Chapter 10

Constraints, Assertions, and Referential Integrity

10.1 Introduction

Your organization's databases are only as good as the integrity of the data contained within their tables and records. That is, the reliability and consistency of the data is critical to your applications, decision support and query tools, and other aspects of your information systems environment that utilize your database. Outright errors in the data are an obvious problem, although not the only one. Missing information, lost linkages among data elements, and other problems can result in incorrect reports and can lead the database users to lose confidence in the overall database environment.

The integrity of the data can be ensured by applying *constraints,* or rules, to the structure of the database and its contents. There are two primary methods by which these constraints can be specified, applied, and enforced:

- through the applications programs and software systems that use the database
- through the database management system itself

Traditionally, application programs have been the medium through which such constraints have been applied. Examples include

- Verifying that an input value for a particular data item is within the correct range of values (e.g., a new employee's age must be between 16 and 90, and hourly wage must be greater than the legal minimum wage rate) or is from the correct list of values (e.g., a new movie's type can only be CHILDREN, HORROR, COMEDY, OTHER).
- Verifying that the relationship between two or more values is correct (e.g., comparing the number of copies of a particular CD that we have on hand with the number requested in an incoming order).

The ever-growing use of relational database management systems introduced the concept of *database-enforced constraints* to many information systems users. That is, instead of having to write COBOL, C, or some other type of procedural code to specify and enforce constraints, the data definition language of the database itself provides facilities to perform these functions.

One of the catalysts encouraging the shift from procedural specification and enforcement of constraints to constraint management within the database realm was the explosion of personal computer-based applications in the early and mid-1980s, particularly those using the dBASE product family (particularly dBASE III and III+) or other products such as Clipper or FoxPro based on the Xbase language. These development environments provided a rich set of database-managed constraints for ranges of values, lists of values, and many of the other capabilities we'll discuss in this chapter, and application developers quickly became enamored with the combination of simplicity and power, and the resultant high productivity, of these features. Many of the more powerful relational DBMS products began featuring constraints within their own data definition languages, which of course led to SQL and standardization efforts.

There are a couple of major advantages to database-enforced constraints as compared with those managed through procedural applications code.

- Since the tables in a relational database are used by many different applications, each and every constraint must be specified—and modified, whenever necessary—in each applicable program when the application-enforced model is used. In contrast, database-enforced constraints can usually be designated in a single location (typically in the data definition language) and still apply to all necessary applications.
- Application-enforced constraints are usually more complex to code and manage than are database-enforced constraints. For example, to validate a list-of-values constraint in procedural code, program constructs such as loop control and if-then-else statements are necessary, as illustrated in the following pseudocode:


```
do until all input is correct:
   get movie title
   get movie cost
 . . .
   get movie type
 . . .
   if movie type = 'Children'
       or if movie type = 'Horror'
       or if . . .
   then movie type is correct
   else
       repeat loop
   end if
end do loop
```

A corresponding database-enforced constraint, however, is usually specified through a much less wordy clause in the DDL itself.

The very nature of the relational model—the division of a database's data among logically related tables, together with logical relationships among those tables—introduces an additional consideration with respect to database constraints, namely, that of keeping update anomalies, redundant data, and similar problems to a minimum. This is done through *normalization*. (See section 1.2.3, subsection called "Normal Forms," for a discussion of this concept in the relational model.) Data normalization typically requires several physical SQL tables to constitute some logical application object. For example, a MOVIE is represented by several different SQL tables: MOVIE_TITLES and MOVIES_STARS. In a more comprehensive database environment, perhaps 10 or more SQL tables would be used to describe various aspects of our movies.

Because of this split among multiple tables, it is highly desirable that constraints be used to ensure that the following types of situations do *not* occur. Consider the following scenarios.

Table 10-1 *Rows in MOVIES_STARS Table*

Wall Street	Douglas	Michael
Wall Street	Sheen	Charlie
Wall Street	Sheen	Martin

- Though the rows Table 10-1 exist, there is no row in our "main movie table" (MOVIE_TITLES) for the movie *Wall Street*. Therefore, we have a lost data problem.

- We received prerelease information about a movie called *The Postman, Part II*, and we enter preliminary information in our MOVIE_TITLES and MOVIES_STARS tables. Several months later when the movie is released on videocassette, the title has been changed to *The Last Postman*. We issue an UPDATE MOVIE_TITLES SET... statement to correct the title in one table but forget to do so in the MOVIES_STARS table. Until we remember that we forgot (?!), we have a problem of a lost relationship between these two tables; simple join operations, for example, will not make the association between applicable rows because of the difference in the titles.

In this chapter we look at the facilities provided by SQL:1999 to handle different types of constraint specification and enforcement. First we look at some basic constraints, such as those used to specify whether or not a column can have a NULL value or what values are permissible in a column, and then we turn our attention to those dealing with referential integrity—that is, rules that require values stored in certain columns of the rows of one table also be stored in certain columns of the rows of another table (under certain circumstances, even of the same table).

10.2 Column Constraints and Table Constraints

Let's expand on our basic table definitions of Chapter 4, "Basic Data Definition Language (DDL)," by adding constraints to the DDL. Let's look at some of the basic constraints you can specify on columns and in tables. Syntax 10-1 shows what these constraints look like in BNF form. Here, we're showing you the high-level syntax of constraints, both table constraints and column constraints. You'll see the syntax for specific types of constraints later: unique constraints in Syntax 10-2, check constraints in Syntax 10-3, and referential constraints in Syntax 10-5.

Syntax 10-1 *Syntax of Constraint Definition*

```
• <table constraint definition> ::=
      [ <constraint name definition> ]
      <table constraint> [ <constraint characteristics> ]

  <table constraint> ::=
        <unique constraint definition>
      | <referential constraint definition>
      | <check constraint definition
```



```
• <column constraint definition> ::=
      [ constraint name definition ]
      <column constraint> [ <constraint characteristics> ]

  <column constraint> ::=
        NOT NULL
      | <unique constraint definition>
      | <referential constraint definition>
      | <check constraint definition>

• <constraint name definition> ::=
      CONSTRAINT <constraint name>

  <constraint characteristics> ::=
        <constraint check time> [ [ NOT ] DEFERRABLE ]
      | [ NOT ] DEFERRABLE [ <constraint check time> ]

  <constraint check time> ::=
        INITIALLY DEFERRED
      | INITIALLY IMMEDIATE
```

SQL:1999, like SQL-92, defines its column constraints (constraints that you define as part of a column definition) to be syntactic shorthand for table constraints (constraints that are defined as elements of a table definition). While this might seem unnecessarily complex and even confusing to application developers like yourself, it did simplify the specification of SQL constraints enough to justify doing so. You'll see in Section 10.2.2 why we mentioned this fact.

10.2.1 Constraint Deferral

Before we begin discussing the various types of constraints, let's consider the issue of just when a constraint is checked. In the introduction to this chapter, we saw that constraints are used to enforce valid states of the database. SQL's policy is that constraints are checked, by default, at the end of execution of every SQL statement. Actually, it's more correct to say that SQL requires that no constraint is violated at the time an SQL statement terminates. Saying it this way avoids the implication that the database has to explicitly check constraints that could not have been violated by a specific statement, such as a SELECT statement that makes no changes to the data.

Remember we said that this is SQL's default policy. It's possible that your environment will have some applications that cause certain inconsistencies in your data to span the execution of two or more SQL statements. If the policies of our

music and video store included a rule that prohibited stocking more than $1,000,000 of DVDs and a second rule that prohibited returning merchandise before it has been accepted into inventory, we could easily find ourselves unable to return unordered merchandise solely because our inventories are nearing, but less than, that limit. However, if we were able to *temporarily* prevent the database system from checking the constraint that enforces the million-dollar limit, we could accept the merchandise into inventory and then return it to the shipper, all without causing a constraint violation.

SQL:1999, like SQL-92 before it, allows you to declare that your constraints are deferrable. Deferred constraints are not checked at the end of every SQL statement. However, every constraint, even a deferred constraint, is always checked at the end of every transaction.

At the end of Syntax 10-1, there are BNF productions named <constraint characteristics> and <constraint check time>. <constraint characteristics> allow you to specify that a particular constraint either is or is not deferrable. If a constraint is deferrable, then you can also specify whether it is initially deferred—that is, deferred by default when each transaction begins. Deferrable constraints that are not initially deferred can be explicitly deferred by the use of the SET CONSTRAINTS statement. Deferred constraints, whether initially deferred or deferred by the use of SET CONSTRAINTS, can be checked upon demand by using SET CONSTRAINTS to set them to immediate checking.

When a nondeferred constraint is checked and is found to have been violated (that is, the data in the database does not satisfy the constraint), then the effects on the database of the statement causing the constraint to be checked are undone. By undoing the offending statement, the database can be restored to a consistent state. However, if a deferred constraint is checked, either at the end of a transaction or as a result of executing a SET CONSTRAINTS statement, the statement that caused the database to become inconsistent is not necessarily identifiable. As a result, it's not possible to just undo the effects of one statement. Instead, unless your application is somehow able to determine the problem and update the database to become consistent, the entire transaction will have to be undone (see Chapter 15, "Transaction Management"). Because of this difficulty, we recommend that you defer constraint evaluation only in limited circumstances.

10.2.2 NOT NULL

In Chapter 4, "Basic Data Definition Language (DDL)," we briefly mentioned the NOT NULL constraint while discussing basic Data Definition Language syntax.

Since NOT NULL is an example of an SQL:1999 constraint (and, arguably, the simplest to specify and understand), we will quickly repeat our discussion.

NOT NULL indicates that null values are not permissible in any row of that table for the specified column. That is,

```
CREATE TABLE movie_titles (
    title CHARACTER (30) NOT NULL,
    |
    |
)
```

will ensure that no row in the movie_titles table will ever have a null value in the title column. As we previewed in Chapter 4, "Basic Data Definition Language (DDL)," and mention later in this chapter, this restriction can also be specified using the multipurpose CHECK constraint.

In Syntax 10-1, you probably noticed that the alternatives for column constraints and for table constraints were exactly the same, except that NOT NULL is not permitted for table constraints. In the transformation of the column constraint NOT NULL into a table constraint, SQL:1999 converts NOT NULL into a check constraint, described in section 10.2.4.

10.2.3 UNIQUE

In many situations, we want to ensure that a specific column or group of columns designates some type of unique identifier for a table. In Chapter 8, "Working with Multiple Tables: The Relational Operators," we showed an example of a table of music distributors from which we order CDs and tapes. Changing our example from that chapter a little bit, since each distributor has a different unique identifier, we can specify the syntax in Example 10-1.

Example 10-1 *Using the UNIQUE Constraint*

```
CREATE TABLE music_distributors (
    distributor_id CHARACTER (15) UNIQUE,
    |          -- {rest of column definitions}
    |
    |
    ...)
```

Alternatively, we can specify the uniqueness constraint in a slightly different way, as shown in Example 10-2.

Example 10-2 *Alternative Form of UNIQUE Constraint*

```
CREATE TABLE music_distributors (
    distributor_id CHARACTER (15),
    |          -- {rest of column definitions}
    |
    |
    UNIQUE (distributor_id),
    ...)
```

You might be wondering what happened to the NOT NULL constraint if you took a peek back at Chapter 8, "Working with Multiple Tables: The Relational Operators," while reading this example. SQL-89 required you to specify NOT NULL UNIQUE in your constraint definition; you were not permitted to say UNIQUE without also saying NOT NULL. Additionally, the ordering of the two constraints was restrictive; you could not say UNIQUE NOT NULL, but rather had to place the NOT NULL constraint first. SQL-92 relaxed that requirement and allowed UNIQUE to be specified without an accompanying specification of NOT NULL. That is, SQL-92 did not include in the uniqueness check implied by a UNIQUE predicate any rows in which any of the columns participating in the UNIQUE predicate had the null value. The SQL-92 syntax and semantics for the UNIQUE predicate continues into SQL:1999. (That is, UNIQUE alone permits multiple null values in a column.)

Still, specifying UNIQUE without NOT NULL is considered a dangerous practice by leading relational database proponents and is not recommended. (However, see section 10.4 for a different predicate that avoids the problems associated with failing to use NOT NULL along with a UNIQUE predicate.) For syntactical completeness, however, we'll show you what happens if you were to use only the UNIQUE constraint.

(Note that SQL:1999 also defines a UNIQUE column constraint to be exactly the same as a CHECK constraint on the table, as we will see later.)

Let's look at our music_distributors example again. With the above constraint definition, the following table contents would be legal within SQL:1999. That's normally not a good idea, particularly if the column is used to uniquely identify a business entity (like the distributor for the music we sell), but having only a single row with a null value in the unique column makes that possible, if a bit unorthodox.

Table 10-2 *Valid MUSIC_DISTRIBUTORS Table Contents*

DISTRIBUTOR_ID...
Music All the Time
CD Wholesalers, Inc.
null
Yesterday's Music Found

It's probably worse that the following is also allowed, since it's no longer possible to identify certain rows uniquely at all, since there are multiple rows with the same value (well, the same "nonvalue") in the unique column.

Table 10-3 *Valid, but Even Less Convenient, MUSIC_DISTRIBUTORS Table Contents*

DISTRIBUTOR_ID...
Music All the Time
null
CD Wholesalers, Inc.
null
Yesterday's Music Found

In SQL:1999 terminology, UNIQUE may be defined as meaning "not equal." In our example, no two music_distributors rows may have distributor_id column values that are equal. (Two rows with null values are considered not equal and are thus not prohibited by UNIQUE.)

The complete syntax for unique constraints can be seen in Syntax 10-2, even though we've described only some of the syntax so far (those alternatives using the keyword UNIQUE); we'll cover primary keys in section 10.4.

Syntax 10-2 *Syntax of UNIQUE Constraints*

```
<unique constraint definition> ::=
      <unique specification>
          <left paren> <unique column list> <right paren>
    | UNIQUE ( VALUE )

<unique specification> ::=
      UNIQUE
    | PRIMARY KEY

<unique column list> ::= <column name list>
```

The alternatives in which a list of column-names is specified permit you to force the database system to check that no two rows in the table have the same values in the combination of the specified columns.

The alternative UNIQUE (VALUE) allows you to ensure that a table contains no duplicate rows when all columns of the table are considered; it is merely a shorthand notation for

```
UNIQUE ( column-name-1, column-name-2,..., column-name-n )
```

in which you list every column of the table.

10.2.4 CHECK

One of the most flexible, and therefore useful, constraints within SQL:1999 is the CHECK constraint. This allows you to specify a wide range of rules for your tables, such as range of values, list of values, and others.

The syntax of the CHECK constraint is shown in Syntax 10-3.

Syntax 10-3 *Syntax of CHECK Constraint*

```
CHECK ( search-condition )
```

As we shall see, the search-condition can be any sort of search condition that we've learned to use. However, there is one unusual aspect to them. Let's say that you're defining a CHECK constraint on table movie_titles. In this search condition, you refer to any column of that table (e.g., our_dvd_cost) just by its name; the context is always the current row of that table. By contrast, if you need to access a column in another table, you have to use a subquery.

As with UNIQUE (discussed above), there are several alternate syntax specifications for CHECK. Assume that we want to ensure that our_dvd_cost for all movies is less than $90.00 (Example 10-3).

Example 10-3 *CHECK Clause Example (Column Constraint)*

```
CREATE TABLE movie_titles (
    ...
    our_dvd_cost     DECIMAL (5,2)
      CHECK ( our_dvd_cost < 90.00 ),
    | --{rest of column definitions}
    |
)
```


Alternatively, the constraint could be specified as shown in Example 10-4.

Example 10-4 *Alternative Form of CHECK Clause (Table Constraint)*

```
CREATE TABLE movie_titles (
    ...
    our_dvd_cost     DECIMAL (5,2),
    | --{rest of column definitions}
    |
    CHECK ( our_dvd_cost < 90.00 ) ,
    |
    |
)
```

It is important to note the rule that applies to the search condition of the CHECK constraint. Specifically, the search condition *must not be False* for any row in the table (as opposed to having to be True for every row in the table). This means that the table can be *empty* and the CHECK constraint will be satisfied (that is, no row violates the search condition because there are no rows!). This is rather subtle but is important to understand, especially when analyzing and debugging your applications. That is, the search condition can evaluate to Unknown and the CHECK constraint is satisfied.

Perhaps you also want to be sure that our_dvd_cost is positive. You could specify this with an additional CHECK clause (Example 10-5).

Example 10-5 *Using Two CHECK Constraints Together*

```
...CHECK ( our_dvd_cost < 90 ),
   CHECK ( our_dvd_cost > 0 ) ...
```

(The comma in Example 10-5 would be used only if these two constraints were specified as table constraints; if they're column constraints, then the comma is prohibited.) Or, you could specify the two conditions in a single constraint (Example 10-6).

Example 10-6 *Combining Two CHECK Constraints into One Using AND*

```
...CHECK ( our_dvd_cost < 90 AND our_dvd_cost > 0 )
```

In this case, all of the CHECK clauses in a table are effectively ANDed together. This is true even if the CHECK clauses reference different columns (Example 10-7).

Example 10-7 *Another Example with Two CHECK Constraints and Then Combining Them*

```
...CHECK ( our_dvd_cost < 90 ),
   CHECK ( tapes_owned > 0 )
```

The example in Example 10-7 is equivalent to

```
...CHECK (our_cost < 90 AND tapes_owned > 0)
```

Let's look at some of the typical business rules you are likely to want to specify through your database and how they might be coded using the CHECK constraint. Don't forget that a CHECK constraint uses an ordinary search condition like those that we learned about in Chapter 8, "Working with Multiple Tables: The Relational Operators." There are, however, a couple of limitations. When the CHECK constraint is a column constraint, then it cannot reference any column other than the column with which it is associated; when it is a table constraint, then it can reference only columns that are in the table with which it is associated.

Minimum or Maximum Values

We already saw how a maximum value such as our_dvd_cost < 90 can be specified. Similarly, a minimum value constraint can be designated. To ensure that all of our e-tailer employees earn at least the current minimum wage rate, you might specify the syntax shown in Example 10-8.

Example 10-8 *Employees Table Definition and Minimum Wage CHECK Constraint*

```
CREATE TABLE employees (
    emp_last_name        VARCHAR (30) NOT NULL,
    emp_first_name       VARCHAR (15) NOT NULL,
    emp_address          VARCHAR (30),
    emp_city             VARCHAR (15),
    emp_state            CHAR (2),
    emp_zip              VARCHAR (9),
    emp_phone            CHAR (8),
    emp_start_date       DATE,
    emp_hourly_rate      DECIMAL (7,2)
    |
    |
    |
```



```
    CHECK (emp_hourly_rate >= 5.25),
    |
    |
    |
)
```

On occasion, you might wish to designate a minimum or maximum value check against some dynamic value rather than a hard-coded one as in the examples above. Assume that some table competition_prices contains information collected about your competitors, and you wish to ensure that none of our DVD format movie titles are sale-priced higher than the maximum price charged by any competitor for any of their DVDs. To accomplish this, you'd use a subquery in the constraint, as shown in Example 10-9.

Example 10-9 *Dynamic Definition of a Maximum Price CHECK Constraint*

```
CREATE TABLE competition_prices (
    |
    |
    |
    max_dvd_price DECIMAL (5,2),
    |
    |
    | )

CREATE TABLE movie_titles (
    |
    |
    |
    current_dvd_sale_price DECIMAL (5,2),
    |
    |
    CHECK ( current_dvd_sale_price <=
            ( SELECT MAX ( max_dvd_price )
              FROM competition_prices ) )
    |
)
```

You should be aware that any constraint that references data in more than one table is checked after a change to any one or more of those tables (both competition_prices and movie_titles, in this case). This constraint, therefore,

¿esto se puede hacer en Oracle? un subquery dentro del check

will also cause a failure if an attempt is made to insert a record into competition_prices with a max_dvd_price that is lower than some current_dvd_sale_price in movie_titles.

We can also specify constraints for a column's values against other values in the same table. Assume that we want to ensure that no DVD movie has a current sales price that is lower than our lowest cost for *any* DVD movie; this is shown in Example 10-10.

Example 10-10 *CHECK Constraint with Dynamic Checking against Column Values in the Same Table*

```
CREATE TABLE movie_titles (
    |
    |
    |
    our_dvd_cost              DECIMAL (5,2),
    current_dvd_sale_price    DECIMAL (5,2),
    |
    |
    CHECK ( current_dvd_sale_price >=
            ( SELECT MIN ( our_dvd_cost )
              FROM  movie_titles ) )
    |
)
```

Range of Values

It's often desirable to place both minimum and maximum permissible values on one or more columns. Many DBMS products allow you to specify a parameter such as RANGE, in which the boundary values are designated. SQL:1999 allows you to use a combination of CHECK and BETWEEN. Assume that we wish to place "reasonableness" boundaries on our numeric columns within movie_titles. We can do so as shown in Example 10-11.

Example 10-11 *CHECK Constraint Used for Range of Values*

```
CREATE TABLE movie_titles (
    title                       CHARACTER ( 30 ) NOT NULL,
    year_released               DATE,
    our_tape_cost               DECIMAL ( 5,2 ),
    our_dvd_cost                DECIMAL ( 5,2 ),
```



```
    regular_tape_sale_price     DECIMAL ( 5,2 ),
    current_tape_sale_price     DECIMAL ( 5,2 ),
    regular_dvd_sale_price      DECIMAL ( 5,2 ),
    current_dvd_sale_price      DECIMAL ( 5,2 ),
    part_of_series              CHARACTER ( 3 ),
    movie_type                  CHARACTER ( 10 ),
    tapes_owned                 INTEGER,
    dvds_owned                  INTEGER,
    tapes_in_stock              INTEGER,
    dvds_in_stock               INTEGER,
    total_tape_units_sold       INTEGER,
    total_dvd_units_sold        INTEGER,
    |
    |
    |
    CHECK (
      ( our_tape_cost BETWEEN .99 AND 90.00 )
    AND
      ( our_dvd_cost BETWEEN .99 AND 90.00 )
    AND
      ( regular_tape_sale_price BETWEEN .25 AND 90.00 )
    AND
      ( current_tape_sale_price BETWEEN .25 AND 90.00 )
    AND
      ( regular_dvd_sale_price BETWEEN .25 AND 90.00 )
    AND
      ( current_dvd_sale_price BETWEEN .25 AND 90.00 )
    AND
      ( tapes_owned BETWEEN 0 AND 1000 )
    AND
      ( dvds_owned BETWEEN 0 AND 1000 )
    AND ... )
    |
    |
    |
)
```

Several points are worth noting about the first example. Note the use of AND as part of the above constraint. If applicable, OR may also be used as part of a CHECK constraint, as well as NOT or any Boolean expression (including any predicate).

Additionally, the sign (positive or negative) of a particular column's permissible values can be designated through the CHECK constraint. By specifying that a value is (BETWEEN 0 and {*some value*}) or (BETWEEN ({*some negative value*} and 0), positive and negative values, respectively, can be enforced. An important point to remember is that the ordering of the values within the BETWEEN clause is *very* important. Remember the meaning of BETWEEN: allowable values are (1) greater than or equal to the first value, and (2) less than or equal to the second value (see Chapter 7, "Predicates"). Therefore, the following syntax is *not* very useful:

```
CHECK (some_negative_number BETWEEN 0 AND -10000)
```

In fact, that CHECK constraint can *never* be satisfied (except by an empty table).

List of Values

It's often desirable to specify that a specific list of values, and no others, are permissible values for a given column. For example, we may wish to specify that each and every row in the movie_titles table must have one of a given set of values for the movie_type column. We can do so as shown in Example 10-12.

Example 10-12 *Using a CHECK Constraint for List-of-Values Constraint*

```
CREATE TABLE movie_titles (
    title           CHAR (30) NOT NULL,
    |
    |
    |
    movie_type      CHAR (10),
    |
    |
    |
    CHECK ( movie_type IN
          ( 'Horror', 'Children', 'Comedy', 'Musical',
            'Romance', 'Western', 'Adventure', 'Other' ) ) )
```

Exclusion from Another Table

On occasion, you might wish to make certain that values in some column in a table are different from those of some other table. For example, we may have some table discontinued_music_titles in which we store rows for each music

title that has been discontinued. We can ensure that no discontinued title appears in our music_titles table, as shown in Example 10-13.

Example 10-13 *Using CHECK Constraint to Maintain Exclusivity of Contents between Two Tables*

```
CREATE TABLE music_titles (
    music_id                CHAR (12) NOT NULL,
    title                   CHAR (30) NOT NULL,
    artist                  CHAR (40),
    artist_more             CHAR (50),
    |
    |
    |
)

CREATE TABLE discontinued_music_titles (
    music_id                CHAR (12) NOT NULL,
    title                   CHAR (30) NOT NULL,
    artist                  CHAR (40),
    artist_more             CHAR (50),
    |
    |
    |
    CHECK ( music_id <> ANY
          ( SELECT music_id FROM music_titles ) )
    |
    |
    |
)
```

This sort of constraint allows you to effectively form a unique constraint across more than one table. Another way to achieve the same effect is shown in Example 10-14.

Example 10-14 *Alternative Constraint Expression for Exclusivity of Contents*

```
CHECK ( UNIQUE ( SELECT music_id
                 FROM music_titles
                 UNION ALL
                 SELECT music_id
                 FROM discontinued_music_titles ) )
```

10.2.5 Constraint Names

Each of SQL's table constraints and column constraints may, optionally, have a name assigned to it. In reality, all constraints have a name, although not necessarily one assigned by the user. For example, alternative expressions for several of the earlier constraints in this chapter are shown in Example 10-15.

Example 10-15 *Named Constraints*

```
CREATE TABLE movie_titles (
    title                CHARACTER (30)
      CONSTRAINT title_not_null NOT NULL,
    |
    |
    |
    movie_type           CHARACTER(10),
    |
    |
    |
    CONSTRAINT check_movie_type
      CHECK ( movie_type IN
            ( 'Horror', 'Children', 'Comedy', 'Musical',
              'Romance', 'Western', 'Adventure', 'Other' ) )
```

We strongly recommend that constraint names always be used, even though they are optional. Applications can then report more complete information when constraint violations occur, because they can get the name of the violated constraint out of the diagnostics area (see Chapter 20, "Diagnostics and Error Management"), thereby helping with the problem analysis process. Constraint names were new to SQL-92 and have proven to be very useful. If you don't give names to your constraints, the DBMS will assign them for you, but they will probably be less meaningful and less mnemonic than those you have chosen yourself.

Whether you or the DBMS assigns the constraint name, it will be reported with any error caused by violating the constraint, and you can use the name to SET the constraint to DEFERRED or IMMEDIATE (see Chapter 15, "Transaction Management") or to DROP the constraint (see Chapter 4, "Basic Data Definition Language (DDL)").

Note also that constraint names may be assigned as well to the two constraints that don't directly utilize CHECK: NOT NULL and UNIQUE. Additionally, these two constraints could, if desired, be expressed using CHECK, as follows:

De aqui salieron mas cuentas láminas.


```
CONSTRAINT column_not_null
  CHECK (columnname IS NOT NULL)

CONSTRAINT column_unique
  CHECK (UNIQUE (SELECT columnname FROM table) )
```

10.3 Assertions

As we have seen, SQL has several types of constraints. Of those that we've examined so far, all have been "attached" to tables or, equivalently, to columns in tables. However, SQL:1999 also provides you with a different kind of constraint that isn't attached to a particular table. This constraint, called an *assertion*, is a standalone constraint in a schema and is normally used to specify a restriction that affects more than one table; it was introduced in SQL-92.

A table (or column) constraint is normally used to make a restriction on the data that is stored in the table (or column) to which the constraint is attached. It is possible (given some awkwardness, such as using subqueries) to express restrictions involving multiple tables with regular table CHECK constraints, but this is not really recommended (among other things, performance tends to suffer using this approach, since the database system is less able to determine your real intent). Instead, you should use assertions for that purpose.

ojo

Let's suppose that you want to ensure that the total number of movies (DVDs and videocassettes) plus the total number of music items (CDs plus cassettes) in stock at any one time is never greater than some limit imposed by your insurance company or banker, say, 500,000. (This sort of situation illustrates a primary use of constraints: to specify the rules of your business directly in the database.) To express this restriction as a table constraint would be awkward at best. First, you'd have to decide whether to put the constraint on the movie_titles table or on the music_inventory table. Then you'd have to write a statement something like that shown in Example 10-16.

Example 10-16 *A Complicated Way of Using Constraints to Handle Inventory Limits*

```
CREATE TABLE movie_titles (
    title...
    ...
    CONSTRAINT maximum_inventory
      CHECK ( ( SELECT SUM ( tapes_in_stock + dvds_in_stock )
                FROM movie_titles )
```

```
            + ( SELECT SUM ( number_cd_now_in_stock +
                             number_cassette_now_in_stock )
                FROM music_inventory )
            < 500000 ),
      ...
)
```

That, of course, will work—at least most of the time—but it sure doesn't feel natural to put it here. Of course, that constraint implies that the two sums have to be produced every time any of the *_in_stock values are changed, which may cause performance problems.

A more natural (if not necessarily better performing) way to express the same restriction is to state it as a *standalone* constraint—that is, as an assertion.

```
CREATE ASSERTION maximum_inventory
  CHECK ( ( SELECT SUM ( tapes_in_stock + dvds_in_stock )
            FROM movie_titles )
          + ( SELECT SUM ( number_cd_now_in_stock +
                           number_cassette_now_in_stock )
            FROM music_inventory )
          < 500000 )
```

Note that the syntax of the CHECK portion of the assertion is identical to the CHECK constraint in Example 10-16 and that we even named the constraint the same. However, as an assertion, this CHECK is *not* attached to any particular table. This has two advantages. We've already stated the first: it is more natural to express restrictions not specific to one table as standalone constraints instead of as table constraints.

Also, we're not home free with the use of assertions when their referenced table(s) are empty. In this example, in the unlikely case that either the movie_titles table or the music_inventory table happened to be empty, the corresponding SUM expression would return a null value, making the condition of the assertion Unknown.

The other advantage is due to a subtle characteristic of table constraints: because table constraints are intended to govern the restrictions on data stored in that table, they are required to be True if *and only if* there is some data stored in the table to which they are attached. This means that the constraint in our first example would *always* be satisfied if the movie_titles table were empty. Therefore, if we were to decide to no longer handle movie media, but to concentrate solely on music, we could stock as many CDs as we'd like without violating the constraint. (And the Wall Street analysts gazing through our quarterly reports

might not be happy to learn that we've now stocked 2,000,000 CDs on hand in a down economy—and there goes our stock price, down by about 35% the next day!) By contrast, an assertion expresses a restriction that must always be obeyed (or satisfied) in the database, regardless of whether any particular table has data stored in it or not. (Of course, some assertions might require that a table have data stored in it, but we think you know what we mean.)

Note that some table constraints can never be violated.

```
CREATE TABLE sample (
    col1...
    ...
    CONSTRAINT table_never_empty
      CHECK ( ( SELECT COUNT(*)
                FROM sample ) > 0 ),
    ...
)
```

This constraint can never be violated, because it's satisfied when the table is empty and it's True when the table is not empty. That surprising statement—that the constraint is satisfied when the table is empty—is caused by SQL's rule that a constraint can be violated *only* by one or more rows in a table, and not by the table itself. If we were to express this as an assertion, though, the assertion would, indeed, guarantee that the table always had at least one row stored in it.

```
CREATE ASSERTION sample_table_never_empty
    CHECK ( ( SELECT COUNT(*) FROM sample ) > 0 )
```

The format of the statement to create an assertion is shown in Syntax 10-4.

Syntax 10-4 *Syntax of Assertion Definition*

```
CREATE ASSERTION constraint-name
    CHECK ( search-condition ) [ attributes ]
```

The constraint name is always required for assertions. We've already mentioned that all constraints have names, but the system will assign a name for table or column constraints that you don't name yourself (mainly to be compatible with programs written before SQL-92 was implemented). However, assertions are standalone objects in the schema, so they must have a name, and SQL:1999 requires you to give them a name of your choosing, as did SQL-92.

The search-condition is described in Chapter 7, "Predicates"; basically, it consists of one or more predicates (multiple predicates are connected with ANDs, ORs, and NOTs) that express the condition that must be satisfied (alternately, that must not be violated) in the database. It may reference one or more tables, but it has no inherent table context, so you'll have to provide the context by using a query expression (actually, you have to use a subquery; see discussion in Chapter 9, "Advanced SQL Query Expressions").

Ordinary constraints are checked at the end of every SQL statement unless they are deferred. See section 10.2.1 for information about constraint deferral. The attributes are the same as for any other constraint, as discussed at the very end of section 10.2.5.

Assertions, like other constraints, have some restrictions. For example, all of the values used must be literals or database values; you can't reference host variables or parameters, datetime functions, CURRENT_USER, SESSION_USER, or SYSTEM_USER. The reason for this limitation is subtle: since the value returned by those functions and special variables can change frequently, their use could cause the data in the database to violate a constraint even when the data itself is not changed. Assertion names, like any constraint names, can be qualified with a schema name. If you choose to do this and your CREATE ASSERTION statement is part of a CREATE SCHEMA, then the schema names have to match. Most assertions will reference one or more database columns; you must have REFERENCES privileges on each column that your assertions reference. If your assertion doesn't reference a particular column, but does reference a table (e.g., using COUNT(*)), then you must have REFERENCES privileges on at least one column of the table.

10.4 PRIMARY KEY

The relational database model contains the concept of a *key*, that is, one or more columns within a table that have some unique value by which individual rows can be identified. In some cases, a single column (such as a social security number or employee ID number) is enough to uniquely identify given rows. In other cases, a combination of columns (as in the example below) is required. We showed you the syntax used to specify a primary key in Syntax 10-2.

Let's look at our movie_stars table. To make the explanation easier to understand, let's disregard the year_released and assume that all movie titles are unique. We will also, again for purposes of this example, assume that no two actors or actresses with exactly the same name appear in the same movie. And finally, another restriction for simplicity is that we'll ignore the actors' and

actresses' middle names if they are used (e.g., Mary *Tyler* Moore, James *Earl* Jones).

Therefore, we initially have the following table definition (we will need to modify this, as we will see):

```
CREATE TABLE movies_stars (
    movie_title        CHARACTER (30) NOT NULL,
    actor_last_name    CHARACTER (35) NOT NULL,
    actor_first_name   CHARACTER (25) );
```

Table 10-4 *Result of CREATE TABLE*

MOVIE_TITLE	*ACTOR_LAST_NAME*	*ACTOR_FIRST_NAME*
The Way We Were	Redford	Robert
The Way We Were	Streisand	Barbra
Prince of Tides	Nolte	Nick
Prince of Tides	Streisand	Barbra
|		
|		
|		

In Table 10-4, no single column is sufficient to uniquely identify given rows. Since any movie may have (and usually does have) more than one star, there will be multiple rows in which the same movie title appears. Similarly, the combination of actor_last_name and actor_first_name is also not enough to constitute a key, since actors (especially the stars, who appear in our database table) appear in many movies over the course of their careers. Only the combination of three columns—in this case, all of the columns of the table—is sufficient to constitute a key.

In some tables, there are multiple *candidate keys*. That is, there are several different options from which you can choose a column or combination of columns for the unique identification process. Assume for a moment that our MOVIE_TITLES table has a column MOVIE_NUMBER, a column that has a unique value in each row. MOVIE_NUMBER is a candidate key, as is TITLE (or in our original example, the combination of TITLE and YEAR_RELEASED).

Whichever candidate key you choose becomes your *primary key*. The primary key and its related syntax is the concept with which we are concerned. (You are free, if you wish, to use UNIQUE constraints to identify other candidate keys of interest to your applications.) Since movie titles are unique for the purposes of this example, we can designate that column as a primary key:

```
CREATE TABLE movie_titles (
    title CHARACTER (30) PRIMARY KEY,
    |
    |
    |
)
```

SQL-89 required you to say NOT NULL PRIMARY KEY. SQL-92 (in the Intermediate and Full levels) permitted you to state only PRIMARY KEY, although this means precisely the same thing. Primary keys have the inherent property that no row can have a null value for the primary key columns; this continues in SQL:1999. There can never be more than one primary key defined for a table, but a primary key can use more than one column.

Primary keys may also be designated through table constraints (discussed earlier in this chapter in conjunction with CHECK). Let's look at our alternative primary key definition.

```
CREATE TABLE movies_stars (
    movie_title          CHARACTER (30) NOT NULL,
    actor_last_name      CHARACTER (35) NOT NULL,
    actor_first_name     CHARACTER (25) NOT NULL,

    CONSTRAINT stars_pk PRIMARY KEY
      ( movie_title, actor_last_name, actor_first_name ),
)
```

The restriction prohibiting null values for primary key columns means that *no* column that forms part of a multiple column primary key is allowed to have a null value. This restriction presents somewhat of a problem for our application as currently defined. Recall from Table 10-4 that the column actor_first_name does *not* have a NOT NULL constraint applied, in order to allow for single-name actors and actresses (Sting, Cher) or for groups (the Marx Brothers). However, as we also said above, we need all three columns in the movies_stars table to uniquely identify any given row. Therefore, we have a dilemma: how to resolve these two conflicting problems.

Our view is that primary keys (and foreign keys; see section 10.5.1) are important characteristics of your database and should be implemented and maintained through your SQL syntax whenever possible. Therefore, we will add a NOT NULL constraint to the actor_first_name column definition in order to be able to create key definitions. Single-name actors and groups will then have 25 blanks in the actor_first_name column, likely placed there by some host application

program. Alternatively, you could choose to not specify a PRIMARY KEY for the table, but to specify a UNIQUE constraint for all of the columns in the primary key and separately specify a CHECK constraint that prohibits the combination of all of those columns from being null:

```
CONSTRAINT stars_pk1
  UNIQUE ( movie_title, actor_last_name, actor_first_name ),
CONSTRAINT stars_pk2
  CHECK ( ( movie_title, actor_last_name, actor_first_name )
          NOT NULL )
```

This is representative of the types of tradeoff decisions you must often make when designing and implementing your SQL databases. When facets such as those discussed above conflict, you must decide which to implement and which to work around.

Most SQL implementations use the idea of an *index* (as in the typical CREATE INDEX . . . statement) to enforce uniqueness in a table. However, we feel that because uniqueness is an inherent property of data, it should not be relegated to being an implementation issue but should be designated in the standard. Therefore, whether or not a specific SQL:1999 implementation supports indices (indices also have additional uses, such as performance enhancement), the PRIMARY KEY does designate support for key fields.

Now that we've seen how to define primary keys, let's turn our attentions to FOREIGN KEYs and the concept of referential integrity.

10.5 Referential Integrity

Referential integrity reflects the purpose of constraints whose job it is to ensure that values stored in one place in your database reference existing data in some other place in the database. That is, *references* to data must be valid. (By contrast, the kinds of constraints we've discussed up to this point in this chapter are sometimes said to enforce "semantic integrity.") Referential integrity in a database allows you to require that your sales invoices never reference a part number that is not tracked by your database, or that you never accept shipments from a supplier with whom you do not have an existing relationship. Since SQL is primarily a *value-based* system, referential integrity is not implemented by means of *pointers* (a technique that was used in some earlier data models). Instead, it is implemented by means of constraints that specify that a value in a column of a row *here* is the same as the value in the corresponding column of some row *there*.

The SQL:1999 syntax for the constraints dealing with referential integrity can be seen in Syntax 10-5. The syntax is rather lengthy and perhaps intimidating, but don't worry: we'll cover it in small bites.

Syntax 10-5 *Syntax of Referential Integrity Constraints*

```
<referential constraint definition> ::=
    FOREIGN KEY <left paren> <referencing columns> <right paren>
      <references specification>

<references specification> ::=
    REFERENCES <referenced table and columns>
      [ MATCH <match type> ]
      [ <referential triggered action> ]

<match type> ::=
      FULL
    | PARTIAL
    | SIMPLE

<referencing columns> ::=
    <reference column list>

<referenced table and columns> ::=
    <table name>
      [ <left paren> <reference column list> <right paren> ]

<reference column list> ::= <column name list>

<referential triggered action> ::=
      <update rule> [ <delete rule> ]
    | <delete rule> [ <update rule> ]

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::=
      CASCADE
    | SET NULL
    | SET DEFAULT
    | RESTRICT
    | NO ACTION
```


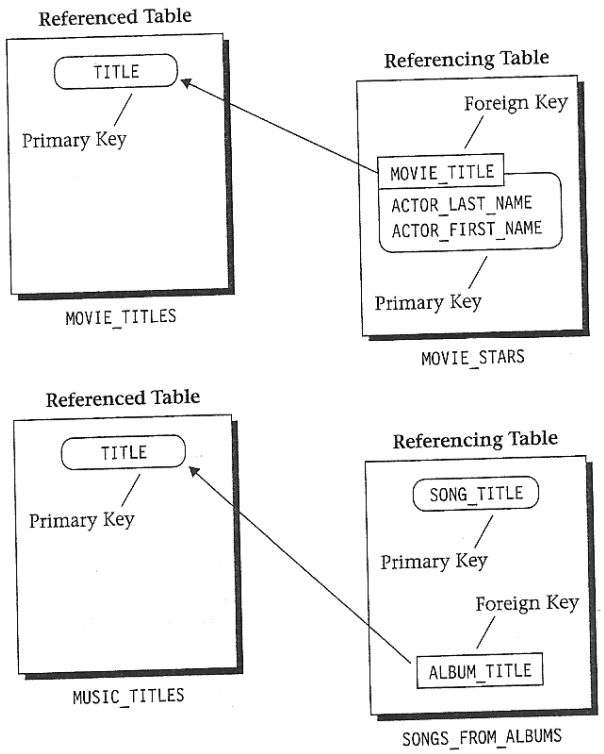

Figure 10-1 Foreign Key Usage in SQL:1999

10.5.1 FOREIGN KEY

A *foreign key* is a column or group of columns within a table that references, or relates to, some other table through its values (see Figure 10-1). The foreign key must always include enough columns in its definition to uniquely identify a row in the referenced table.

The primary reason for defining foreign keys within the data definition language is to ensure that rows in one table always have corresponding rows in another table; that is, to ensure that integrity of data is maintained. Recall, for example, the potential data integrity problem we mentioned at the beginning of

this chapter: several rows exist in the MOVIES_STARS table for the movie *Man on the Moon,* but there is no corresponding row for that movie in the MOVIE_TITLES table. In this situation, there is a problem with lost linkages between data elements of one table and those of another.

To ensure that such a problem does not occur, we can define a foreign key in the MOVIES_STARS table that corresponds to our earlier defined primary key of the MOVIE_TITLES table. The syntax to accomplish this is shown in Example 10-17.

Example 10-17 *FOREIGN KEY Constraint*

```
CREATE TABLE movies_stars (
    |
    |
    |
    CONSTRAINT titles_fk FOREIGN KEY ( movie_title )
      REFERENCES movie_titles (title)
)
```

In the code segment in Example 10-17, the keywords FOREIGN KEY flag the designated column (movie_title) as the foreign key through which the table movie_titles is referenced (through the latter table's primary key column title). Note that you can use both PRIMARY KEY and FOREIGN KEY definitions within any table, as shown in Example 10-18; the primary key of a given table (in our current example, movie_stars) has no bearing on the definition of a foreign key through which another table is referenced.

Example 10-18 *Using Both PRIMARY KEY and FOREIGN KEY Constraints Together*

```
CREATE TABLE movies_stars (
    movie_title        CHARACTER (30) NOT NULL,
    actor_last_name    CHARACTER (35) NOT NULL,
    actor_first_name   CHARACTER (25) NOT NULL,

    CONSTRAINT stars_pk PRIMARY KEY
      ( movie_title, actor_last_name, actor_first_name ),

    CONSTRAINT titles_fk FOREIGN KEY ( movie_title )
      REFERENCES movie_titles ,
    |
    |
    |
)
```


As with primary keys, there are several different syntactical ways in which you can define foreign keys. You could, for example, state the following:

```
CREATE TABLE movies_stars (
    movie_title CHARACTER (30)
      CONSTRAINT title_not_null NOT NULL
      CONSTRAINT titles_fk
        REFERENCES movie_titles (title),
    |
    |
    |
)
```

If your foreign key references the primary key of the referenced table (as opposed to some candidate key that wasn't selected as the primary key; refer to our discussion earlier in this chapter), you can omit the column reference, as shown in Example 10-19.

Example 10-19 *Omitting the Column Reference in REFERENCES*

```
CREATE TABLE movies_stars (
    movie_title CHARACTER (30) NOT NULL
      REFERENCES movie_titles,
    |
    |
    |
)
```

You can also omit the primary key reference for foreign key definitions coded as table constraints (see Example 10-20), as we did in our first foreign key example (Example 10-17).

Example 10-20 *Omitting the Primary Key Reference for a FOREIGN KEY Constraint*

```
CREATE TABLE movies_stars (
    |
    |
    |
    CONSTRAINT titles_fk FOREIGN KEY
      ( movie_title ) REFERENCES movie_titles
)
```

Finally, the constraint name for the foreign key *may* be omitted, as shown in Example 10-21, although as we stated earlier in this chapter, we strongly recommend naming all of your constraints for diagnostic purposes.

Example 10-21 *Omitting the Name of the FOREIGN KEY Constraint*

```
CREATE TABLE movies_stars (
    |
    |
    |
    FOREIGN KEY ( movie_title )
      REFERENCES movie_titles ( title )
)
```

In the above example, `title` could be omitted since the primary key is being referenced rather than a nonprimary candidate key.

One more rule: If your foreign and primary or candidate keys are one-column keys, you can use the earlier "short" form (with the column definition). If they are multicolumn, you must use the later form (separate table constraint).

10.5.2 Referential Constraint Actions

When you specify a `FOREIGN KEY`, you may only wish to prohibit the execution of any SQL statement that might violate the referential integrity constraints, which is what we've demonstrated so far. This was the only referential integrity capability defined by SQL-89. However, as applications have become increasingly complex, it's been desirable to put the burden of data integrity on the database system. The feature that does this is called *referential actions*.

The following sections describe the SQL:1999 capabilities for referential actions.

Setting a Default

Suppose you have a business rule that requires that whenever you drop a distributor, you automatically switch the distributor for those movies to some specific other distributor—that is, to your default distributor. If all you wanted to do was to specify a restriction rule in your database that prohibits dropping a distributor as long as you stock movies distributed by that company, you would write something like the definition shown in Example 10-22.

Example 10-22 *REFERENCES between Movie Titles and Distributors*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
    distributor        CHARACTER VARYING(25)
      REFERENCES distributors,
    |
    |
    |
)
```

However, to tell the database system to automatically set the distributor to the default when you delete the corresonding row in the `distributor` table, you would code something like that shown in Example 10-23.

Example 10-23 *Setting a Default Value*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
    distributor        CHARACTER VARYING(25)
      DEFAULT 'Big East, Inc.'
      REFERENCES distributors
        ON DELETE SET DEFAULT,
    |
    |
    |
)
```

SET NULL

You may want to establish a business rule that if a distributor goes out of business, all of the movies available from that company have the `distributor` column set to null, indicating "no distributor currently available (until we get another one)." You can specify this automatic action as shown in Example 10-24.

Example 10-24 *Setting a NULL Value*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
    distributor        CHARACTER VARYING (25)
      REFERENCES distributors
        ON DELETE SET NULL,
    |
    |
    |
)
```

CASCADE

Suppose you receive prerelease information about a movie entitled *The Certifier*, an epic starring Arnold Schwarzenegger as a postal worker (Kevin Costner has a cameo appearance—but imagine the scene where Arnie "goes postal"!). You enter information about this movie in your database, including tables MOVIE_TITLES and MOVIES_STARS. But, before you actually receive the tapes and begin renting this blockbuster, the title is changed to *The Last Certifier*. The name change can be automatically handled in any tables (such as MOVIES_STARS), through the use of CASCADE, as shown in Example 10-25.

Example 10-25 *Using CASCADE*

```
CREATE TABLE movies_stars (
    |
    |
    |
    CONSTRAINT titles_fk
      FOREIGN KEY ( movie_title )
        REFERENCES movie_titles
          ON UPDATE CASCADE,
)
```

Likewise, referring back to our earlier example with distributors and the movies (Example 10-23), we can ensure that any company name changes among the distributors cascade through to the MOVIE_TITLES table by the syntax shown in Example 10-26.

Example 10-26 *Cascading Name Changes across Multiple Tables*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
    distributor        CHARACTER VARYING(25)
      REFERENCES distributors
        ON UPDATE CASCADE,
    |
    |
    |
)
```

Of course, CASCADE applies equally well to the ON DELETE referential action. Suppose that our business model keeps track of DVDs that are on order from various distributors in a separate table (DVDS_ON_ORDER, perhaps). We'd probably have a business rule that says those rows must not exist if the distributor from whom the DVDs were ordered does not exist. That means that we'd undoubtedly have a referential action to delete rows from DVDS_ON_ORDER whenever the referenced rows in the DISTRIBUTORS table is deleted because the distributor from whom we'd ordered the DVDs went out of business.

NO ACTION

You can set any of the three referential actions (CASCADE, SET NULL, or SET DEFAULT) or either of the two "nonactions" (NO ACTION and RESTRICT) on each of the two activities DELETE and UPDATE as we illustrate in Figure 10-2.

The default for either activity (that is, if you don't specify the referential action syntax at all) is NO ACTION, but you can also code the defaults to say that you want the database system to ensure that you have a valid state at the end of your statement (Example 10-27).

Example 10-27 *NO ACTION*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
```

ACTIVITY	ACTION: CASCADE	SET NULL	SET DEFAULT	RESTRICT	NO ACTION
DELETE	✔	✔	✔	✔	✔
UPDATE	✔	✔	✔	✔	✔

Figure 10-2 Referential Actions in SQL:1999

```
    distributor        CHARACTER VARYING(25)
      REFERENCES distributors
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    |
    |
    |
)
```

NO ACTION means just what it says: if the referential constraint remains unsatisfied at the end of the SQL statement, then no actions are performed. Actually, the phrase was chosen to indicate that no *referential* actions were performed; however, the effect is that the SQL statement making the changes is "undone" and therefore effectively performs "no action" either. Instead, the DBMS will give you a constraint violation error.

Let's consider some examples. Suppose we have our movie_titles with the PRIMARY KEY constraint on title and the movies_stars table with a FOREIGN KEY constraint on its column movie_title that specifies ON UPDATE NO ACTION. If we try to execute

```
UPDATE movie_titles
  SET title = 'One Extremely Strange Movie'
WHERE title = 'Rocky Horror Picture Show'
```

then the presence of rows in movies_stars listing Tim Curry, Susan Sarandon, Barry Bostwick, and others will cause the UPDATE statement to fail. If that FOREIGN KEY had said ON UPDATE CASCADE, then Tim, Susan, Barry, et al. would now be shown as starring in *One Extremely Strange Movie*.

On the other hand, ON UPDATE SET NULL would attempt to have them starring in no movie at all, which would violate the PRIMARY KEY of movies_stars and would thus cause the UPDATE statement to fail, too.

The action (or lack of it) doesn't have to be the same for the two activities. You can choose to SET NULL for DELETE and have CASCADE for UPDATE if you wish, as shown in Example 10-28.

Example 10-28 *Mix-and-Match Actions for ON UPDATE and ON DELETE*

```
CREATE TABLE movie_titles (
    title              CHARACTER (30) NOT NULL,
    |
    |
    |
    distributor        CHARACTER VARYING(25)
      REFERENCES distributors
      ON UPDATE CASCADE
      ON DELETE SET NULL,
    |
    |
    |
)
```

10.5.3 RESTRICT

SQL:1999 provides one more option for referential actions (SQL-92 didn't provide this option). If you use NO ACTION as your (implicit or explicit) referential action specification, then you've instructed the database system to ensure that your referential constraints are satisfied by the time the SQL statement's actions (or, for deferred constraints, the transaction's actions) have finished, including application of any other referential actions and any trigger executions (see Chapter 11, "Active Databases," for information on triggers).

The semantics of referential integrity sometimes allow changes to a table that *temporarily* (during the execution of the statement) causes a referential constraint to be violated, even though the state of the table is correct after the statement's effects are fully applied. Specifying a referential action of NO ACTION allows such temporary violations. However, your application might have a requirement that the state of a table *never* violate a specific referential constraint, even temporarily during the execution of an SQL statement. If you want to enforce this prohibition, you can use the referential action keyword RESTRICT.

10.6 Multiple Cascades

So far, we have dealt with referential integrity between only two tables. There are probably occasions when you would want multiple levels of cascades specified in your referential integrity constraints. Let's take our MOVIE_TITLES and MOVIES_STARS example (the key data definition segments are reproduced below) and add a new table. Assume our Web site has a section through which online shoppers can browse to find out information about various awards won by movies we have in stock; perhaps it's a link to content from the online movie database at *www.imdb.com*. On Academy Awards weekend, we may want to put on sale all movies we have in stock that have won the Best Picture award. We can access our system and produce a quick list of Best Picture winners. To accomplish this, we first add a new table to our other two with awards information, as shown in Example 10-29.

Example 10-29 *A New Table for Movie Awards*

```
CREATE TABLE movie_titles (
    title                    CHARACTER (30) PRIMARY KEY,
    |
    |
    |
)

CREATE TABLE movies_stars (
    movie_title              CHARACTER (30) NOT NULL,
    actor_last_name          CHARACTER (35) NOT NULL,
    actor_first_name         CHARACTER (25) NOT NULL,

    CONSTRAINT stars_pk PRIMARY KEY
      ( movie_title, actor_last_name, actor_first_name ),
    CONSTRAINT titles_fk FOREIGN KEY ( movie_title )
      REFERENCES movie_titles
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    |
    |
    |
)
```



```
CREATE TABLE movie_awards (
    movie_title              CHARACTER (30),
    first_name               CHARACTER (35),
    last_name                CHARACTER (25),
    award                    CHARACTER VARYING(10),

    FOREIGN_KEY (movie_title, first_name, last_name)
      REFERENCES movies_stars
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    |
    |
    | )
```

With a structure like this, when you change the name (or just the spelling of the name) of a movie, that change is cascaded to the movie_stars table, changing the spelling of the movie's name in every row associated with that movie in the other table, too. Those changes will, in turn, be cascaded to the movie_awards table, changing the name in that table. Also, if you change the spelling of a movie star's name, that change will also be reflected in the movie_awards table. Similarly, if you delete a movie (because you no longer carry it in stock), that deletion will cause the movie_stars entries to be deleted along with the movie_awards that those stars won. Figure 10-3 illustrates the three-level cascade.

FOREIGN KEYs can even reference the table in which they're defined. For example, an employees table may have a column emp_id and another column mgr_id. Of course, you'd want all managers to also be employees; you could ensure that by specifying

```
|
|
|
emp_id              INTEGER PRIMARY KEY,
mgr_id              INTEGER REFERENCES employees,
|
|
|
```

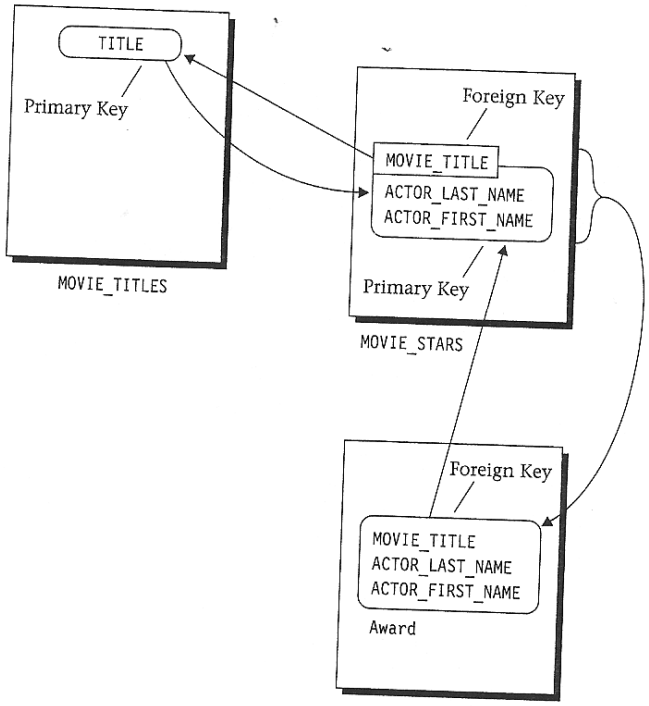

Figure 10-3 Cascading among Multiple SQL:1999 Tables

10.7 More about Referential Integrity Constraints

To better understand how referential integrity works, we have to know a little bit more about how SQL:1999 expects statements to behave. Realize that various DBMS products might implement this in a different way, but the end result will be the same. Think of every SQL statement as having three phases: a *setup* phase, an *execution* phase, and a *cleanup* phase. During the setup phase, the DBMS first identifies every row that will be directly affected by the statement and then identifies every *matching* row for every directly affected row. (A matching row is a row in a referencing table whose foreign key values identify some directly affected row in the referenced table.)

During the execution phase, the DBMS performs the action of the statement on the directly affected rows (if the statement is a DELETE statement, then the row isn't actually deleted but is "marked for deletion"). No new matching rows are ever added to the list of matching rows as a result of any direct actions.

During the cleanup phase, the referential actions are performed. In general, when a directly affected row causes some referential action, that action is performed, and any referential actions caused by *that* action are then performed. Although this may sound like a depth-first tree walk, for those of you who stayed awake during graph theory, referential integrity actions have been very carefully designed to allow them to be performed in any order at all to arrive at a correct, consistent state of the database. (If any of those referential actions involve a row deletion, then those rows are also merely marked for deletion. The reason for this is to avoid certain anomalies that could result from actually deleting the rows before the referential actions have "died down.") At the end of the cleanup phase, any rows that are marked for deletion are actually deleted.

10.8 More about Constraints and Assertions

Let's spend just a little more time on the subject of constraints and assertions to really grasp some of the subtle issues. For example, we told you that constraint checking was done only at the end of statements and not during processing of individual rows, but we didn't tell you why it's that way. (An obvious exception to the "not during processing" policy is when you specify a referential action of RESTRICT; in that case, the referential constraint is actually checked as each row is processed to ensure no temporary violation of the referential constraint occurs.)

Consider a table that has unique ID numbers for each movie (the smarter way of doing the table definition, as you might have guessed by now, similar to what we did in the music_titles example in Chapter 8, "Working with Multiple Tables: The Relational Operators," even though there we used a CHARACTER typed unique identifier):

```
CREATE TABLE movie_titles (
    id       INTEGER constraint movie_titles_pk PRIMARY KEY,
    title    CHARACTER (30),
    |
    |
    |
)
```