3324CH05.qxd 9/21/04 9:53 AM Page 239

CHAPTER 5

Joins, Temporary Tables, and Transactions

In this chapter, we'll discuss three additional features you can use to speed up your MySQL applications. While these aren't directly related to one another, each represents an opportunity to decrease the amount of database or code overhead required to perform useful tasks with MySQL by combining queries or operations on the code level into fewer units that perform more work.

- *Joins* allow for the selection of data from multiple tables using a single SQL statement.
- *Temporary tables* provide a way to organize data derived from queries into new tables, which can themselves be queried repeatedly over the lifetime of a MySQL user session.
- *Transactions* allow you to group together related operations into logical units in such a way that all operations either succeed or fail together.

We'll spend some time with each of these features, discussing what it is, how it works, and how you can put it to use in your applications.

The rationale behind joins is relatively simple: it's more efficient to issue a single query than to use a series of them, with the resultset from the first query providing the conditions for one or more additional queries. There are several types of joins, which are distinguished chiefly by how they treat values in one table column that aren't matched in the related column of the other table; we'll cover each of these in turn. In addition, we'll discuss the two basic styles for join notation (theta-style and ANSI-style) and the variations on these that are available in MySQL.

The use of temporary tables is another way to save time and effort, particularly when dealing with several queries that return very large and similar resultsets. When you find yourself dealing with the same subset of table data several times in a single session, it's often faster and more economical to obtain it once and store it in a temporary table, rather than either saving the data in a

3324CH05.qxd 9/21/04 9:53 AM Page 240

programming structure (such as an array or hash) or repeating a complex join several times. If you're using several resultsets that contain a large proportion of data in common, it can also make sense to obtain a single resultset that has all the data that's required, store this in a temporary table, and then select from this temporary table as needed. Temporary tables are very convenient to use in MySQL because they are unique to the user session in which they were created. We'll explain just what this means, as well as how to use temporary tables.

Transactions are beneficial because they make it much easier to guarantee data integrity. It's also much more efficient to use transactions than to attempt to perform each query separately in your application logic, testing for its success or failure, and then undoing any previously successful operations in the event that one does fail. By using InnoDB or Berkeley DB (BDB) tables and transactions, you can let MySQL handle this task for you. Using transactions is not necessarily faster in and of itself than not using them; in fact, MyISAM tables (which don't support transactions) are faster than either InnoDB or BDB tables (which do). However, you'll almost certainly save time in development, and your applications will require less code, because you don't need to test and possibly undo each query individually. In this chapter, we'll cover the basic theory of transactions and how they're implemented in MySQL. Later in this book (in Chapter 7), you'll see how these are used in PHP, Python, and Perl.

Joins

A join in MySQL or any other relational database is simply the selection of data from two or more related tables in a single query based on column values common to all of those tables. The cardinal rule for relating tables can be stated as follows: *Tables to be joined must have one or more columns sharing a set of values that allow those tables to be connected in some meaningful way*.

In other words, if we think of tables as modeling real objects, then joins are simply a way of relating objects according to the attributes they hold in common. The column held in common by both tables is usually referred to as the *common key* or *join key*. Of course, it's possible to have more than one common column between two tables, and so it's possible to use more than one join key in any particular join. Most often, the join key will be the primary key of one table and a foreign key in the other.

Before going any further, let's provide a scenario that we'll employ for generating some examples in the rest of this section. This represents a slight modification of the students/classes schema used in Chapter 3. This updated schema is shown in Figure 5-1.

3324CH05.qxd 9/21/04 9:53 AM Page 241

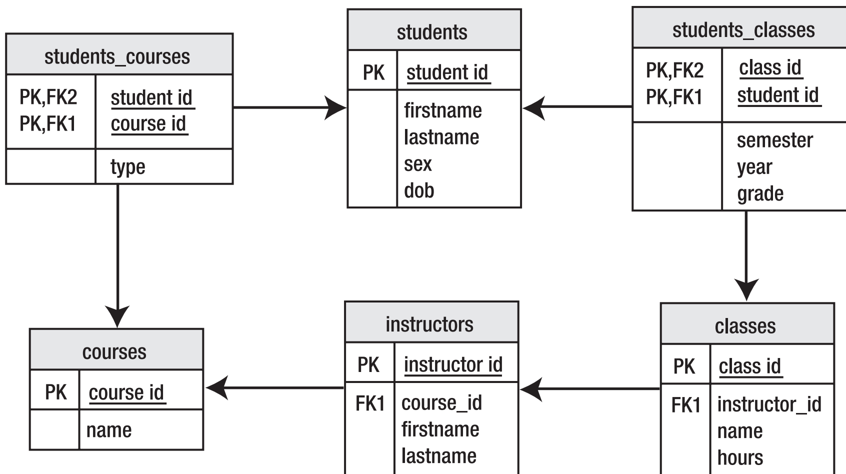


Figure 5-1. Updated students/classes schema

This schema consists of six tables from a database used for tracking students' schedules and grades at a small college. Four of these characterize students, instructors, classes, and courses of study. (Obviously, this is greatly simplified for illustrative purposes.) We also have two lookup tables (**students_classes** and **students_courses**) linking students with classes and students with courses of instruction. In a real-world application, we might do some things differently, but we hope you'll be able to overlook that for the time being.

NOTE *The students-tables.sql file is included in the ch5 folder of the code accompanying this book (available from the Downloads section of* `http://www.apress.com`*). Also in that folder you'll find the students-data.sql file, which contains the SQL statements for inserting the test data we'll be referring to in our examples.*

The following is the SQL code for generating the required tables:

```
CREATE TABLE classes (
  class_id int(11) NOT NULL auto_increment,
  instructor_id int(11) NOT NULL default '0',
  name varchar(50) NOT NULL default '',
  hours int(1) NOT NULL default '0',
  PRIMARY KEY  (class_id)
);
```

```
CREATE TABLE courses (
  course_id int(11) NOT NULL auto_increment,
  name varchar(50) NOT NULL default '',
  PRIMARY KEY  (course_id)
);

CREATE TABLE instructors (
  instructor_id int(11) NOT NULL auto_increment,
  course_id int(11) NOT NULL default '0',
  firstname varchar(50) NOT NULL default '',
  lastname varchar(50) NOT NULL default '',
  PRIMARY KEY  (instructor_id)
);

CREATE TABLE students (
  student_id int(11) NOT NULL auto_increment,
  firstname varchar(50) NOT NULL default '',
  lastname varchar(50) NOT NULL default '',
  sex enum('M','F') NOT NULL default 'M',
  dob date NOT NULL default '0000-00-00',
  PRIMARY KEY  (student_id)
);

CREATE TABLE students_classes (
  student_id int(11) NOT NULL default '0',
  class_id int(11) NOT NULL default '0',
  semester enum('FALL','SPRING','SUMMER') NOT NULL default 'FALL',
  year int(4) NOT NULL default '2005',
  grade int(1) default NULL,
  PRIMARY KEY  (student_id,class_id,semester,year)
);

#  Note: For the grade column, we assume that the US system is being used:
#  A = 4, B = 3, C = 2, D = 1, F = 0; for our purposes we'll assume that
#  a value of NULL represents incomplete status (class in progress, etc.)

CREATE TABLE students_courses (
  student_id int(11) NOT NULL default '0',
  course_id int(11) NOT NULL default '0',
  type enum('MAJOR','MINOR') NOT NULL default 'MAJOR',
  PRIMARY KEY  (student_id,course_id)
);
```

3324CH05.qxd 9/21/04 9:53 AM Page 243

NOTE *We've constructed this in such a way that students may have double majors and/or minors. Limiting students to no more than two of each would need to be done in the application, as MySQL doesn't yet support triggers; we'll discuss this further in Chapter 8.*

While we've shown the foreign key relationships in Figure 5-1, we have not bothered to include them in the table definition statements. However, you should keep them mind, since join keys at least imply a foreign key relationship between the tables being joined, even if it's not made mandatory through the use of constraints.

As for the use of joins, consider the following problem: Suppose we want to know the name of the course area in which a given instructor teaches classes. We could do this by using two separate queries. First, we get the course area number from that instructor's record in the **instructors** table, and then we plug that number into the **courses** table to obtain the name of the corresponding course:

```
SELECT @cnum := course_id FROM instructors
  WHERE firstname = 'Mary' AND lastname = 'Williams';
SELECT name FROM courses WHERE course_id = @cnum;
```

Notice that we employ a user variable in order to preserve the result of the first query and make it available to the second. As you learned in Chapter 4, this frees us from the need to create, set, and refer to an additional application variable in programming code.

Here's what happens when we run these two queries from the MySQL command line:

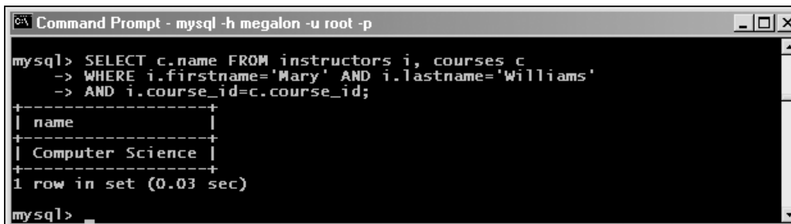
```
Command Prompt - mysql -h megalon -u root -p

mysql> SELECT @cnum:=course_id FROM instructors
    -> WHERE firstname='Mary' AND lastname='Williams';
+------------------+
| @cnum:=course_id |
+------------------+
|                1 |
+------------------+
1 row in set (0.11 sec)

mysql> SELECT name FROM courses WHERE course_id=@cnum;
+------------------+
| name             |
+------------------+
| Computer Science |
+------------------+
1 row in set (0.12 sec)

mysql> _
```

Since the course number (**course_id** column) is common to both tables, we can write a single query joining both tables using this as the common key for the

join. We merely combine any conditions required by each of the original two queries and set the columns common to both tables equal to one another:

```
SELECT c.name
  FROM instructors i, courses c
  WHERE i.firstname = 'Mary' AND i.lastname = 'Williams'
  AND i.course_id = c.course_id;
```

Here's the result:



This result is the same as that obtained by using the two previous queries in succession.

Theta-Style Joins vs. ANSI-Style Joins

There are two accepted styles for writing joins, known as *theta-style joins* and *ANSI-style joins*. Perhaps the best way to explain the difference is to show an example. Let's suppose we want a listing of all instructors that shows the names of the courses of study for which they teach classes. Since the names of the instructors are in one table (**instructors**) and those of the courses are in another (**courses**), we'll need to execute a join on these two tables in order to obtain the desired set of data.

Theta-style join syntax uses commas to separate multiple table names and aliases, just as in the previous example:

```
SELECT c.name
  FROM instructors i, courses c
  WHERE i.firstname = 'Mary' AND i.lastname = 'Williams'
  AND i.course_id = c.course_id;
```

ANSI syntax uses the `JOIN` and `ON` keywords instead:

```
SELECT c.name
  FROM instructors i JOIN courses c
  ON i.course_id = c.course_id
  WHERE i.firstname = 'Mary' AND i.lastname = 'Williams';
```

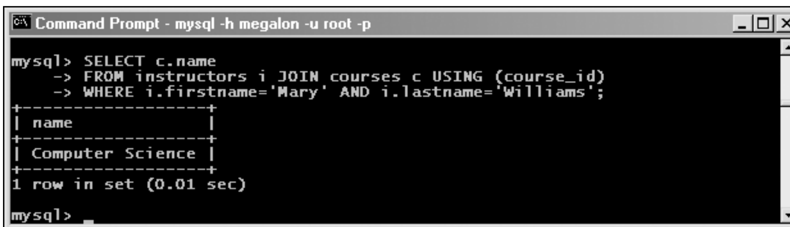
3324CH05.qxd 9/21/04 9:53 AM Page 245

The JOIN keyword is used to separate the names of the tables being joined, and the ON clause contains the equality relation showing which column is being used as the join key. Both varieties of join syntax are permissible in MySQL; however, the ANSI syntax is generally preferable because it's usually easier to read and understand, particularly when writing joins involving three or more tables. There are also some types of joins that can't be written using theta-style notation in MySQL, as you'll see shortly.

In addition, MySQL supports a nonstandard extension of the ANSI syntax that can be used as a sort of shorthand for when the join column has the same name in both tables to be joined:

```
SELECT c.name
  FROM instructors i JOIN courses c
  USING (course_id)
  WHERE i.firstname = 'Mary' AND i.lastname = 'Williams';
```

This has the same result as our earlier example.



The USING keyword is not supported in other database systems; however, if portability isn't an issue, it can be handy for eliminating a bit of typing, as well as for conceptualization purposes.

Join Types

When joining two tables together, MySQL can handle rows that are or aren't matched in one or both tables in several different ways. We'll look briefly at each of these in turn.

Cross Join

Each row from the first table in a *cross join* is joined to all rows from the second. Also known as the *Cartesian product* of two tables, this type of join yields

3324CH05.qxd 9/21/04 9:53 AM Page 246

extremely large resultsets, the size of the resultset being the product of the number of rows in each table. Here is an example of a cross join written using theta-style notation:

```
SELECT i.firstname, i.lastname, c.name
FROM instructors i, courses c;
```

Using ANSI-style notation, we would write this as follows:

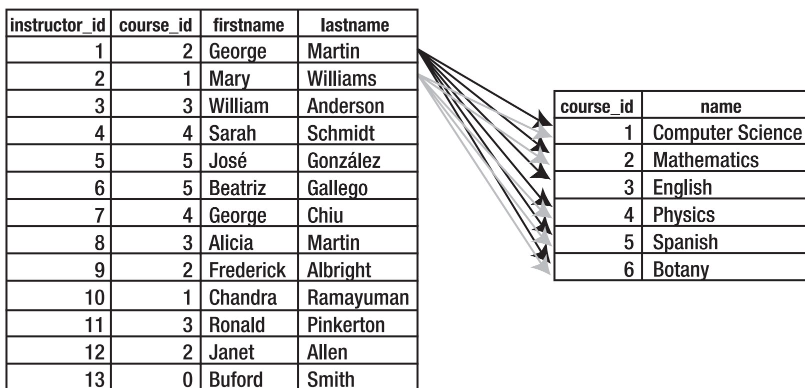
```
SELECT i.firstname, i.lastname, c.name
  FROM instructors i
  JOIN courses c;
```

or like this:

```
SELECT i.firstname, i.lastname, c.name
  FROM instructors i
  CROSS JOIN courses c;
```

The two ANSI-style forms are equivalent in MySQL.

The reason for this multiplication might be more apparent if you visualize a cross join as shown in Figure 5-2. Very simply, every row in the left-hand table of the join is matched to every row in the table on the right. For the sake of clarity, we've indicated only the matches on the first two rows of the **instructors** table, but you should be able to extrapolate from this and see that there will be 6 × 13 = 78 rows in the resultset. (Don't worry that we're asking you to take this as merely an



instructor_id	course_id	firstname	lastname
1	2	George	Martin
2	1	Mary	Williams
3	3	William	Anderson
4	4	Sarah	Schmidt
5	5	José	González
6	5	Beatriz	Gallego
7	4	George	Chiu
8	3	Alicia	Martin
9	2	Frederick	Albright
10	1	Chandra	Ramayuman
11	3	Ronald	Pinkerton
12	2	Janet	Allen
13	0	Buford	Smith

course_id	name
1	Computer Science
2	Mathematics
3	English
4	Physics
5	Spanish
6	Botany

Figure 5-2. A cross join matches every row in the tables.

3324CH05.qxd 9/21/04 9:53 AM Page 247

article of faith; we'll offer proof of a more concrete sort very shortly.) Note that the last row in **instructors**, for which there are no records in the **courses** table having the same value in the **course_id** column, is still matched against every row in the right-hand table.

Assuming that we're using the data supplied in students-data.sql, the resultset produced by this query (written in any of the three ways shown) would contain 78 rows (for 13 instructors and 6 course areas). Cross joins are very inefficient due to the sheer size of their resultsets and to the fact that, given *a* equal to the number of rows in the first table and *b* equal to the number of rows in the second, the proportion of redundant data in the result will be:

$$\frac{a(b-1)}{ba} = 1 - \frac{1}{b}$$

In the example shown, approximately 92% (12/13) of the data returned is repetitive and therefore useless to us.

NOTE *If a join condition is not specified for any other type of join (except a natural join), MySQL will treat it as a cross join. This is true for most other databases as well.*

Inner Join

An *inner join* is defined as a join in which unmatched rows from either table are not to be returned.

Writing inner joins using the theta-style notation is just a matter of adding an appropriate `WHERE` clause that relates the columns comprising the join key:

```
SELECT i.firstname, i.lastname, c.name
  FROM instructors i, courses c
  WHERE i.course_id = c.course_id;
```

This join will produce a list of all instructors with the names of the course areas in which they teach.

To accomplish the same thing in an ANSI-style join, use an `ON` or `USING` clause to define the join key:

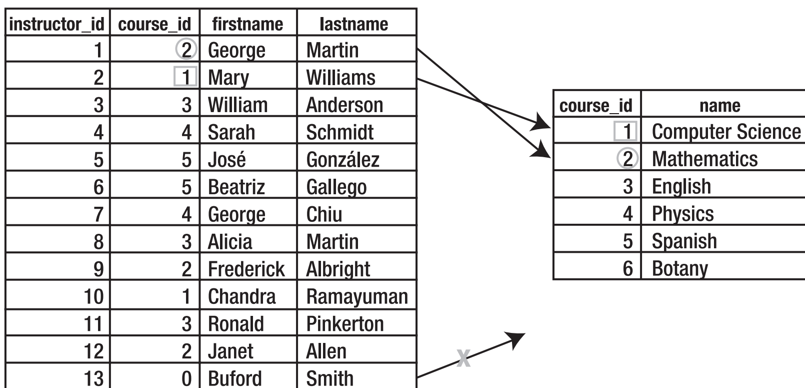
```
SELECT i.firstname, i.lastname, c.name
  FROM instructors i
  JOIN courses c
  USING (course_id);
```

3324CH05.qxd 9/21/04 9:53 AM Page 248

The USING keyword is specific to MySQL; the ANSI-standard equivalent to this join is as follows:

```
SELECT i.firstname, i.lastname, c.name
  FROM instructors i
  JOIN courses c
  ON i.course_id = c.course_id;
```

This query is illustrated in Figure 5-3, which shows how just a few of the rows on the left correspond to rows in the table on the right. The last row in the **instructors** table has a **course_id** value of 0; since there are no rows in courses with that value in the **course_id** column, the row from **instructors** isn't included in the resultset. This is indicated by the X over the arrow in the diagram in Figure 5-3.



instructor_id	course_id	firstname	lastname
1	2	George	Martin
2	1	Mary	Williams
3	3	William	Anderson
4	4	Sarah	Schmidt
5	5	José	González
6	5	Beatriz	Gallego
7	4	George	Chiu
8	3	Alicia	Martin
9	2	Frederick	Albright
10	1	Chandra	Ramayuman
11	3	Ronald	Pinkerton
12	2	Janet	Allen
13	0	Buford	Smith

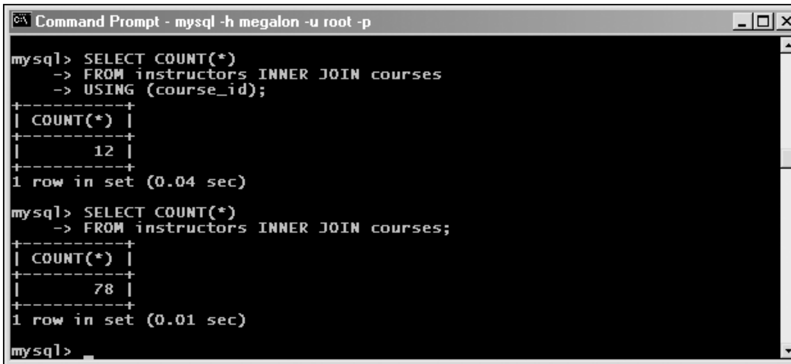
course_id	name
1	Computer Science
2	Mathematics
3	English
4	Physics
5	Spanish
6	Botany

Figure 5-3. An inner join does not return unmatched rows.

CAUTION *You should never place restrictions on the rows to be returned in a join's* ON *clause; only join conditions of the form* t1.col1 = t2.col2 *(where* t1 *and* t2 *are table aliases) should be placed here. Any restrictions intended to filter the resultset should be placed in a* WHERE *clause.*

3324CH05.qxd 9/21/04 9:53 AM Page 249

You may use the optional INNER keyword as well. However, you should note that using this does *not* by itself make your query into an inner join; in fact, without an ON or USING clause, MySQL will still treat the query as a cross join. Compare the following two queries.



Unless there's an actual need to find rows in one table that aren't matched in another (and sometimes there can be, as you'll see shortly), inner joins are generally the most efficient joins to use. There's no point in returning records you don't need, and you'll save time and effort by not being required to filter out NULL rows from your results.

NOTE *If a join condition but no join type is specified in an ANSI-style join, MySQL will treat the join as an inner join, just as SQL Server and PostgreSQL will.*

Left (Outer) Join

Outer joins differ from inner joins in that outer joins will return records in one table that aren't matched in another. In a *left outer join* (or more simply, *left-hand join* or even just *left join*), all records from the first (left-hand) table in a join that meet any conditions set in the WHERE clause are returned, whether or not there's a match in the second (right-hand) table.

For example, let's suppose we would like a list of all instructors whose last name begins with the letter *A*, along with any classes that they teach. If we want a list including only those instructors who actually teach any classes, we use an inner join:

3324CH05.qxd 9/21/04 9:53 AM Page 250


```
Command Prompt - mysql -h megalon -u root -p
mysql> SELECT i.firstname, i.lastname, c.name
    -> FROM instructors i
    -> INNER JOIN classes c
    -> USING (instructor_id)
    -> WHERE i.lastname LIKE 'A%';
+-----------+-----------+--------------------------------+
| firstname | lastname  | name                           |
+-----------+-----------+--------------------------------+
| Frederick | Albright  | Calculus                       |
| Frederick | Albright  | Introduction to Matrices       |
| Frederick | Albright  | Modern Geometry                |
| Frederick | Albright  | Probability & Statistics       |
| William   | Anderson  | English Grammar and Composition |
| William   | Anderson  | The Romantic Period            |
| William   | Anderson  | Shakespearean Plays and Sonnets |
+-----------+-----------+--------------------------------+
7 rows in set (0.07 sec)

mysql> _
```

As previously mentioned, the `INNER` keyword is optional. We could also use theta-style syntax for this query:

```
SELECT i.firstname, i.lastname, c.name
  FROM instructors I, classes c
  WHERE i.instructor_id = c.instructor_id
  AND i.lastname LIKE 'A%';
```

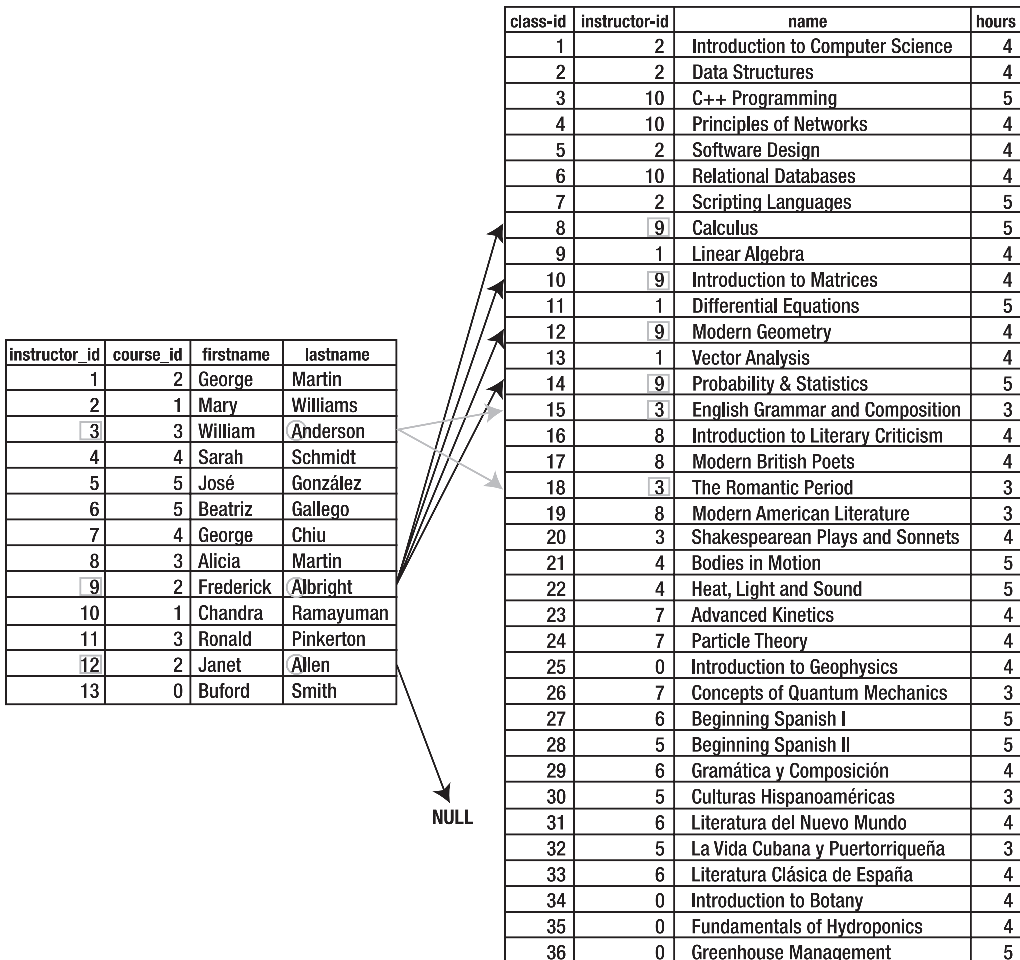
Using a left join, we can obtain a list of all instructors whose last name begins with *A*, whether or not there are any matching entries for those instructors in the **classes** table. As you can see from Figure 5-4, there are three instructors whose last names begin with the letter *A*: William Anderson's instructor ID matches that listed for three classes, and the instructor ID for Frederick Albright is the same as that of four classes. Janet Allen's instructor ID doesn't match with that for any classes at all; since this is a left join, we show an arrow pointing from her record in the **instructors** table to the word *NULL*.

The query and its result are as follows.

```
Command Prompt - mysql -h megalon -u root -p
mysql> SELECT i.firstname, i.lastname, c.name
    -> FROM instructors i
    -> LEFT JOIN classes c
    -> USING (instructor_id)
    -> WHERE i.lastname LIKE 'A%';
+-----------+-----------+--------------------------------+
| firstname | lastname  | name                           |
+-----------+-----------+--------------------------------+
| William   | Anderson  | English Grammar and Composition |
| William   | Anderson  | The Romantic Period            |
| William   | Anderson  | Shakespearean Plays and Sonnets |
| Frederick | Albright  | Calculus                       |
| Frederick | Albright  | Introduction to Matrices       |
| Frederick | Albright  | Modern Geometry                |
| Frederick | Albright  | Probability & Statistics       |
| Janet     | Allen     | NULL                           |
+-----------+-----------+--------------------------------+
8 rows in set (0.01 sec)

mysql> _
```

There are no matching class records for the instructor named Janet Allen, so MySQL dutifully returns a row containing her first and last names in the corresponding columns from **instructors** and a `NULL` value for the **name** column that was requested from the **classes** table.

3324CH05.qxd 9/21/04 9:53 AM Page 251



instructor_id	course_id	firstname	lastname
1	2	George	Martin
2	1	Mary	Williams
3	3	William	Anderson
4	4	Sarah	Schmidt
5	5	José	González
6	5	Beatriz	Gallego
7	4	George	Chiu
8	3	Alicia	Martin
9	2	Frederick	Albright
10	1	Chandra	Ramayuman
11	3	Ronald	Pinkerton
12	2	Janet	Allen
13	0	Buford	Smith

class-id	instructor-id	name	hours
1	2	Introduction to Computer Science	4
2	2	Data Structures	4
3	10	C++ Programming	5
4	10	Principles of Networks	4
5	2	Software Design	4
6	10	Relational Databases	4
7	2	Scripting Languages	5
8	9	Calculus	5
9	1	Linear Algebra	4
10	9	Introduction to Matrices	4
11	1	Differential Equations	5
12	9	Modern Geometry	4
13	1	Vector Analysis	4
14	9	Probability & Statistics	5
15	3	English Grammar and Composition	3
16	8	Introduction to Literary Criticism	4
17	8	Modern British Poets	4
18	3	The Romantic Period	3
19	8	Modern American Literature	3
20	3	Shakespearean Plays and Sonnets	4
21	4	Bodies in Motion	5
22	4	Heat, Light and Sound	5
23	7	Advanced Kinetics	4
24	7	Particle Theory	4
25	0	Introduction to Geophysics	4
26	7	Concepts of Quantum Mechanics	3
27	6	Beginning Spanish I	5
28	5	Beginning Spanish II	5
29	6	Gramática y Composición	4
30	5	Culturas Hispanoaméricas	3
31	6	Literatura del Nuevo Mundo	4
32	5	La Vida Cubana y Puertorriqueña	3
33	6	Literatura Clásica de España	4
34	0	Introduction to Botany	4
35	0	Fundamentals of Hydroponics	4
36	0	Greenhouse Management	5

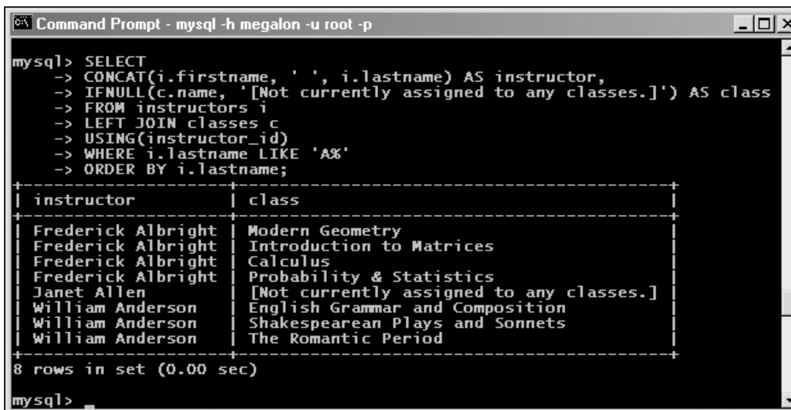
Figure 5-4. A left (outer) join returns all records from the left-hand table that meet any conditions in the `WHERE` *clause, whether or not there's a match in the right-hand table.*

Let's consider for a minute what happens when we encounter this `NULL` value in an application. We probably wouldn't want to display the word *NULL*, since it's not very descriptive. Instead, we would likely prefer something a bit more user-friendly, along the lines of "No classes assigned." Rather than test for the `NULL` value in our application code and make a suitable substitution there, we can use a flow-control operator to accomplish the same thing in the join itself. While we're at it, let's reduce the number of columns in the output to two by using the `CONCAT()` function on the instructor's first and last names to form a single **instructor** column. We'll also include an `ORDER BY` clause in the query to alphabetize the results by the instructor's last name.

3324CH05.qxd 9/21/04 9:53 AM Page 252


```
SELECT
  CONCAT(i.firstname, ' ', i.lastname) AS instructor,
  IFNULL(c.name, '[Not currently assigned to any classes.]') AS class
FROM instructors i
LEFT JOIN classes c
USING(instructor_id)
WHERE i.lastname LIKE 'A%'
ORDER BY i.lastname;
```

The result looks like this in the MySQL Monitor:



Generally speaking, you can employ any operators, built-in functions, and flow-control operators in multiple-table joins that you could use in `SELECT` queries from a single table.

NOTE *MySQL does* not *support Oracle-style* `(+) =` *or* `= (+)` *theta syntax for outer joins. If you need to specify a left join, you must use ANSI syntax with* `LEFT JOIN` *or* `LEFT OUTER JOIN`. *The same is true with respect to right joins: use* `RIGHT JOIN` *or* `RIGHT OUTER JOIN`. *Oracle 9 also implements the SQL92 syntax supported by MySQL, as does PostgreSQL 7.1 and later.*

3324CH05.qxd 9/21/04 9:53 AM Page 253

Right (Outer) Join

A *right outer join* (or more, commonly, *right* or *right-hand join*) is similar to a left join, except that all rows from the second (or right-hand) table in the join that satisfy any included `WHERE` clause are returned, whether or not matching rows are found in the first (left-hand) table. MySQL supports ANSI-style right joins using either the `RIGHT JOIN` or `RIGHT OUTER JOIN` keywords.

As you can see in Figure 5-5, this works as you would expect: in the opposite fashion from a left join.

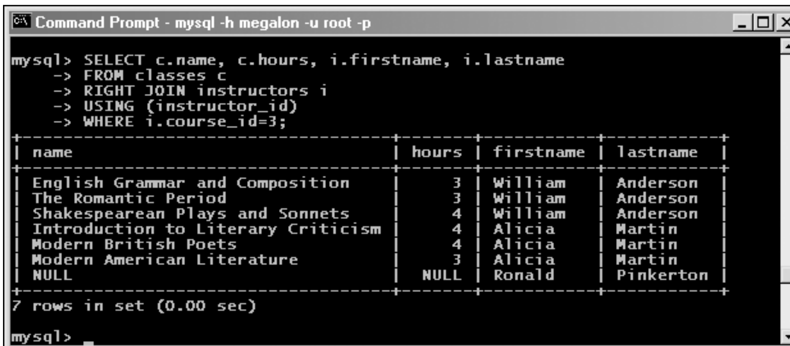
class-id	instructor-id	name	hours
1	2	Introduction to Computer Science	4
2	2	Data Structures	4
3	10	C++ Programming	5
4	10	Principles of Networks	4
5	2	Software Design	4
6	10	Relational Databases	4
7	2	Scripting Languages	5
8	9	Calculus	5
9	1	Linear Algebra	4
10	9	Introduction to Matrices	4
11	1	Differential Equations	5
12	9	Modern Geometry	4
13	1	Vector Analysis	4
14	9	Probability & Statistics	5
15	3	English Grammar and Composition	3
16	8	Introduction to Literary Criticism	4
17	8	Modern British Poets	4
18	3	The Romantic Period	3
19	8	Modern American Literature	3
20	3	Shakespearean Plays and Sonnets	4
21	4	Bodies in Motion	5
22	4	Heat, Light and Sound	5
23	7	Advanced Kinetics	4
24	7	Particle Theory	4
25	0	Introduction to Geophysics	4
26	7	Concepts of Quantum Mechanics	3
27	6	Beginning Spanish I	5
28	5	Beginning Spanish II	5
29	6	Gramática y Composición	4
30	5	Culturas Hispanoaméricas	3
31	6	Literatura del Nuevo Mundo	4
32	5	La Vida Cubana y Puertorriqueña	3
33	6	Literatura Clásica de España	4
34	0	Introduction to Botany	4
35	0	Fundamentals of Hydroponics	4
36	0	Greenhouse Management	5

instructor_id	course_id	firstname	lastname
1	2	George	Martin
2	1	Mary	Williams
3	3	William	Anderson
4	4	Sarah	Schmidt
5	5	José	González
6	5	Beatriz	Gallego
7	4	George	Chiu
8	3	Alicia	Martin
9	2	Frederick	Albright
10	1	Chandra	Ramayuman
11	3	Ronald	Pinkerton
12	2	Janet	Allen
13	0	Buford	Smith

NULL

Figure 5-5. A right (outer) join returns all records from the right-hand table that meet any conditions in the `WHERE` *clause, whether or not there's a match in the left-hand table.*

The following shows the query represented in Figure 5-5 being run in the MySQL Monitor.



You can see that there are three instructors whose **course_id** column value is 3 (that is, they all teach English classes). Two instructors teach three classes each, and the third (Ronald Pinkerton) isn't listed as teaching any classes at all.

In this case, `NULL` values are returned in both columns in the **classes** table for rows that don't match any **instructor_id** values from the instructors table. If this seems a bit confusing, try turning it into a left join:

```
SELECT c.name, c.hours, i.firstname, i.lastname
FROM instructors i
LEFT JOIN classes c
USING(instructor_id)
WHERE i.course_id = 3;
```

If you run this query in the MySQL Monitor, you'll find that the results are exactly the same as those produced by the previous right join.

TIP *Left or right? In most cases, it really doesn't matter whether you use left joins or right joins, as long as the tables to be joined are in the correct order. However, the recommended practice by most professionals is to use left joins whenever possible. Using either one or the other (but not both) is desirable for reasons of consistency. In addition, most people seem to find left joins easier to visualize than right joins when reading and writing queries.*

Full Join

A *full join* returns all rows from both tables being joined that otherwise fulfill any conditions set in a query's `WHERE` clause. All columns in rows from either table that don't have matches in the other one are filled with `NULL` values.

3324CH05.qxd 9/21/04 9:53 AM Page 255

MySQL doesn't support explicit full joins; instead we'll offer a couple of alternative ways to simulate a full join later in this chapter, in the "Emulating a Full Join Using a UNION Query" and "Emulating a Full Join Using a Temporary Table" sections.

CAUTION *Some references state that the default join type in MySQL is the full join, but this is incorrect usage of the term* full join, *where* cross join *is what's really meant.*

Natural Join

A *natural join* is a MySQL-specific shortcut that performs the same task as an inner or left join in which the `ON` or `USING` clause refers to all columns that the tables to be joined have in common. Using this form:

```
SELECT i.firstname, i.lastname, c.name
FROM instructors i
NATURAL JOIN classes c
WHERE i.lastname LIKE 'A%';
```

is the same as using this form:

```
SELECT i.firstname, i.lastname, c.name
FROM instructors i
INNER JOIN classes c
USING(instructor_id)
WHERE i.lastname LIKE 'A%';
```

Similarly, you can make MySQL assume automatically that all same-named columns are to be used as join keys for a left outer join:

```
SELECT i.firstname, i.lastname, c.name
FROM instructors i
NATURAL LEFT JOIN classes c
WHERE i.lastname LIKE 'A%';
```

This yields the same result as the following:

```
SELECT i.firstname, i.lastname, c.name
FROM instructors i
LEFT JOIN classes c
USING(instructor_id)
WHERE i.lastname LIKE 'A%';
```

CAUTION *You can't use the* INNER *keyword with* NATURAL JOIN. *You'll get a syntax error if you try to do this.*

You can also perform natural right joins, as in the following example.

```
Command Prompt - mysql -h megalon -u root -p

mysql> SELECT i.firstname, i.lastname, c.name
    -> FROM instructors i
    -> NATURAL RIGHT JOIN courses c;
+-----------+-----------+------------------+
| firstname | lastname  | name             |
+-----------+-----------+------------------+
| Mary      | Williams  | Computer Science |
| Chandra   | Ramayuman | Computer Science |
| George    | Martin    | Mathematics      |
| Frederick | Albright  | Mathematics      |
| Janet     | Allen     | Mathematics      |
| William   | Anderson  | English          |
| Alicia    | Martin    | English          |
| Ronald    | Pinkerton | English          |
| Sarah     | Schmidt   | Physics          |
| George    | Chiu      | Physics          |
| Jos⊖      | GonzßIez  | Spanish          |
| Beatriz   | Gallego   | Spanish          |
| NULL      | NULL      | Botany           |
+-----------+-----------+------------------+
13 rows in set (0.00 sec)

mysql> _
```

Since there are no instructors in the Botany department, the columns from the left-hand table in the row containing "Botany" from the right-hand table are filled with NULL values.

By using the same name for related columns in different tables and NATURAL [LEFT | RIGHT] JOIN syntax, you can save a lot of typing in your joins. The principal drawback to the USING notation is that this isn't portable from MySQL to other databases. It's also true that someone who is not familiar with your table schemas may need to look them up before being to able to know for certain on which columns the tables in the query are being joined.

TIP *You can also use* NATURAL LEFT OUTER JOIN *and* NATURAL RIGHT OUTER JOIN*, in addition to what's shown in the examples.*

Self Join

Self joins aren't used often, but they are very handy for one particular purpose: retrieving information that represents a hierarchy. Suppose we want to model the supervisory responsibilities for personnel in an office department, such as that represented by the tree graph in Figure 5-6.

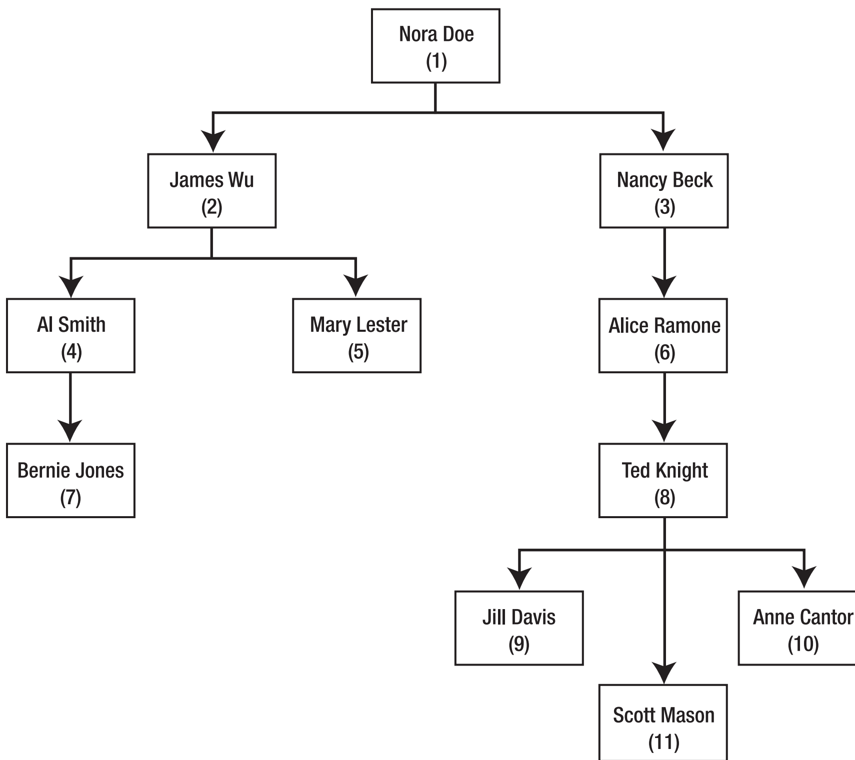


Figure 5-6. A hierarchy of department personnel

As shown here, Nora Doe (Employee #1) supervises James Wu (Employee #2) and Nancy Beck (Employee #3), James Wu supervises Al Smith (Employee #4) and Mary Lester (Employee #5), and so on. It's fairly straightforward to construct a table to hold this data:

```
CREATE TABLE employees (
  employee_id int(11) NOT NULL auto_increment,
  supervisor_id int(11) default NULL,
  firstname varchar(50) NOT NULL default '',
  lastname varchar(50) NOT NULL default '',
  PRIMARY KEY  (employee_id)
);
```

This table contains an **employee_id** column to store the employee's ID number, a **supervisor_id** column to hold the employee ID of the supervisor to whom the employee reports, and **firstname** and **lastname** columns to store the employee's first and last names. We'll allow the **supervisor_id** column of this table to take a `NULL` value to indicate an employee with no supervisors. The first two statements for inserting the data into this table look like this:

3324CH05.qxd 9/21/04 9:53 AM Page 258


```
INSERT INTO employees (employee_id, supervisor_id, firstname, lastname)
  VALUES (1, NULL, 'Nora', 'Doe');
INSERT INTO employees (employee_id, supervisor_id, firstname, lastname)
  VALUES (2, 1, 'James', 'Wu');
```

NOTE *Writing the remaining* `INSERT` *statements based on the diagram in Figure 5-6 should be a trivial exercise. Alternatively, you can use the self-join.sql script included in the ch5 directory of the code download for this book (available from the Downloads section of* `http://www.apress.com`*) to create and populate the* ***employees*** *table.*

Data that refers to other data stored in the same table exhibits what's commonly referred to as a *recursive,* or *reflexive, relationship*. A self join is used to extract this data in such a way that this recursive relationship is preserved, and it works by joining the table to itself. Since we can refer to the same table identifier (table name or alias) only once in a given query, we indulge in a couple of bits of alias trickery here:

- We use the same table name twice, but use a different table alias each time. In effect, we're telling MySQL to refer to the same table under two different guises.
- Since the columns which we're retrieving have the same names, we use column aliases to distinguish the columns returned from the table under the first table alias from those returned from the table under the table second alias.

We'll take the additional step of concatenating the first and last names of each employee as well. The resulting query might look like this:

```
Command Prompt - mysql -h megalon -u root -p

mysql> SELECT
    -> CONCAT(e1.firstname, ' ', e1.lastname) AS supervisor,
    -> CONCAT(e2.firstname, ' ', e2.lastname) AS employee
    -> FROM employees e1
    -> JOIN employees e2 ON e1.employee_id = e2.supervisor_id;
+--------------+--------------+
| supervisor   | employee     |
+--------------+--------------+
| Nora Doe     | James Wu     |
| Nora Doe     | Nancy Beck   |
| James Wu     | Al Smith     |
| James Wu     | Mary Lester  |
| Nancy Beck   | Alice Ramone |
| Al Smith     | Bernie Jones |
| Alice Ramone | Ted Knight   |
| Ted Knight   | Jill Davis   |
| Ted Knight   | Anne Cantor  |
| Ted Knight   | Scott Mason  |
+--------------+--------------+
10 rows in set (0.05 sec)

mysql> _
```

3324CH05.qxd 9/21/04 9:53 AM Page 259

If we want to include a record showing that Nora Doe is the department head (that she has no immediate supervisor), we can do that using an outer join. Since we're displaying the supervisor column on the left side of the output, we'll need to use a right join:

```
Command Prompt - mysql -h megalon -u root -p

mysql> SELECT
    -> IFNULL(CONCAT(e1.firstname, ' ', e1.lastname), "[DEPARTMENT HEAD]")
    ->   AS supervisor,
    -> CONCAT(e2.firstname, ' ', e2.lastname) AS employee
    -> FROM employees e1
    -> RIGHT JOIN employees e2 ON e1.employee_id = e2.supervisor_id;
+-------------------+--------------+
| supervisor        | employee     |
+-------------------+--------------+
| [DEPARTMENT HEAD] | Nora Doe     |
| Nora Doe          | James Wu     |
| Nora Doe          | Nancy Beck   |
| James Wu          | Al Smith     |
| James Wu          | Mary Lester  |
| Nancy Beck        | Alice Ramone |
| Al Smith          | Bernie Jones |
| Alice Ramone      | Ted Knight   |
| Ted Knight        | Jill Davis   |
| Ted Knight        | Anne Cantor  |
| Ted Knight        | Scott Mason  |
+-------------------+--------------+
11 rows in set (0.00 sec)

mysql> _
```

Notice that we use `IFNULL()` once again in order to substitute a descriptive message in place of the word *NULL* in Nora's employee record, making use of the fact that the result of concatenating any value to `NULL` is also `NULL`. If we wanted to use a left join, we could rewrite this query as follows:

```
SELECT
  IFNULL(CONCAT(e2.firstname, ' ', e2.lastname), "[DEPARTMENT HEAD]")
    AS supervisor,
  CONCAT(e1.firstname, ' ', e1.lastname) AS employee
FROM employees e1
LEFT JOIN employees e2 ON e2.employee_id = e1.supervisor_id;
```

We just switch the aliases used for the columns to be selected and in the `ON` clause.

Other likely scenarios for using self joins include relating parts of items that themselves are used as parts of other items; department hierarchies in an organization; and sections and subsections of a document, an application, or a web site.

Unions

Beginning with MySQL 4.0, you can use the `UNION` keyword to combine the results of multiple `SELECT` queries into a single resultset. This can be very useful

3324CH05.qxd 9/21/04 9:53 AM Page 260

in eliminating the need to store resultsets in programming data structures such as arrays or to employ temporary tables (which we'll look at very shortly) in order to preserve intermediate results.

The basic syntax for UNION is as follows:

```
(SELECT ...)
UNION [DISTINCT | ALL]
(SELECT ...)
[UNION
(SELECT ...) [...]]
```

The SELECT statements can be any that are legal in MySQL, as long as each query yields the same number of columns. The parentheses surrounding the individual SELECT statements are required if you want to use an ORDER BY clause that affects the combined resultset. However, it's good practice to use parentheses in any case, to make your queries easier to read.

CAUTION *Prior to MySQL 4.1.1, there's a further restriction in that the values of the columns in the first* SELECT *query's resultset are used to determine the result types and lengths of the same columns for the combined resultset. This means that column values from the second query and any additional ones might be truncated or otherwise altered in order to match the sizes and types of those resulting from the first* SELECT.

Let's look at a simple example. Suppose we have two tables listing a small firm's salespeople and service technicians. Here's the definition for the **sales** table:

```
CREATE TABLE sales (
  firstname varchar(50) NULL,
  lastname varchar(50) NULL
);
```

NOTE *The table definitions and data for this example are included in the union.sql file in the ch5 directory of this book's downloadable code.*

The structure of the **service_techs** table is identical to this one. In order to obtain a combined listing of all the employees from both tables in a single result, we can do this:

```
Command Prompt - mysql -h megalon -u root -p

mysql> (SELECT lastname, firstname FROM sales)
    -> UNION
    -> (SELECT lastname, firstname FROM service_techs)
    -> ORDER BY lastname;
+----------+-----------+
| lastname | firstname |
+----------+-----------+
| Anderson | Jane      |
| Bridges  | Lucinda   |
| Fields   | Hope      |
| Griffith | George    |
| Miller   | Lisette   |
| Nelson   | Mike      |
| Norton   | Steve     |
| Roberts  | Peter     |
| Roberts  | Denise    |
| Thomas   | Jerry     |
| Williams | Franklin  |
| Yates    | Mandy     |
+----------+-----------+
12 rows in set (0.11 sec)

mysql> _
```

Notice that the `ORDER BY` clause following the second `SELECT` (and outside the parentheses surrounding it) controls the sort order for the entire resultset.

By default, MySQL eliminates any duplicated rows from the combined resultset. Beginning with MySQL 4.0.17, you can indicate this behavior using the `DISTINCT` keyword. While it actually has no effect (since it represents the default behavior), it can serve as a reminder that you're dropping any duplicates. The `DISTINCT` keyword is also required by the SQL standard, so using it will make your code more portable as well.

If we want *all* rows to be included in the final result, we can use the `ALL` keyword, like so:

```
Command Prompt - mysql -h megalon -u root -p

mysql> (SELECT lastname, firstname FROM sales)
    -> UNION ALL
    -> (SELECT lastname, firstname FROM service_techs)
    -> ORDER BY lastname;
+----------+-----------+
| lastname | firstname |
+----------+-----------+
| Anderson | Jane      |
| Anderson | Jane      |
| Bridges  | Lucinda   |
| Fields   | Hope      |
| Griffith | George    |
| Griffith | George    |
| Miller   | Lisette   |
| Nelson   | Mike      |
| Norton   | Steve     |
| Roberts  | Denise    |
| Roberts  | Peter     |
| Thomas   | Jerry     |
| Thomas   | Jerry     |
| Williams | Franklin  |
| Yates    | Mandy     |
+----------+-----------+
15 rows in set (0.00 sec)

mysql> _
```

Emulating a Full Join Using a UNION Query

In some cases, you can `UNION` together a left join and a right join to simulate a full join, as shown in the following example.

3324CH05.qxd 9/21/04 9:53 AM Page 262


```
Command Prompt - mysql -h megalon -u root -p
mysql> (
    -> SELECT CONCAT(s.firstname, ' ', s.lastname) AS salesperson,
    -> CONCAT(t.firstname, ' ', t.lastname) AS tech
    -> FROM sales s
    -> LEFT JOIN service_techs t
    -> USING (firstname, lastname)
    -> )
    -> UNION
    -> (
    -> SELECT CONCAT(s.firstname, ' ', s.lastname) AS salesperson,
    -> CONCAT(t.firstname, ' ', t.lastname) AS tech
    -> FROM sales s
    -> RIGHT JOIN service_techs t
    -> USING (firstname, lastname)
    -> );
+-------------------+-----------------+
| salesperson       | tech            |
+-------------------+-----------------+
| Jane Anderson     | Jane Anderson   |
| Franklin Williams | NULL            |
| Jerry Thomas      | Jerry Thomas    |
| Lisette Miller    | NULL            |
| Peter Roberts     | NULL            |
| Mandy Yates       | NULL            |
| Lucinda Bridges   | NULL            |
| George Griffith   | George Griffith |
| Hope Fields       | NULL            |
| NULL              | Denise Roberts  |
| NULL              | Steve Norton    |
| NULL              | Mike Nelson     |
+-------------------+-----------------+
12 rows in set (0.39 sec)

mysql> _
```

In order for this to work, all the columns in the first query's resultset must accept `NULL` values. (This is true through MySQL 5.0.0.)

As an exercise, try writing a query (using the union.sql tables and data as a basis) whose output looks something like this:

EMPLOYEE	SALES	TECH
Jane Anderson	X	X
Franklin Williams	X	
Jerry Thomas	X	X
[*etc. . .*		]
Steve Norton		X
Mike Nelson		X

CAUTION *Any column names used in an* `ORDER BY` *clause applying to an entire union must be common to the resultsets produced by all of the* `SELECT` *queries making up the union.*

We'll look at another method for simulating full joins using temporary tables a little later in this chapter.

Temporary Tables

Now let's talk about another advanced MySQL feature: *temporary tables*. These allow you to create a short-term storage place within the database itself for a set of data that you need to use several times in a single series of operations. One advantage of this is that you can use SQL to access the data, rather than using programming code, which means that if you need to port your application from, say, PHP to Java, there's that much less code to be translated. There are additional benefits to using temporary tables, as you'll see shortly.

Most often, it's best to obtain a desired set of data in a single `SELECT` query. However, sometimes this simply isn't possible, or you may want to work with subsets of the same, larger resultset over several successive operations. You can handle intermediate or temporary results for reuse within a single session in two basic ways:

- Using programming constructs such as arrays, hashes, or objects and retrieving data from these when required by the application logic
- Using database tables

The second option is preferable because it tends to be faster, there's less likelihood of bugs in the programming code, and applications are more easily ported (the latter two reasons derive from the simple fact that there's less code to manage). The one drawback to doing this is that you're then required to manage the tables for storing the intermediate results. The solution to this problem is to use temporary tables, which are supported in MySQL 3.23 and later.

NOTE *Beginning with MySQL 5.0.1, views may offer another alternative for handling intermediate or temporary results for reuse. See Chapter 8 for more information about views.*

Creating Temporary Tables

To create a temporary table, simply include the `TEMPORARY` keyword in a table creation statement. Otherwise, the statements used to create them are no different than those used to create normal tables.

A temporary table can be of type MyISAM, HEAP, MERGE, or InnoDB. (Your MySQL installation must support InnoDB tables in order to use these as temporary tables, of course.) You can also use ISAM as the table type for temporary tables in MySQL versions through the 4.0.*x* series. (Don't forget that ISAM tables are disabled in MySQL 4.1 and removed altogether beginning with MySQL 5.0.0.)

Temporary tables differ from normal tables in that temporary tables exist only for the duration of the current session and are automatically deleted after it ends. (For web applications, this means that temporary tables generally cease to exist upon completion of the current page or script.) The following is a simple example (this particular example was produced on a PC running Windows 2000 Server, but the results will be the same regardless of operating system or platform).

```
Command Prompt - mysql -h megalon -u root -p

C:\>mysql -h megalon -u root -p
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 5.0.0-alpha-max-nt-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE test;
Database changed
mysql> CREATE TEMPORARY TABLE temptable (val INT);
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO temptable (val) VALUES (123);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM temptable;
+------+
| val  |
+------+
|  123 |
+------+
1 row in set (0.00 sec)

mysql> \q
Bye

C:\>mysql -h megalon -u root -p
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.0.0-alpha-max-nt-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE test;
Database changed
mysql> SELECT * FROM temptable;
ERROR 1146 (42S02): Table 'test.temptable' doesn't exist
mysql> _
```

Notice that the table was deleted as soon as the session was ended using the `\q` (or `quit`) command. You can verify this by logging in again as the same user and trying to query the table you created previously. One interesting and potentially useful side effect of this behavior is that multiple users can employ the same table names for temporary tables without fear of collisions between them. This also means that different users can't access each other's temporary tables. If one user or process needs to access a table created by a different one, you'll need to omit the `TEMPORARY` keyword from the table definition and take care of maintenance issues (such as deleting the table once there's no further use for it) in your application code.

CAUTION *In MySQL, you can't refer to the same temporary table more than once in a single query. This means you can't do a self join on a temporary table.*

Emulating a Full Join Using a Temporary Table

We've already looked at one way of duplicating what a full join does by using a UNION query. Now we'll demonstrate how to simulate a full join with a temporary table, in the event you're working with a MySQL version earlier than 4.0. This will be fairly straightforward, but should serve as a good example of the use of a temporary table.

We'll use the same **sales** and **service_techs** tables defined and populated in the union.sql script, which we used in our earlier example of emulating a full join. We perform the emulation in four steps:

1. Create the temporary table used to store the intermediate result.
2. Insert the data returned by a left join on the two employees tables.
3. Insert the data from a right join on the same two tables.
4. Select the final data set from the temporary table.

We'll actually combine steps 1 and 2 using CREATE TABLE ... SELECT syntax, as this is just as valid for temporary tables as it is for normal ones. The left join is simply the one we used earlier in the first part of the UNION join.

```
Command Prompt - mysql -h megalon -u root -p
mysql> CREATE TEMPORARY TABLE temptable
    -> SELECT
    ->   CONCAT(s.firstname, ' ', s.lastname) AS salesperson,
    ->   CONCAT(t.firstname, ' ', t.lastname ) AS tech
    -> FROM sales s
    -> LEFT JOIN service_techs t
    -> USING (firstname, lastname);
Query OK, 9 rows affected (0.02 sec)
Records: 9  Duplicates: 0  Warnings: 0

mysql> _
```

Here is the schema of the temporary table we just created:

```
Command Prompt - mysql -h megalon -u root -p

mysql> DESCRIBE temptable;
+-------------+-----------+------+-----+---------+-------+
| Field       | Type      | Null | Key | Default | Extra |
+-------------+-----------+------+-----+---------+-------+
| salesperson | char(101) | YES  |     | NULL    |       |
| tech        | char(101) | YES  |     | NULL    |       |
+-------------+-----------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> _
```

3324CH05.qxd 9/21/04 9:53 AM Page 266

And here's the data that we inserted:

```
mysql> SELECT * FROM temptable;
+-------------------+-----------------+
| salesperson       | tech            |
+-------------------+-----------------+
| Jane Anderson     | Jane Anderson   |
| Franklin Williams | NULL            |
| Jerry Thomas      | Jerry Thomas    |
| Lisette Miller    | NULL            |
| Peter Roberts     | NULL            |
| Mandy Yates       | NULL            |
| Lucinda Bridges   | NULL            |
| George Griffith   | George Griffith |
| Hope Fields       | NULL            |
+-------------------+-----------------+
9 rows in set (0.00 sec)

mysql> _
```

Now we need to insert the data from a right join on the same two tables. We already have the necessary `SELECT` query for this. All we need to do is turn it into an `INSERT SELECT`, as follows.

```
mysql> INSERT INTO temptable (salesperson, tech)
    ->   SELECT
    ->     CONCAT(s.firstname, ' ', s.lastname) AS salesperson,
    ->     CONCAT(t.firstname, ' ', t.lastname) AS tech
    ->   FROM sales s
    ->   RIGHT JOIN service_techs t
    ->   USING (firstname, lastname);
Query OK, 6 rows affected (0.04 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> _
```

The temporary table now contains the following data:

```
mysql> SELECT * FROM temptable;
+-------------------+-----------------+
| salesperson       | tech            |
+-------------------+-----------------+
| Jane Anderson     | Jane Anderson   |
| Franklin Williams | NULL            |
| Jerry Thomas      | Jerry Thomas    |
| Lisette Miller    | NULL            |
| Peter Roberts     | NULL            |
| Mandy Yates       | NULL            |
| Lucinda Bridges   | NULL            |
| George Griffith   | George Griffith |
| Hope Fields       | NULL            |
| Jane Anderson     | Jane Anderson   |
| NULL              | Denise Roberts  |
| Jerry Thomas      | Jerry Thomas    |
| NULL              | Steve Norton    |
| NULL              | Mike Nelson     |
| George Griffith   | George Griffith |
+-------------------+-----------------+
15 rows in set (0.00 sec)

mysql> _
```

We have some duplicate rows. We can take care of this problem easily enough by using a `SELECT DISTINCT` instead:

3324CH05.qxd 9/21/04 9:54 AM Page 267


```
Command Prompt - mysql -h megalon -u root -p

mysql> SELECT DISTINCT * FROM temptable;
+--------------------+------------------+
| salesperson        | tech             |
+--------------------+------------------+
| Jane Anderson      | Jane Anderson    |
| Franklin Williams  | NULL             |
| Jerry Thomas       | Jerry Thomas     |
| Lisette Miller     | NULL             |
| Peter Roberts      | NULL             |
| Mandy Yates        | NULL             |
| Lucinda Bridges    | NULL             |
| George Griffith    | George Griffith  |
| Hope Fields        | NULL             |
| NULL               | Denise Roberts   |
| NULL               | Steve Norton     |
| NULL               | Mike Nelson      |
+--------------------+------------------+
12 rows in set (0.00 sec)

mysql> _
```

This is the same data that we obtained using the `UNION` join earlier.

NOTE *You can also use temporary tables to simulate subselects in older versions of MySQL that don't support these. We'll look at how this can be accomplished when we discuss subqueries in Chapter 8.*

Transactions

Both joins and temporary tables can help reduce the amount of application code needed to get a given amount of work out of MySQL. A third way to streamline your applications that use MySQL is to employ transactions.

A *transaction* is simply a group of related operations that make up a logical or conceptual whole. For example, when you make a purchase using a credit card from a web site, your cardholder account must be debited and the site owner's merchant account must be credited. When problems such as a power outage, a system failure, or loss of a network connection occur, it is possible that all of these necessary operations may not take place, causing data integrity problems. (For example, your card is debited, but the merchant's account doesn't get the credit for the sale.) Transactional features have been added to most popular databases in order to provide a solution to exactly this type of problem. Since version 3.23-max and all 4.0 versions and later, MySQL has provided transactional support using the InnoDB and BDB storage engines.

In MySQL (when using InnoDB or BDB tables) or in any other transactional database, a transaction must follow what are known as the *ACID rules*:

- **Atomicity:** All operations associated with any given transaction must occur as a single unit. If any single operation fails, then the transaction does not take place, and the database is returned to its previous state. This is often stated like this: In the event of the failure of an operation that makes up part of the transaction, the transaction is not *committed*, and the transaction is *rolled back*.

- **Consistency:** The system's state following the transaction must be consistent with its original state. For example, if you're transferring money between two checking accounts, the total of the two accounts must be the same before and after the transfer takes place. If the balance of one account has increased by $100, then the balance of the other account must have decreased by the same amount.

- **Isolation:** Each transaction must appear to be independent of all other actions taking place in the system. In other words (to use our banking example again), the system must behave as though the two accounts between which funds are being transferred have exclusive use of the system while this transaction is taking place. (In practice, true concurrent isolation is fairly difficult to achieve without causing major performance problems, and what usually happens instead is some sort of sequential prioritization.)

- **Durability:** Simply put, once a transaction is completed, it must stay completed. This is achieved in MySQL and other transaction-safe databases by means of a transaction log file, which is updated whenever a transaction is completed.

The main benefit of using transactional tables with your application is that you can ensure data integrity and concurrency in the event that something unexpected may occur. This is very important in the business world, such as for financial institutions.

NOTE *Both InnoDB and BDB tables support transactions, but there are some problems with BDB tables. They tend to be slow when large numbers of them are simultaneously open; they can't be moved between directories; and you may not be able to delete BDB tables unless you're running in auto-commit mode. Also, BDB tables currently aren't supported on Linux running on other than 32-bit Intel processors, and they aren't supported at all on Mac OS X. For these and other reasons, we recommend that you use InnoDB tables when you need to support transactions in a production environment.*

MySQL Transaction Commands

MySQL has three commands for use in performing transactions:

BEGIN: This command begins a transaction. MySQL supports BEGIN WORK as an alias for this command. Since MySQL 4.0.11, you can also use the standard SQL command START TRANSACTION. Alternatively, you may use SET AUTOCOMMIT = 0;. With all of these commands, changes are not written to disk and logged until a COMMIT statement has been issued. The difference between SET AUTOCOMMIT = 0; and the others is that it disables auto-commit mode until you explicitly turn it back on (by issuing a SET AUTOCOMMIT = 1;); the others merely disable auto-commit mode temporarily, until either a COMMIT or ROLLBACK command has been issued.

ROLLBACK: If there is a failure in any of the queries required for the transaction, you can issue this command to cancel the transaction and return the database to its previous state. Issuing a ROLLBACK requires the "undoing" of each query that was successful prior to the advent of the error condition that necessitated it. Fortunately, thanks to improved versioning techniques, this doesn't take as long as you might think it would.

COMMIT: Once all required queries are completed successfully, this commits the transaction; that is, all changes are saved to the transaction log on disk or in other permanent storage. COMMIT statements execute fairly rapidly, since there is no requirement for any actual additional work in the database to be done, only that operations be recorded.

In MySQL you cannot nest transactions. If you issue a BEGIN statement for a given user while that user has a pending transaction, MySQL will treat it as a COMMIT followed by a BEGIN. In other words, MySQL allows individual users to perform only *sequential* transactions.

NOTE *MySQL does not support save points as do some other databases. You can only commit or roll back a complete transaction.*

Transaction Processing Considerations

It's best to keep transactions as small as possible. Since MySQL must guarantee that transactions belonging to different users are kept separate, this means that table rows involved in those transactions must be kept locked (and thus not accessible by other others) for the duration of each transaction. In addition, since transactions must be logged as sets of queries, using the minimum

number of SQL statements possible per transaction ensures that your application isn't slowed down by the need to log a large number of queries before the next transaction can commence.

You should perform all transactions before and after obtaining user input. Don't write your application in such a way that a transaction is in progress while awaiting user response. Imagine what could happen if a user decided to leave for his lunch break while the database was waiting for one or more of his transactions to finish! Plan your application in such a way that all necessary data is obtained, then the transaction performed, before collecting the next item or set of data.

Summary

In this chapter, we've looked at three features that can help increase the efficiency of your MySQL applications: joins (including `UNION` joins), temporary tables, and transactions. Each of these can be used in different ways to cut down on the number of queries required to isolate the data you need, reduce or eliminate the need to store data as part of application logic, decrease the number of bugs in programming code, and help guarantee data integrity.

Using joins reduces the number of queries needed to obtain a given set of data by allowing you to combine queries. These also tend to make applications more efficient because they can often eliminate the need to store data in programming constructs so that they can be used in subsequent queries. They make applications less error-prone because less code means less of a chance for bugs to creep into the codebase. We looked at all of the major join types supported by MySQL.

Temporary tables are useful when you want to store the results of queries in tables for reuse several times during a single user or application session, rather than in programming constructs such as arrays or hashes. MySQL's temporary tables are unique to each session, and they are cleaned up for you at the end of each session, which cuts down drastically on table management requirements and worries about issues such as table name collisions. They're also useful in cases where it's unwieldy or even impossible to derive a desired set of data in a single query, such as in our example using a temporary table to simulate a full join, which isn't currently supported in MySQL.

Transactions, supported in MySQL via the InnoDB and BDB storage engines, are useful because they provide a mechanism whereby queries can be grouped together logically and performed as a unit. This is extremely important in scenarios where the state of the database must be preserved; for example, in the case of transferring funds between checking accounts, where the total of the amounts in both accounts must be the same before and after the transfer. Without transactions, you must lock all affected tables and track the success or failure of each single query required to complete the transfer, and in the event of failure, you

3324CH05.qxd 9/21/04 9:54 AM Page 271

must be prepared to perform the inverse of each query that succeeded up to that point. (There are also concurrency and other issues involved, but we won't dwell on those here.) Because transactions handle commits and rollbacks for you automatically, you're saved a tremendous amount of code overhead. Many database APIs that are compatible with MySQL provide enhanced support for transactions, including Python's MySQLdb module and `ext/mysqli` in PHP 5, as you'll see in Chapter 7, when we discuss MySQL application programming.

What's Next

In the next chapter, we'll look at additional ways to speed up and improve the efficiency of MySQL-backed applications by identifying bottlenecks in them as they're running. Some of these methods include analysis of log files and status variables, evaluation of table design and queries, caching issues, application bloat, and the configuration of the MySQL server.

3324CH05.qxd 9/21/04 9:54 AM Page 272