```
 1)  CREATE TRIGGER AvgNetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD TABLE AS OldStuff,
 5)      NEW TABLE AS NewStuff
 6)  FOR EACH STATEMENT
 7)  WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
 8)  BEGIN
 9)      DELETE FROM MovieExec
10)      WHERE (name, address, cert#, netWorth) IN NewStuff;
11)      INSERT INTO MovieExec
12)          (SELECT * FROM OldStuff);
13)  END;
```

Figure 7.6: Constraining the average net worth

If the operation is an update, then tables `NewStuff` and `OldStuff` are the new and old versions of the updated tuples, respectively. If an analogous trigger were written for deletions, then the deleted tuples would be in `OldStuff`, and there would be no declaration of a relation name like `NewStuff` for `NEW TABLE` in this trigger. Likewise, in the analogous trigger for insertions, the new tuples would be in `NewStuff`, and there would be no declaration of `OldStuff`.

Line (6) tells us that this trigger is executed once for a statement, regardless of how many tuples are modified. Line (7) is the condition. This condition is satisfied if the average net worth *after* the update is less than $500,000.

The action of lines (8) through (13) consists of two statements that restore the old relation `MovieExec` if the condition of the `WHEN` clause is satisfied; i.e., the new average net worth is too low. Lines (9) and (10) remove all the new tuples, i.e., the updated versions of the tuples, while lines (11) and (12) restore the tuples as they were before the update. □

Example 7.15: An important use of `BEFORE` triggers is to fix up the inserted tuples in some way before they are inserted. Suppose that we want to insert movie tuples into

```
Movies(title, year, length, genre, studioName, producerC#)
```

but sometimes, we will not know the year of the movie. Since `year` is part of the primary key, we cannot have `NULL` for this attribute. However, we could make sure that `year` is not `NULL` with a trigger and replace `NULL` by some suitable value, perhaps one that we compute in a complex way. In Fig. 7.7 is a trigger that takes the simple expedient of replacing `NULL` by 1915 (something that could be handled by a default value, but which will serve as an example).

Line (2) says that the condition and action execute before the insertion event. In the referencing-clause of lines (3) through (5), we define names for

```
1)  CREATE TRIGGER FixYearTrigger
2)  BEFORE INSERT ON Movies
3)  REFERENCING
4)      NEW ROW AS NewRow
5)      NEW TABLE AS NewStuff
6)  FOR EACH ROW
7)  WHEN NewRow.year IS NULL
8)  UPDATE NewStuff SET year = 1915;
```

Figure 7.7: Fixing NULL's in inserted tuples

both the new row being inserted and a table consisting of only that row. Even though the trigger executes once for each inserted tuple [because line (6) declares this trigger to be row-level], the condition of line (7) needs to be able to refer to an attribute of the inserted row, while the action of line (8) needs to refer to a table in order to describe an update. □

7.5.3 Exercises for Section 7.5

Exercise 7.5.1: Write the triggers analogous to Fig. 7.6 for the insertion and deletion events on `MovieExec`.

Exercise 7.5.2: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the "PC" example of Exercise 2.4.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) When updating the price of a PC, check that there is no lower priced PC with the same speed.

b) When inserting a new printer, check that the model number exists in `Product`.

! c) When making any modification to the `Laptop` relation, check that the average price of laptops for each manufacturer is at least $1500.

! d) When updating the RAM or hard disk of any PC, check that the updated PC has at least 100 times as much hard disk as RAM.

! e) When inserting a new PC, laptop, or printer, make sure that the model number did not previously appear in any of `PC`, `Laptop`, or `Printer`.

Exercise 7.5.3: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the battleships example of Exercise 2.4.3.

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) When a new class is inserted into `Classes`, also insert a ship with the name of that class and a `NULL` launch date.

b) When a new class is inserted with a displacement greater than 35,000 tons, allow the insertion, but change the displacement to 35,000.

! c) If a tuple is inserted into `Outcomes`, check that the ship and battle are listed in `Ships` and `Battles`, respectively, and if not, insert tuples into one or both of these relations, with `NULL` components where necessary.

! d) When there is an insertion into `Ships` or an update of the `class` attribute of `Ships`, check that no country has more than 20 ships.

!! e) Check, under all circumstances that could cause a violation, that no ship fought in a battle that was at a later date than another battle in which that ship was sunk.

! **Exercise 7.5.4:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with `NULL` or default values, rather than rejecting the attempted modification.

a) Assure that at all times, any star appearing in `StarsIn` also appears in `MovieStar`.

b) Assure that at all times every movie executive appears as either a studio president, a producer of a movie, or both.

c) Assure that every movie has at least one male and one female star.

d) Assure that the number of movies made by any studio in any year is no more than 100.

e) Assure that the average length of all movies made in any year is no more than 120.

7.6 Summary of Chapter 7

- ✦ *Referential-Integrity Constraints*: We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attribute(s) of some tuple of the same or another relation. To do so, we use a `REFERENCES` or `FOREIGN KEY` declaration in the relation schema.
- ✦ *Attribute-Based Check Constraints*: We can place a constraint on the value of an attribute by adding the keyword `CHECK` and the condition to be checked after the declaration of that attribute in its relation schema.
- ✦ *Tuple-Based Check Constraints*: We can place a constraint on the tuples of a relation by adding the keyword `CHECK` and the condition to be checked to the declaration of the relation itself.
- ✦ *Modifying Constraints*: A tuple-based check can be added or deleted with an `ALTER` statement for the appropriate table.
- ✦ *Assertions*: We can declare an assertion as an element of a database schema. The declaration gives a condition to be checked. This condition may involve one or more relations of the database schema, and may involve the relation as a whole, e.g., with aggregation, as well as conditions about individual tuples.
- ✦ *Invoking the Checks*: Assertions are checked whenever there is a change to one of the relations involved. Attribute- and tuple-based checks are only checked when the attribute or relation to which they apply changes by insertion or update. Thus, the latter constraints can be violated if they have subqueries.
- ✦ *Triggers*: The SQL standard includes triggers that specify certain events (e.g., insertion, deletion, or update to a particular relation) that awaken them. Once awakened, a condition can be checked, and if true, a specified sequence of actions (SQL statements such as queries and database modifications) will be executed.

7.7 References for Chapter 7

References [5] and [4] survey all aspects of active elements in database systems. [1] discusses recent thinking regarding active elements in SQL-99 and future

standards. References [2] and [3] discuss HiPAC, an early prototype system that offered active database elements.

1. R. J. Cochrane, H. Pirahesh, and N. Mattos, "Integrating triggers and declarative constraints in SQL database systems," *Intl. Conf. on Very Large Databases*, pp. 567–579, 1996.

2. U. Dayal et al., "The HiPAC project: combining active databases and timing constraints," *SIGMOD Record* **17**:1, pp. 51–70, 1988.

3. D. R. McCarthy and U. Dayal, "The architecture of an active database management system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.

4. N. W. Paton and O. Diaz, "Active database systems," *Computing Surveys* **31**:1 (March, 1999), pp. 63–103.

5. J. Widom and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

Chapter 8

Views and Indexes

We begin this chapter by introducing virtual views, which are relations that are defined by a query over other relations. Virtual views are not stored in the database, but can be queried as if they existed. The query processor will replace the view by its definition in order to execute the query.

Views can also be materialized, in the sense that they are constructed periodically from the database and stored there. The existence of these materialized views can speed up the execution of queries. A very important specialized type of "materialized view" is the index, a stored data structure whose sole purpose is to speed up the access to specified tuples of one of the stored relations. We introduce indexes here and consider the principal issues in selecting the right indexes for a stored table.

8.1 Virtual Views

Relations that are defined with a `CREATE TABLE` statement actually exist in the database. That is, a SQL system stores tables in some physical organization. They are persistent, in the sense that they can be expected to exist indefinitely and not to change unless they are explicitly told to change by a SQL modification statement.

There is another class of SQL relations, called (*virtual*) *views*, that do not exist physically. Rather, they are defined by an expression much like a query. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify views.

8.1.1 Declaring Views

The simplest form of view definition is:

`CREATE VIEW` <view-name> `AS` <view-definition>;

The view definition is a SQL query.

Relations, Tables, and Views

SQL programmers tend to use the term "table" instead of "relation." The reason is that it is important to make a distinction between stored relations, which are "tables," and virtual relations, which are "views." Now that we know the distinction between a table and a view, we shall use "relation" only where either a table or view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term "base relation" or "base table."

There is also a third kind of relation, one that is neither a view nor stored permanently. These relations are temporary results, as might be constructed for some subquery. Temporaries will also be referred to as "relations" subsequently.

Example 8.1: Suppose we want to have a view that is a part of the

```
Movies(title, year, length, genre, studioName, producerC#)
```

relation, specifically, the titles and years of the movies made by Paramount Studios. We can define this view by

```
1)  CREATE VIEW ParamountMovies AS
2)      SELECT title, year
3)      FROM Movies
4)      WHERE studioName = 'Paramount';
```

First, the name of the view is `ParamountMovies`, as we see from line (1). The attributes of the view are those listed in line (2), namely `title` and `year`. The definition of the view is the query of lines (2) through (4). □

Example 8.2: Let us consider a more complicated query used to define a view. Our goal is a relation `MovieProd` with movie titles and the names of their producers. The query defining the view involves two relations:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

The following view definition

```
CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

joins the two relations and requires that the certificate numbers match. It then extracts the movie title and producer name from pairs of tuples that agree on the certificates. □

8.1.2 Querying Views

A view may be queried exactly as if it were a stored table. We mention its name in a FROM clause and rely on the DBMS to produce the needed tuples by operating on the relations used to define the virtual view.

Example 8.3: We may query the view ParamountMovies just as if it were a stored table, for instance:

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

finds the movies made by Paramount in 1979. □

Example 8.4: It is also possible to write queries involving both views and base tables. An example is:

```
SELECT DISTINCT starName
FROM ParamountMovies, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

This query asks for the name of all stars of movies made by Paramount. □

The simplest way to interpret what a query involving virtual views means is to replace each view in a FROM clause by a subquery that is identical to the view definition. That subquery is followed by a tuple variable, so we can refer to its tuples. For instance, the query of Example 8.4 can be thought of as the query of Fig. 8.1.

```
SELECT DISTINCT starName
FROM (SELECT title, year
      FROM Movies
      WHERE studioName = 'Paramount'
     ) Pm, StarsIn
WHERE Pm.title = movieTitle AND Pm.year = movieYear;
```

Figure 8.1: Interpreting the use of a virtual view as a subquery

8.1.3 Renaming Attributes

Sometimes, we might prefer to give a view's attributes names of our own choosing, rather than use the names that come out of the query defining the view. We may specify the attributes of the view by listing them, surrounded by parentheses, after the name of the view in the CREATE VIEW statement. For instance, we could rewrite the view definition of Example 8.2 as:

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

The view is the same, but its columns are headed by attributes `movieTitle` and `prodName` instead of `title` and `name`.

8.1.4 Exercises for Section 8.1

Exercise 8.1.1: From the following base tables of our running example

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Construct the following views:

a) A view `RichExec` giving the name, address, certificate number and net worth of all executives with a net worth of at least $10,000,000.

b) A view `StudioPres` giving the name, address, and certificate number of all executives who are studio presidents.

c) A view `ExecutiveStar` giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

Exercise 8.1.2: Write each of the queries below, using one or more of the views from Exercise 8.1.1 and no base tables.

a) Find the names of females who are both stars and executives.

b) Find the names of those executives who are both studio presidents and worth at least $10,000,000.

! c) Find the names of studio presidents who are also stars and are worth at least $50,000,000.

8.2 Modifying Views

In limited circumstances it is possible to execute an insertion, deletion, or update to a view. At first, this idea makes no sense at all, since the view does not exist the way a base table (stored relation) does. What could it mean, say, to insert a new tuple into a view? Where would the tuple go, and how would the database system remember that it was supposed to be in the view?

For many views, the answer is simply "you can't do that." However, for sufficiently simple views, called *updatable views*, it is possible to translate the

modification of the view into an equivalent modification on a base table, and the modification can be done to the base table instead. In addition, "instead-of" triggers can be used to turn a view modification into modifications of base tables. In that way, the programmer can force whatever interpretation of a view modification is desired.

8.2.1 View Removal

An extreme modification of a view is to delete it altogether. This modification may be done whether or not the view is updatable. A typical DROP statement is

```
DROP VIEW ParamountMovies;
```

Note that this statement deletes the definition of the view, so we may no longer make queries or issue modification commands involving this view. However dropping the view does not affect any tuples of the underlying relation `Movies`. In contrast,

```
DROP TABLE Movies
```

would not only make the `Movies` table go away. It would also make the view `ParamountMovies` unusable, since a query that used it would indirectly refer to the nonexistent relation `Movies`.

8.2.2 Updatable Views

SQL provides a formal definition of when modifications to a view are permitted. The SQL rules are complex, but roughly, they permit modifications on views that are defined by selecting (using `SELECT`, not `SELECT DISTINCT`) some attributes from one relation R (which may itself be an updatable view). Two important technical points:

- The `WHERE` clause must not involve R in a subquery.
- The `FROM` clause can only consist of one occurrence of R and no other relation.
- The list in the `SELECT` clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with `NULL` values or the proper default. For example, it is not permitted to project out an attribute that is declared `NOT NULL` and has no default.

An insertion on the view can be applied directly to the underlying relation R. The only nuance is that we need to specify that the attributes in the `SELECT` clause of the view are the only ones for which values are supplied.

Example 8.5: Suppose we insert into view ParamountMovies of Example 8.1 a tuple like:

```
INSERT INTO ParamountMovies
VALUES('Star Trek', 1979);
```

View ParamountMovies meets the SQL updatability conditions, since the view asks only for some components of some tuples of one base table:

```
Movies(title, year, length, genre, studioName, producerC#)
```

The insertion on ParamountMovies is executed as if it were the same insertion on Movies:

```
INSERT INTO Movies(title, year)
VALUES('Star Trek', 1979);
```

Notice that the attributes title and year had to be specified in this insertion, since we cannot provide values for other attributes of Movies.

The tuple inserted into Movies has values 'Star Trek' for title, 1979 for year, and NULL for the other four attributes. Curiously, the inserted tuple, since it has NULL as the value of attribute studioName, will not meet the selection condition for the view ParamountMovies, and thus, the inserted tuple has no effect on the view. For instance, the query of Example 8.3 would not retrieve the tuple ('Star Trek', 1979).

To fix this apparent anomaly, we could add studioName to the SELECT clause of the view, as:

```
CREATE VIEW ParamountMovies AS
   SELECT studioName, title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

Then, we could insert the *Star-Trek* tuple into the view by:

```
INSERT INTO ParamountMovies
VALUES('Paramount', 'Star Trek', 1979);
```

This insertion has the same effect on Movies as:

```
INSERT INTO Movies(studioName, title, year)
VALUES('Paramount', 'Star Trek', 1979);
```

Notice that the resulting tuple, although it has NULL in the attributes not mentioned, does yield the appropriate tuple for the view ParamountMovies. □

We may also delete from an updatable view. The deletion, like the insertion, is passed through to the underlying relation R. However, to make sure that only tuples that can be seen in the view are deleted, we add (using AND) the condition of the WHERE clause in the view to the WHERE clause of the deletion.

Example 8.6: Suppose we wish to delete from the updatable ParamountMovies view all movies with "Trek" in their titles. We may issue the deletion statement

```
DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the Movies base table; the only difference is that the condition defining the view ParamountMovies is added to the conditions of the WHERE clause.

```
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

is the resulting delete statement. □

Similarly, an update on an updatable view is passed through to the underlying relation. The view update thus has the effect of updating all tuples of the underlying relation that give rise in the view to updated view tuples.

Example 8.7: The view update

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

is equivalent to the base-table update

```
UPDATE Movies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
    studioName = 'Paramount';
```

□

8.2.3 Instead-Of Triggers on Views

When a trigger is defined on a view, we can use INSTEAD OF in place of BEFORE or AFTER. If we do so, then when an event awakens the trigger, the action of the trigger is done instead of the event itself. That is, an instead-of trigger intercepts attempts to modify the view and in its place performs whatever action the database designer deems appropriate. The following is a typical example.

Why Some Views Are Not Updatable

Consider the view MovieProd of Example 8.2, which relates movie titles and producers' names. This view is not updatable according to the SQL definition, because there are two relations in the FROM clause: Movies and MovieExec. Suppose we tried to insert a tuple like

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

We would have to insert tuples into both Movies and MovieExec. We could use the default value for attributes like length or address, but what could be done for the two equated attributes producerC# and cert# that both represent the unknown certificate number of DeMille? We could use NULL for both of these. However, when joining relations with NULL's, SQL does not recognize two NULL values as equal (see Section 6.1.6). Thus, 'Greatest Show on Earth' would not be connected with 'Cecil B. DeMille' in the MovieProd view, and our insertion would not have been done correctly.

Example 8.8: Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovies AS
   SELECT title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

from Example 8.1. As we discussed in Example 8.5, this view is updatable, but it has the unexpected flaw that when you insert a tuple into ParamountMovies, the system cannot deduce that the studioName attribute is surely Paramount, so studioName is NULL in the inserted Movies tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 8.2. Much of the trigger is unsurprising. We see the keyword INSTEAD OF on line (2), establishing that an attempt to insert into ParamountMovies will never take place.

Rather, lines (5) and (6) is the action that replaces the attempted insertion. There is an insertion into Movies, and it specifies the three attributes that we know about. Attributes title and year come from the tuple we tried to insert into the view; we refer to these values by the tuple variable NewRow that was declared in line (3) to represent the tuple we are trying to insert. The value of attribute studioName is the constant 'Paramount'. This value is not part of the inserted view tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view ParamountMovies. □

```
1) CREATE TRIGGER ParamountInsert
2) INSTEAD OF INSERT ON ParamountMovies
3) REFERENCING NEW ROW AS NewRow
4) FOR EACH ROW
5) INSERT INTO Movies(title, year, studioName)
6) VALUES(NewRow.title, NewRow.year, 'Paramount');
```

Figure 8.2: Trigger to replace an insertion on a view by an insertion on the underlying base table

8.2.4 Exercises for Section 8.2

Exercise 8.2.1: Which of the views of Exercise 8.1.1 are updatable?

Exercise 8.2.2: Suppose we create the view:

```
CREATE VIEW DisneyComedies AS
    SELECT title, year, length FROM Movies
    WHERE studioName = 'Disney' AND genre = 'comedy';
```

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the length for a movie (given by title and year) in this view.

Exercise 8.2.3: Using the base tables

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

suppose we create the view:

```
CREATE VIEW NewPC AS
SELECT maker, model, speed, ram, hd, price
FROM Product, PC
WHERE Product.model = PC.model AND type = 'pc';
```

Notice that we have made a check for consistency: that the model number not only appears in the `PC` relation, but the `type` attribute of `Product` indicates that the product is a PC.

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the price.

d) Write an instead-of trigger to handle a deletion of a specified tuple from this view.

8.3 Indexes in SQL

An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A. We could think of the index as a binary search tree of (key, value) pairs, in which a key a (one of the values that attribute A may have) is associated with a "value" that is the set of locations of the tuples that have a in the component for attribute A. Such an index may help with queries in which the attribute A is compared with a constant, for instance $A = 3$, or even $A \leq 3$. Note that the key for the index can be any attribute or set of attributes, and need not be the key for the relation on which the index is built. We shall refer to the attributes of the index as the *index key* when a distinction needs to be made.

The technology of implementing indexes on large relations is of central importance in the implementation of DBMS's. The most important data structure used by a typical DBMS is the "B-tree," which is a generalization of a balanced binary tree. We shall take up B-trees when we talk about DBMS implementation, but for the moment, thinking of indexes as binary search trees will suffice.

8.3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. For example, consider the first query we examined:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

from Example 6.1. There might be 10,000 `Movies` tuples, of which only 200 were made in 1990.

The naive way to implement this query is to get all 10,000 tuples and test the condition of the `WHERE` clause on each. It would be much more efficient if we had some way of getting only the 200 tuples from the year 1990 and testing each of them to see if the studio was Disney. It would be even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the `WHERE` clause — that the studio is Disney and the year is 1990; see the discussion of "multiattribute indexes," in Section 8.3.2.

Indexes may also be useful in queries that involve a join. The following example illustrates the point.

Example 8.9: Recall the query

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

from Example 6.12 that asks for the name of the producer of *Star Wars*. If there is an index on `title` of `Movies`, then we can use this index to get the tuple for *Star Wars*. From this tuple, we can extract the `producerC#` to get the certificate of the producer.

Now, suppose that there is also an index on `cert#` of `MovieExec`. Then we can use the `producerC#` with this index to find the tuple of `MovieExec` for the producer of *Star Wars*. From this tuple, we can extract the producer's name. Notice that with these two indexes, we look at only the two tuples, one from each relation, that are needed to answer the query. Without indexes, we have to look at every tuple of the two relations. □

8.3.2 Declaring Indexes

Although the creation of indexes is not part of any SQL standard up to and including SQL-99, most commercial systems have a way for the database designer to say that the system should create an index on a certain attribute for a certain relation. The following syntax is typical. Suppose we want to have an index on attribute `year` for the relation `Movies`. Then we say:

```
CREATE INDEX YearIndex ON Movies(year);
```

The result will be that an index whose name is `YearIndex` will be created on attribute `year` of the relation `Movies`. Henceforth, SQL queries that specify a year may be executed by the SQL query processor in such a way that only those tuples of `Movies` with the specified year are ever examined; there is a resulting decrease in the time needed to answer the query.

Often, a DBMS allows us to build a single index on multiple attributes. This type of index takes values for several attributes and efficiently finds the tuples with the given values for these attributes.

Example 8.10: Since `title` and `year` form a key for `Movies`, we might expect it to be common that values for both these attributes will be specified, or neither will. The following is a typical declaration of an index on these two attributes:

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

Since (`title`, `year`) is a key, if follows that when we are given a title and year, we know the index will find only one tuple, and that will be the desired tuple. In contrast, if the query specifies both the title and year, but only `YearIndex` is available, then the best the system can do is retrieve all the movies of that year and check through them for the given title.

If, as is often the case, the key for the multiattribute index is really the concatenation of the attributes in some order, then we can even use this index to find all the tuples with a given value in the first of the attributes. Thus, part of the design of a multiattribute index is the choice of the order in which the attributes are listed. For instance, if we were more likely to specify a title

than a year for a movie, then we would prefer to order the attributes as above; if a year were more likely to be specified, then we would ask for an index on `(year, title)`. □

If we wish to delete the index, we simply use its name in a statement like:

```
DROP INDEX YearIndex;
```

8.3.3 Exercises for Section 8.3

Exercise 8.3.1: For our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

a) `studioName`.

b) `address` of `MovieExec`.

c) `genre` and `length`.

8.4 Selection of Indexes

Choosing which indexes to create requires the database designer to analyze a trade-off. In practice, this choice is one of the principal factors that influence whether a database design gives acceptable performance. Two important factors to consider are:

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

8.4.1 A Simple Cost Model

To understand how to choose indexes for a database, we first need to know where the time is spent answering a query. The details of how relations are stored will be taken up when we consider DBMS implementation. But for the moment, let us state that the tuples of a relation are normally distributed

among many pages of a disk.[1] One page, which is typically several thousand bytes at least, will hold many tuples.

To examine even one tuple requires that the whole page be brought into main memory. On the other hand, it costs little more time to examine all the tuples on a page than to examine only one. There is a great time saving if the page you want is already in main memory, but for simplicity we shall assume that never to be the case, and every page we need must be retrieved from the disk.

8.4.2 Some Useful Indexes

Often, the most useful index we can put on a relation is an index on its key. There are two reasons:

1. Queries in which a value for the key is specified are common. Thus, an index on the key will get used frequently.

2. Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple. Thus, at most one page must be retrieved to get that tuple into main memory (although there may be other pages that need to be retrieved to use the index itself).

The following example shows the power of key indexes, even in a query that involves a join.

Example 8.11: Recall Figure 6.3, where we suggested an exhaustive pairing of tuples of `Movies` and `MovieExec` to compute a join. Implementing the join this way requires us to read each of the pages holding tuples of `Movies` and each of the pages holding tuples of `MovieExec` at least once. In fact, since these pages may be too numerous to fit in main memory at the same time, we may have to read each page from disk many times. With the right indexes, the whole query might be done with as few as two page reads.

An index on the key `title` and `year` for `Movies` would help us find the one `Movies` tuple for *Star Wars* quickly. Only one page — the page containing that tuple — would be read from disk. Then, after finding the producer-certificate number in that tuple, an index on the key `cert#` for `MovieExec` would help us quickly find the one tuple for the producer in the `MovieExec` relation. Again, only one page with `MovieExec` tuples would be read from disk, although we might need to read a small number of other pages to use the `cert#` index. □

When the index is not on a key, it may or may not be able to improve the time spent retrieving from disk the tuples needed to answer a query. There are two situations in which an index can be effective, even if it is not on a key.

[1] Pages are usually referred to as "blocks" in discussion of databases, but if you are familiar with a paged-memory system from operating systems you should think of the disk as divided into pages.

1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute. Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from disk.

2. If the tuples are "clustered" on that attribute. We *cluster* a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example 8.12: As an example of an index of the first kind, suppose `Movies` had an index on `title` rather than `title` and `year`. Since `title` by itself is not a key for the relation, there would be titles such as *King Kong*, where several tuples matched the index key `title`. If we compared use of the index on `title` with what happens in Example 8.11, we would find that a search for movies with title *King Kong* would produce three tuples (because there are three movies with that title, from years 1933, 1976, and 2005). It is possible that these tuples are on three different pages, so all three pages would be brought into main memory, roughly tripling the amount of time this step takes. However, since the relation `Movies` probably is spread over many more than three pages, there is still a considerable time saving in using the index.

At the next step, we need to get the three `producerC#` values from these three tuples, and find in the relation `MovieExec` the producers of these three movies. We can use the index on `cert#` to find the three relevant tuples of `MovieExec`. Possibly they are on three different pages, but we still spend less time than we would if we had to bring the entire `MovieExec` relation into main memory. □

Example 8.13: Now, suppose the only index we have on `Movies` is one on `year`, and we want to answer the query:

```
SELECT *
FROM Movies
WHERE year = 1990;
```

First, suppose the tuples of `Movies` are not clustered by year; say they are stored alphabetically by title. Then this query gains little from the index on `year`. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990. Thus, a large fraction of the pages used to hold the relation `Movies` will have to be brought to main memory.

However, suppose the tuples of `Movies` are clustered on `year`. Then we could use the index on `year` to find only the small number of pages that contained tuples with `year` = 1990. In this case, the `year` index will be of great help. In comparison, an index on the combination of `title` and `year` would be of little help, no matter what attribute or attributes we used to cluster `Movies`. □

8.4.3 Calculating the Best Indexes to Create

It might seem that the more indexes we create, the more likely it is that an index useful for a given query will be available. However, if modifications are the most frequent action, then we should be very conservative about creating indexes. Each modification on a relation R forces us to change any index on one or more of the modified attributes of R. Thus, we must read and write not only the pages of R that are modified, but also read and write certain pages that hold the index. But even when modifications are the dominant form of database action, it may be an efficiency gain to create an index on a frequently used attribute. In fact, since some modification commands involve querying the database (e.g., an `INSERT` with a select-from-where subquery or a `DELETE` with a condition) one must be very careful how one estimates the relative frequency of modifications and queries.

Remember that the typical relation is stored over many disk blocks (pages), and the principal cost of a query or modification is often the number of pages that need to be brought to main memory. Thus, indexes that let us find a tuple without examining the entire relation can save a lot of time. However, the indexes themselves have to be stored, at least partially, on disk, so accessing and modifying the indexes themselves cost disk accesses. In fact, modification, since it requires one disk access to read a page and another disk access to write the changed page, is about twice as expensive as accessing the index or the data in a query.

To calculate the new value of an index, we need to make assumptions about which queries and modifications are most likely to be performed on the database. Sometimes, we have a history of queries that we can use to get good information, on the assumption that the future will be like the past. In other cases, we may know that the database supports a particular application or applications, and we can see in the code for those applications all the SQL queries and modifications that they will ever do. In either situation, we are able to list what we expect are the most common query and modification forms. These forms can have variables in place of constants, but should otherwise look like real SQL statements. Here is a simple example of the process, and of the calculations that we need to make.

Example 8.14: Let us consider the relation

```
StarsIn(movieTitle, movieYear, starName)
```

Suppose that there are three database operations that we sometimes perform on this relation:

Q_1: We look for the title and year of movies in which a given star appeared. That is, we execute a query of the form:

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

for some constant s.

Q_2: We look for the stars that appeared in a given movie. That is, we execute a query of the form:

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear = y;
```

for constants t and y.

I: We insert a new tuple into `StarsIn`. That is, we execute an insertion of the form:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

for constants t, y, and s.

Let us make the following assumptions about the data:

1. `StarsIn` occupies 10 pages, so if we need to examine the entire relation the cost is 10.
2. On the average, a star has appeared in 3 movies and a movie has 3 stars.
3. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of `StarsIn`, even if we have an index on `starName` or on the combination of `movieTitle` and `movieYear`, it will take 3 disk accesses to find the (average of) 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.
4. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.
5. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Figure 8.3 gives the costs of each of the three operations; Q_1 (query given a star), Q_2 (query given a movie), and I (insertion). If there is no index, then we must scan the entire relation for Q_1 or Q_2 (cost 10),[2] while an insertion requires

[2]There is a subtle point that we shall ignore here. In many situations, it is possible to store a relation on disk using consecutive pages or tracks. In that case, the cost of retrieving the entire relation may be significantly less than retrieving the same number of pages chosen randomly.

Action	No Index	Star Index	Movie Index	Both Indexes
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Figure 8.3: Costs associated with the three actions, as a function of which indexes are selected

merely that we access a page with free space and rewrite it with the new tuple (cost of 2, since we assume that page can be found without an index). These observations explain the column labeled "No Index."

If there is an index on stars only, then Q_2 still requires a scan of the entire relation (cost 10). However, Q_1 can be answered by accessing one index page to find the three tuples for a given star and then making three more accesses to find those tuples. Insertion I requires that we read and write both a page for the index and a page for the data, for a total of 4 disk accesses.

The case where there is an index on movies only is symmetric to the case for stars only. Finally, if there are indexes on both stars and movies, then it takes 4 disk accesses to answer either Q_1 or Q_2. However, insertion I requires that we read and write two index pages as well as a data page, for a total of 6 disk accesses. That observation explains the last column in Fig. 8.3.

The final row in Fig. 8.3 gives the average cost of an action, on the assumption that the fraction of the time we do Q_1 is p_1 and the fraction of the time we do Q_2 is p_2; therefore, the fraction of the time we do I is $1 - p_1 - p_2$.

Depending on p_1 and p_2, any of the four choices of index/no index can yield the best average cost for the three actions. For example, if $p_1 = p_2 = 0.1$, then the expression $2 + 8p_1 + 8p_2$ is the smallest, so we would prefer not to create any indexes. That is, if we are doing mostly insertion, and very few queries, then we don't want an index. On the other hand, if $p_1 = p_2 = 0.4$, then the formula $6 - 2p_1 - 2p_2$ turns out to be the smallest, so we would prefer indexes on both `starName` and on the (`movieTitle, movieYear`) combination. Intuitively, if we are doing a lot of queries, and the number of queries specifying movies and stars are roughly equally frequent, then both indexes are desired.

If we have $p_1 = 0.5$ and $p_2 = 0.1$, then an index on stars only gives the best average value, because $4 + 6p_2$ is the formula with the smallest value. Likewise, $p_1 = 0.1$ and $p_2 = 0.5$ tells us to create an index on only movies. The intuition is that if only one type of query is frequent, create only the index that helps that type of query. □

8.4.4 Automatic Selection of Indexes to Create

"Tuning" a database is a process that includes not only index selection, but the choice of many different parameters. We have not yet discussed much about

physical implementation of databases, but some examples of tuning issues are the amount of main memory to allocate to various processes and the rate at which backups and checkpoints are made (to facilitate recovery from a crash). There are a number of tools that have been designed to take the responsibility from the database designer and have the system tune itself, or at least advise the designer on good choices.

We shall mention some of these projects in the bibliographic notes for this chapter. However, here is an outline of how the index-selection portion of tuning advisors work.

1. The first step is to establish the query workload. Since a DBMS normally logs all operations anyway, we may be able to examine the log and find a set of representative queries and database modifications for the database at hand. Or it is possible that we know, from the application programs that use the database, what the typical queries will be.

2. The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen.

3. The tuning advisor generates a set of possible *candidate* indexes, and evaluates each one. Typical queries are given to the query optimizer of the DBMS. The query optimizer has the ability to estimate the running times of these queries under the assumption that one particular set of indexes is available.

4. The index set resulting in the lowest cost for the given workload is suggested to the designer, or it is automatically created.

A subtle issue arises when we consider possible indexes in step (3). The existence of previously chosen indexes may influence how much *benefit* (improvement in average execution time of the query mix) another index offers. A "greedy" approach to choosing indexes has proven effective.

a) Initially, with no indexes selected, evaluate the benefit of each of the candidate indexes. If at least one provides positive benefit (i.e., it reduces the average execution time of queries), then choose that index.

b) Then, reevaluate the benefit of each of the remaining candidate indexes, assuming that the previously selected index is also available. Again, choose the index that provides the greatest benefit, assuming that benefit is positive.

c) In general, repeat the evaluation of candidate indexes under the assumption that all previously selected indexes are available. Pick the index with maximum benefit, until no more positive benefits can be obtained.

8.4.5 Exercises for Section 8.4

Exercise 8.4.1: Suppose that the relation `StarsIn` discussed in Example 8.14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of p_1 and p_2 to measure the cost of queries Q_1 and Q_2 and insertion I, under the four combinations of index/no index discussed there.

! **Exercise 8.4.2:** In this problem, we consider indexes for the relation

```
Ships(name, class, launched)
```

from our running battleships exercise. Assume:

i. `name` is the key.

ii. The relation `Ships` is stored over 50 pages.

iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.

iv. On average, there are 5 ships of a class, and 25 ships launched in any given year.

v. With probability p_1 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE name =` n.

vi. With probability p_2 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE class =` c.

vii. With probability p_3 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE launched =` y.

viii. With probability $1 - p_1 - p_2 - p_3$ the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 8.14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of p_1, p_2, and p_3, what is the best choice of indexes?

8.5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. Until now, we have thought of views only as logical descriptions of relations. However, if a view is used frequently enough, it may even be efficient to *materialize* it; that is, to maintain its value at all times. As with maintaining indexes, there is a cost involved in maintaining a materialized view, since we must recompute parts of the materialized view each time one of the underlying base tables changes.

8.5.1 Maintaining a Materialized View

In principle, the DBMS needs to recompute a materialized view every time one of its base tables changes in any way. For simple views, it is possible to limit the number of times we need to consider changing the materialized view, and it is possible to limit the amount of work we do when we must maintain the view. We shall take up an example of a join view, and see that there are a number of opportunities to simplify our work.

Example 8.15: Suppose we frequently want to find the name of the producer of a given movie. We might find it advantageous to materialize a view:

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#
```

To start, the DBMS does not have to consider the effect on `MovieProd` of an update on any attribute of `Movies` or `MovieExec` that is not mentioned in the query that defines the materialized view. Surely any modification to a relation that is neither `Movies` nor `MovieExec` can be ignored as well. However, there are a number of other simplifications that enable us to handle other modifications to `Movies` or `MovieExec` more efficiently than a re-execution of the query that defines the materialized view.

1. Suppose we insert a new movie into `Movies`, say `title = 'Kill Bill'`, `year` = 2003, and `producerC#` = 23456. Then we only need to look up `cert#` = 23456 in `MovieExec`. Since `cert#` is the key for `MovieExec`, there can be at most one name returned by the query

   ```
   SELECT name FROM MovieExec
   WHERE cert# = 23456;
   ```

   As this query returns `name = 'Quentin Tarantino'`, the DBMS can insert the proper tuple into `MovieProd` by:

   ```
   INSERT INTO MovieProd
   VALUES('Kill Bill', 2003, 'Quentin Tarantino');
   ```

   Note that, since `MovieProd` is materialized, it is stored like any base table, and this operation makes sense; it does not have to be reinterpreted by an instead-of trigger or any other mechanism.

2. Suppose we delete a movie from `Movies`, say the movie with `title = 'Dumb & Dumber'` and `year` = 1994. The DBMS has only to delete this one movie from `MovieProd` by:

```
DELETE FROM MovieProd
WHERE title = 'Dumb & Dumber' AND year = 1994;
```

3. Suppose we insert a tuple into `MovieExec`, and that tuple has `cert#` = 34567 and `name = 'Max Bialystock'`. Then the DBMS may have to insert into `MovieProd` some movies that were not there because their producer was previously unknown. The operation is:

```
INSERT INTO MovieProd
    SELECT title, year, 'Max Bialystock'
    FROM Movies
    WHERE producerC# = 34567;
```

4. Suppose we delete the tuple with `cert#` = 45678 from `MovieExec`. Then the DBMS must delete from `MovieProd` all movies that have `producerC#` = 45678, because there now can be no matching tuple in `MovieExec` for their underlying `Movies` tuple. Thus, the DBMS executes:

```
DELETE FROM MovieProd
WHERE (title, year) IN
    (SELECT title, year FROM Movies
     WHERE producerC# = 45678);
```

   Notice that it is not sufficient to look up the `name` corresponding to 45678 in `MovieExec` and delete all movies from `MovieProd` that have that producer name. The reason is that, because `name` is not a key for `MovieExec`, there could be two producers with the same name.

We leave as an exercise the consideration of how updates to `Movies` that involve `title` or `year` are handled, and how updates to `MovieExec` involving `cert#` are handled. □

The most important thing to take away from Example 8.15 is that all the changes to the materialized view are *incremental*. That is, we never have to reconstruct the whole view from scratch. Rather, insertions, deletions, and updates to a base table can be implemented in a join view such as `MovieProd` by a small number of queries to the base tables followed by modification statements on the materialized view. Moreover, these modifications do not affect all the tuples of the view, but only those that have at least one attribute with a particular constant.

It is not possible to find rules such as those in Example 8.15 for any materialized view we could construct; some are just too complicated. However, many common types of materialized view *do* allow the view to be maintained incrementally. We shall explore another common type of materialized view — aggregation views — in the exercises.

8.5.2 Periodic Maintenance of Materialized Views

There is another setting in which we may use materialized views, yet not have to worry about the cost or complexity of maintaining them up-to-date as the underlying base tables change. We shall encounter the option when we study OLAP in Section 10.6, but for the moment let us remark that it is common for databases to serve two purposes. For example, a department store may use its database to record its current inventory; this data changes with every sale. The same database may be used by analysts to study buyer patterns and to predict when the store is going to need to restock an item.

The analysts' queries may be answered more efficiently if they can query materialized views, especially views that aggregate data (e.g., sum the inventories of different sizes of shirt after grouping by style). But the database is updated with each sale, so modifications are far more frequent than queries. When modifications dominate, it is costly to have materialized views, or even indexes, on the data.

What is usually done is to create materialized views, but not to try to keep them up-to-date as the base tables change. Rather, the materialized views are reconstructed periodically (typically each night), when other activity in the database is low. The materialized views are only used by analysts, and their data might be out of date by as much as 24 hours. However, in normal situations, the rate at which an item is bought by customers changes slowly. Thus, the data will be "good enough" for the analysts to predict items that are selling well and those that are selling poorly. Of course if Brad Pitt is seen wearing a Hawaiian shirt one morning, and every cool guy has to buy one by that evening, the analysts will not notice they are out of Hawaiian shirts until the next morning, but the risk of that sort of occurrence is low.

8.5.3 Rewriting Queries to Use Materialized Views

A materialized view can be referred to in the `FROM` clause of a query, just as a virtual view can (Section 8.1.2). However, because a materialized view is stored in the database, it is possible to rewrite a query to use a materialized view, even if that view was not mentioned in the query as written. Such a rewriting may enable the query to execute much faster, because the hard parts of the query, e.g., joining of relations, may have been carried out already when the materialized view was constructed.

However,we must be very careful to check that the query can be rewritten to use a materialized view. A complete set of rules that will let us use materialized views of any kind is beyond the scope of this book. However, we shall offer a relatively simple rule that applies to the view of Example 8.15 and similar views.

Suppose we have a materialized view V defined by a query of the form:

SELECT L_V
FROM R_V
WHERE C_V

where L_V is a list of attributes, R_V is a list of relations, and C_V is a condition. Similarly, suppose we have a query Q of the same form:

SELECT L_Q
FROM R_Q
WHERE C_Q

Here are the conditions under which we can replace part of the query Q by the view V.

1. The relations in list R_V all appear in the list R_Q.
2. The condition C_Q is equivalent to C_V AND C for some condition C. As a special case, C_Q could be equivalent to C_V, in which case the "AND C" is unnecessary.
3. If C is needed, then the attributes of relations on list R_V that C mentions are attributes on the list L_V.
4. Attributes on the list L_Q that come from relations on the list R_V are also on the list L_V.

If all these conditions are met, then we can rewrite Q to use V, as follows:

a) Replace the list R_Q by V and the relations that are on list R_Q but not on R_V.

b) Replace C_Q by C. If C is not needed (i.e., $C_V = C_Q$), then there is no WHERE clause.

Example 8.16: Suppose we have the materialized view MovieProd from Example 8.15. This view is defined by the query V:

```
SELECT title, year, name
FROM Movies, MovieExec
WHERE producerC# = cert#
```

Suppose also that we need to answer the query Q that asks for the names of the stars of movies produced by Max Bialystock. For this query we need the relations:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

The query Q can be written:

```
SELECT starName
FROM StarsIn, Movies, MovieExec
WHERE movieTitle = title AND movieYear = year AND
    producerC# = cert# AND name = 'Max Bialystock';
```

Let us compare the view definition V with the query Q, to see that they meet the conditions listed above.

1. The relations in the FROM clause of V are all in the FROM clause of Q.
2. The condition from Q can be written as the condition from V AND C, where $C =$

```
movieTitle = title AND movieYear = year AND
    name = 'Max Bialystock'
```

3. The attributes of C that come from relations of V (Movies and MovieExec) are title, year, and name. These attributes all appear in the SELECT clause of V.
4. No attribute from the SELECT list of Q is from a relation that appears in the FROM list of V.

We may thus use V in Q, yielding the rewritten query:

```
SELECT starName
FROM StarsIn, MovieProd
WHERE movieTitle = title AND movieYear = year AND
    name = 'Max Bialystock';
```

That is, we replaced Movies and MovieExec in the FROM clause by the materialized view MovieProd. We also removed the condition of the view from the WHERE clause, leaving only the condition C. Since the rewritten query involves the join of only two relations, rather than three, we expect the rewritten query to execute in less time than the original. □

8.5.4 Automatic Creation of Materialized Views

The ideas that were discussed in Section 8.4.4 for indexes can apply as well to materialized views. We first need to establish or approximate the query workload. An automated materialized-view-selection advisor needs to generate candidate views. This task can be far more difficult than generating candidate indexes. In the case of indexes, there is only one possibile index for each attribute of each relation. We could also consider indexes on small sets of attributes of a relation, but even if we do, generating all the candidate indexes is straightforward. However, with materialized views, any query could in principle define a view, so there is no limit on what views we need to consider.

The process can be limited if we remember that there is no point in creating a materialized view that does not help for at least one query of our expected workload. For example, suppose some or all of the queries in our workload have the form considered in Section 8.5.3. Then we can use the analysis of that section to find the views that can help a given query. We can limit ourselves to candidate materialized views that:

1. Have a list of relations in the FROM clause that is a subset of those in the FROM clause of at least one query of the workload.
2. Have a WHERE clause that is the AND of conditions that each appear in at least one query.
3. Have a list of attributes in the SELECT clause that is sufficient to be used in at least one query.

To evaluate the benefit of a materialized view, let the query optimizer estimate the running times of the queries, both with and without the materialized view. Of course, the optimizer must be designed to take advantage of materialized views; all modern optimizers know how to exploit indexes, but not all can exploit materialized views. Section 8.5.3 was an example of the reasoning that would be necessary for a query optimizer to perform, if it were to take advantage of such views.

There is another issue that comes up when we consider automatic choice of materialized views, but that did not surface for indexes. An index on a relation is generally smaller than the relation itself, and all indexes on one relation take roughly the same amount of space. However, materialized views can vary radically in size, and some — those involving joins — can be very much larger than the relation or relations on which they are built. Thus, we may need to rethink the definition of the "benefit" of a materialized view. For example, we might want to define the benefit to be the improvement in average running time of the query workload divided by the amount of space the view occupies.

8.5.5 Exercises for Section 8.5

Exercise 8.5.1: Complete Example 8.15 by considering updates to either of the base tables.

! **Exercise 8.5.2:** Suppose the view NewPC of Exercise 8.2.3 were a materialized view. What modifications to the base tables Product and PC would require a modification of the materialized view? How would you implement those modifications incrementally?

! **Exercise 8.5.3:** This exercise explores materialized views that are based on aggregation of data. Suppose we build a materialized view on the base tables

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our running battleships exercise, as follows:

```
CREATE MATERIALIZED VIEW ShipStats AS
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = Ships.class
    GROUP BY country;
```

What modifications to the base tables `Classes` and `Ships` would require a modification of the materialized view? How would you implement those modifications incrementally?

! **Exercise 8.5.4:** In Section 8.5.3 we gave conditions under which a materialized view of simple form could be used in the execution of a query of similar form. For the view of Example 8.15, describe all the queries of that form, for which this view could be used.

8.6 Summary of Chapter 8

- ✦ *Virtual Views*: A virtual view is a definition of how one relation (the view) may be constructed logically from tables stored in the database or other views. Views may be queried as if they were stored relations. The query processor modifies queries about a view so the query is instead about the base tables that are used to define the view.

- ✦ *Updatable Views*: Some virtual views on a single relation are updatable, meaning that we can insert into, delete from, and update the view as if it were a stored table. These operations are translated into equivalent modifications to the base table over which the view is defined.

- ✦ *Instead-Of Triggers*: SQL allows a special type of trigger to apply to a virtual view. When a modification to the view is called for, the instead-of trigger turns the modification into operations on base tables that are specified in the trigger.

- ✦ *Indexes*: While not part of the SQL standard, commercial SQL systems allow the declaration of indexes on attributes; these indexes speed up certain queries or modifications that involve specification of a value, or range of values, for the indexed attribute(s).

- ✦ *Choosing Indexes*: While indexes speed up queries, they slow down database modifications, since the indexes on the modified relation must also be modified. Thus, the choice of indexes is a complex problem, depending on the actual mix of queries and modifications performed on the database.

- ✦ *Automatic Index Selection*: Some DBMS's offer tools that choose indexes for a database automatically. They examine the typical queries and modifications performed on the database and evaluate the cost trade-offs for different indexes that might be created.

- ✦ *Materialized Views*: Instead of treating a view as a query on base tables, we can use the query as a definition of an additional stored relation, whose value is a function of the values of the base tables.

- *Maintaining Materialized Views*: As the base tables change, we must make the corresponding changes to any materialized view whose value is affected by the change. For many common kinds of materialized views, it is possible to make the changes to the view incrementally, without recomputing the entire view.

- *Rewriting Queries to Use Materialized Views*: The conditions under which a query can be rewritten to use a materialized view are complex. However, if the query optimizer can perform such rewritings, then an automatic design tool can consider the improvement in performance that results from creating materialized views and can select views to materialize, automatically.

8.7 References for Chapter 8

The technology behind materialized views is surveyed in [2] and [7]. Reference [3] introduces the greedy algorithm for selecting materialized views.

Two projects for automatically tuning databases are AutoAdmin at Microsoft and SMART at IBM. Current information on AutoAdmin can be found on-line at [8]. A description of the technology behind this system is in [1].

A survey of the SMART project is in [4]. The index-selection aspect of the project is described in [6].

Reference [5] surveys index selection, materialized views, automatic tuning, and related subjects covered in this chapter.

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," *Intl. Conf. on Very Large Databases*, pp. 496–505, 2000.

2. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA, 1999.

3. V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.

4. S. S. Lightstone, G. Lohman, and S. Zilio, "Toward autonomic computing with DB2 universal database," *SIGMOD Record* **31**:3, pp. 55–61, 2002.

5. S. S. Lightstone, T. Teorey, and T. Nadeau, *Physical Database Design*, Morgan-Kaufmann, San Francisco, 2007.

6. G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley, "DB2 Advisor: an optimizer smart enough to recommend its own indexes," *Proc. Sixteenth IEEE Conf. on Data Engineering*, pp. 101–110, 2000.

7. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).

8. Microsoft on-line description of the AutoAdmin project.
`http://research.microsoft.com/dmx/autoadmin/`

Chapter 9

SQL in a Server Environment

We now turn to the question of how SQL fits into a complete programming environment. The typical server environment is introduced in Section 9.1. Section 9.2 introduces the SQL terminology for client-server computing and connecting to a database.

Then, we turn to how programming is really done, when SQL must be used to access a database as part of a typical application. In Section 9.3 we see how to embed SQL in programs that are written in an ordinary programming language, such as C. A critical issue is how we move data between SQL relations and the variables of the surrounding, or "host," language. Section 9.4 considers another way to combine SQL with general-purpose programming: persistent stored modules, which are pieces of code stored as part of a database schema and executable on command from the user.

A third programming approach is a "call-level interface," where we program in some conventional language and use a library of functions to access the database. In Section 9.5 we discuss the SQL-standard library called SQL/CLI, for making calls from C programs. Then, in Section 9.6 we meet Java's JDBC (database connectivity), which is an alternative call-level interface. Finally, another popular call-level interface, PHP, is covered in Section 9.7.

9.1 The Three-Tier Architecture

Databases are used in many different settings, including small, standalone databases. For example, a scientist may run a copy of MySQL or Microsoft Access on a laboratory computer to store experimental data. However, there is a very common architecture for large database installations; this architecture motivates the discussion of the entire chapter. The architecture is called *three-tier* or *three-layer*, because it distinguishes three different, interacting functions:

1. *Web Servers.* These are processes that connect clients to the database system, usually over the Internet or possibly a local connection.

2. *Application Servers.* These processes perform the "business logic," whatever it is the system is intended to do.

3. *Database Servers.* These processes run the DBMS and perform queries and modifications at the request of the application servers.

The processes may all run on the same processor in a small system, but it is common to dedicate a large number of processors to each of the tiers. Figure 9.1 suggests how a large database installation would be organized.

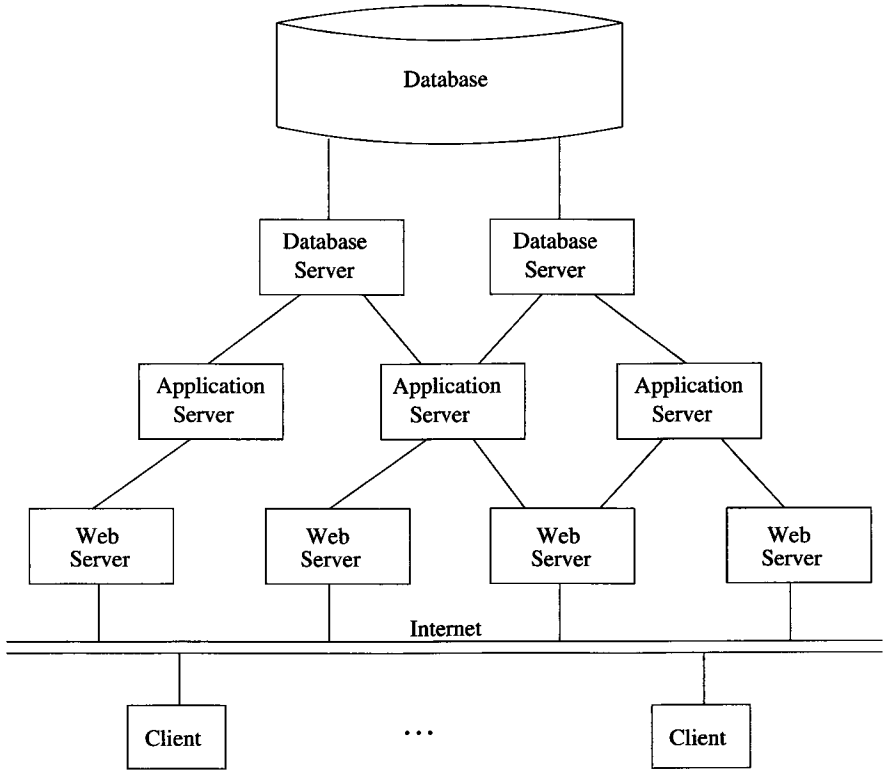


Figure 9.1: The Three-Tier Architecture

9.1.1 The Web-Server Tier

The web-server processes manage the interactions with the user. When a user makes contact, perhaps by opening a URL, a web server, typically running

Apache/Tomcat, responds to the request. The user then becomes a *client* of this web-server process. Typically, the client's actions are performed by the web-browser, e.g., managing of the filling of forms, which are then posted to the web server.

As an example, let us consider a site such as Amazon.com. A user (customer) opens a connection to the Amazon database system by entering the URL `www.amazon.com` into their browser. The Amazon web-server presents a "home page" to the user, which includes forms, menus, and buttons enabling the user to express what it is they want to do. For example, the user may set a menu to `Books` and enter into a form the title of the book they are interested in. The client web-browser transmits this information to the Amazon web-server, and that web-server must negotiate with the next tier — the application tier — to fulfill the client's request.

9.1.2 The Application Tier

The job of the application tier is to turn data, from the database, into a response to the request that it receives from the web-server. Each web-server process can invoke one or more application-tier processes to handle the request; these processes can be on one machine or many, and they may be on the same or different machines from the web-server processes.

The actions performed by the application tier are often referred to as the *business logic* of the organization operating the database. That is, one designs the application tier by reasoning out what the response to a request by the potential customer should be, and then implementing that strategy.

In the case of our example of a book at Amazon.com, this response would be the elements of the page that Amazon displays about a book. That data includes the title, author, price, and several other pieces of information about the book. It also includes links to more information, such as reviews, alternative sellers of the book, and similar books.

In a simple system, the application tier may issue database queries directly to the database tier, and assemble the results of those queries, perhaps in an HTML page. In a more complex system, there can be several subtiers to the application tier, and each may have its own processes. A common architecture is to have a subtier that supports "objects." These objects can contain data such as the title and price of a book in a "book object." Data for this object is obtained by a database query. The object may also have methods that can be invoked by the application-tier processes, and these methods may in turn cause additional queries to be issued to the database when and if they are invoked.

Another subtier may be present to support *database integration*. That is, there may be several quite independent databases that support operations, and it may not be possible to issue queries involving data from more than one database at a time. The results of queries to different sources may need to be combined at the integration subtier. To make integration more complex, the databases may not be compatable in a number of important ways. We shall

examine the technology of information integration elsewhere. However, for the moment, consider the following hypothetical example.

Example 9.1: The Amazon database containing information about a book may have a price in dollars. But the customer is in Europe, and their account information is in another database, located in Europe, with billing information in Euros. The integration subtier needs to know that there is a difference in currencies, when it gets a price from the books database and uses that price to enter data into a bill that is displayed to the customer. □

9.1.3 The Database Tier

Like the other tiers, there can be many processes in the database tier, and the processes can be distributed over many machines, or all be together on one. The database tier executes queries that are requested from the application tier, and may also provide some buffering of data. For example, a query that produces many tuples may be fed one-at-a-time to the requesting process of the application tier.

Since creating connections to the database takes significant time, we normally keep a large number of connections open and allow application processes to share these connections. Each application process must return the connection to the state in which it was found, to avoid unexpected interactions between application processes.

The balance of this chapter is about how we implement a database tier. Especially, we need to learn:

1. How do we enable a database to interact with "ordinary" programs that are written in a conventional language such as C or Java?

2. How do we deal with the differences in data-types supported by SQL and conventional languages? In particular, relations are the results of queries, and these are not directly supported by conventional languages.

3. How do we manage connections to a database when these connections are shared between many short-lived processes?

9.2 The SQL Environment

In this section we shall take the broadest possible view of a DBMS and the databases and programs it supports. We shall see how databases are defined and organized into clusters, catalogs, and schemas. We shall also see how programs are linked with the data they need to manipulate. Many of the details depend on the particular implementation, so we shall concentrate on the general ideas that are contained in the SQL standard. Sections 9.5, 9.6, and 9.7 illustrate how these high-level concepts appear in a "call-level interface," which requires the programmer to make explicit connections to databases.

9.2.1 Environments

A *SQL environment* is the framework under which data may exist and SQL operations on data may be executed. In practice, we should think of a SQL environment as a DBMS running at some installation. For example, ABC company buys a license for the Megatron 2010 DBMS to run on a collection of ABC's machines. The system running on these machines constitutes a SQL environment.

All the database elements we have discussed — tables, views, triggers, and so on — are defined within a SQL environment. These elements are organized into a hierarchy of structures, each of which plays a distinct role in the organization. The structures defined by the SQL standard are indicated in Fig. 9.2.

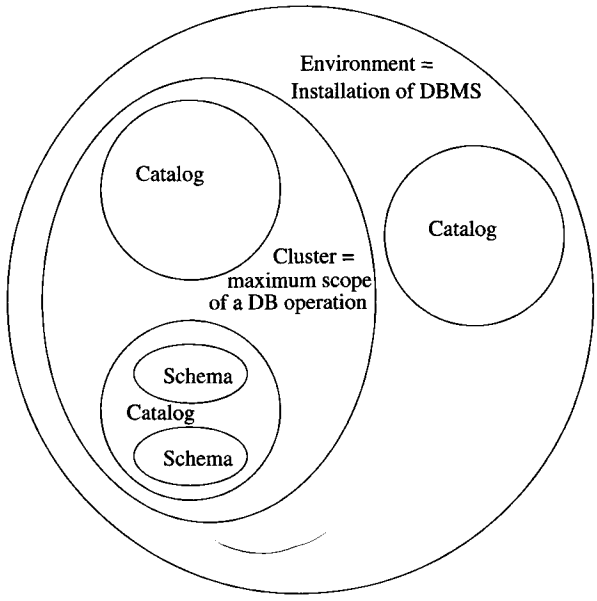


Figure 9.2: Organization of database elements within the environment

Briefly, the organization consists of the following structures:

1. *Schemas.* These are collections of tables, views, assertions, triggers, and some other types of information (see the box on "More Schema Elements" in Section 9.2.2). Schemas are the basic units of organization, close to what we might think of as a "database," but in fact somewhat less than a database as we shall see in point (3) below.

2. *Catalogs.* These are collections of schemas. They are the basic unit for supporting unique, accessible terminology. Each catalog has one or more schemas; the names of schemas within a catalog must be unique, and

each catalog contains a special schema called INFORMATION_SCHEMA that contains information about all the schemas in the catalog.

3. *Clusters.* These are collections of catalogs. Each user has an associated cluster: the set of all catalogs accessible to the user (see Section 10.1 for an explanation of how access to catalogs and other elements is controlled). A cluster is the maximum scope over which a query can be issued, so in a sense, a cluster is "the database" as seen by a particular user.

9.2.2 Schemas

The simplest form of schema declaration is:

CREATE SCHEMA <schema name> <element declarations>

The element declarations are of the forms discussed in various places, such as Sections 2.3, 8.1.1, 7.5.1, and 9.4.1.

Example 9.2: We could declare a schema that includes the five relations about movies that we have been using in our running example, plus some of the other elements we have introduced, such as views. Figure 9.3 sketches the form of such a declaration. □

```
CREATE SCHEMA MovieSchema
   CREATE TABLE MovieStar ... as in Fig. 7.3
      Create-table statements for the four other tables
   CREATE VIEW MovieProd ... as in Example 8.2
      Other view declarations
   CREATE ASSERTION RichPres ... as in Example 7.11
```

Figure 9.3: Declaring a schema

It is not necessary to declare the schema all at once. One can modify or add to the "current" schema using the appropriate CREATE, DROP, or ALTER statement, e.g., CREATE TABLE followed by the declaration of a new table for the schema. We change the "current" schema with a SET SCHEMA statement. For example,

```
SET SCHEMA MovieSchema;
```

makes the schema described in Fig. 9.3 the current schema. Then, any declarations of schema elements are added to that schema, and any DROP or ALTER statements refer to elements already in that schema.

More Schema Elements

Some schema elements that we have not already mentioned, but that occasionally are useful are:

- *Domains*: These are sets of values or simple data types. They are little used today, because object-relational DBMS's provide more powerful type-creation mechanisms; see Section 10.4.
- *Character sets*: These are sets of symbols and methods for encoding them. ASCII and Unicode are common options.
- *Collations*: A collation specifies which characters are "less than" which others. For example, we might use the ordering implied by the ASCII code, or we might treat lower-case and capital letters the same and not compare anything that isn't a letter.
- *Grant statements*: These concern who has access to schema elements. We shall discuss the granting of privileges in Section 10.1.
- *Stored Procedures*: These are executable code; see Section 9.4.

9.2.3 Catalogs

Just as schema elements like tables are created within a schema, schemas are created and modified within a catalog. In principle, we would expect the process of creating and populating catalogs to be analogous to the process of creating and populating schemas. Unfortunately, SQL does not define a standard way to do so, such as a statement

`CREATE CATALOG` <catalog name>

followed by a list of schemas belonging to that catalog and the declarations of those schemas.

However, SQL does stipulate a statement

`SET CATALOG` <catalog name>

This statement allows us to set the "current" catalog, so new schemas will go into that catalog and schema modifications will refer to schemas in that catalog should there be a name ambiguity.

9.2.4 Clients and Servers in the SQL Environment

A SQL environment is more than a collection of catalogs and schemas. It contains elements whose purpose is to support operations on the database or

Complete Names for Schema Elements

Formally, the name for a schema element such as a table is its catalog name, its schema name, and its own name, connected by dots in that order. Thus, the table `Movies` in the schema `MovieSchema` in the catalog `MovieCatalog` can be referred to as

```
MovieCatalog.MovieSchema.Movies
```

If the catalog is the default or current catalog, then we can omit that component of the name. If the schema is also the default or current schema, then that part too can be omitted, and we are left with the element's own name, as is usual. However, we have the option to use the full name if we need to access something outside the current schema or catalog.

databases represented by those catalogs and schemas. According to the SQL standard, within a SQL environment are two special kinds of processes: SQL clients and SQL servers.

In terms of Fig. 9.1, a "SQL server" plays the role what we called a "database server there. A "SQL client" is like the application servers from that figure. The SQL standard does not define processes analogous to what we called "Web servers" or "clients" in Fig. 9.1.

9.2.5 Connections

If we wish to run some program involving SQL at a host where a SQL client exists, then we may open a connection between the client and server by executing a SQL statement

```
CONNECT TO <server name> AS <connection name>
    AUTHORIZATION <name and password>
```

The server name is something that depends on the installation. The word `DEFAULT` can substitute for a name and will connect the user to whatever SQL server the installation treats as the "default server." We have shown an authorization clause followed by the user's name and password. The latter is the typical method by which a user would be identified to the server, although other strings following `AUTHORIZATION` might be used.

The connection name can be used to refer to the connection later on. The reason we might have to refer to the connection is that SQL allows several connections to be opened by the user, but only one can be active at any time. To switch among connections, we can make `conn1` become the active connection by the statement:

```
SET CONNECTION conn1;
```

Whatever connection was currently active becomes *dormant* until it is reactivated with another `SET CONNECTION` statement that mentions it explicitly.

We also use the name when we drop the connection. We can drop connection `conn1` by

```
DISCONNECT conn1;
```

Now, `conn1` is terminated; it is not dormant and cannot be reactivated.

However, if we shall never need to refer to the connection being created, then `AS` and the connection name may be omitted from the `CONNECT TO` statement. It is also permitted to skip the connection statements altogether. If we simply execute SQL statements at a host with a SQL client, then a default connection will be established on our behalf.

9.2.6 Sessions

The SQL operations that are performed while a connection is active form a *session*. The session lasts as long as the connection that created it. For example, when a connection is made dormant, its session also becomes dormant, and reactivation of the connection by a `SET CONNECTION` statement also makes the session active. Thus, we have shown the session and connection as two aspects of the link between client and server in Fig. 9.4.

Each session has a current catalog and a current schema within that catalog. These may be set with statements `SET SCHEMA` and `SET CATALOG`, as discussed in Sections 9.2.2 and 9.2.3. There is also an authorized user for every session, as we shall discuss in Section 10.1.

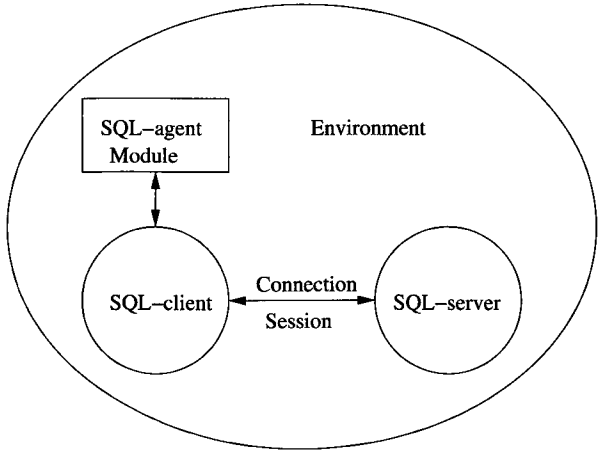


Figure 9.4: The SQL client-server interactions

The Languages of the SQL Standard

Implementations conforming to the SQL standard are required to support at least one of the following seven host languages: ADA, C, Cobol, Fortran, M (formerly called Mumps, and used primarily in the medical community), Pascal, and PL/I. We shall use C in our examples.

9.2.7 Modules

A *module* is the SQL term for an application program. The SQL standard suggests that there are three kinds of modules, but insists only that a SQL implementation offer the user at least one of these types.

1. *Generic SQL Interface.* The user may type SQL statements that are executed by a SQL server. In this mode, each query or other statement is a module by itself. It is this mode that we imagined for most of our examples in this book, although in practice it is rarely used.

2. *Embedded SQL.* This style will be discussed in Section 9.3. Typically, a preprocessor turns the embedded SQL statements into suitable function or procedure calls to the SQL system. The compiled host-language program, including these function calls, is a module.

3. *True Modules.* The most general style of modules envisioned by SQL is a collection of stored functions or procedures, some of which are host-language code and some of which are SQL statements. They communicate among themselves by passing parameters and perhaps via shared variables. PSM modules (Section 9.4) are an example of this type of module.

An execution of a module is called a *SQL agent.* In Fig. 9.4 we have shown both a module and an SQL agent, as one unit, calling upon a SQL client to establish a connection. However, we should remember that the distinction between a module and an SQL agent is analogous to the distinction between a program and a process; the first is code, the second is an execution of that code.

9.3 The SQL/Host-Language Interface

To this point, we have used the *generic SQL interface* in our examples. That is, we have assumed there is a SQL interpreter, which accepts and executes the sorts of SQL queries and commands that we have learned. Although provided as an option by almost all DBMS's, this mode of operation is actually rare. In real systems, such as those described in Section 9.1, there is a program in some

conventional *host* language such as C, but some of the steps in this program are actually SQL statements.

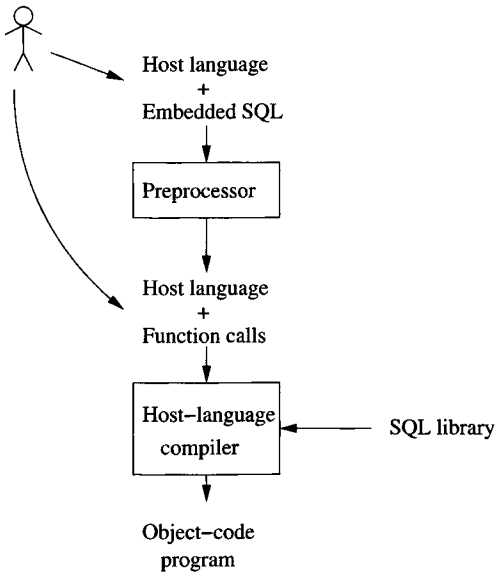


Figure 9.5: Processing programs with SQL statements embedded

A sketch of a typical programming system that involves SQL statements is in Fig. 9.5. There, we see the programmer writing programs in a host language, but with some special "embedded" SQL statements. There are two ways this embedding could take place.

1. *Call-Level Interface.* A library is provided, and the embedding of SQL in the host language is really calls to functions or methods in this library. SQL statements are usually string arguments of these methods. This approach, often referred to as a call-level interface or CLI, is discussed in Section 9.5 and is represented by the curved arrow in Fig. 9.5 from the user directly to the host language.

2. *Directly Embedded SQL.* The entire host-language program, with embedded SQL statements, is sent to a preprocessor, which changes the embedded SQL statements into something that makes sense in the host language. Typically, the SQL statements are replaced by calls to library functions or methods, so the difference between a CLI and direct embedding of SQL is more a matter of "look and feel" than of substance. The preprocessed host-language program is then compiled in the usual manner and operates on the database through execution of the library calls.

In this section, we shall learn the SQL standard for direct embedding in a host language — C in particular. We are also introduced to a number of concepts, such as cursors, that appear in all, or almost all, systems for embedding SQL.

9.3.1 The Impedance Mismatch Problem

The basic problem of connecting SQL statements with those of a conventional programming language is *impedance mismatch*: the fact that the data model of SQL differs so much from the models of other languages. As we know, SQL uses the relational data model at its core. However, C and similar languages use a data model with integers, reals, arithmetic, characters, pointers, record structures, arrays, and so on. Sets are not represented directly in C or these other languages, while SQL does not use pointers, loops and branches, or many other common programming-language constructs. As a result, passing data between SQL and other languages is not straightforward, and a mechanism must be devised to allow the development of programs that use both SQL and another language.

One might first suppose that it is preferable to use a single language. Either do all computation in SQL or forget SQL and do all computation in a conventional language. However, we can dispense with the idea of omitting SQL when there are database operations involved. SQL systems greatly aid the programmer in writing database operations that can be executed efficiently, yet that can be expressed at a very high level. SQL takes from the programmer's shoulders the need to understand how data is organized in storage or how to exploit that storage structure to operate efficiently on the database.

On the other hand, there are many important things that SQL cannot do at all. For example, one cannot write a SQL query to compute n factorial, something that is an easy exercise in C or similar languages.[1] As another example, SQL cannot format its output directly into a convenient form such as a graphic. Thus, real database programming requires both SQL and a host language.

9.3.2 Connecting SQL to the Host Language

When we wish to use a SQL statement within a host-language program, we warn the preprocessor that SQL code is coming with the keywords `EXEC SQL` in front of the statement. We transfer information between the database, which is accessed only by SQL statements, and the host-language program through *shared variables*, which are allowed to appear in both host-language statements

[1] We should be careful here. There are extensions to the basic SQL language, such as recursive SQL discussed in Section 10.2 or SQL/PSM discussed in Section 9.4, that do offer "Turing completeness" — the ability to compute anything that can be computed in any other programming language. However, these extensions were never intended for general-purpose calculation, and we do not regard them as general-purpose languages.

and SQL statements. Shared variables are prefixed by a colon within a SQL statement, but they appear without the colon in host-language statements.

A special variable, called SQLSTATE in the SQL standard, serves to connect the host-language program with the SQL execution system. The type of SQLSTATE is an array of five characters. Each time a function of the SQL library is called, a code is put in the variable SQLSTATE that indicates any problems found during that call. The SQL standard also specifies a large number of five-character codes and their meanings.

For example, '00000' (five zeroes) indicates that no error condition occurred, and '02000' indicates that a tuple requested as part of the answer to a SQL query could not be found. The latter code is very important, since it allows us to create a loop in the host-language program that examines tuples from some relation one-at-a-time and to break the loop after the last tuple has been examined.

9.3.3 The DECLARE Section

To declare shared variables, we place their declarations between two embedded SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
EXEC SQL END DECLARE SECTION;
```

What appears between them is called the *declare section.* The form of variable declarations in the declare section is whatever the host language requires. It only makes sense to declare variables to have types that both the host language and SQL can deal with, such as integers, reals, and character strings or arrays.

Example 9.3: The following statements might appear in a C function that updates the Studio relation:

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

The first and last statements are the required beginning and end of the declare section. In the middle is a statement declaring two shared variables, studioName and studioAddr. These are both character arrays and, as we shall see, they can be used to hold a name and address of a studio that are made into a tuple and inserted into the Studio relation. The third statement declares SQLSTATE to be a six-character array.[2] □

[2] We shall use six characters for the five-character value of SQLSTATE because in programs to follow we want to use the C function strcmp to test whether SQLSTATE has a certain value. Since strcmp expects strings to be terminated by '\0', we need a sixth character for this endmarker. The sixth character must be set initially to '\0', but we shall not show this assignment in programs to follow.

9.3.4 Using Shared Variables

A shared variable can be used in SQL statements in places where we expect or allow a constant. Recall that shared variables are preceded by a colon when so used. Here is an example in which we use the variables of Example 9.3 as components of a tuple to be inserted into relation `Studio`.

Example 9.4: In Fig. 9.6 is a sketch of a C function `getStudio` that prompts the user for the name and address of a studio, reads the responses, and inserts the appropriate tuple into `Studio`. Lines (1) through (4) are the declarations from Example 9.3. We omit the C code that prints requests and scans entered text to fill the two arrays `studioName` and `studioAddr`.

```
    void getStudio() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50], studioAddr[256];
3)          char SQLSTATE[6];
4)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name and address
           be entered and read response into variables
           studioName and studioAddr */

5)      EXEC SQL INSERT INTO Studio(name, address)
6)               VALUES (:studioName, :studioAddr);
    }
```

Figure 9.6: Using shared variables to insert a new studio

Then, in lines (5) and (6) is an embedded SQL `INSERT` statement. This statement is preceded by the keywords `EXEC SQL` to indicate that it is indeed an embedded SQL statement rather than ungrammatical C code. The values inserted by lines (5) and (6) are not explicit constants, as they were in all previous examples; rather, the values appearing in line (6) are shared variables whose current values become components of the inserted tuple. □

Any SQL statement that does not return a result (i.e., is not a query) can be embedded in a host-language program by preceding it with `EXEC SQL`. Examples of embeddable SQL statements include insert-, delete-, and update-statements and those statements that create, modify, or drop schema elements such as tables and views.

However, select-from-where queries are not embeddable directly into a host language, because of the "impedance mismatch." Queries produce bags of tuples as a result, while none of the major host languages support a set or bag

data type directly. Thus, embedded SQL must use one of two mechanisms for connecting the result of queries with a host-language program:

1. *Single-Row* SELECT *Statements.* A query that produces a single tuple can have that tuple stored in shared variables, one variable for each component of the tuple.

2. *Cursors.* Queries producing more than one tuple can be executed if we declare a *cursor* for the query. The cursor ranges over all tuples in the answer relation, and each tuple in turn can be fetched into shared variables and processed by the host-language program.

We shall consider each of these mechanisms in turn.

9.3.5 Single-Row Select Statements

The form of a single-row select is the same as an ordinary select-from-where statement, except that following the `SELECT` clause is the keyword `INTO` and a list of shared variables. These shared variables each are preceded by a colon, as is the case for all shared variables within a SQL statement. If the result of the query is a single tuple, this tuple's components become the values of these variables. If the result is either no tuple or more than one tuple, then no assignment to the shared variables is made, and an appropriate error code is written in the variable `SQLSTATE`.

Example 9.5: We shall write a C function to read the name of a studio and print the net worth of the studio's president. A sketch of this function is shown in Fig. 9.7. It begins with a declare section, lines (1) through (5), for the variables we shall need. Next, C statements that we do not show explicitly obtain a studio name from the standard input.

Lines (6) through (9) are the single-row select statement. It is quite similar to queries we have already seen. The two differences are that the value of variable `studioName` is used in place of a constant string in the condition of line (9), and there is an `INTO` clause at line (7) that tells us where to put the result of the query. In this case, we expect a single tuple, and tuples have only one component, that for attribute `netWorth`. The value of this one component of one tuple is placed in the shared variable `presNetWorth`. □

9.3.6 Cursors

The most versatile way to connect SQL queries to a host language is with a cursor that runs through the tuples of a relation. This relation can be a stored table, or it can be something that is generated by a query. To create and use a cursor, we need the following statements:

1. A cursor declaration, whose simplest form is:

```
    void printNetWorth() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50];
3)          int presNetWorth;
4)          char SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name be entered.
           read response into studioName */

6)      EXEC SQL SELECT netWorth
7)               INTO :presNetWorth
8)               FROM Studio, MovieExec
9)               WHERE presC# = cert# AND
                       Studio.name = :studioName;

        /* check that SQLSTATE has all 0's and if so, print
           the value of presNetWorth */
    }
```

Figure 9.7: A single-row select embedded in a C function

```
EXEC SQL DECLARE <cursor name> CURSOR FOR <query>
```

The query can be either an ordinary select-from-where query or a relation name. The cursor *ranges* over the tuples of the relation produced by the query.

2. A statement `EXEC SQL OPEN`, followed by the cursor name. This statement initializes the cursor to a position where it is ready to retrieve the first tuple of the relation over which the cursor ranges.

3. One or more uses of a *fetch statement*. The purpose of a fetch statement is to get the next tuple of the relation over which the cursor ranges. The fetch statement has the form:

```
EXEC SQL FETCH FROM <cursor name> INTO <list of variables>
```

There is one variable in the list for each attribute of the tuple's relation. If there is a tuple available to be fetched, these variables are assigned the values of the corresponding components from that tuple. If the tuples have been exhausted, then no tuple is returned, and the value of `SQLSTATE` is set to `'02000'`, a code that means "no tuple found."

4. The statement `EXEC SQL CLOSE` followed by the name of the cursor. This statement closes the cursor, which now no longer ranges over tuples of the relation. It can, however, be reinitialized by another `OPEN` statement, in which case it ranges anew over the tuples of this relation.

Example 9.6: Suppose we wish to determine the number of movie executives whose net worths fall into a sequence of bands of exponentially growing size, each band corresponding to a number of digits in the net worth. We shall design a query that retrieves the `netWorth` field of all the `MovieExec` tuples into a shared variable called `worth`. A cursor called `execCursor` will range over all these one-component tuples. Each time a tuple is fetched, we compute the number of digits in the integer `worth` and increment the appropriate element of an array `counts`.

The C function `worthRanges` begins in line (1) of Fig. 9.8. Line (2) declares some variables used only by the C function, not by the embedded SQL. The array `counts` holds the counts of executives in the various bands, `digits` counts the number of digits in a net worth, and `i` is an index ranging over the elements of array `counts`.

Lines (3) through (6) are a SQL declare section in which shared variable `worth` and the usual `SQLSTATE` are declared. Lines (7) and (8) declare `execCursor` to be a cursor that ranges over the values produced by the query on line (8). This query simply asks for the `netWorth` components of all the tuples in `MovieExec`. This cursor is then opened at line (9). Line (10) completes the initialization by zeroing the elements of array `counts`.

The main work is done by the loop of lines (11) through (16). At line (12) a tuple is fetched into shared variable `worth`. Since tuples produced by the query of line (8) have only one component, we need only one shared variable, although in general there would be as many variables as there are components of the retrieved tuples. Line (13) tests whether the fetch has been successful. Here, we use a macro `NO_MORE_TUPLES`, defined by

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

Recall that `"02000"` is the `SQLSTATE` code that means no tuple was found. If there are no more tuples, we break out of the loop and go to line (17).

If a tuple has been fetched, then at line (14) we initialize the number of digits in the net worth to 1. Line (15) is a loop that repeatedly divides the net worth by 10 and increments `digits` by 1. When the net worth reaches 0 after division by 10, `digits` holds the correct number of digits in the value of `worth` that was originally retrieved. Finally, line (16) increments the appropriate element of the array `counts` by 1. We assume that the number of digits is no more than 14. However, should there be a net worth with 15 or more digits, line (16) will not increment any element of the `counts` array, since there is no appropriate range; i.e., enormous net worths are thrown away and do not affect the statistics.

Line (17) begins the wrap-up of the function. The cursor is closed, and lines (18) and (19) print the values in the `counts` array. □

```
1)  void worthRanges() {

2)      int i, digits, counts[15];
3)      EXEC SQL BEGIN DECLARE SECTION;
4)          int worth;
5)          char SQLSTATE[6];
6)      EXEC SQL END DECLARE SECTION;
7)      EXEC SQL DECLARE execCursor CURSOR FOR
8)          SELECT netWorth FROM MovieExec;

9)      EXEC SQL OPEN execCursor;
10)     for(i=1; i<15; i++) counts[i] = 0;
11)     while(1) {
12)         EXEC SQL FETCH FROM execCursor INTO :worth;
13)         if(NO_MORE_TUPLES) break;
14)         digits = 1;
15)         while((worth /= 10) > 0) digits++;
16)         if(digits <= 14) counts[digits]++;
        }
17)     EXEC SQL CLOSE execCursor;
18)     for(i=0; i<15; i++)
19)         printf("digits = %d: number of execs = %d\n",
                i, counts[i]);
    }
```

Figure 9.8: Grouping executive net worths into exponential bands

9.3.7 Modifications by Cursor

When a cursor ranges over the tuples of a base table (i.e., a relation that is stored in the database), then one can not only read the current tuple, but one can update or delete the current tuple. The syntax of these `UPDATE` and `DELETE` statements are the same as we encountered in Section 6.5, with the exception of the `WHERE` clause. That clause may only be `WHERE CURRENT OF` followed by the name of the cursor. Of course it is possible for the host-language program reading the tuple to apply whatever condition it likes to the tuple before deciding whether or not to delete or update it.

Example 9.7: In Fig. 9.9 we see a C function that looks at each tuple of `MovieExec` and decides either to delete the tuple or to double the net worth. In lines (3) and (4) we declare variables that correspond to the four attributes of `MovieExec`, as well as the necessary `SQLSTATE`. Then, at line (6), `execCursor` is declared to range over the stored relation `MovieExec` itself.

Lines (8) through (14) are the loop, in which the cursor `execCursor` refers to each tuple of `MovieExec`, in turn. Line (9) fetches the current tuple into

```
1)  void changeWorth() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          int certNo, worth;
4)          char execName[31], execAddr[256], SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;
6)      EXEC SQL DECLARE execCursor CURSOR FOR MovieExec;

7)      EXEC SQL OPEN execCursor;
8)      while(1) {
9)          EXEC SQL FETCH FROM execCursor INTO :execName,
                :execAddr, :certNo, :worth;
10)         if(NO_MORE_TUPLES) break;
11)         if (worth < 1000)
12)             EXEC SQL DELETE FROM MovieExec
                        WHERE CURRENT OF execCursor;
13)         else
14)             EXEC SQL UPDATE MovieExec
                        SET netWorth = 2 * netWorth
                        WHERE CURRENT OF execCursor;
        }
15)     EXEC SQL CLOSE execCursor;
    }
```

Figure 9.9: Modifying executive net worths

the four variables used for this purpose; note that only `worth` is actually used. Line (10) tests whether we have exhausted the tuples of `MovieExec`. We have again used the macro `NO_MORE_TUPLES` for the condition that variable `SQLSTATE` has the "no more tuples" code `"02000"`.

In the test of line (11) we ask if the net worth is under $1000. If so, the tuple is deleted by the `DELETE` statement of line (12). Note that the `WHERE` clause refers to the cursor, so the current tuple of `MovieExec`, the one we just fetched, is deleted from `MovieExec`. If the net worth is at least $1000, then at line (14), the net worth in the same tuple is doubled, instead. □

9.3.8 Protecting Against Concurrent Updates

Suppose that as we examine the net worths of movie executives using the function `worthRanges` of Fig. 9.8, some other process is modifying the underlying `MovieExec` relation. What should we do about this possibility? Perhaps nothing. We might be happy with approximate statistics, and we don't care whether or not we count an executive who was in the process of being deleted, for example. Then, we simply accept what tuples we get through the cursor.

However, we may not wish to allow concurrent changes to affect the tuples we see through this cursor. Rather, we may insist on the statistics being taken on the relation as it exists at some point in time. In terms of the transactions of Section 6.6, we want the code that runs the cursor through the relation to be serializable with any other operations on the relation. To obtain this guarantee, we may declare the cursor *insensitive* to concurrent changes.

Example 9.8: We could modify lines (7) and (8) of Fig. 9.8 to be:

```
7)    EXEC SQL DECLARE execCursor INSENSITIVE CURSOR FOR
8)        SELECT netWorth FROM MovieExec;
```

If `execCursor` is so declared, then the SQL system will guarantee that changes to relation `MovieExec` made between one opening and closing of `execCursor` will not affect the set of tuples fetched. □

There are certain cursors ranging over a relation R about which we may say with certainty that they will not change R. Such a cursor can run simultaneously with an insensitive cursor for R, without risk of changing the relation R that the insensitive cursor sees. If we declare a cursor `FOR READ ONLY`, then the database system can be sure that the underlying relation will not be modified because of access to the relation through this cursor.

Example 9.9: We could append after line (8) of `worthRanges` in Fig. 9.8 a line

```
FOR READ ONLY;
```

If so, then any attempt to execute a modification through cursor `execCursor` would cause an error. □

9.3.9 Dynamic SQL

Our model of SQL embedded in a host language has been that of specific SQL queries and commands within a host-language program. An alternative style of embedded SQL has the statements themselves be computed by the host language. Such statements are not known at compile time, and thus cannot be handled by a SQL preprocessor or a host-language compiler.

An example of such a situation is a program that prompts the user for an SQL query, reads the query, and then executes that query. The generic interface for ad-hoc SQL queries that we assumed in Chapter 6 is an example of just such a program. If queries are read and executed at run-time, there is nothing that can be done at compile-time. The query has to be parsed and a suitable way to execute the query found by the SQL system, immediately after the query is read.

The host-language program must instruct the SQL system to take the character string just read, to turn it into an executable SQL statement, and finally to execute that statement. There are two *dynamic SQL* statements that perform these two steps.

1. EXEC SQL PREPARE V FROM <expression>, where V is a SQL variable. The expression can be any host-language expression whose value is a string; this string is treated as a SQL statement. Presumably, the SQL statement is parsed and a good way to execute it is found by the SQL system, but the statement is not executed. Rather, the plan for executing the SQL statement becomes the value of V.

2. EXEC SQL EXECUTE V. This statement causes the SQL statement denoted by variable V to be executed.

Both steps can be combined into one, with the statement:

EXEC SQL EXECUTE IMMEDIATE <expression>

The disadvantage of combining these two parts is seen if we prepare a statement once and then execute it many times. With EXECUTE IMMEDIATE the cost of preparing the statement is paid each time the statement is executed, rather than paid only once, when we prepare it.

Example 9.10: In Fig. 9.10 is a sketch of a C program that reads text from standard input into a variable query, prepares it, and executes it. The SQL variable SQLquery holds the prepared query. Since the query is only executed once, the line:

```
EXEC SQL EXECUTE IMMEDIATE :query;
```

could replace lines (6) and (7) of Fig. 9.10. □

```
1)  void readQuery() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          char *query;
4)      EXEC SQL END DECLARE SECTION;

5)      /* prompt user for a query, allocate space (e.g.,
           use malloc) and make shared variable :query point
           to the first character of the query */
6)      EXEC SQL PREPARE SQLquery FROM :query;
7)      EXEC SQL EXECUTE SQLquery;
    }
```

Figure 9.10: Preparing and executing a dynamic SQL query

9.3.10 Exercises for Section 9.3

Exercise 9.3.1: Write the following embedded SQL queries, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1. You may use any host language with which you are familiar, and details of host-language programming may be replaced by clear comments if you wish.

a) Ask the user for a price and find the PC whose price is closest to the desired price. Print the maker, model number, and speed of the PC.

b) Ask the user for minimum values of the speed, RAM, hard-disk size, and screen size that they will accept. Find all the laptops that satisfy these requirements. Print their specifications (all attributes of `Laptop`) and their manufacturer.

! c) Ask the user for a manufacturer. Print the specifications of all products by that manufacturer. That is, print the model number, product-type, and all the attributes of whichever relation is appropriate for that type.

!! d) Ask the user for a "budget" (total price of a PC and printer), and a minimum speed of the PC. Find the cheapest "system" (PC plus printer) that is within the budget and minimum speed, but make the printer a color printer if possible. Print the model numbers for the chosen system.

e) Ask the user for a manufacturer, model number, speed, RAM, hard-disk size, and price of a new PC. Check that there is no PC with that model number. Print a warning if so, and otherwise insert the information into tables `Product` and `PC`.

Exercise 9.3.2: Write the following embedded SQL queries, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3.

a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore of the guns. Find the class with the largest firepower.

! b) Ask the user for the name of a battle. Find the countries of the ships involved in the battle. Print the country with the most ships sunk and the country with the most ships damaged.

c) Ask the user for the name of a class and the other information required for a tuple of table `Classes`. Then ask for a list of the names of the ships of that class and their dates launched. However, the user need not give the first name, which will be the name of the class. Insert the information gathered into `Classes` and `Ships`.

! d) Examine the `Battles`, `Outcomes`, and `Ships` relations for ships that were in battle before they were launched. Prompt the user when there is an error found, offering the option to change the date of launch or the date of the battle. Make whichever change is requested.

9.4 Stored Procedures

In this section, we introduce you to *Persistent, Stored Modules* (SQL/PSM, or just PSM). PSM is part of the latest revision to the SQL standard, called SQL:2003. It allows us to write procedures in a simple, general-purpose language and to store them in the database, as part of the schema. We can then use these procedures in SQL queries and other statements to perform computations that cannot be done with SQL alone. Each commercial DBMS offers its own extension of PSM. In this book, we shall describe the SQL/PSM standard, which captures the major ideas of these facilities, and which should help you understand the language associated with any particular system. References to PSM extensions provided with several major commercial systems are in the bibliographic notes.

9.4.1 Creating PSM Functions and Procedures

In PSM, you define *modules*, which are collections of function and procedure definitions, temporary relation declarations, and several other optional declarations. The major elements of a procedure declaration are:

```
CREATE PROCEDURE <name> (<parameters>)
    <local declarations>
    <procedure body>;
```

This form should be familiar from a number of programming languages; it consists of a procedure name, a parenthesized list of parameters, some optional local-variable declarations, and the executable body of code that defines the procedure. A function is defined in almost the same way, except that the keyword `FUNCTION` is used, and there is a return-value type that must be specified. That is, the elements of a function definition are:

```
CREATE FUNCTION <name> (<parameters>) RETURNS <type>
    <local declarations>
    <function body>;
```

The parameters of a PSM procedure are mode-name-type triples. That is, the parameter name is not only followed by its declared type, as usual in programming languages, but it is preceded by a "mode," which is either `IN`, `OUT`, or `INOUT`. These three keywords indicate that the parameter is input-only, output-only, or both input and output, respectively. `IN` is the default, and can be omitted.

Function parameters, on the other hand, may only be of mode `IN`. That is, PSM forbids side-effects in functions, so the only way to obtain information from a function is through its return-value. We shall not specify the `IN` mode for function parameters, although we do so in procedure definitions.

Example 9.11: While we have not yet learned the variety of statements that can appear in procedure and function bodies, one kind should not surprise us: an SQL statement. The limitation on these statements is the same as for embedded SQL, as we introduced in Section 9.3.4: only single-row-select statements and cursor-based accesses are permitted as queries. In Fig. 9.11 is a PSM procedure that takes two addresses — an old address and a new address — as parameters and replaces the old address by the new everywhere it appears in `MovieStar`.

```
1)  CREATE PROCEDURE Move(
2)      IN oldAddr VARCHAR(255),
3)      IN newAddr VARCHAR(255)
    )
4)  UPDATE MovieStar
5)  SET address = newAddr
6)  WHERE address = oldAddr;
```

Figure 9.11: A procedure to change addresses

Line (1) introduces the procedure and its name, `Move`. Lines (2) and (3) declare two input parameters, both of whose types are `VARCHAR(255)`. This type is consistent with the type we declared for the attribute `address` of `MovieStar` in Fig. 2.8. Lines (4) through (6) are a conventional `UPDATE` statement. However, notice that the parameter names can be used as if they were constants. Unlike host-language variables, which require a colon prefix when used in SQL (see Section 9.3.2), parameters and other local variables of PSM procedures and functions require no colon. □

9.4.2 Some Simple Statement Forms in PSM

Let us begin with a potpourri of statement forms that are easy to master.

1. *The call-statement*: The form of a procedure call is:

   CALL <procedure name> (<argument list>);

   That is, the keyword CALL is followed by the name of the procedure and a parenthesized list of arguments, as in most any language. This call can, however, be made from a variety of places:

   i. From a host-language program, in which it might appear as

   ```
   EXEC SQL CALL Foo(:x, 3);
   ```

   for instance.

   ii. As a statement of another PSM function or procedure.

   iii. As a SQL command issued to the generic SQL interface. For example, we can issue a statement such as

   ```
   CALL Foo(1, 3);
   ```

   to such an interface, and have stored procedure Foo executed with its two parameters set equal to 1 and 3, respectively.

   Note that it is not permitted to call a function. You invoke functions in PSM as you do in C: use the function name and suitable arguments as part of an expression.

2. *The return-statement*: Its form is

   RETURN <expression>;

   This statement can only appear in a function. It evaluates the expression and sets the return-value of the function equal to that result. However, at variance with common programming languages, the return-statement of PSM does *not* terminate the function. Rather, control continues with the following statement, and it is possible that the return-value will be changed before the function completes.

3. *Declarations of local variables*: The statement form

   DECLARE <name> <type>;

   declares a variable with the given name to have the given type. This variable is local, and its value is not preserved by the DBMS after a running of the function or procedure. Declarations must precede executable statements in the function or procedure body.

4. *Assignment Statements*: The form of an assignment is:

SET <variable> = <expression>;

Except for the introductory keyword SET, assignment in PSM is quite like assignment in other languages. The expression on the right of the equal-sign is evaluated, and its value becomes the value of the variable on the left. NULL is a permissible expression. The expression may even be a query, as long as it returns a single value.

5. *Statement groups*: We can form a list of statements ended by semicolons and surrounded by keywords BEGIN and END. This construct is treated as a single statement and can appear anywhere a single statement can. In particular, since a procedure or function body is expected to be a single statement, we can put any sequence of statements in the body by surrounding them by BEGIN...END.

6. *Statement labels*: We label a statement by prefixing it with a name (the label) and a colon.

9.4.3 Branching Statements

For our first complex PSM statement type, let us consider the if-statement. The form is only a little strange; it differs from C or similar languages in that:

1. The statement ends with keywords END IF.

2. If-statements nested within the else-clause are introduced with the single word ELSEIF.

Thus, the general form of an if-statement is as suggested by Fig. 9.12. The condition is any boolean-valued expression, as can appear in the WHERE clause of SQL statements. Each statement list consists of statements ended by semicolons, but does not need a surrounding BEGIN...END. The final ELSE and its statement(s) are optional; i.e., IF...THEN...END IF alone or with ELSEIF's is acceptable.

Example 9.12: Let us write a function to take a year y and a studio s, and return a boolean that is TRUE if and only if studio s produced at least one comedy in year y or did not produce any movies at all in that year. The code appears in Fig. 9.13.

Line (1) introduces the function and includes its arguments. We do not need to specify a mode for the arguments, since that can only be IN for a function. Lines (2) and (3) test for the case where there are no movies at all by studio s in year y, in which case we set the return-value to TRUE at line (4). Note that line (4) does not cause the function to return. Technically, it is the flow of control dictated by the if-statements that causes control to jump from line (4) to line (9), where the function completes and returns.

```
IF <condition> /tt THEN
   <statement list>
ELSEIF <condition> /tt THEN
   <statement list>
ELSEIF
      ...
ELSE
  <statement list>
END IF;
```

Figure 9.12: The form of an if-statement

```
1)  CREATE FUNCTION BandW(y INT, s CHAR(15)) RETURNS BOOLEAN

2)  IF NOT EXISTS(
3)      SELECT * FROM Movies WHERE year = y AND
            studioName = s)
4)  THEN RETURN TRUE;
5)  ELSEIF 1 <=
6)      (SELECT COUNT(*) FROM Movies WHERE year = y AND
            studioName = s AND genre = 'comedy')
7)  THEN RETURN TRUE;
8)  ELSE RETURN FALSE;
9)  END IF;
```

Figure 9.13: If there are any movies at all, then at least one has to be a comedy

If studio s made movies in year y, then lines (5) and (6) test if at least one of them was a comedy. If so, the return-value is again set to true, this time at line (7). In the remaining case, studio s made movies but only in color, so we set the return-value to FALSE at line (8). □

9.4.4 Queries in PSM

There are several ways that select-from-where queries are used in PSM.

1. Subqueries can be used in conditions, or in general, any place a subquery is legal in SQL. We saw two examples of subqueries in lines (3) and (6) of Fig. 9.13, for instance.

2. Queries that return a single value can be used as the right sides of assignment statements.

3. A single-row select statement is a legal statement in PSM. Recall this statement has an INTO clause that specifies variables into which the com-

ponents of the single returned tuple are placed. These variables could be local variables or parameters of a PSM procedure. The general form was discussed in the context of embedded SQL in Section 9.3.5.

4. We can declare and use a cursor, essentially as it was described in Section 9.3.6 for embedded SQL. The declaration of the cursor, `OPEN`, `FETCH`, and `CLOSE` statements are all as described there, with the exceptions that:
   (a) No `EXEC SQL` appears in the statements, and
   (b) The variables do not use a colon prefix.

```
CREATE PROCEDURE SomeProc(IN studioName CHAR(15))

DECLARE presNetWorth INTEGER;

SELECT netWorth
INTO presNetWorth
FROM Studio, MovieExec
WHERE presC# = cert# AND Studio.name = studioName;
    ...
```

Figure 9.14: A single-row select in PSM

Example 9.13: In Fig. 9.14 is the single-row select of Fig. 9.7, redone for PSM and placed in the context of a hypothetical procedure definition. Note that, because the single-row select returns a one-component tuple, we could also get the same effect from an assignment statement, as:

```
SET presNetWorth = (SELECT netWorth
    FROM Studio, MovieExec
    WHERE presC# = cert# AND Studio.name = studioName);
```

We shall defer examples of cursor use until we learn the PSM loop statements in the next section. □

9.4.5 Loops in PSM

The basic loop construct in PSM is:

```
LOOP
   <statement list>
END LOOP;
```

One often labels the `LOOP` statement, so it is possible to break out of the loop, using a statement:

```
LEAVE <loop label>;
```

In the common case that the loop involves the fetching of tuples via a cursor, we often wish to leave the loop when there are no more tuples. It is useful to declare a *condition* name for the SQLSTATE value that indicates no tuple found ('02000', recall); we do so with:

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

More generally, we can declare a condition with any desired name corresponding to any SQLSTATE value by

```
DECLARE <name> CONDITION FOR SQLSTATE <value>;
```

We are now ready to take up an example that ties together cursor operations and loops in PSM.

Example 9.14: Figure 9.15 shows a PSM procedure that takes a studio name s as an input argument and produces in output arguments `mean` and `variance` the mean and variance of the lengths of all the movies owned by studio s. Lines (1) through (4) declare the procedure and its parameters.

Lines (5) through (8) are local declarations. We define `Not_Found` to be the name of the condition that means a `FETCH` failed to return a tuple at line (5). Then, at line (6), the cursor `MovieCursor` is defined to return the set of the lengths of the movies by studio s. Lines (7) and (8) declare two local variables that we'll need. Integer `newLength` holds the result of a `FETCH`, while `movieCount` counts the number of movies by studio s. We need `movieCount` so that, at the end, we can convert a sum of lengths into an average (mean) of lengths and a sum of squares of the lengths into a variance.

The rest of the lines are the body of the procedure. We shall use `mean` and `variance` as temporary variables, as well as for "returning" the results at the end. In the major loop, `mean` actually holds the sum of the lengths, and `variance` actually holds the sum of the squares of the lengths. Thus, lines (9) through (11) initialize these variables and the count of the movies to 0. Line (12) opens the cursor, and lines (13) through (19) form the loop labeled `movieLoop`.

Line (14) performs a fetch, and at line (15) we check that another tuple was found. If not, we leave the loop. Lines (16) through (18) accumulate values; we add 1 to `movieCount`, add the length to `mean` (which, recall, is really computing the sum of lengths), and we add the square of the length to `variance`.

When all movies by studio s have been seen, we leave the loop, and control passes to line (20). At that line, we turn `mean` into its correct value by dividing the sum of lengths by the count of movies. At line (21), we make `variance` truly hold the variance by dividing the sum of squares of the lengths by the number of movies and subtracting the square of the mean. See Exercise 9.4.4 for a discussion of why this calculation is correct. Line (22) closes the cursor, and we are done. □

```
1)  CREATE PROCEDURE MeanVar(
2)      IN s CHAR(15),
3)      OUT mean REAL,
4)      OUT variance REAL
    )
5)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
6)  DECLARE MovieCursor CURSOR FOR
        SELECT length FROM Movies WHERE studioName = s;
7)  DECLARE newLength INTEGER;
8)  DECLARE movieCount INTEGER;

    BEGIN
9)      SET mean = 0.0;
10)     SET variance = 0.0;
11)     SET movieCount = 0;
12)     OPEN MovieCursor;
13)     movieLoop: LOOP
14)         FETCH FROM MovieCursor INTO newLength;
15)         IF Not_Found THEN LEAVE movieLoop END IF;
16)         SET movieCount = movieCount + 1;
17)         SET mean = mean + newLength;
18)         SET variance = variance + newLength * newLength;
19)     END LOOP;
20)     SET mean = mean/movieCount;
21)     SET variance = variance/movieCount - mean * mean;
22)     CLOSE MovieCursor;
    END;
```

Figure 9.15: Computing the mean and variance of lengths of movies by one studio

9.4.6 For-Loops

There is also in PSM a for-loop construct, but it is used only to iterate over a cursor. The form of the statement is shown in Fig. 9.16. This statement not only declares a cursor, but it handles for us a number of "grubby details": the opening and closing of the cursor, the fetching, and the checking whether there are no more tuples to be fetched. However, since we are not fetching tuples for ourselves, we can not specify the variable(s) into which component(s) of a tuple are placed. Thus, the names used for the attributes in the result of the query are also treated by PSM as local variables of the same type.

Example 9.15: Let us redo the procedure of Fig. 9.15 using a for-loop. The code is shown in Fig. 9.17. Many things have not changed. The declaration of the procedure in lines (1) through (4) of Fig. 9.17 are the same, as is the

Other Loop Constructs

PSM also allows while- and repeat-loops, which have the expected meaning, as in C. That is, we can create a loop of the form

```
WHILE <condition> DO
   <statement list>
END WHILE;
```

or a loop of the form

```
REPEAT
   <statement list>
UNTIL <condition>
END REPEAT;
```

Incidentally, if we label these loops, or the loop formed by a loop-statement or for-statement, then we can place the label as well after the `END LOOP` or other ender. The advantage of doing so is that it makes clearer where each loop ends, and it allows the PSM compiler to catch some syntactic errors involving the omission of an `END`.

```
FOR <loop name> AS <cursor name> CURSOR FOR
   <query>
DO
   <statement list>
END FOR;
```

Figure 9.16: The PSM for-statement

declaration of local variable `movieCount` at line (5).

However, we no longer need to declare a cursor in the declaration portion of the procedure, and we do not need to define the condition `Not_Found`. Lines (6) through (8) initialize the variables, as before. Then, in line (9) we see the for-loop, which also defines the cursor `MovieCursor`. Lines (11) through (13) are the body of the loop. Notice that in lines (12) and (13), we refer to the length retrieved via the cursor by the attribute name `length`, rather than by the local variable name `newLength`, which does not exist in this version of the procedure. Lines (15) and (16) compute the correct values for the output variables, exactly as in the earlier version of this procedure. □

```
1)  CREATE PROCEDURE MeanVar(
2)      IN s CHAR(15),
3)      OUT mean REAL,
4)      OUT variance REAL
    )
5)  DECLARE movieCount INTEGER;

    BEGIN
6)      SET mean = 0.0;
7)      SET variance = 0.0;
8)      SET movieCount = 0;
9)      FOR movieLoop AS MovieCursor CURSOR FOR
            SELECT length FROM Movies WHERE studioName = s;
10)     DO
11)         SET movieCount = movieCount + 1;
12)         SET mean = mean + length;
13)         SET variance = variance + length * length;
14)     END FOR;
15)     SET mean = mean/movieCount;
16)     SET variance = variance/movieCount - mean * mean;
    END;
```

Figure 9.17: Computing the mean and variance of lengths using a for-loop

9.4.7 Exceptions in PSM

A SQL system indicates error conditions by setting a nonzero sequence of digits in the five-character string SQLSTATE. We have seen one example of these codes: '02000' for "no tuple found." For another example, '21000' indicates that a single-row select has returned more than one row.

PSM allows us to declare a piece of code, called an *exception handler*, that is invoked whenever one of a list of these error codes appears in SQLSTATE during the execution of a statement or list of statements. Each exception handler is associated with a block of code, delineated by BEGIN...END. The handler appears within this block, and it applies only to statements within the block.

The components of the handler are:

1. A list of exception conditions that invoke the handler when raised.
2. Code to be executed when one of the associated exceptions is raised.
3. An indication of where to go after the handler has finished its work.

The form of a handler declaration is:

DECLARE <where to go next> HANDLER FOR <condition list>
  <statement>

Why Do We Need Names in For-Loops?

Notice that `movieLoop` and `MovieCursor`, although declared at line (9) of Fig. 9.17, are never used in that procedure. Nonetheless, we have to invent names, both for the for-loop itself and for the cursor over which it iterates. The reason is that the PSM interpreter will translate the for-loop into a conventional loop, much like the code of Fig. 9.15, and in this code, there is a need for both names.

The choices for "where to go" are:

a) `CONTINUE`, which means that after executing the statement in the handler declaration, we execute the statement after the one that raised the exception.

b) `EXIT`, which means that after executing the handler's statement, control leaves the `BEGIN...END` block in which the handler is declared. The statement after this block is executed next.

c) `UNDO`, which is the same as `EXIT`, except that any changes to the database or local variables that were made by the statements of the block executed so far are undone. That is, the block is a transaction, which is aborted by the exception.

The "condition list" is a comma-separated list of conditions, which are either declared conditions, like `Not_Found` in line (5) of Fig. 9.15, or expressions of the form `SQLSTATE` and a five-character string.

Example 9.16: Let us write a PSM function that takes a movie title as argument and returns the year of the movie. If there is no movie of that title or more than one movie of that title, then `NULL` must be returned. The code is shown in Fig. 9.18.

Lines (2) and (3) declare symbolic conditions; we do not have to make these definitions, and could as well have used the SQL states for which they stand in line (4). Lines (4), (5), and (6) are a block, in which we first declare a handler for the two conditions in which either zero tuples are returned, or more than one tuple is returned. The action of the handler, on line (5), is simply to set the return-value to `NULL`.

Line (6) is the statement that does the work of the function `GetYear`. It is a `SELECT` statement that is expected to return exactly one integer, since that is what the function `GetYear` returns. If there is exactly one movie with title t (the input parameter of the function), then this value will be returned. However, if an exception is raised at line (6), either because there is no movie with title t or several movies with that title, then the handler is invoked, and `NULL` instead

```
1)  CREATE FUNCTION GetYear(t VARCHAR(255)) RETURNS INTEGER

2)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
3)  DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

    BEGIN
4)      DECLARE EXIT HANDLER FOR Not_Found, Too_Many
5)          RETURN NULL;
6)      RETURN (SELECT year FROM Movies WHERE title = t);
    END;
```

Figure 9.18: Handling exceptions in which a single-row select returns other than one tuple

becomes the return-value. Also, since the handler is an `EXIT` handler, control next passes to the point after the `END`. Since that point is the end of the function, `GetYear` returns at that time, with the return-value `NULL`. □

9.4.8 Using PSM Functions and Procedures

As we mentioned in Section 9.4.2, we can call a PSM procedure anywhere SQL statements can appear, e.g., as embedded SQL, from PSM code itself, or from SQL issued to the generic interface. We invoke a procedure by preceding it by the keyword `CALL`. In addition, a PSM function can be used as part of an expression, e.g., in a `WHERE` clause. Here is an example of how a function can be used within an expression.

Example 9.17: Suppose that our schema includes a module with the function `GetYear` of Fig. 9.18. Imagine that we are sitting at the generic interface, and we want to enter the fact that Denzel Washington was a star of *Remember the Titans*. However, we forget the year in which that movie was made. As long as there was only one movie of that name, and it is in the `Movies` relation, we don't have to look it up in a preliminary query. Rather, we can issue to the generic SQL interface the following insertion:

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES('Remember the Titans', GetYear('Remember the Titans'),
    'Denzel Washington');
```

Since `GetYear` returns `NULL` if there is not a unique movie by the name of *Remember the Titans*, it is possible that this insertion will have `NULL` in the middle component. □

9.4.9 Exercises for Section 9.4

Exercise 9.4.1: Using our running movie database:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write PSM procedures or functions to perform the following tasks:

a) Given the name of a movie studio, produce the net worth of its president.

b) Given a name and address, return 1 if the person is a movie star but not an executive, 2 if the person is an executive but not a star, 3 if both, and 4 if neither.

! c) Given a studio name, assign to output parameters the titles of the two longest movies by that studio. Assign `NULL` to one or both parameters if there is no such movie (e.g., if there is only one movie by a studio, there is no "second-longest").

! d) Given a star name, find the earliest (lowest year) movie of more than 120 minutes length in which they appeared. If there is no such movie, return the year 0.

e) Given an address, find the name of the unique star with that address if there is exactly one, and return `NULL` if there is none or more than one.

f) Given the name of a star, delete them from `MovieStar` and delete all their movies from `StarsIn` and `Movies`.

Exercise 9.4.2: Write the following PSM functions or procedures, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1.

a) Take a price as argument and return the model number of the PC whose price is closest.

b) Take a maker and model as arguments, and return the price of whatever type of product that model is.

! c) Take model, speed, ram, hard-disk, and price information as arguments, and insert this information into the relation `PC`. However, if there is already a PC with that model number (tell by assuming that violation of a key constraint on insertion will raise an exception with `SQLSTATE` equal to `'23000'`), then keep adding 1 to the model number until you find a model number that is not already a PC model number.

! d) Given a price, produce the number of PC's, the number of laptops, and the number of printers selling for more than that price.

Exercise 9.4.3: Write the following PSM functions or procedures, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3.

a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore. Given a class, find its firepower.

! b) Given the name of a battle, produce the two countries whose ships were involved in the battle. If there are more or fewer than two countries involved, produce `NULL` for both countries.

c) Take as arguments a new class name, type, country, number of guns, bore, and displacement. Add this information to `Classes` and also add the ship with the class name to `Ships`.

! d) Given a ship name, determine if the ship was in a battle with a date before the ship was launched. If so, set the date of the battle and the date the ship was launched to 0.

! **Exercise 9.4.4:** In Fig. 9.15, we used a tricky formula for computing the variance of a sequence of numbers $x_1, x_2, \ldots, x_n$. Recall that the variance is the average square of the deviation of these numbers from their mean. That is, the variance is $(\sum_{i=1}^{n}(x_i - \overline{x})^2)/n$, where the mean $\overline{x}$ is $(\sum_{i=1}^{n} x_i)/n$. Prove that the formula for the variance used in Fig. 9.15, which is

$$(\sum_{i=1}^{n}(x_i)^2)/n - ((\sum_{i=1}^{n} x_i)/n)^2$$

yields the same value.

9.5 Using a Call-Level Interface

When using a *call-level interface* (CLI), we write ordinary host-language code, and we use a library of functions that allow us to connect to and access a database, passing SQL statements to that database. The differences between this approach and embedded SQL programming are, in one sense, cosmetic, since the preprocessor replaces embedded SQL by calls to library functions much like the functions in the standard SQL/CLI.

We shall give three examples of call-level interfaces. In this section, we cover the standard SQL/CLI, which is an adaptation of ODBC (Open Database Connectivity). We cover JDBC, which is a collection of classes that support database access from Java programs. Then, we explore PHP, which is a way to embed database access in Web pages described by HTML.

9.5.1 Introduction to SQL/CLI

A program written in C and using SQL/CLI (hereafter, just CLI) will include the header file `sqlcli.h`, from which it gets a large number of functions, type definitions, structures, and symbolic constants. The program is then able to create and deal with four kinds of records (structs, in C):

1. *Environments.* A record of this type is created by the application (client) program in preparation for one or more connections to the database server.

2. *Connections.* One of these records is created to connect the application program to the database. Each connection exists within some environment.

3. *Statements.* An application program can create one or more statement records. Each holds information about a single SQL statement, including an implied cursor if the statement is a query. At different times, the same CLI statement can represent different SQL statements. Every CLI statement exists within some connection.

4. *Descriptions.* These records hold information about either tuples or parameters. The application program or the database server, as appropriate, sets components of description records to indicate the names and types of attributes and/or their values. Each statement has several of these created implicitly, and the user can create more if needed. In our presentation of CLI, description records will generally be invisible.

Each of these records is represented in the application program by a *handle*, which is a pointer to the record. The header file `sqlcli.h` provides types for the handles of environments, connections, statements, and descriptions: `SQLHENV`, `SQLHDBC`, `SQLHSTMT`, and `SQLHDESC`, respectively, although we may think of them as pointers or integers. We shall use these types and also some other defined types with obvious interpretations, such as `SQL_CHAR` and `SQL_INTEGER`, that are provided in `sqlcli.h`.

We shall not go into detail about how descriptions are set and used. However, (handles for) the other three types of records are created by the use of a function

`SQLAllocHandle(`*hType, hIn, hOut*`)`

Here, the three arguments are:

1. *hType* is the type of handle desired. Use `SQL_HANDLE_ENV` for a new environment, `SQL_HANDLE_DBC` for a new connection, or `SQL_HANDLE_STMT` for a new statement.

2. *hIn* is the handle of the higher-level element in which the newly allocated element lives. This parameter is `SQL_NULL_HANDLE` if you want an environment; the latter name is a defined constant telling `SQLAllocHandle` that there is no relevant value here. If you want a connection handle, then *hIn* is the handle of the environment within which the connection will exist, and if you want a statement handle, then *hIn* is the handle of the connection within which the statement will exist.

3. *hOut* is the address of the handle that is created by `SQLAllocHandle`.

`SQLAllocHandle` also returns a value of type `SQLRETURN` (an integer). This value is 0 if no errors occurred, and there are certain nonzero values returned in the case of errors.

Example 9.18: Let us see how the function `worthRanges` of Fig. 9.8, which we used as an example of embedded SQL, would begin in CLI. Recall this function examines all the tuples of `MovieExec` and breaks their net worths into ranges. The initial steps are shown in Fig. 9.19.

```
1)  #include sqlcli.h
2)  SQLHENV myEnv;
3)  SQLHDBC myCon;
4)  SQLHSTMT execStat;
5)  SQLRETURN errorCode1, errorCode2, errorCode3;

6)  errorCode1 = SQLAllocHandle(SQL_HANDLE_ENV,
        SQL_NULL_HANDLE, &myEnv);
7)  if(!errorCode1) {
8)      errorCode2 = SQLAllocHandle(SQL_HANDLE_DBC,
            myEnv, &myCon);
9)  if(!errorCode2)
10)     errorCode3 = SQLAllocHandle(SQL_HANDLE_STMT,
            myCon, &execStat); }
```

Figure 9.19: Declaring and creating an environment, a connection, and a statement

Lines (2) through (4) declare handles for an environment, connection, and statement, respectively; their names are `myEnv`, `myCon`, and `execStat`, respectively. We plan that `execStat` will represent the SQL statement

```
SELECT netWorth FROM MovieExec;
```

much as did the cursor `execCursor` in Fig. 9.8, but as yet there is no SQL statement associated with `execStat`. Line (5) declares three variables into which function calls can place their response and indicate an error. A value of 0 indicates no error occurred in the call.

Line (6) calls `SQLAllocHandle`, asking for an environment handle (the first argument), providing a null handle in the second argument (because none is needed when we are requesting an environment handle), and providing the address of `myEnv` as the third argument; the generated handle will be placed there. If line (6) is successful, lines (7) and (8) use the environment handle to get a connection handle in `myCon`. Assuming that call is also successful, lines (9) and (10) get a statement handle for `execStat`. □

9.5.2 Processing Statements

At the end of Fig. 9.19, a statement record whose handle is `execStat`, has been created. However, there is as yet no SQL statement with which that record is associated. The process of associating and executing SQL statements with statement handles is analogous to the dynamic SQL described in Section 9.3.9. There, we associated the text of a SQL statement with what we called a "SQL variable," using `PREPARE`, and then executed it using `EXECUTE`.

The situation in CLI is quite analogous, if we think of the "SQL variable" as a statement handle. There is a function

`SQLPrepare(`*sh, st, sl*`)`

that takes:

1. A statement handle *sh*,
2. A pointer to a SQL statement *st*, and
3. A length *sl* for the character string pointed to by *st*. If we don't know the length, a defined constant `SQL_NTS` tells `SQLPrepare` to figure it out from the string itself. Presumably, the string is a "null-terminated string," and it is sufficient for `SQLPrepare` to scan it until encountering the endmarker `'\0'`.

The effect of this function is to arrange that the statement referred to by the handle *sh* now represents the particular SQL statement *st*.

Another function

`SQLExecute(`*sh*`)`

causes the statement to which handle *sh* refers to be executed. For many forms of SQL statement, such as insertions or deletions, the effect of executing this statement on the database is obvious. Less obvious is what happens when the SQL statement referred to by *sh* is a query. As we shall see in Section 9.5.3,

there is an implicit cursor for this statement that is part of the statement record itself. The statement is in principle executed, so we can imagine that all the answer tuples are sitting somewhere, ready to be accessed. We can fetch tuples one at a time, using the implicit cursor, much as we did with real cursors in Sections 9.3 and 9.4.

Example 9.19: Let us continue with the function `worthRanges` that we began in Fig. 9.19. The following two function calls associate the query

```
SELECT netWorth FROM MovieExec;
```

with the statement referred to by handle `execStat`:

```
11)  SQLPrepare(execStat, "SELECT netWorth FROM MovieExec",
         SQL_NTS);
12)  SQLExecute(execStat);
```

These lines could appear right after line (10) of Fig. 9.19. Remember that `SQL_NTS` tells `SQLPrepare` to determine the length of the null-terminated string to which its second argument refers. □

As with dynamic SQL, the prepare and execute steps can be combined into one if we use the function `SQLExecDirect`. An example that combines lines (11) and (12) above is:

```
SQLExecDirect(execStat, "SELECT netWorth FROM MovieExec",
    SQL_NTS);
```

9.5.3 Fetching Data From a Query Result

The function that corresponds to a `FETCH` command in embedded SQL or PSM is

`SQLFetch(`*sh*`)`

where *sh* is a statement handle. We presume the statement referred to by *sh* has been executed already, or the fetch will cause an error. `SQLFetch`, like all CLI functions, returns a value of type `SQLRETURN` that indicates either success or an error. The return value `SQL_NO_DATA` tells us tuples were left in the query result. As in our previous examples of fetching, this value will be used to get us out of a loop in which we repeatedly fetch new tuples from the result.

However, if we follow the `SQLExecute` of Example 9.19 by one or more `SQLFetch` calls, where does the tuple appear? The answer is that its components go into one of the description records associated with the statement whose handle appears in the `SQLFetch` call. We can extract the same component at each fetch by binding the component to a host-language variable, before we begin fetching. The function that does this job is:

`SQLBindCol(`*sh, colNo, colType, pVar, varSize, varInfo*`)`

The meanings of these six arguments are:

1. *sh* is the handle of the statement involved.
2. *colNo* is the number of the component (within the tuple) whose value we obtain.
3. *colType* is a code for the type of the variable into which the value of the component is to be placed. Examples of codes provided by `sqlcli.h` are `SQL_CHAR` for character arrays and strings, and `SQL_INTEGER` for integers.
4. *pVar* is a pointer to the variable into which the value is to be placed.
5. *varSize* is the length in bytes of the value of the variable pointed to by *pVar*.
6. *varInfo* is a pointer to an integer that can be used by `SQLBindCol` to provide additional information about the value produced.

Example 9.20: Let us redo the entire function `worthRanges` from Fig. 9.8, using CLI calls instead of embedded SQL. We begin as in Fig. 9.19, but for the sake of succinctness, we skip all error checking except for the test whether `SQLFetch` indicates that no more tuples are present. The code is shown in Fig. 9.20.

Line (3) declares the same local variables that the embedded-SQL version of the function uses, and lines (4) through (7) declare additional local variables using the types provided in `sqlcli.h`; these are variables that involve SQL in some way. Lines (4) through (6) are as in Fig. 9.19. New are the declarations on line (7) of `worth` (which corresponds to the shared variable of that name in Fig. 9.8) and `worthInfo`, which is required by `SQLBindCol`, but not used.

Lines (8) through (10) allocate the needed handles, as in Fig. 9.19, and lines (11) and (12) prepare and execute the SQL statement, as discussed in Example 9.19. In line (13), we see the binding of the first (and only) column of the result of this query to the variable `worth`. The first argument is the handle for the statement involved, and the second argument is the column involved, 1 in this case. The third argument is the type of the column, and the fourth argument is a pointer to the place where the value will be placed: the variable `worth`. The fifth argument is the size of that variable, and the final argument points to `worthInfo`, a place for `SQLBindCol` to put additional information (which we do not use here).

The balance of the function resembles closely lines (11) through (19) of Fig. 9.8. The while-loop begins at line (14) of Fig. 9.20. Notice that we fetch a tuple and check that we are not out of tuples, all within the condition of the while-loop, on line (14). If there is a tuple, then in lines (15) through (17) we determine the number of digits the integer (which is bound to `worth`) has and increment the appropriate count. After the loop finishes, i.e., all tuples returned

```
1)  #include sqlcli.h
2)  void worthRanges() {

3)      int i, digits, counts[15];
4)      SQLHENV myEnv;
5)      SQLHDBC myCon;
6)      SQLHSTMT execStat;
7)      SQLINTEGER worth, worthInfo;

8)      SQLAllocHandle(SQL_HANDLE_ENV,
            SQL_NULL_HANDLE, &myEnv);
9)      SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
10)     SQLAllocHandle(SQL_HANDLE_STMT, myCon, &execStat);
11)     SQLPrepare(execStat,
            "SELECT netWorth FROM MovieExec", SQL_NTS);
12)     SQLExecute(execStat);
13)     SQLBindCol(execStat, 1, SQL_INTEGER, &worth,
            sizeof(worth), &worthInfo);
14)     while(SQLFetch(execStat) != SQL_NO_DATA) {
15)         digits = 1;
16)         while((worth /= 10) > 0) digits++;
17)         if(digits <= 14) counts[digits]++;
        }
18)     for(i=0; i<15; i++)
19)         printf("digits = %d: number of execs = %d\n",
                i, counts[i]);
    }
```

Figure 9.20: Grouping executive net worths: CLI version

by the statement execution of line (12) have been examined, the resulting counts are printed out at lines (18) and (19). □

9.5.4 Passing Parameters to Queries

Embedded SQL gives us the ability to execute a SQL statement, part of which consists of values determined by the current contents of shared variables. There is a similar capability in CLI, but it is rather more complicated. The steps needed are:

1. Use `SQLPrepare` to prepare a statement in which some portions, called *parameters*, are replaced by a question-mark. The ith question-mark represents the ith parameter.

Extracting Components with SQLGetData

An alternative to binding a program variable to an output of a query's result relation is to fetch tuples without any binding and then transfer components to program variables as needed. The function to use is `SQLGetData`, and it takes the same arguments as `SQLBindCol`. However, it only copies data once, and it must be used after each fetch in order to have the same effect as initially binding the column to a variable.

2. Use function `SQLBindParameter` to bind values to the places where the question-marks are found. This function has ten arguments, of which we shall explain only the essentials.
3. Execute the query with these bindings, by calling `SQLExecute`. Note that if we change the values of one or more parameters, we need to call `SQLExecute` again.

The following example will illustrate the process, as well as indicate the important arguments needed by `SQLBindParameter`.

Example 9.21: Let us reconsider the embedded SQL code of Fig. 9.6, where we obtained values for two variables `studioName` and `studioAddr` and used them as the components of a tuple, which we inserted into `Studio`. Figure 9.21 sketches how this process would work in CLI. It assumes that we have a statement handle `myStat` to use for the insertion statement.

```
    /* get values for studioName and studioAddr */

1)  SQLPrepare(myStat,
        "INSERT INTO Studio(name, address) VALUES(?, ?)",
        SQL_NTS);
2)  SQLBindParameter(myStat, 1,..., studioName,...);
3)  SQLBindParameter(myStat, 2,..., studioAddr,...);
4)  SQLExecute(myStat);
```

Figure 9.21: Inserting a new studio by binding parameters to values

The code begins with steps (not shown) to give `studioName` and `studioAddr` values. Line (1) shows statement `myStat` being prepared to be an insertion statement with two parameters (the question-marks) in the `VALUE` clause. Then, lines (2) and (3) bind the first and second question-marks, to the current contents of `studioName` and `studioAddr`, respectively. Finally, line (4) executes the insertion. If the entire sequence of steps in Fig. 9.21, including the unseen work to obtain new values for `studioName` and `studioAddr`, are placed

in a loop, then each time around the loop, a new tuple, with a new name and address for a studio, is inserted into `Studio`. □

9.5.5 Exercises for Section 9.5

Exercise 9.5.1: Repeat the problems of Exercise 9.3.1, but write the code in C with CLI calls.

Exercise 9.5.2: Repeat the problems of Exercise 9.3.2, but write the code in C with CLI calls.

9.6 JDBC

Java Database Connectivity, or JDBC, is a facility similar to CLI for allowing Java programs to access SQL databases. The concepts resemble those of CLI, although Java's object-oriented flavor is evident in JDBC.

9.6.1 Introduction to JDBC

The first steps we must take to use JDBC are:

1. include the line:

   ```
   import java.sql.*;
   ```

   to make the JDBC classes available to your Java program.

2. Load a "driver" for the database system we shall use. The driver we need depends on which DBMS is available to us, but we load the needed driver with the statement:

   `Class.forName(`<driver name>`);`

   For example, to get the driver for a MySQL database, execute:

   ```
   Class.forName("com.mysql.jdbc.Driver");
   ```

   The effect is that a class called `DriverManager` is available. This class is analogous in many ways to the environment whose handle we get as the first step in using CLI.

3. Establish a connection to the database. A variable of class Connection is created if we apply the method `getConnection` to `DriverManager`.

The Java statement to establish a connection looks like:

```
Connection myCon = DriverManager.getConnection(<URL>,
                    <user name>, <password>);
```

That is, the method `getConnection` takes as arguments the URL for the database to which you wish to connect, your user name, and your password. It returns an object of class Connection, which we have chosen to call `myCon`.

Example 9.22: Each DBMS has its own way of specifying the URL in the `getConnection` method. For instance, if you want to connect to a MySQL database, the form of the URL is

```
jdbc:mysql://<host name>/<database name>
```

□

A JDBC Connection object is quite analogous to a CLI connection, and it serves the same purpose. By applying the appropriate methods to a Connection like `myCon`, we can create statement objects, place SQL statements "in" those objects, bind values to SQL statement parameters, execute the SQL statements, and examine results a tuple at a time.

9.6.2 Creating Statements in JDBC

There are two methods we can apply to a Connection object in order to create statements:

1. `createStatement()` returns a Statement object. This object has no associated SQL statement yet, so method `createStatement()` may be thought of as analogous to the CLI call to `SQLAllocHandle` that takes a connection handle and returns a statement handle.

2. `prepareStatement(`Q`)`, where Q is a SQL query passed as a string argument, returns a PreparedStatement object. Thus, we may draw an analogy between executing `prepareStatement(`Q`)` in JDBC with the two CLI steps in which we get a statement handle with `SQLAllocHandle` and then apply `SQLPrepare` to that handle and the query Q.

There are four different methods that execute SQL statements. Like the methods above, they differ in whether or not they take a SQL statement as an argument. However, these methods also distinguish between SQL statements that are queries and other statements, which are collectively called "updates." Note that the SQL `UPDATE` statement is only one small example of what JDBC terms an "update." The latter include all modification statements, such as inserts, and all schema-related statements such as `CREATE TABLE`. The four "execute" methods are:

a) `executeQuery(`Q`)` takes a statement Q, which must be a query, and is applied to a Statement object. This method returns a ResultSet object, which is the set (bag, to be precise) of tuples produced by the query Q. We shall see how to access these tuples in Section 9.6.3.

b) `executeQuery()` is applied to a PreparedStatement object. Since a prepared statement already has an associated query, there is no argument. This method also returns a ResultSet object.

c) `executeUpdate(`U`)` takes a nonquery statement U and, when applied to a Statement object, executes U. The effect is felt on the database only; no ResultSet object is returned.

d) `executeUpdate()`, with no argument, is applied to a PreparedStatement object. In that case, the SQL statement associated with the prepared statement is executed. This SQL statement must not be a query, of course.

Example 9.23: Suppose we have a Connection object `myCon`, and we wish to execute the query

```
SELECT netWorth FROM MovieExec;
```

One way to do so is to create a Statement object `execStat`, and then use it to execute the query directly.

```
Statement execStat = myCon.createStatement();
ResultSet worths = execStat.executeQuery(
    "SELECT netWorth FROM MovieExec");
```

The result of the query is a ResultSet object, which we have named `worths`. We'll see in Section 9.6.3 how to extract the tuples from `worths` and process them.

An alternative is to prepare the query immediately and later execute it. This approach would be preferable should we want to execute the same query repeatedly. Then, it makes sense to prepare it once and execute it many times, rather than having the DBMS prepare the same query many times. The JDBC steps needed to follow this approach are:

```
PreparedStatement execStat = myCon.prepareStatement(
    "SELECT netWorth FROM MovieExec");
ResultSet worths = execStat.executeQuery();
```

The result of executing the query is again a ResultSet object, which we have called `worths`. □

Example 9.24: If we want to execute a parameterless nonquery, we can perform analogous steps in both styles. There is no result set, however. For instance, suppose we want to insert into `StarsIn` the fact that Denzel Washington starred in *Remember the Titans* in the year 2000. We may create and use a statement `starStat` in either of the following ways:

```
Statement starStat = myCon.createStatement();
starStat.executeUpdate("INSERT INTO StarsIn VALUES(" +
    "'Remember the Titans', 2000, 'Denzel Washington')");
```

or

```
PreparedStatement starStat = myCon.prepareStatement(
    "INSERT INTO StarsIn VALUES('Remember the Titans'," +
    "2000, 'Denzel Washington')");
starStat.executeUpdate();
```

Notice that each of these sequences of Java statements takes advantage of the fact that + is the Java operator that concatenates strings. Thus, we are able to extend SQL statements over several lines of Java, as needed. □

9.6.3 Cursor Operations in JDBC

When we execute a query and obtain a result-set object, we may, in effect, run a cursor through the tuples of the result set. To do so, the `ResultSet` class provides the following useful methods:

1. `next()`, when applied to a ResultSet object, causes an implicit cursor to move to the next tuple (to the first tuple the first time it is applied). This method returns `FALSE` if there is no next tuple.

2. `getString(`i`)`, `getInt(`i`)`, `getFloat(`i`)`, and analogous methods for the other types that SQL values can take, each return the ith component of the tuple currently indicated by the cursor. The method appropriate to the type of the ith component must be used.

Example 9.25: Having obtained the result set `worths` as in Example 9.23, we may access its tuples one at a time. Recall that these tuples have only one component, of type integer. The form of the loop is:

```
while(worths.next()) {
    int worth = worths.getInt(1);
    /* process this net worth */
};
```

□

9.6.4 Parameter Passing

As in CLI, we can use a question-mark in place of a portion of a query, and then bind values to those *parameters*. To do so in JDBC, we need to create a prepared statement, and we need to apply to that PreparedStatement object methods such as `setString(`i`, `v`)` or `setInt(`i`, `v`)` that bind the value v, which must be of the appropriate type for the method, to the ith parameter in the query.

Example 9.26: Let us mimic the CLI code in Example 9.21, where we prepared a statement to insert a new studio into relation `Studio`, with parameters for the name and address of that studio. The Java code to prepare this statement, set its parameters, and execute it is shown in Fig. 9.22. We continue to assume that connection object `myCon` is available to us.

```
1)  PreparedStatement studioStat = myCon.prepareStatement(
2)      "INSERT INTO Studio(name, address) VALUES(?, ?)");
    /* get values for variables studioName and studioAddr
       from the user */
3)  studioStat.setString(1, studioName);
4)  studioStat.setString(2, studioAddr);
5)  studioStat.executeUpdate();
```

Figure 9.22: Setting and using parameters in JDBC

In lines (1) and (2), we create and prepare the insertion statement. It has parameters for each of the values to be inserted. After line (2), we could begin a loop in which we repeatedly ask the user for a studio name and address, and place these strings in the variables `studioName` and `studioAddr`. This assignment is not shown, but represented by a comment. Lines (3) and (4) set the first and second parameters to the strings that are the current values of `studioName` and `studioAddr`, respectively. Finally, at line (5), we execute the insertion statement with the current values of its parameters. After line (5), we could go around the loop again, beginning with the steps represented by the comment. □

9.6.5 Exercises for Section 9.6

Exercise 9.6.1: Repeat Exercise 9.3.1, but write the code in Java using JDBC.

Exercise 9.6.2: Repeat Exercise 9.3.2, but write the code in Java using JDBC.

9.7 PHP

PHP is a scripting language for helping to create HTML Web pages. It provides support for database operations through an available library, much as JDBC

What Does PHP Stand For?

Originally, PHP was an acronym for "Personal Home Page." More recently, it is said to be the recursive acronym "PHP: Hypertext Preprocessor" in the spirit of other recursive acronyms such as GNU (= "GNU is Not Unix").

does. In this section we shall give a brief overview of PHP and show how database operations are performed in this language.

9.7.1 PHP Basics

All PHP code is intended to exist inside HTML text. A browser will recognize that text is PHP code by placing it inside a special tag, which looks like:

```
<?php
   PHP code goes here
?>
```

Many aspects of PHP, such as assignment statements, branches, and loops, will be familiar to the C or Java programmer, and we shall not cover them explicitly. However, there are some interesting features of PHP of which we should be aware.

Variables

Variables are untyped and need not be declared. All variable names begin with $.

Often, a variable will be declared to be a member of a "class," in which case certain *functions* (analogous to methods in Java) may be applied to that variable. The function-application operator is ->, comparable to the dot in Java or C++.

Strings

String values in PHP can be surrounded by either single or double quotes, but there is an important difference. Strings surrounded by single quotes are treated literally, just like SQL strings. However, when a string has double quotes around it, any variable names within the string are replaced by their values.

Example 9.27: In the following code:

```
$foo = 'bar';
$x = 'Step up to the $foo';
```

the value of `$x` is `Step up to the $foo`. However, if the following code is executed instead:

```
$foo = "bar";
$x = "Step up to the $foo";
```

the value of `$x` is `Step up to the bar`. It doesn't matter whether `bar` has single or double quotes, since it contains no dollar-signs and therefore no variables. However, the variable `$foo` is replaced only when surrounded by double quotes, as in the second example. □

Concatenation of strings is denoted by a dot. Thus,

```
$y = "$foo" . 'bar';
```

gives `$y` the value `barbar`.

9.7.2 Arrays

PHP has ordinary arrays (called *numeric*), which are indexed $0, 1, \ldots$. It also has arrays that are really mappings, called *associative arrays*. The indexes (*keys*) of an associative array can be any strings, and the array associates a single value with each key. Both kinds of arrays use the conventional square brackets for indexing, but for associative arrays, an array element is represented by:

<key> => <value>

Example 9.28: The following line:

```
$a = array(30,20,10,0);
```

sets `$a` to be a numeric array of length four, with `$a[0]` equal to 30, `$a[1]` equal to 20, and so on. □

Example 9.29: The following line:

```
$seasons = array('spring' => 'warm',  'summer' => 'hot',
                 'fall' => 'warm', 'winter' => 'cold');
```

makes `$seasons` be an array of length four, but it is an associative array. For instance, `$seasons['summer']` has the value `'hot'`. □

9.7.3 The PEAR DB Library

PHP has a collection of libraries called PEAR (PHP Extension and Application Repository). One of these libraries, DB, has generic functions that are analogous to the methods of JDBC. We tell the function `DB::connect` which vendor's DBMS we wish to access, but none of the other functions of DB need to know about which DBMS we are using. Note that the double colon in `DB::connect` is PHP's way of saying "the function `connect` in the DB library." We make the DB library available to our PHP program with the statement:

```
include(DB.php);
```

9.7.4 Creating a Database Connection Using DB

The form of an invocation of the `connect` function is:

```
$myCon = DB::connect(<vendor>://<user name>:<password>
                <host name>/<database name>);
```

The components of this call are like those in the analogous JDBC statement that creates a connection (see Section 9.6.1). The one exception is the vendor, which is a code used by the DB library. For example, `mysqli` is the code for recent versions of the MySQL database.

After executing this statement, the variable `$myCon` is a connection. Like all PHP variables, `$myCon` can change its type. But as long as it is a connection, we may apply to it a number of useful functions that enable us to manipulate the database to which the connection was made. For example, we can disconnect from the database by

```
$myCon->disconnect();
```

Remember that -> is the PHP way of applying a function to an "object."

9.7.5 Executing SQL Statements

All SQL statements are referred to as "queries" and are executed by the function `query`, which takes the statement as an argument and is applied to the connection variable.

Example 9.30: Let us duplicate the insertion statement of Example 9.24, where we inserted Denzel Washington and *Remember the Titans* into the `StarsIn` table. Assuming that `$myCon` has connected to our movie database, We can simply say:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(" .
    "'Denzel Washington', 2000, 'Remember the Titans')");
```

Note that the dot concatenates the two strings that form the query. We only broke the query into two strings because it was necessary to break it over two lines.

The variable `$result` will hold an error code if the insert-statement failed to execute. If the "query" were really a SQL query, then `$result` is a cursor to the tuples of the result (see Section 9.7.6). □

PHP allows SQL to have parameters, denoted by question-marks, as we shall discuss in Section 9.7.7. However, the ability to expand variables in doubly quoted strings gives us another easy way to execute SQL statements that depend on user input. In particular, since PHP is used within Web pages, there are built-in ways to exploit HTML's capabilities.

We often get information from a user of a Web page by showing them a form and having their answers "posted." PHP provides an associative array called `$_POST` with all the information provided by the user. Its keys are the names of the form elements, and the associated values are what the user has entered into the form.

Example 9.31: Suppose we ask the user to fill out a form whose elements are `title`, `year`, and `starName`. These three values will form a tuple that we may insert into the table `StarsIn`. The statement:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(
    $_POST['title'], $_POST['year'], $_POST['starName'])");
```

will obtain the posted values for these three form elements. Since the query argument is a double-quoted string, PHP evaluates terms like `$_POST['title']` and replaces them by their values. □

9.7.6 Cursor Operations in PHP

When the `query` function gets a true query as argument, it returns a result object, that is, a list of tuples. Each tuple is a numeric array, indexed by integers starting at 0. The essential function that we can apply to a result object is `fetchRow()`, which returns the next row, or 0 (false) if there is no next row.

```
1) $worths = $myCon->query("SELECT netWorth FROM MovieExec");
2) while ($tuple = $worths->fetchRow()) {
3)     $worth = $tuple[0];
          // process this value of $worth
   }
```

Figure 9.23: Finding and processing net worths in PHP

Example 9.32: In Fig. 9.23 is PHP code that is the equivalent of the JDBC in Examples 9.23 and 9.25. It assumes that connection `$myCon` is available, as before.

Line (1) passes the query to the connection `$myCon`, and the result object is assigned to the variable `$worths`. We then enter a loop, in which we repeatedly get a tuple from the result and assign this tuple to the variable `$tuple`, which technically becomes an array of length 1, with only a component for the column `netWorth`. As in C, the value returned by `fetchRow()` becomes the value of the condition in the while-statement. Thus, if no tuple is found, this value, 0, terminates the loop. At line (3), the value of the tuple's first (and only) component is extracted and assigned to the variable `$worth`. We do not show the processing of this value. □

9.7.7 Dynamic SQL in PHP

As in JDBC, PHP allows a SQL query to contain question-marks. These question-marks are placeholders for values that can be filled in later, during the execution of the statement. The process of doing so is as follows.

We may apply `prepare` and `execute` functions to a connection; these functions are analogous to similarly named functions discussed in Section 9.3.9 and elsewhere. Function `prepare` takes a SQL statement as argument and returns a prepared version of that statement. Function `execute` takes two arguments: the prepared statement and an array of values to substitute for the question-marks in the statement. If there is only one question-mark, a simple variable, rather than an array, suffices.

Example 9.33: Let us again look at the problem of Example 9.26, where we prepared to insert many name-address pairs into relation `Studio`. To begin, we prepare the query, with parameters, by:

```
$prepQuery = $myCon->prepare("INSERT INTO Studio(name, " .
                    "address) VALUES(?,?)");
```

Now, `$prepQuery` is a "prepared query." We can use it as an argument to `execute` along with an array of two values, a studio name and address. For example, we could perform the following statements:

```
$args = array('MGM', 'Los Angeles');
$result = $myCon->execute($prepQuery, $args);
```

The advantage of this arrangement is the same as for all implementations of dynamic SQL. If we insert many different tuples this way, we only have to prepare the insertion statement once and can execute it many times. □

9.7.8 Exercises for Section 9.7

Exercise 9.7.1: Repeat Exercise 9.3.1, but write the code using PHP.

Exercise 9.7.2: Repeat Exercise 9.3.2, but write the code using PHP.

! **Exercise 9.7.3:** In Example 9.31 we exploited the feature of PHP that strings in double-quotes have variables expanded. How essential is this feature? Could we have done something analogous in JDBC? If so, how?

9.8 Summary of Chapter 9

- *Three-Tier Architectures*: Large database installations that support large-scale user interactions over the Web commonly use three tiers of processes: web servers, application servers, and database servers. There can be many processes active at each tier, and these processes can be at one processor or distributed over many processors.

- *Client-Server Systems in the SQL Standard*: The standard talks of SQL clients connecting to SQL servers, creating a connection (link between the two processes) and a session (sequence of operations). The code executed during the session comes from a module, and the execution of the module is called a SQL agent.

- *The Database Environment*: An installation using a SQL DBMS creates a SQL environment. Within the environment, database elements such as relations are grouped into (database) schemas, catalogs, and clusters. A catalog is a collection of schemas, and a cluster is the largest collection of elements that one user may see.

- *Impedance Mismatch*: The data model of SQL is quite different from the data models of conventional host languages. Thus, information passes between SQL and the host language through shared variables that can represent components of tuples in the SQL portion of the program.

- *Embedded SQL*: Instead of using a generic query interface to express SQL queries and modifications, it is often more effective to write programs that embed SQL queries in a conventional host language. A preprocessor converts the embedded SQL statements into suitable function calls of the host language.

- *Cursors*: A cursor is a SQL variable that indicates one of the tuples of a relation. Connection between the host language and SQL is facilitated by having the cursor range over each tuple of the relation, while the components of the current tuple are retrieved into shared variables and processed using the host language.

- ✦ *Dynamic SQL*: Instead of embedding particular SQL statements in a host-language program, the host program may create character strings that are interpreted by the SQL system as SQL statements and executed.
- ✦ *Persistent Stored Modules*: We may create collections of procedures and functions as part of a database schema. These are written in a special language that has all the familiar control primitives, as well as SQL statements.
- • *The Call-Level Interface*: There is a standard library of functions, called SQL/CLI or ODBC, that can be linked into any C program. These functions give capabilities similar to embedded SQL, but without the need for a preprocessor.
- ✦ *JDBC*: Java Database Connectivity is a collection of Java classes analogous to CLI for connecting Java programs to a database.
- ✦ *PHP*: Another popular system for implementing a call-level interface is PHP. This language is found embedded in HTML pages and enables these pages to interact with a database.

9.9 References for Chapter 9

The PSM standard is [4], and [5] is a comprehensive book on the subject. Oracle's version of PSM is called PL/SQL; a summary can be found in [2]. SQL Server has a version called Transact-SQL [6]. IBM's version is SQL PL [1].

[3] is a popular reference on JDBC. [7] is one on PHP, which was originally developed by one of the book's authors, R. Lerdorf.

1. D. Bradstock et al., *DB2 SQL Procedure Language for Linux, Unix, and Windows*, IBM Press, 2005.
2. Y.-M. Chang et al., "Using Oracle PL/SQL"
   `http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html`
3. M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, Prentice-Hall, Upper Saddle River, NJ, 2003.
4. ISO/IEC Report 9075-4, 2003.
5. J. Melton, *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*, Morgan-Kaufmann, San Francisco, 1998.
6. Microsoft Corp., "Transact-SQL Reference"
   `http://msdn2.microsoft.com/en-us/library/ms189826.aspx`
7. K. Tatroe, R. Lerdorf, and P. MacIntyre, *Programming PHP*, O'Reilly Media, Cambridge, MA, 2006.

Chapter 10

Advanced Topics in Relational Databases

This chapter introduces additional topics that are of interest to the database programmer. We begin with a section on the SQL standard for authorization of access to database elements. Next, we see the SQL extension that allows for recursive programming in SQL — queries that use their own results. Then, we look at the object-relational model, and how it is implemented in the SQL standard.

The remainder of the chapter concerns "OLAP," or on-line analytic processing. OLAP refers to complex queries of a nature that causes them to take significant time to execute. Because they are so expensive, some special technology has developed to handle them efficiently. One important direction is an implementation of relations, called the "data cube," that is rather different from the conventional bag-of-tuples approach of SQL.

10.1 Security and User Authorization in SQL

SQL postulates the existence of *authorization ID's*, which are essentially user names. SQL also has a special authorization ID called `PUBLIC`, which includes any user. Authorization ID's may be granted privileges, much as they would be in the file system environment maintained by an operating system. For example, a UNIX system generally controls three kinds of privileges: read, write, and execute. That list of privileges makes sense, because the protected objects of a UNIX system are files, and these three operations characterize well the things one typically does with files. However, databases are much more complex than file systems, and the kinds of privileges used in SQL are correspondingly more complex.

In this section, we shall first learn what privileges SQL allows on database elements. We shall then see how privileges may be acquired by users (by au-

thorization ID's, that is). Finally, we shall see how privileges may be taken away.

10.1.1 Privileges

SQL defines nine types of privileges: SELECT, INSERT, DELETE, UPDATE, REFERENCES, USAGE, TRIGGER, EXECUTE, and UNDER. The first four of these apply to a relation, which may be either a base table or a view. As their names imply, they give the holder of the privilege the right to query (select from) the relation, insert into the relation, delete from the relation, and update tuples of the relation, respectively.

A SQL statement cannot be executed without the privileges appropriate to that statement; e.g., a select-from-where statement requires the SELECT privilege on every table it accesses. We shall see how the module can get those privileges shortly. SELECT, INSERT, and UPDATE may also have an associated list of attributes, for instance, `SELECT(name, addr)`. If so, then only those attributes may be seen in a selection, specified in an insertion, or changed in an update, respectively. Note that, when granted, privileges such as these will be associated with a particular relation, so it will be clear at that time to what relation attributes `name` and `addr` belong.

The REFERENCES privilege on a relation is the right to refer to that relation in an integrity constraint. These constraints may take any of the forms mentioned in Chapter 7, such as assertions, attribute- or tuple-based checks, or referential integrity constraints. The REFERENCES privilege may also have an attached list of attributes, in which case only those attributes may be referenced in a constraint. A constraint cannot be created unless the owner of the schema in which the constraint appears has the REFERENCES privilege on all data involved in the constraint.

USAGE is a privilege that applies to several kinds of schema elements other than relations and assertions (see Section 9.2.2); it is the right to use that element in one's own declarations. The TRIGGER privilege on a relation is the right to define triggers on that relation. EXECUTE is the right to execute a piece of code, such as a PSM procedure or function. Finally, UNDER is the right to create subtypes of a given type. The matter of types appears in Section 10.4.

Example 10.1: Let us consider what privileges are needed to execute the insertion statement of Fig. 6.15, which we reproduce here as Fig. 10.1. First, it is an insertion into the relation `Studio`, so we require an INSERT privilege on `Studio`. However, since the insertion specifies only the component for attribute `name`, it is acceptable to have either the privilege INSERT or the privilege `INSERT(name)` on relation `Studio`. The latter privilege allows us to insert `Studio` tuples that specify only the `name` component and leave other components to take their default value or NULL, which is what Fig. 10.1 does.

However, notice that the insertion statement of Fig. 10.1 involves two subqueries, starting at lines (2) and (5). To carry out these selections we require

Triggers and Privileges

It is a bit subtle how privileges are handled for triggers. First, if you have the TRIGGER privilege for a relation, you can attempt to create any trigger you like on that relation. However, since the condition and action portions of the trigger are likely to query and/or modify portions of the database, the trigger creator must have the necessary privileges for those actions. When someone performs an activity that awakens the trigger, they do not need the privileges that the trigger condition and action require; the trigger is executed under the privileges of its creator.

```
1) INSERT INTO Studio(name)
2)     SELECT DISTINCT studioName
3)     FROM Movies
4)     WHERE studioName NOT IN
5)         (SELECT name
6)          FROM Studio);
```

Figure 10.1: Adding new studios

the privileges needed for the subqueries. Thus, we need the SELECT privilege on both relations involved in FROM clauses: Movies and Studio. Note that just because we have the INSERT privilege on Studio doesn't mean we have the SELECT privilege on Studio, or vice versa. Since it is only particular attributes of Movies and Studio that get selected, it is sufficient to have the privilege SELECT(studioName) on Movies and the privilege SELECT(name) on Studio, or privileges that include these attributes within a list of attributes. □

10.1.2 Creating Privileges

There are two aspects to the awarding of privileges: how they are created initially, and how they are passed from user to user. We shall discuss initialization here and the transmission of privileges in Section 10.1.4.

First, SQL elements such as schemas or modules have an owner. The owner of something has all privileges associated with that thing. There are three points at which ownership is established in SQL.

1. When a schema is created, it and all the tables and other schema elements in it are owned by the user who created it. This user thus has all possible privileges on elements of the schema.

2. When a session is initiated by a CONNECT statement, there is an opportunity to indicate the user with an AUTHORIZATION clause. For instance,

the connection statement

```
CONNECT TO Starfleet-sql-server AS conn1
    AUTHORIZATION kirk;
```

would create a connection called `conn1` to a database server whose name is `Starfleet-sql-server`, on behalf of user `kirk`. Presumably, the SQL implementation would verify that the user name is valid, for example by asking for a password. It is also possible to include the password in the `AUTHORIZATION` clause, as we discussed in Section 9.2.5. That approach is somewhat insecure, since passwords are then visible to someone looking over Kirk's shoulder.

3. When a module is created, there is an option to give it an owner by using an `AUTHORIZATION` clause. For instance, a clause

```
AUTHORIZATION picard;
```

in a module-creation statement would make user `picard` the owner of the module. It is also acceptable to specify no owner for a module, in which case the module is publicly executable, but the privileges necessary for executing any operations in the module must come from some other source, such as the user associated with the connection and session during which the module is executed.

10.1.3 The Privilege-Checking Process

As we saw above, each module, schema, and session has an associated user; in SQL terms, there is an associated authorization ID for each. Any SQL operation has two parties:

1. The database elements upon which the operation is performed and
2. The agent that causes the operation.

The privileges available to the agent derive from a particular authorization ID called the *current authorization ID*. That ID is either

a) The module authorization ID, if the module that the agent is executing has an authorization ID, or

b) The session authorization ID if not.

We may execute the SQL operation only if the current authorization ID possesses all the privileges needed to carry out the operation on the database elements involved.

Example 10.2: To see the mechanics of checking privileges, let us reconsider Example 10.1. We might suppose that the referenced tables — `Movies` and `Studio` — are part of a schema called `MovieSchema`, which was created by and is owned by user `janeway`. At this point, user `janeway` has all privileges on these tables and any other elements of the schema `MovieSchema`. She may choose to grant some privileges to others by the mechanism to be described in Section 10.1.4, but let us assume none have been granted yet. There are several ways that the insertion of Example 10.1 can be executed.

1. The insertion could be executed as part of a module created by user `janeway` and containing an `AUTHORIZATION janeway` clause. The module authorization ID, if there is one, always becomes the current authorization ID. Then, the module and its SQL insertion statement have exactly the same privileges user `janeway` has, which includes all privileges on the tables `Movies` and `Studio`.

2. The insertion could be part of a module that has no owner. User `janeway` opens a connection with an `AUTHORIZATION janeway` clause in the `CONNECT` statement. Now, `janeway` is again the current authorization ID, so the insertion statement has all the privileges needed.

3. User `janeway` grants all privileges on tables `Movies` and `Studio` to user `archer`, or perhaps to the special user `PUBLIC`, which stands for "all users." Suppose the insertion statement is in a module with the clause

   ```
   AUTHORIZATION archer
   ```

   Since the current authorization ID is now `archer`, and this user has the needed privileges, the insertion is again permitted.

4. As in (3), suppose user `janeway` has given user `archer` the needed privileges. Also, suppose the insertion statement is in a module without an owner; it is executed in a session whose authorization ID was set by an `AUTHORIZATION archer` clause. The current authorization ID is thus `archer`, and that ID has the needed privileges.

□

There are several principles that are illustrated by Example 10.2. We shall summarize them below.

- The needed privileges are always available if the data is owned by the same user as the user whose ID is the current authorization ID. Scenarios (1) and (2) above illustrate this point.

- The needed privileges are available if the user whose ID is the current authorization ID has been granted those privileges by the owner of the data, or if the privileges have been granted to user `PUBLIC`. Scenarios (3) and (4) illustrate this point.

- Executing a module owned by the owner of the data, or by someone who has been granted privileges on the data, makes the needed privileges available. Of course, one needs the EXECUTE privilege on the module itself. Scenarios (1) and (3) illustrate this point.

- Executing a publicly available module during a session whose authorization ID is that of a user with the needed privileges is another way to execute the operation legally. Scenarios (2) and (4) illustrate this point.

10.1.4 Granting Privileges

So far, the only way we have seen to have privileges on a database element is to be the creator and owner of that element. SQL provides a GRANT statement to allow one user to give a privilege to another. The first user retains the privilege granted, as well; thus GRANT can be thought of as "copy a privilege."

There is one important difference between granting privileges and copying. Each privilege has an associated *grant option*. That is, one user may have a privilege like SELECT on table Movies "with grant option," while a second user may have the same privilege, but without the grant option. Then the first user may grant the privilege SELECT on Movies to a third user, and moreover that grant may be with or without the grant option. However, the second user, who does not have the grant option, may not grant the privilege SELECT on Movies to anyone else. If the third user got the privilege with the grant option, then that user may grant the privilege to a fourth user, again with or without the grant option, and so on.

A *grant statement* has the form:

GRANT <privilege list> ON <database element> TO <user list>

possibly followed by WITH GRANT OPTION.

The database element is typically a relation, either a base table or a view. If it is another kind of element, the name of the element is preceded by the type of that element, e.g., ASSERTION. The privilege list is a list of one or more privileges, e.g., SELECT or INSERT(name). Optionally, the keywords ALL PRIVILEGES may appear here, as a shorthand for all the privileges that the grantor may legally grant on the database element in question.

In order to execute this grant statement legally, the user executing it must possess the privileges granted, and these privileges must be held with the grant option. However, the grantor may hold a more general privilege (with the grant option) than the privilege granted. For instance, the privilege INSERT(name) on table Studio might be granted, while the grantor holds the more general privilege INSERT on Studio, with grant option.

Example 10.3: User janeway, who is the owner of the MovieSchema schema that contains tables

```
Movies(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

grants the INSERT and SELECT privileges on table Studio and privilege SELECT on Movies to users kirk and picard. Moreover, she includes the grant option with these privileges. The grant statements are:

```
GRANT SELECT, INSERT ON Studio TO kirk, picard
    WITH GRANT OPTION;
GRANT SELECT ON Movies TO kirk, picard
    WITH GRANT OPTION;
```

Now, picard grants to user sisko the same privileges, but without the grant option. The statements executed by picard are:

```
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Also, kirk grants to sisko the minimal privileges needed for the insertion of Fig. 10.1, namely SELECT and INSERT(name) on Studio and SELECT on Movies. The statements are:

```
GRANT SELECT, INSERT(name) ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Note that sisko has received the SELECT privilege on Movies and Studio from two different users. He has also received the INSERT(name) privilege on Studio twice: directly from kirk and via the generalized privilege INSERT from picard. □

10.1.5 Grant Diagrams

Because of the complex web of grants and overlapping privileges that may result from a sequence of grants, it is useful to represent grants by a graph called a *grant diagram*. A SQL system maintains a representation of this diagram to keep track of both privileges and their origins (in case a privilege is revoked; see Section 10.1.6).

The nodes of a grant diagram correspond to a user and a privilege. Note that the ability to do something (e.g., SELECT on relation R) with the grant option and the same ability without the grant option are different privileges. These two different privileges, even if they belong to the same user, must be represented by two different nodes. Likewise, a user may hold two privileges, one of which is strictly more general than the other (e.g., SELECT on R and SELECT on $R(A)$. These two privileges are also represented by two different nodes.

If user U grants privilege P to user V, and this grant was based on the fact that U holds privilege Q (Q could be P with the grant option, or it could be

some generalization of P, again with the grant option), then we draw an arc from the node for U/Q to the node for V/P. As we shall see, privileges may be lost when arcs of this graph are deleted. That is why we use separate nodes for a pair of privileges, one of which includes the other, such as a privilege with and without the grant option. If the more powerful privilege is lost, the less powerful one might still be retained.

Example 10.4: Figure 10.2 shows the grant diagram that results from the sequence of grant statements of Example 10.3. We use the convention that a * after a user-privilege combination indicates that the privilege includes the grant option. Also, ** after a user-privilege combination indicates that the privilege derives from ownership of the database element in question and was not due to a grant of the privilege from elsewhere. This distinction will prove important when we discuss revoking privileges in Section 10.1.6. A doubly starred privilege automatically includes the grant option. □

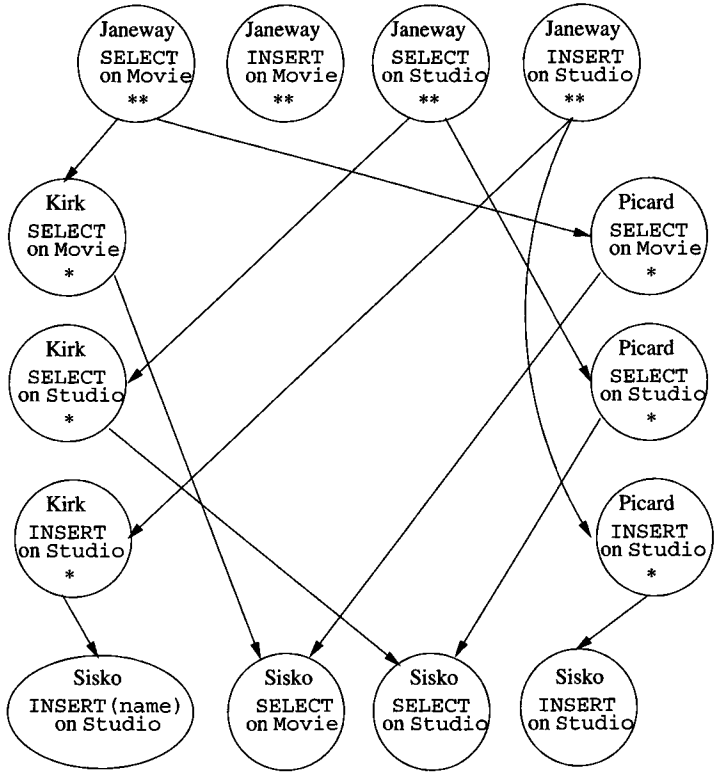


Figure 10.2: A grant diagram

10.1.6 Revoking Privileges

A granted privilege can be revoked at any time. The revoking of privileges may be required to *cascade*, in the sense that revoking a privilege with the grant option that has been passed on to other users may require those privileges to be revoked too. The simple form of a *revoke statement* begins:

`REVOKE` <privilege list> `ON` <database element> `FROM` <user list>

The statement ends with one of the following:

1. `CASCADE`. If chosen, then when the specified privileges are revoked, we also revoke any privileges that were granted *only* because of the revoked privileges. More precisely, if user U has revoked privilege P from user V, based on privilege Q belonging to U, then we delete the arc in the grant diagram from U/Q to V/P. Now, any node that is not accessible from some ownership node (doubly starred node) is also deleted.

2. `RESTRICT`. In this case, the revoke statement cannot be executed if the cascading rule described in the previous item would result in the revoking of any privileges due to the revoked privileges having been passed on to others.

It is permissible to replace `REVOKE` by `REVOKE GRANT OPTION FOR`, in which case the core privileges themselves remain, but the option to grant them to others is removed. We may have to modify a node, redirect arcs, or create a new node to reflect the changes for the affected users. This form of `REVOKE` also must be followed by either `CASCADE` or `RESTRICT`.

Example 10.5: Continuing with Example 10.3, suppose that `janeway` revokes the privileges she granted to `picard` with the statements:

```
REVOKE SELECT, INSERT ON Studio FROM picard CASCADE;
REVOKE SELECT ON Movies FROM picard CASCADE;
```

We delete the arcs of Fig. 10.2 from these `janeway` privileges to the corresponding `picard` privileges. Since `CASCADE` was stipulated, we also have to see if there are any privileges that are not reachable in the graph from a doubly starred (ownership-based) privilege. Examining Fig. 10.2, we see that `picard`'s privileges are no longer reachable from a doubly starred node (they might have been, had there been another path to a `picard` node). Also, `sisko`'s privilege to `INSERT` into `Studio` is no longer reachable. We thus delete not only `picard`'s privileges from the grant diagram, but we delete `sisko`'s `INSERT` privilege.

Note that we do not delete `sisko`'s `SELECT` privileges on `Movies` and `Studio` or his `INSERT(name)` privilege on `Studio`, because these are all reachable from `janeway`'s ownership-based privileges via `kirk`'s privileges. The resulting grant diagram is shown in Fig. 10.3. □

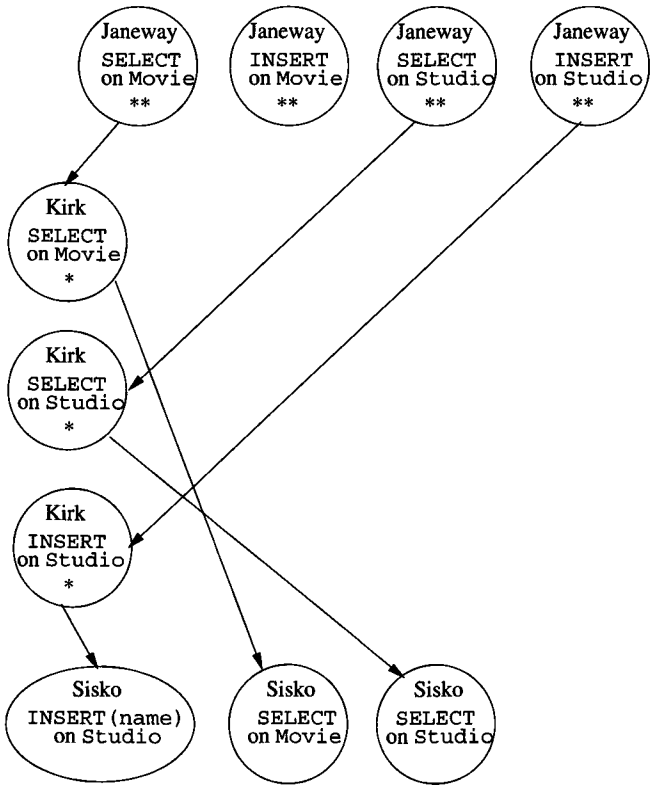


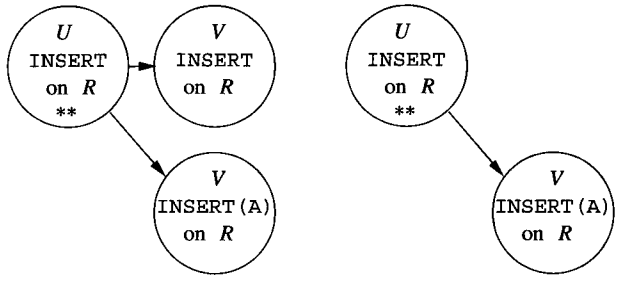
Figure 10.3: Grant diagram after revocation of picard's privileges

Example 10.6: There are a few subtleties that we shall illustrate with abstract examples. First, when we revoke a general privilege p, we do not also revoke a privilege that is a special case of p. For instance, consider the following sequence of steps, whereby user U, the owner of relation R, grants the INSERT privilege on relation R to user V, and also grants the INSERT(A) privilege on the same relation.

Step	By	Action
1	U	GRANT INSERT ON R TO V
2	U	GRANT INSERT(A) ON R TO V
3	U	REVOKE INSERT ON R FROM V RESTRICT

When U revokes INSERT from V, the INSERT(A) privilege remains. The grant diagrams after steps (2) and (3) are shown in Fig. 10.4.

Notice that after step (2) there are two separate nodes for the two similar but distinct privileges that user V has. Also observe that the RESTRICT option in step (3) does not prevent the revocation, because V had not granted the



(a) After step (2) (b) After step (3)

Figure 10.4: Revoking a general privilege leaves a more specific privilege

option to any other user. In fact, V could not have granted either privilege, because V obtained them without grant option. □

Example 10.7: Now, let us consider a similar example where U grants V a privilege $p*$ that includes the grant option and then revokes only the grant option. Assume the grant by U was based on its privilege $q*$. In this case, we must replace the arc from the $U/q*$ node to $V/p*$ by an arc from $U/q*$ to V/p, i.e., the same privilege without the grant option. If there was no such node V/p, it must be created. In normal circumstances, the node $V/p*$ becomes unreachable, and any grants of p made by V will also be unreachable. However, it may be that V was granted $p*$ by some other user besides U, in which case the $V/p*$ node remains accessible.

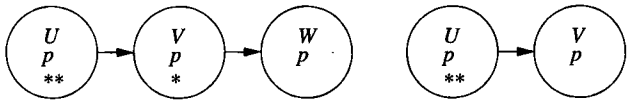
Here is a typical sequence of steps:

Step	By	Action
1	U	`GRANT` p `TO` V `WITH GRANT OPTION`
2	V	`GRANT` p `TO` W
3	U	`REVOKE GRANT OPTION FOR` p `FROM` V `CASCADE`

In step (1), U grants the privilege p to V with the grant option. In step (2), V uses the grant option to grant p to W. The diagram is then as shown in Fig. 10.5(a).

Then in step (3), U revokes the grant option for privilege p from V, but does not revoke the privilege itself. Since there is no node V/p, we create one. The arc from $U/p**$ to $V/P*$ is removed and replaced by one from $U/p**$ to V/p.

Now, the nodes $V/p*$ and W/p are not reachable from any ** node. Thus, these nodes are deleted from the diagram. The resulting grant diagram is shown in Fig. 10.5(b). □



(a) After step (2) (b) After step (3)

Figure 10.5: Revoking a grant option leaves the underlying privilege

10.1.7 Exercises for Section 10.1

Exercise 10.1.1: Indicate what privileges are needed to execute the following queries. In each case, mention the most specific privileges as well as general privileges that are sufficient.

a) The query of Fig. 6.5.

b) The query of Fig. 6.7.

c) The insertion of Fig. 6.15.

d) The deletion of Example 6.37.

e) The update of Example 6.39.

f) The tuple-based check of Fig. 7.3.

g) The assertion of Example 7.11.

Exercise 10.1.2: Show the grant diagrams after steps (4) through (6) of the sequence of actions listed in Fig. 10.6. Assume A is the owner of the relation to which privilege p refers.

Step	By	Action
1	A	GRANT p TO B WITH GRANT OPTION
2	A	GRANT p TO C
3	B	GRANT p TO D WITH GRANT OPTION
4	D	GRANT p TO B, C, E WITH GRANT OPTION
5	B	REVOKE p FROM D CASCADE
6	A	REVOKE p FROM C CASCADE

Figure 10.6: Sequence of actions for Exercise 10.1.2

Exercise 10.1.3: Show the grant diagrams after steps (5) and (6) of the sequence of actions listed in Fig. 10.7. Assume A is the owner of the relation to which privilege p refers.

Step	By	Action
1	A	GRANT p TO B, E WITH GRANT OPTION
2	B	GRANT p TO C WITH GRANT OPTION
3	C	GRANT p TO D WITH GRANT OPTION
4	E	GRANT p TO C
5	E	GRANT p TO D WITH GRANT OPTION
6	A	REVOKE GRANT OPTION FOR p FROM B CASCADE

Figure 10.7: Sequence of actions for Exercise 10.1.3

! **Exercise 10.1.4:** Show the final grant diagram after the following steps, assuming A is the owner of the relation to which privilege p refers.

Step	By	Action
1	A	GRANT p TO B WITH GRANT OPTION
2	B	GRANT p TO B WITH GRANT OPTION
3	A	REVOKE p FROM B CASCADE

10.2 Recursion in SQL

The SQL-99 standard includes provision for recursive definitions of queries. Although this feature is not part of the "core" SQL-99 standard that every DBMS is expected to implement, at least one major system — IBM's DB2 — does implement the SQL-99 proposal, which we describe in this section.

10.2.1 Defining Recursive Relations in SQL

The WITH statement in SQL allows us to define temporary relations, recursive or not. To define a recursive relation, the relation can be used within the WITH statement itself. A simple form of the WITH statement is:

WITH R AS <definition of R> <query involving R>

That is, one defines a temporary relation named R, and then uses R in some query. The temporary relation is not available outside the query that is part of the WITH statement.

More generally, one can define several relations after the WITH, separating their definitions by commas. Any of these definitions may be recursive. Several defined relations may be mutually recursive; that is, each may be defined in terms of some of the other relations, optionally including itself. However, any relation that is involved in a recursion must be preceded by the keyword RECURSIVE. Thus, a more general form of WITH statement is shown in Fig. 10.8.

Example 10.8: Many examples of the use of recursion can be found in a study of paths in a graph. Figure 10.9 shows a graph representing some flights of two

```
WITH
   [RECURSIVE] R1 AS <definition of R1>,
   [RECURSIVE] R2 AS <definition of R2>,
     ...
   [RECURSIVE] Rn AS <definition of Rn>
<query involving R1, R2, ... , Rn >
```

Figure 10.8: Form of a `WITH` statement defining several temporary relations

hypothetical airlines — *Untried Airlines* (UA), and *Arcane Airlines* (AA) — among the cities San Francisco, Denver, Dallas, Chicago, and New York. The data of the graph can be represented by a relation

```
Flights(airline, frm, to, departs, arrives)
```

and the particular tuples in this table are shown in Fig. 10.9.

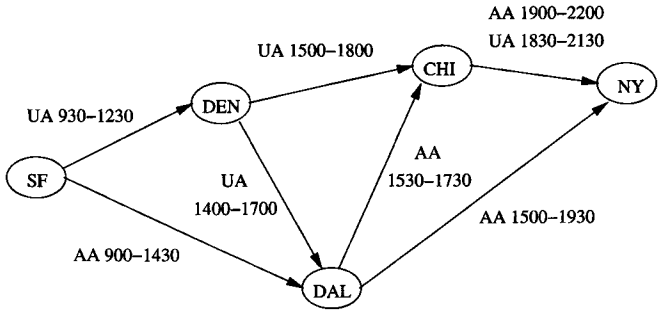


Figure 10.9: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Figure 10.10: Tuples in the relation `Flights`

The simplest recursive question we can ask is "For what pairs of cities (x, y) is it possible to get from city x to city y by taking one or more flights?" Before

writing this query in recursive SQL, it is useful to express the recursion in the Datalog notation of Section 5.3. Since many concepts involving recursion are easier to express in Datalog than in SQL, you may wish to review the terminology of that section before proceeding. The following two Datalog rules describe a relation `Reaches(x,y)` that contains exactly these pairs of cities.

1. `Reaches(x,y)` ← `Flights(a,x,y,d,r)`
2. `Reaches(x,y)` ← `Reaches(x,z) AND Reaches(z,y)`

The first rule says that `Reaches` contains those pairs of cities for which there is a direct flight from the first to the second; the airline a, departure time d, and arrival time r are arbitrary in this rule. The second rule says that if you can reach from city x to city z and you can reach from z to city y, then you can reach from x to y.

Evaluating a recursive relation requires that we apply the Datalog rules repeatedly, starting by assuming there are no tuples in `Reaches`. We begin by using Rule (1) to get the following pairs in `Reaches`: (`SF`, `DEN`), (`SF`, `DAL`), (`DEN`, `CHI`), (`DEN`, `DAL`), (`DAL`, `CHI`), (`DAL`, `NY`), and (`CHI`, `NY`). These are the seven pairs represented by arcs in Fig. 10.9.

In the next round, we apply the recursive Rule (2) to put together pairs of arcs such that the head of one is the tail of the next. That gives us the additional pairs (`SF`, `CHI`), (`DEN`, `NY`), and (`SF`, `NY`). The third round combines all one- and two-arc pairs together to form paths of length up to four arcs. In this particular diagram, we get no new pairs. The relation `Reaches` thus consists of the ten pairs (x, y) such that y is reachable from x in the diagram of Fig. 10.9. Because of the way we drew the diagram, these pairs happen to be exactly those (x, y) such that y is to the right of x in Fig 10.9.

From the two Datalog rules for `Reaches` in Example 10.8, we can develop a SQL query that produces the relation `Reaches`. This SQL query places the Datalog rules for `Reaches` in a `WITH` statement, and follows it by a query. In our example, the desired result was the entire `Reaches` relation, but we could also ask some query about `Reaches`, for instance the set of cities reachable from Denver.

```
1)  WITH RECURSIVE Reaches(frm, to) AS
2)          (SELECT frm, to FROM Flights)
3)      UNION
4)          (SELECT R1.frm, R2.to
5)           FROM Reaches R1, Reaches R2
6)           WHERE R1.to = R2.frm)
7)  SELECT * FROM Reaches;
```

Figure 10.11: Recursive SQL query for pairs of reachable cities

Figure 10.11 shows how to express `Reaches` as a SQL query. Line (1) introduces the definition of `Reaches`, while the actual definition of this relation is in

Mutual Recursion

There is a graph-theoretic way to check whether two relations or predicates are mutually recursive. Construct a *dependency graph* whose nodes correspond to the relations (or predicates if we are using Datalog rules). Draw an arc from relation A to relation B if the definition of B depends directly on the definition of A. That is, if Datalog is being used, then A appears in the body of a rule with B at the head. In SQL, A would appear in a `FROM` clause, somewhere in the definition of B, possibly in a subquery. If there is a cycle involving nodes R and S, then R and S are *mutually recursive*. The most common case will be a loop from R to R, indicating that R depends recursively upon itself.

lines (2) through (6).

That definition is a union of two queries, corresponding to the two Datalog rules by which `Reaches` was defined. Line (2) is the first term of the union and corresponds to the first, or basis rule. It says that for every tuple in the `Flights` relation, the second and third components (the `frm` and `to` components) are a tuple in `Reaches`.

Lines (4) through (6) correspond to Rule (2), the recursive rule, in the definition of `Reaches`. The two `Reaches` subgoals in Rule (2) are represented in the `FROM` clause by two aliases `R1` and `R2` for `Reaches`. The first component of `R1` corresponds to x in Rule (2), and the second component of `R2` corresponds to y. Variable z is represented by both the second component of `R1` and the first component of `R2`; note that these components are equated in line (6).

Finally, line (7) describes the relation produced by the entire query. It is a copy of the `Reaches` relation. As an alternative, we could replace line (7) by a more complex query. For instance,

```
7)  SELECT to FROM Reaches WHERE frm = 'DEN';
```

would produce all those cities reachable from Denver. □

10.2.2 Problematic Expressions in Recursive SQL

The SQL standard for recursion does not allow an arbitrary collection of mutually recursive relations to be written in a `WITH` clause. There is a small matter that the standard requires only that *linear* recursion be supported. A linear recursion, in Datalog terms, is one in which no rule has more than one subgoal that is mutually recursive with the head. Notice that Rule (2) in Example 10.8 has two subgoals with predicate `Reaches` that are mutually recursive with the head (a predicate is always mutually recursive with itself; see the box on Mutual

Recursion). Thus, technically, a DBMS might refuse to execute Fig. 10.11 and yet conform to the standard.[1]

But there is a more important restriction on SQL recursions, one that, if violated leads to recursions that cannot be executed by the query processor in any meaningful way. To be a legal SQL recursion, the definition of a recursive relation R may involve only the use of a mutually recursive relation S (including R itself) if that use is "monotone" in S. A use of S is *monotone* if adding an arbitrary tuple to S might add one or more tuples to R, or it might leave R unchanged, but it can never cause any tuple to be deleted from R. The following example suggests what can happen if the monotonicity requirement is not respected.

Example 10.9: Suppose relation R is a unary (one-attribute) relation, and its only tuple is (0). R is used as an EDB relation in the following Datalog rules:

```
1. P(x) ← R(x) AND NOT Q(x)
2. Q(x) ← R(x) AND NOT P(x)
```

Informally, the two rules tell us that an element x in R is either in P or in Q but not both. Notice that P and Q are mutually recursive.

If we start out, assuming that both P and Q are empty, and apply the rules once, we find that $P = \{(0)\}$ and $Q = \{(0)\}$; that is, (0) is in both IDB relations. On the next round, we apply the rules to the new values for P and Q again, and we find that now both are empty. This cycle repeats as long as we like, but we never converge to a solution.

In fact, there are two "solutions" to the Datalog rules:

a)	$P = \{(0)\}$	$Q = \emptyset$
b)	$P = \emptyset$	$Q = \{(0)\}$

However, there is no reason to assume one over the other, and the simple iteration we suggested as a way to compute recursive relations never converges to either. Thus, we cannot answer a simple question such as "Is $P(0)$ true?"

The problem is not restricted to Datalog. The two Datalog rules of this example can be expressed in recursive SQL. Figure 10.12 shows one way of doing so. This SQL does not adhere to the standard, and no DBMS should execute it. □

The problem in Example 10.9 is that the definitions of P and Q in Fig. 10.12 are not monotone. Look at the definition of P in lines (2) through (5) for instance. P depends on Q, with which it is mutually recursive, but adding a tuple to Q can delete a tuple from P. Notice that if $R = \{(0)\}$ and Q is empty, then $P = \{(0)\}$. But if we add (0) to Q, then we delete (0) from P. Thus, the definition of P is not monotone in Q, and the SQL code of Fig. 10.12 does not meet the standard.

[1]Note, however, that we can replace either one of the uses of **Reaches** in line (5) of Fig. 10.11 by **Flights**, and thus make the recursion linear. Nonlinear recursions can frequently — although not always — be made linear in this fashion.

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          EXCEPT
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)              (SELECT * FROM R)
 8)          EXCEPT
 9)              (SELECT * FROM P)

10)  SELECT * FROM P;
```

Figure 10.12: Query with nonmonotonic behavior, illegal in SQL

Example 10.10: Aggregation can also lead to nonmonotonicity. Suppose we have unary (one-attribute) relations P and Q defined by the following two conditions:

1. P is the union of Q and an EDB relation R.
2. Q has one tuple that is the sum of the members of P.

We can express these conditions by a `WITH` statement, although this statement violates the monotonicity requirement of SQL. The query shown in Fig. 10.13 asks for the value of P.

```
1)  WITH
2)      RECURSIVE P(x) AS
3)              (SELECT * FROM R)
4)          UNION
5)              (SELECT * FROM Q),

6)      RECURSIVE Q(x) AS
7)          SELECT SUM(x) FROM P

8)  SELECT * FROM P;
```

Figure 10.13: Nonmonotone query involving aggregation, illegal in SQL

Suppose that R consists of the tuples (12) and (34), and initially P and Q are both empty. Figure 10.14 summarizes the values computed in the first six rounds. Note that both relations are computed, in one round, from the values of the relations at the previous round. Thus, P is computed in the first round

Round	P	Q
1)	{(12), (34)}	{NULL}
2)	{(12), (34), NULL}	{(46)}
3)	{(12), (34), (46)}	{(46)}
4)	{(12), (34), (46)}	{(92)}
5)	{(12), (34), (92)}	{(92)}
6)	{(12), (34), (92)}	{(138)}

Figure 10.14: Iterative calculation for a nonmonotone aggregation

to be the same as R, and Q is {NULL}, since the old, empty value of P is used in line (7).

At the second round, the union of lines (3) through (5) is the set

$$R \cup \{\texttt{NULL}\} = \{(12),\ (34),\ \texttt{NULL}\}$$

so that set becomes the new value of P. The old value of P was {(12), (34)}, so on the second round $Q = \{(46)\}$. That is, 46 is the sum of 12 and 34.

At the third round, we get $P = \{(12), (34), (46)\}$ at lines (2) through (5). Using the old value of P, {(12), (34), NULL}, Q is defined by lines (6) and (7) to be {(46)} again. Remember that NULL is ignored in a sum.

At the fourth round, P has the same value, {(12), (34), (46)}, but Q gets the value {(92)}, since 12+34+46=92. Notice that Q has lost the tuple (46), although it gained the tuple (92). That is, adding the tuple (46) to P has caused a tuple (by coincidence the same tuple) to be deleted from Q. That behavior is the nonmonotonicity that SQL prohibits in recursive definitions, confirming that the query of Fig. 10.13 is illegal. In general, at the $2i$th round, P will consist of the tuples (12), (34), and $(46i - 46)$, while Q consists only of the tuple $(46i)$. □

10.2.3 Exercises for Section 10.2

Exercise 10.2.1: The relation

```
Flights(airline, frm, to, departs, arrives)
```

from Example 10.8 has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning.

a) Write this recursion in Datalog.

b) Write the recursion in SQL.

! **Exercise 10.2.2:** In Example 10.8 we used `frm` as an attribute name. Why did we not use the more obvious name `from`?

Exercise 10.2.3: Suppose we have a relation

```
SequelOf(movie, sequel)
```

that gives the immediate sequels of a movie, of which there can be more than one. We want to define a recursive relation `FollowOn` whose pairs (x, y) are movies such that y was either a sequel of x, a sequel of a sequel, or so on.

a) Write the definition of `FollowOn` as recursive Datalog rules.

b) Write the definition of `FollowOn` as a SQL recursion.

c) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on to movie x, but is not a sequel of x.

d) Write a recursive SQL query that returns the set of pairs (x, y) meaning that y is a follow-on of x, but is neither a sequel nor a sequel of a sequel.

! e) Write a recursive SQL query that returns the set of movies x that have at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.

! f) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on of x but y has at most one follow-on.

Exercise 10.2.4: Suppose we have a relation

```
Rel(class, rclass, mult)
```

that describes how one ODL class is related to other classes. Specifically, this relation has tuple (c, d, m) if there is a relation from class c to class d. This relation is multivalued if $m =$ `'multi'` and it is single-valued if $m =$ `'single'`. It is possible to view `Rel` as defining a graph whose nodes are classes and in which there is an arc from c to d labeled m if and only if (c, d, m) is a tuple of `Rel`. Write a recursive SQL query that produces the set of pairs (c, d) such that:

a) There is a path from class c to class d in the graph described above.

b) There is a path from c to d along which every arc is labeled `single`.

! c) There is a path from c to d along which at least one arc is labeled `multi`.

d) There is a path from c to d but no path along which all arcs are labeled `single`.

! e) There is a path from c to d along which arc labels alternate `single` and `multi`.

f) There are paths from c to d and from d to c along which every arc is labeled `single`.

10.3 The Object-Relational Model

The relational model and the object-oriented model typified by ODL are two important points in a spectrum of options that could underlie a DBMS. For an extended period, the relational model was dominant in the commercial DBMS world. Object-oriented DBMS's made limited inroads during the 1990's, but never succeeded in winning significant market share from the vendors of relational DBMS's. Rather, the vendors of relational systems have moved to incorporate many of the ideas found in ODL or other object-oriented-database proposals. As a result, many DBMS products that used to be called "relational" are now called "object-relational."

This section extends the abstract relational model to incorporate several important object-relational ideas. It is followed by sections that cover object-relational extensions of SQL. We introduce the concept of object-relations in Section 10.3.1, then discuss one of its earliest embodiments — nested relations — in Section 10.3.2. ODL-like references for object-relations are discussed in Section 10.3.3, and in Section 10.3.4 we compare the object-relational model with the pure object-oriented approach.

10.3.1 From Relations to Object-Relations

While the relation remains the fundamental concept, the relational model has been extended to the *object-relational model* by incorporation of features such as:

1. *Structured types for attributes.* Instead of allowing only atomic types for attributes, object-relational systems support a type system like ODL's: types built from atomic types and type constructors for structs, sets, and bags, for instance. Especially important is a type that is a bag of structs, which is essentially a relation. That is, a value of one component of a tuple can be an entire relation, called a "nested relation."

2. *Methods.* These are similar to methods in ODL or any object-oriented programming system.

3. *Identifiers for tuples.* In object-relational systems, tuples play the role of objects. It therefore becomes useful in some situations for each tuple to have a unique ID that distinguishes it from other tuples, even from tuples that have the same values in all components. This ID, like the object-identifier assumed in ODL, is generally invisible to the user, although

there are even some circumstances where users can see the identifier for a tuple in an object-relational system.

4. *References.* While the pure relational model has no notion of references or pointers to tuples, object-relational systems can use these references in various ways.

In the next sections, we shall elaborate upon and illustrate each of these additional capabilities of object-relational systems.

10.3.2 Nested Relations

In the *nested-relational model*, we allow attributes of relations to have a type that is not atomic; in particular, a type can be a relation schema. As a result, there is a convenient, recursive definition of the types of attributes and the types (schemas) of relations:

BASIS: An atomic type (integer, real, string, etc.) can be the type of an attribute.

INDUCTION: A relation's type can be any *schema* consisting of names for one or more attributes, and any legal type for each attribute. In addition, a schema also can be the type of any attribute.

In what follows, we shall generally omit atomic types where they do not matter. An attribute that is a schema will be represented by the attribute name and a parenthesized list of the attributes of its schema. Since those attributes may themselves have structure, parentheses can be nested to any depth.

Example 10.11: Let us design a nested-relation schema for stars that incorporates within the relation an attribute `movies`, which will be a relation representing all the movies in which the star has appeared. The relation schema for attribute `movies` will include the title, year, and length of the movie. The relation schema for the relation `Stars` will include the name, address, and birthdate, as well as the information found in `movies`. Additionally, the `address` attribute will have a relation type with attributes `street` and `city`. We can record in this relation several addresses for the star. The schema for `Stars` can be written:

```
Stars(name, address(street, city), birthdate,
   movies(title, year, length))
```

An example of a possible relation for nested relation `Stars` is shown in Fig. 10.15. We see in this relation two tuples, one for Carrie Fisher and one for Mark Hamill. The values of components are abbreviated to conserve space, and the dashed lines separating tuples are only for convenience and have no notational significance.

name	*address*	*birthdate*	*movies*
Fisher	*street* | *city* Maple | H'wood Locust | Malibu	9/9/99	*title* | *year* | *length* Star Wars | 1977 | 124 Empire | 1980 | 127 Return | 1983 | 133
Hamill	*street* | *city* Oak | B'wood	8/8/88	*title* | *year* | *length* Star Wars | 1977 | 124 Empire | 1980 | 127 Return | 1983 | 133

Figure 10.15: A nested relation for stars and their movies

In the Carrie Fisher tuple, we see her name, an atomic value, followed by a relation for the value of the address component. That relation has two attributes, `street` and `city`, and there are two tuples, corresponding to her two houses. Next comes the birthdate, another atomic value. Finally, there is a component for the `movies` attribute; this attribute has a relation schema as its type, with components for the title, year, and length of a movie. The relation for the `movies` component of the Carrie Fisher tuple has tuples for her three best-known movies.

The second tuple, for Mark Hamill, has the same components. His relation for `address` has only one tuple, because in our imaginary data, he has only one house. His relation for `movies` looks just like Carrie Fisher's because their best-known movies happen, by coincidence, to be the same. Note that these two relations are two different tuple-components. These components happen to be identical, just like two components that happened to have the same integer value, e.g., 124. □

10.3.3 References

The fact that movies like *Star Wars* will appear in several relations that are values of the `movies` attribute in the nested relation `Stars` is a cause of redundancy. In effect, the schema of Example 10.11 has the nested-relation analog of not being in BCNF. However, decomposing this `Stars` relation will not eliminate the redundancy. Rather, we need to arrange that among all the tuples of all the `movies` relations, a movie appears only once.

To cure the problem, object-relations need the ability for one tuple t to refer to another tuple s, rather than incorporating s directly in t. We thus add to our model an additional inductive rule: the type of an attribute also can be a

reference to a tuple with a given schema or a set of references to tuples with a given schema.

If an attribute A has a type that is a reference to a single tuple with a relation schema named R, we show the attribute A in a schema as $A(*R)$. Notice that this situation is analogous to an ODL relationship A whose type is R; i.e., it connects to a single object of type R. Similarly, if an attribute A has a type that is a set of references to tuples of schema R, then A will be shown in a schema as $A(\{*R\})$. This situation resembles an ODL relationship A that has type `Set<`R`>`.

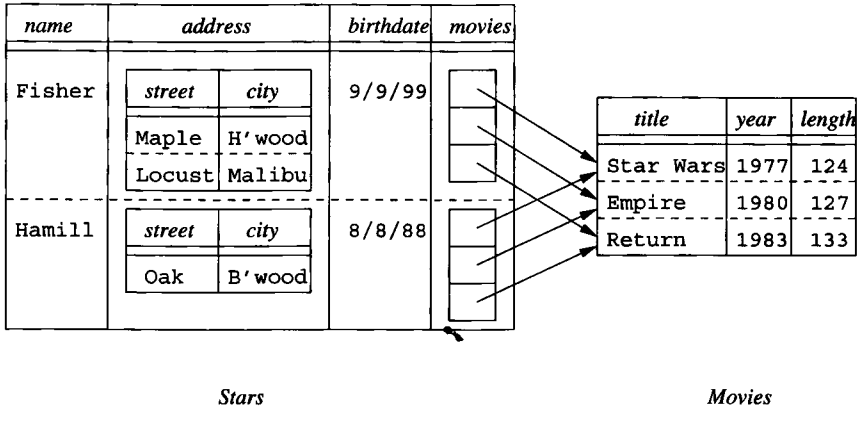


Figure 10.16: Sets of references as the value of an attribute

Example 10.12: An appropriate way to fix the redundancy in Fig. 10.15 is to use two relations, one for stars and one for movies. In this example only, we shall use a relation called `Movies` that is an ordinary relation with the same schema as the attribute `movies` in Example 10.11. A new relation `Stars` has a schema similar to the nested relation `Stars` of that example, but the movies attribute will have a type that is a set of references to `Movies` tuples. The schemas of the two relations are thus:

```
Movies(title, year, length)
Stars(name, address(street, city), birthdate,
    movies({*Movies}))
```

The data of Fig. 10.15, converted to this new schema, is shown in Fig. 10.16. Notice that, because each movie has only one tuple, although it can have many references, we have eliminated the redundancy inherent in the schema of Example 10.11. □

10.3.4 Object-Oriented Versus Object-Relational

The object-oriented data model, as typified by ODL, and the object-relational model discussed here, are remarkably similar. Some of the salient points of comparison follow.

Objects and Tuples

An object's value is really a struct with components for its attributes and relationships. It is not specified in the ODL standard how relationships are to be represented, but we may assume that an object is connected to related objects by some collection of references. A tuple is likewise a struct, but in the conventional relational model, it has components for only the attributes. Relationships would be represented by tuples in another relation, as suggested in Section 4.5.2. However the object-relational model, by allowing sets of references to be a component of tuples, also allows relationships to be incorporated directly into the tuples that represent an "object" or entity.

Methods

We did not discuss the use of methods as part of an object-relational schema. However, in practice, the SQL-99 standard and all implementations of object-relational ideas allow the same ability as ODL to declare and define methods associated with any class or type.

Type Systems

The type systems of the object-oriented and object-relational models are quite similar. Each is based on atomic types and construction of new types by struct- and collection-type-constructors. The choice of collection types may vary, but all variants include at least sets and bags. Moreover, the set (or bag) of structs type plays a special role in both models. It is the type of classes in ODL, and the type of relations in the object-relational model.

References and Object-ID's

A pure object-oriented model uses object-ID's that are completely hidden from the user, and thus cannot be seen or queried. The object-relational model allows references to be part of a type, and thus it is possible under some circumstances for the user to see their values and even remember them for future use. You may regard this situation as anything from a serious bug to a stroke of genius, depending on your point of view, but in practice it appears to make little difference.

Backwards Compatibility

With little difference in essential features of the two models, it is interesting to consider why object-relational systems have dominated the pure object-oriented systems in the marketplace. The reason, we believe, is as follows. As relational DBMS's evolved into object-relational DBMS's, the vendors were careful to maintain backwards compatibility. That is, newer versions of the system would still run the old code and accept the same schemas, should the user not care to adopt any of the object-oriented features. On the other hand, migration to a pure object-oriented DBMS would require the installations to rewrite and reorganize extensively. Thus, whatever competitive advantage could be argued for object-oriented database systems was insufficient to motivate many to make the switch.

10.3.5 Exercises for Section 10.3

Exercise 10.3.1: Using the notation developed for nested relations and relations with references, give one or more relation schemas that represent the following information. In each case, you may exercise some discretion regarding what attributes of a relation are included, but try to keep close to the attributes found in our running movie example. Also, indicate whether your schemas exhibit redundancy, and if so, what could be done to avoid it.

a) Movies, with the usual attributes plus all their stars and the usual information about the stars.

! b) Studios, all the movies made by that studio, and all the stars of each movie, including all the usual attributes of studios, movies, and stars.

c) Movies with their studio, their stars, and all the usual attributes of these.

Exercise 10.3.2: Represent the banking information of Exercise 4.1.1 in the object-relational model developed in this section. Make sure that it is easy, given the tuple for a customer, to find their account(s) and *also* easy, given the tuple for an account to find the customer(s) that hold that account. Also, try to avoid redundancy.

! **Exercise 10.3.3:** If the data of Exercise 10.3.2 were modified so that an account could be held by only one customer [as in Exercise 4.1.2(a)], how could your answer to Exercise 10.3.2 be simplified?

! **Exercise 10.3.4:** Render the players, teams, and fans of Exercise 4.1.3 in the object-relational model.

! **Exercise 10.3.5:** Render the genealogy of Exercise 4.1.6 in the object-relational model.

10.4 User-Defined Types in SQL

We now turn to the way SQL-99 incorporates many of the object-oriented features that we saw in Section 10.3. The central extension that turns the relational model into the object-relational model in SQL is the *user-defined type*, or UDT. We find UDT's used in two distinct ways:

1. A UDT can be the type of a table.
2. A UDT can be the type of an attribute belonging to some table.

10.4.1 Defining Types in SQL

The SQL-99 standard allows the programmer to define UDT's in several ways. The simplest is as a renaming of an existing type.

CREATE TYPE T AS <primitive type>;

renames a primitive type such as INTEGER. Its purpose is to prevent errors caused by accidental coercions among values that logically should not be compared or interchanged, even though they have the same primitive data type. An example should make the purpose clear.

Example 10.13: In our running movies example, there are several attributes of type INTEGER. These include `length` of `Movies`, `cert#` of `MovieExec`, and `presC#` of `Studio`. It makes sense to compare a value of `cert#` with a value of `presC#`, and we could even take a value from one of these two attributes and store it in a tuple as the value of the other attribute. However, It would not make sense to compare a movie length with the certificate number of a movie executive, or to take a `length` value from a `Movies` tuple and store it in the `cert#` attribute of a `MovieExec` tuple.

If we create types:

```
CREATE TYPE CertType AS INTEGER;
CREATE TYPE LengthType AS INTEGER;
```

then we can declare `cert#` and `presC#` to be of type `CertType` instead of INTEGER in their respective relation declarations, and we can declare `length` to be of type `LengthType` in the `Movies` declaration. In that case, an object-relational DBMS will intercept attempts to compare values of one type with the other, or to use a value of one type in place of the other. □

A more powerful form of UDT declaration in SQL is similar to a class declaration in ODL, with some distinctions. First, key declarations for a relation with a user-defined type are part of the table definition, not the type definition; that is, many SQL relations can be declared to have the same UDT but different keys and other constraints. Second, in SQL we do not treat relationships as properties. A relationship can be represented by a separate relation,

as was discussed in Section 4.10.5, or through references, which are covered in Section 10.4.5. This form of UDT definition is:

CREATE TYPE T AS (<attribute declarations>);

Example 10.14: Figure 10.17 shows two UDT's, `AddressType` and `StarType`. A tuple of type `AddressType` has two components, whose attributes are `street` and `city`. The types of these components are character strings of length 50 and 20, respectively. A tuple of type `StarType` also has two components. The first is attribute `name`, whose type is a 30-character string, and the second is `address`, whose type is itself a UDT `AddressType`, that is, a tuple with `street` and `city` components. □

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);

CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

Figure 10.17: Two type definitions

10.4.2 Method Declarations in UDT's

The declaration of a method resembles the way a function in PSM is introduced; see Section 9.4.1. There is no analog of PSM procedures as methods. That is, every method returns a value of some type. While function declarations and definitions in PSM are combined, a method needs both a declaration, which follows the parenthesized list of attributes in the `CREATE TYPE` statement, and a separate definition, in a `CREATE METHOD` statement. The actual code for the method need not be PSM, although it could be. For example, the method body could be Java with JDBC used to access the database.

A method declaration looks like a PSM function declaration, with the keyword `METHOD` replacing `CREATE FUNCTION`. However, SQL methods typically have no arguments; they are applied to rows, just as ODL methods are applied to objects. In the definition of the method, `SELF` refers to this tuple, if necessary.

Example 10.15: Let us extend the definition of the type `AddressType` of Fig. 10.17 with a method `houseNumber` that extracts from the `street` component the portion devoted to the house address. For instance, if the `street`

component were '123 Maple St.', then houseNumber should return '123'. Exactly how houseNumber works is not visible in its declaration; the details are left for the definition. The revised type definition is thus shown in Fig. 10.18.

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
    )
    METHOD houseNumber() RETURNS CHAR(10);
```

Figure 10.18: Adding a method declaration to a UDT

We see the keyword METHOD, followed by the name of the method and a parenthesized list of its arguments and their types. In this case, there are no arguments, but the parentheses are still needed. Had there been arguments, they would have appeared, followed by their types, such as (a INT, b CHAR(5)). □

10.4.3 Method Definitions

Separately, we need to define the method. A simple form of method definition is:

```
CREATE METHOD <method name, arguments, and return type>
  FOR <UDT name>
    <method body>
```

That is, the UDT for which the method is defined is indicated in a FOR clause. The method definition need not be contiguous to, or part of, the definition of the type to which it belongs.

Example 10.16: For instance, we could define the method houseNumber from Example 10.15 as in Fig. 10.19. We have omitted the body of the method because accomplishing the intended separation of the string string as intended is nontrivial, even if a general-purpose host language is used. □

```
CREATE METHOD houseNumber() RETURNS CHAR(10)
FOR AddressType
BEGIN
    ...
END;
```

Figure 10.19: Defining a method

10.4.4 Declaring Relations with a UDT

Having declared a type, we may declare one or more relations whose tuples are of that type. The form of relation declarations is like that of Section 2.3.3, but the attribute declarations are omitted from the parenthesized list of elements, and replaced by a clause with `OF` and the name of the UDT. That is, the alternative form of a `CREATE TABLE` statement, using a UDT, is:

```
CREATE TABLE <table name> OF <UDT name>
    (<list of elements>);
```

The parenthesized list of elements can include keys, foreign keys, and tuple-based constraints. Note that all these elements are declared for a particular table, not for the UDT. Thus, there can be several tables with the same UDT as their row type, and these tables can have different constraints, and even different keys. If there are no constraints or key declarations desired for the table, then the parentheses are not needed.

Example 10.17: We could declare `MovieStar` to be a relation whose tuples are of type `StarType` by

```
CREATE TABLE MovieStar OF StarType (
    PRIMARY KEY (name)
);
```

As a result, table `MovieStar` has two attributes, `name` and `address`. The first attribute, `name`, is an ordinary character string, but the second, `address`, has a type that is itself a UDT, namely the type `AddressType`. Attribute `name` is a key for this relation, so it is not possible to have two tuples with the same name. □

10.4.5 References

The effect of object identity in object-oriented languages is obtained in SQL through the notion of a *reference*. A table may have a *reference column* that serves as the "identity" for its tuples. This column could be the primary key of the table, if there is one, or it could be a column whose values are generated and maintained unique by the DBMS, for example. We shall defer to Section 10.4.6 the matter of defining reference columns until we first see how reference types are used.

To refer to the tuples of a table with a reference column, an attribute may have as its type a reference to another type. If T is a UDT, then `REF`(T) is the type of a reference to a tuple of type T. Further, the reference may be given a *scope*, which is the name of the relation whose tuples are referred to. Thus, an attribute A whose values are references to tuples in relation R, where R is a table whose type is the UDT T, would be declared by:

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movies
);
```

Figure 10.20: Adding a best movie reference to StarType

A REF(T) SCOPE R

If no scope is specified, the reference can go to any relation of type T.

Example 10.18: Let us record in MovieStar the best movie for each star. Assume that we have declared an appropriate relation Movies, and that the type of this relation is the UDT MovieType; we shall define both MovieType and Movies later, in Fig. 10.21. Figure 10.20 is a new definition of StarType that includes an attribute bestMovies that is a reference to a movie. Now, if relation MovieStar is defined to have the UDT of Fig. 10.20, then each star tuple will have a component that refers to a Movies tuple — the star's best movie. □

10.4.6 Creating Object ID's for Tables

In order to refer to rows of a table, such as Movies in Example 10.18, that table needs to have an "object-ID" for its tuples. Such a table is said to be *referenceable*. In a CREATE TABLE statement where the type of the table is a UDT (as in Section 10.4.4), we may include an element of the form:

REF IS <attribute name> <how generated>

The attribute name is a name given to the column that will serve as the object-ID for tuples. The "how generated" clause can be:

1. SYSTEM GENERATED, meaning that the DBMS is responsible for maintaining a unique value in this column of each tuple, or
2. DERIVED, meaning that the DBMS will use the primary key of the relation to produce unique values for this column.

Example 10.19: Figure 10.21 shows how the UDT MovieType and relation Movies could be declared so that Movies is referenceable. The UDT is declared in lines (1) through (4). Then the relation Movies is defined to have this type in lines (5) through (7). Notice that we have declared title and year, together, to be the key for relation Movies in line (7).

We see in line (6) that the name of the "identity" column for Movies is movieID. This attribute, which automatically becomes a fourth attribute of

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      genre   CHAR(10)
    );

5)  CREATE TABLE Movies OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

Figure 10.21: Creating a referenceable table

`Movies`, along with `title`, `year`, and `genre`, may be used in queries like any other attribute of `Movies`.

Line (6) also says that the DBMS is responsible for generating the value of `movieID` each time a new tuple is inserted into `Movies`. Had we replaced `SYSTEM GENERATED` by `DERIVED`, then new tuples would get their value of `movieID` by some calculation, performed by the system, on the values of the primary-key attributes `title` and `year` taken from the new tuple. □

Example 10.20: Now, let us see how to represent the many-many relationship between movies and stars using references. Previously, we represented this relationship by a relation like `StarsIn` that contains tuples with the keys of `Movies` and `MovieStar`. As an alternative, we may define `StarsIn` to have references to tuples from these two relations.

First, we need to redefine $\widehat{\text{MovieStar}}$ so it is a referenceable table, thusly:

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED,
    PRIMARY KEY (name)
);
```

Then, we may declare the relation `StarsIn` to have two attributes, which are references, one to a movie tuple and one to a star tuple. Here is a direct definition of this relation:

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movies
);
```

Optionally, we could have defined a UDT as above, and then declared `StarsIn` to be a table of that type. □

10.4.7 Exercises for Section 10.4

Exercise 10.4.1: For our running movies example, choose type names for the attributes of each of the relations. Give attributes the same UDT if their values can reasonably be compared or exchanged, and give them different UDT's if they should not have their values compared or exchanged.

Exercise 10.4.2: Write type declarations for the following types:

a) `NameType`, with components for first, middle, and last names and a title.

b) `PersonType`, with a name of the person and references to the persons that are their mother and father. You must use the type from part (a) in your declaration.

c) `MarriageType`, with the date of the marriage and references to the husband and wife.

Exercise 10.4.3: Redesign our running products database schema of Exercise 2.4.1 to use type declarations and reference attributes where appropriate. In particular, in the relations `PC`, `Laptop`, and `Printer` make the `model` attribute be a reference to the `Product` tuple for that model.

! **Exercise 10.4.4:** In Exercise 10.4.3 we suggested that model numbers in the tables `PC`, `Laptop`, and `Printer` could be references to tuples of the `Product` table. Is it also possible to make the `model` attribute in `Product` a reference to the tuple in the relation for that type of product? Why or why not?

Exercise 10.4.5: Redesign our running battleships database schema of Exercise 2.4.3 to use type declarations and reference attributes where appropriate. Look for many-one relationships and try to represent them using an attribute with a reference type.

10.5 Operations on Object-Relational Data

All appropriate SQL operations from previous chapters apply to tables that are declared with a UDT or that have attributes whose type is a UDT. There are also some entirely new operations we can use, such as reference-following. However, some familiar operations, especially those that access or modify columns whose type is a UDT, involve new syntax.

10.5.1 Following References

Suppose x is a value of type `REF(`T`)`. Then x refers to some tuple t of type T. We can obtain tuple t itself, or components of t, by two means:

1. Operator -> has essentially the same meaning as this operator does in C. That is, if x is a reference to a tuple t, and a is an attribute of t, then `x->a` is the value of the attribute a in tuple t.

2. The `DEREF` operator applies to a reference and produces the tuple referenced.

Example 10.21: Let us use the relation `StarsIn` from Example 10.20 to find the movies in which Brad Pitt starred. Recall that the schema is

```
StarsIn(star, movie)
```

where `star` and `movie` are references to tuples of `MovieStar` and `Movies`, respectively. A possible query is:

```
1)  SELECT DEREF(movie)
2)  FROM StarsIn
3)  WHERE star->name = 'Brad Pitt';
```

In line (3), the expression `star->name` produces the value of the `name` component of the `MovieStar` tuple referred to by the `star` component of any given `StarsIn` tuple. Thus, the `WHERE` clause identifies those `StarsIn` tuples whose `star` components are references to the Brad-Pitt `MovieStar` tuple. Line (1) then produces the movie tuple referred to by the `movie` component of those tuples. All three attributes — `title`, `year`, and `genre` — will appear in the printed result.

Note that we could have replaced line (1) by:

```
1)  SELECT  movie
```

However, had we done so, we would have gotten a list of system-generated gibberish that serves as the internal unique identifiers for certain movie tuples. We would not see the information in the referenced tuples. □

10.5.2 Accessing Components of Tuples with a UDT

When we define a relation to have a UDT, the tuples must be thought of as single objects, rather than lists with components corresponding to the attributes of the UDT. As a case in point, consider the relation `Movies` declared in Fig. 10.21. This relation has UDT `MovieType`, which has three attributes: `title`, `year`, and `genre`. However, a tuple t in `Movies` has only *one* component, not three. That component is the object itself.

If we "drill down" into the object, we can extract the values of the three attributes in the type `MovieType`, as well as use any methods defined for that type. However, we have to access these attributes properly, since they are not attributes of the tuple itself. Rather, every UDT has an implicitly defined *observer method* for each attribute of that UDT. The name of the observer

method for an attribute x is $x()$. We apply this method as we would any other method for this UDT; we attach it with a dot to an expression that evaluates to an object of this type. Thus, if t is a variable whose value is of type T, and x is an attribute of T, then $t.x()$ is the value of x in the tuple (object) denoted by t.

Example 10.22: Let us find, from the relation `Movies` of Fig. 10.21 the year(s) of movies with title *King Kong*. Here is one way to do so:

```
SELECT m.year()
FROM Movies m
WHERE m.title() = 'King Kong';
```

Even though the tuple variable m would appear not to be needed here, we need a variable whose value is an object of type `MovieType` — the UDT for relation `Movies`. The condition of the `WHERE` clause compares the constant `'King Kong'` to the value of `m.title()`, the observer method for attribute `title` applied to a `MovieType` object m. Similarly, the value in the `SELECT` clause is expressed `m.year()`; this expression applies the observer method for `year` to the object m. □

In practice, object-relational DBMS's do not use method syntax to extract an attribute from an object. Rather, the parentheses are dropped, and we shall do so in what follows. For instance, the query of Example 10.22 will be written:

```
SELECT m.year
FROM Movies m
WHERE m.title = 'King Kong';
```

The tuple variable m is still necessary, however.

The dot operator can be used to apply methods as well as to find attribute values within objects. These methods should have the parentheses attached, even if they take no arguments.

Example 10.23: Suppose relation `MovieStar` has been declared to have UDT `StarType`, which we should recall from Example 10.14 has an attribute `address` of type `AddressType`. That type, in turn, has a method `houseNumber()`, which extracts the house number from an object of type `AddressType` (see Example 10.15). Then the query

```
SELECT MAX(s.address.houseNumber())
FROM MovieStar s
```

extracts the address component from a `StarType` object s, then applies the `houseNumber()` method to that `AddressType` object. The result returned is the largest house number of any movie star. □