCES states (state_name);
```

When you add a foreign key constraint to a table, the DBMS verifies that all existing data in the table meet that constraint. If they do not, the ALTER TABLE statement will fail.

Modifying Columns

You can modify columns by changing any characteristic of the column, including the data type, size, and constraints.

Changing Column Definitions

To replace a complete column definition, use the MODIFY command with the current column name and the new column characteristics. For example, to change the customer number in *Antique Opticals' customer* table from an integer to a character column, they would use

```
ALTER TABLE customer
MODIFY customer_numb char (4);
```

When you change the data type of a column, the DBMS will attempt to convert any existing values to the new data type. If the current values cannot be converted, then the table modification will not be performed. In general, most columns can be converted to character. However, conversions from a character data type to numbers, dates, and/or times require that existing data represent legal values in the new data type.

Given that the DBMS converts values whenever it can, changing a column data type may seem like a simple change, but it isn't. In this particular example, the customer number is referenced by foreign keys and therefore the foreign key columns must be modified as well. You need to remove the foreign key constraints, change the foreign key columns, change the

primary key column, and then add the foreign key constraints back to the tables containing the foreign keys. Omitting the changes to the foreign keys will make it impossible to add any rows to those foreign key tables because integer customer numbers will never match character customer numbers. Moral to the story: before changing column characteristics, consider the effect of those changes on other tables in the database.

Changing Default Values

To add or change a default value only (without changing the data type or size of the column), include the DEFAULT keyword:

```
ALTER TABLE order_line
MODIFY discount_percent DEFAULT 0;
```

Changing Null Status

To switch between allowing nulls and not allowing nulls, without changing any other characteristics, add NULL or NOT NULL as appropriate:

```
ALTER TABLE customer
MODIFY customer_zip NOT NULL;
```

or

```
ALTER TABLE customer
MODIFY customer_zip NULL;
```

Changing Column Constraints

To modify a column constraint without changing any other column characteristics, include a CHECK clause:

```
ALTER TABLE item
MODIFY retail_price
    CHECK (VALUE >= 12.95);
```

Deleting Table Elements

You can also delete structural elements from a table as needed, without deleting the entire table:

- To delete a column:

```
ALTER TABLE order_line
```

```
DELETE line_cost;
```

- To delete a CHECK table constraint (a CHECK that has been applied to an entire table, rather than to a specific column):

```
ALTER TABLE customer
DELETE CHECK;
```

- To remove the UNIQUE constraint from one or more columns:

```
ALTER TABLE item
DELETE UNIQUE (title);
```

- To remove a table's primary key:

```
ALTER TABLE customer
DELETE PRIMARY KEY;
```

Although you can delete a table's primary key, keep in mind that if you do not add a new one, you will not be able to modify any data in that table.

- To delete a foreign key:

```
ALTER TABLE item
DELETE FOREIGN KEY item2distributor;
```

Renaming Table Elements

You can rename both tables and columns:

- To rename a table, place the new table name after the RENAME keyword:

```
ALTER TABLE order_line
RENAME line item;
```

- To rename a column, include both the old and new column names, separated by the keyword TO:

```
ALTER TABLE item
RENAME title TO item_title;
```

Deleting Database Elements

To delete a structural element from a database, you "drop" the element. For example, to delete a table you would type:

```
DROP TABLE table_name
```

Dropping a table (or any database structural element, for that matter) is irrevocable. In most cases, the DBMS will not bother to ask you "are you sure?" but will immediately delete the structure of the table and all of its data, if it can. A table deletion will fail, for example, if it has foreign keys referencing it and one or more of the foreign key constraints contain ON DELETE RESTRICT. Dropping a table or view will also fail if the element being dropped is currently in use by another user.

Note: There is one exception to the irrevocability of a delete. If an element is deleted during a program-controlled transaction and the transaction is rolled back, the deletion will be undone. Undoing transactions is covered in Chapter 22.

You can remove the following elements from a database with the DROP statement:

- Tables
- Views

```
DROP VIEW view_name
```

- Indexes

```
DROP INDEX index_name
```

- Domains

```
DROP DOMAIN domain_name
```

For Further Reading

The Web sites in the following citations provide extensive SQL tutorials that you can use as references when needed.

SQLcourse.com. Creating Tables. http://www.sqlcourse.com/create.html
tutorialspoint. SQL – Constraints. http://www.tutorialspoint.com/sql/sql-constraints.htm
tutorialspoint. SQL – DROP or DELETE Table. http://www.tutorialspoint.com/sql/sql-drop-table.htm
w3schools.com. The SQL CREATE DATABASE Statement. http://www.w3schools.com/sql/sql_create_db.asp
w3schools.com. The SQL CREATE TABLE Statement. http://www.w3schools.com/sql/sql_create_table.asp

[1] The SUBSTRING function extracts characters from text data, beginning at the character following FROM. The number of characters to extract appears after the FOR.