MySQL SQL Syntax and Use

By Paul Dubois

Feb 7, 2003

Retrieving Records from Multiple Tables

It does no good to put records in a database unless you retrieve them eventually and do something with them. That's the purpose of the SELECT statement—to help you get at your data. SELECT probably is used more often than any other in the SQL language, but it can also be the trickiest; the constraints you use to choose rows can be arbitrarily complex and can involve comparisons between columns in many tables.

The basic syntax of the SELECT statement looks like this:

```
SELECT selection_list          # What columns to select
FROM table_list                # Where to select rows from
WHERE primary_constraint       # What conditions rows must satisfy
GROUP BY grouping_columns      # How to group results
ORDER BY sorting_columns       # How to sort results
HAVING secondary_constraint    # Secondary conditions rows must sat
LIMIT count;                   # Limit on results
```

Everything in this syntax is optional except the word SELECT and the *selection_list* part that specifies what you want to retrieve. Some databases require the FROM clause as well. MySQL

does not, which allows you to evaluate expressions without referring to any tables:

```
SELECT SQRT(POW(3,2)+POW(4,2));
```

In Chapter 1, we devoted quite a bit of attention to single-table SELECT statements, concentrating primarily on the output column list and the WHERE, GROUP BY, ORDER BY, HAVING, and LIMIT clauses. This section covers an aspect of SELECT that is often confusing —writing joins; that is, SELECT statements that retrieve records from multiple tables. We'll discuss the types of join MySQL supports, what they mean, and how to specify them. This should help you employ MySQL more effectively because, in many cases, the real problem of figuring out how to write a query is determining the proper way to join tables together.

One problem with using SELECT is that when you first encounter a new type of problem, it's not always easy to see how to write a SELECT query to solve it. However, after you figure it out, you can use that experience when you run across similar problems in the future. SELECT is probably the statement for which past experience plays the largest role in being able to use it effectively, simply because of the sheer variety of problems to which it applies.

As you gain experience, you'll be able to adapt joins more easily to new problems, and you'll find yourself thinking things like, "Oh, yes, that's one of those LEFT JOIN things," or, "Aha, that's a three-way join restricted by the common pairs of key columns." (I'm a little reluctant to point that out, actually. You may find it encouraging to hear that experience helps you. On the other hand, you may find it alarming to consider that you could wind up thinking in terms like that!)

Many of the examples that demonstrate how to use the forms of join operations that MySQL supports use the following two tables, t1 and t2. They're small, which makes them simple enough that the effect of each type of join can be seen readily:

```
Table t1:   Table t2:
+----+----+  +----+----+
| i1 | c1 |  | i2 | c2 |
+----+----+  +----+----+
| 1  | a  |  | 2  | c  |
| 2  | b  |  | 3  | b  |
| 3  | c  |  | 4  | a  |
+----+----+  +----+----+
```

The Trivial Join

The simplest join is the trivial join, in which only one table is named. In this case, rows are selected from the named table:

```
mysql> SELECT * FROM t1;
+----+----+
| i1 | c1 |
+----+----+
| 1  | a  |
| 2  | b  |
| 3  | c  |
+----+----+
```

Some people don't consider this form of SELECT a join at all and use the term only for SELECT statements that retrieve records from two or more tables. I suppose it's a matter of perspective.

The Full Join

If a SELECT statement names multiple tables in the FROM clause with the names separated by commas, MySQL performs a full join. For example, if you join t1 and t2 as follows, each row in t1 is combined with each row in t2:

```
mysql> SELECT t1.*, t2.* FROM t1, t2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
| 1  | a  | 2  | c  |
```

```
| 2  | b  | 2  | c  |
| 3  | c  | 2  | c  |
| 1  | a  | 3  | b  |
| 2  | b  | 3  | b  |
| 3  | c  | 3  | b  |
| 1  | a  | 4  | a  |
| 2  | b  | 4  | a  |
| 3  | c  | 4  | a  |
+----+----+----+----+
```

A full join is also called a *cross join* because each row of each table is crossed with each row in every other table to produce all possible combinations. This is also known as the *cartesian product*. Joining tables this way has the potential to produce a very large number of rows because the possible row count is the product of the number of rows in each table. A full join between three tables that contain 100, 200, and 300 rows, respectively, could return 100x200x300 = 6 million rows. That's a lot of rows, even though the individual tables are small. In cases like this, a WHERE clause will normally be used to reduce the result set to a more manageable size.

If you add a WHERE clause causing tables to be matched on the values of certain columns, the join becomes what is known as an *equi-join* because you're selecting only rows with equal values in the specified columns:

```
mysql> SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
| 2  | b  | 2  | c  |
| 3  | c  | 3  | b  |
+----+----+----+----+
```

The JOIN and CROSS JOIN join types are equivalent to the ',' (comma) join operator. For example, the following statements are all the same:

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 CROSS JOIN t2 WHERE t1.i1 = t2.i2;
```

Normally, the MySQL optimizer considers itself free to determine the order in which to scan tables to retrieve rows most quickly. On occasion, the optimizer will make a non-optimal choice. If you find this happening, you can override the optimizer's choice using the STRAIGHT_JOIN keyword. A join performed with STRAIGHT_JOIN is like a cross join but forces the tables to be joined in the order named in the FROM clause.

STRAIGHT_JOIN can be specified at two points in a SELECT statement. You can specify it between the SELECT keyword and the selection list to have a global effect on all cross joins in the statement, or you can specify it in the FROM clause. The following two statements are equivalent:

```
SELECT STRAIGHT_JOIN ... FROM t1, t2, t3 ... ;
SELECT ... FROM t1 STRAIGHT_JOIN t2 STRAIGHT_JOIN t3 ... ;
```

Qualifying Column References

References to table columns throughout a SELECT statement must resolve unambiguously to a single table named in the FROM clause. If only one table is named, there is no ambiguity because all columns must be columns of that table. If multiple tables are named, any column name that appears in only one table is similarly unambiguous. However, if a column name appears in multiple tables, references to the column must be qualified by the table name using *tbl_name.col_name* syntax to specify which table you mean. Suppose a table mytbl1 contains columns a and b, and a table mytbl2 contains columns b and c. In this case, references to columns a or c are unambiguous, but references to b must be qualified as either mytbl1.b or mytbl2.b:

```
SELECT a, mytbl1.b, mytbl2.b, c FROM mytbl1, mytbl2 ... ;
```

Sometimes a table name qualifier is not sufficient to resolve a column reference. For example, if you're joining a table to itself, you're using it multiple times within the query and it doesn't help to qualify a column name with the table name. In this case, table aliases are useful for communicating your intent. You can assign an alias to any instance of the table and refer to columns from that instance as *alias_name.col_name*. The following query joins a table to itself, but assigns an alias to one instance of the table to allow column references to be specified unambiguously:

```
SELECT mytbl.col1, m.col2 FROM mytbl, mytbl AS m WHERE
mytbl.col1 > m.col1;
```

Left and Right Joins

An equi-join shows only rows where a match can be found in both tables. Left and right joins show matches, too, but also show rows in one table that have no match in the other table. The examples in this section use `LEFT JOIN`, which identifies rows in the left table that are not matched by the right table. `RIGHT JOIN` is the same except that the roles of the tables are reversed. (`RIGHT JOIN` is available only as of MySQL 3.23.25.)

A `LEFT JOIN` works like this: You specify the columns to be used for matching rows in the two tables. When a row from the left table matches a row from the right table, the contents of the rows are selected as an output row. When a row in the left table has no match, it is still selected for output, but joined with a "fake" row from the right table in which all the columns have been set to `NULL`. In other words, a `LEFT JOIN` forces the result set to contain a row for every row in the left table whether or not there is a match for it in the right table. The rows with no match can be identified by the fact that all columns from the right table are `NULL`.

Consider once again our two tables, `t1` and `t2`:

```
Table t1:   Table t2:
+----+----+  +----+----+
| i1 | c1 |  | i2 | c2 |
+----+----+  +----+----+
| 1  | a  |  | 2  | c  |
| 2  | b  |  | 3  | b  |
| 3  | c  |  | 4  | a  |
+----+----+  +----+----+
```

If we use a cross join to match these tables on `t1.i1` and `t2.i2`, we'll get output only for the values 2 and 3, which appear in both tables:

```
mysql> SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
| 2  | b  | 2  | c  |
| 3  | c  | 3  | b  |
+----+----+----+----+
```

A left join produces output for every row in `t1`, whether or not `t2` matches it. To write a left join, name the tables with `LEFT JOIN` in between (rather than a comma) and specify the matching condition using an `ON` clause (rather than a `WHERE` clause):

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
| 1  | a  | NULL | NULL |
| 2  | b  |  2   | c    |
| 3  | c  |  3   | b    |
+----+----+------+------+
```

Now there is an output row even for the value 1, which has no match in `t2`.

`LEFT JOIN` is especially useful when you want to find *only* those left table rows that are unmatched by the right table. Do this by adding a `WHERE` clause that looks for rows in the right table that have `NULL` values—in other words, the rows in one table that are missing from the

other:

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
  -> WHERE t2.i2 IS NULL;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
| 1  | a  | NULL | NULL |
+----+----+------+------+
```

Normally, what you're really after are the unmatched values in the left table. The NULL columns from the right table are of no interest for display purposes, so you wouldn't bother naming them in the output column list:

```
mysql> SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
  -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
| 1  | a  |
+----+----+
```

LEFT JOIN actually allows the matching conditions to be specified two ways. ON is one of these; it can be used whether or not the columns you're joining on have the same name:

```
SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
```

The other syntax involves a USING() clause; this is similar in concept to ON, but the name of the joined column or columns must be the same in each table. For example, the following query joins mytbl1.b to mytbl2.b:

```
SELECT mytbl1.*, mytbl2.* FROM mytbl1 LEFT JOIN mytbl2 USING (b);
```

LEFT JOIN has a few synonyms and variants. LEFT OUTER JOIN is a synonym for LEFT JOIN. There is also an ODBC-style notation for LEFT JOIN that MySQL accepts (the OJ means "outer join"):

```
{ OJ tbl_name1 LEFT OUTER JOIN tbl_name2 ON join_expr }
```

NATURAL LEFT JOIN is similar to LEFT JOIN; it performs a LEFT JOIN, matching all columns that have the same name in the left and right tables.

One thing to watch out for with LEFT JOIN is that if the columns that you're joining on are not declared as NOT NULL, you may get problematic rows in the result. For example, if the right table contains columns with NULL values, you won't be able to distinguish those NULL values from NULL values that identify unmatched rows.

As already mentioned, LEFT JOIN is useful for answering "Which values are missing?" questions. When you want to know which values in one table are not present in another table, you use a LEFT JOIN on the two tables and look for rows in which NULL is selected from the second table. Let's consider a more complex example of this type of problem than the one shown earlier using t1 and t2.

For the grade-keeping project first mentioned in Chapter 1, we have a student table listing students, an event table listing the grade events that have occurred, and a score table listing scores for each student for each grade event. However, if a student was ill on the day of some quiz or test, the score table wouldn't have any score for the student for that event, so a makeup quiz or test should be given. How do we find these missing records so that we can make sure those students take the makeup?

The problem is to determine which students have no score for a given grade event and to do this for each grade event. Another way to say this is that we want to find out which combinations of student and event are not represented in the score table. This "which values are not present" wording is a tip-off that we want a `LEFT JOIN`. The join isn't as simple as in the previous example, though, because we aren't just looking for values that are not present in a single column; we're looking for a two-column combination. The combinations we want are all the student/event combinations, which are produced by crossing the `student` table with the `event` table:

```
FROM student, event
```

Then we take the result of that join and perform a `LEFT JOIN` with the `score` table to find the matches:

```
FROM student, event
   LEFT JOIN score ON student.student_id = score.student.id
           AND event.event_id = score.event_id
```

Note that the `ON` clause allows the rows in the `score` table to be joined according to matches in different tables. That's the key for solving this problem. The `LEFT JOIN` forces a row to be generated for each row produced by the cross join of the `student` and `event` tables, even when there is no corresponding `score` table record. The result set rows for these missing score records can be identified by the fact that the columns from the `score` table will all be `NULL`. We can select these records in the `WHERE` clause. Any column from the `score` table will do, but because we're looking for missing scores, it's probably conceptually clearest to test the `score` column:

```
WHERE score.score IS NULL
```

We can put the results in order using an `ORDER BY` clause. The two most logical orderings are by event per student or by student per event. I'll choose the first:

```
ORDER BY student.student_id, event.event_id
```

Now all we need to do is name the columns we want to see in the output, and we're done. Here is the final query:

```
SELECT
  student.name, student.student_id,
  event.date, event.event_id, event.type
FROM
  student, event
  LEFT JOIN score ON student.student_id = score.student_id
          AND event.event_id = score.event_id
WHERE
  score.score IS NULL
ORDER BY
  student.student_id, event.event_id;
```

Running the query produces these results:

```
+-----------+------------+------------+----------+------+
| name      | student_id | date       | event_id | type |
+-----------+------------+------------+----------+------+
| Megan     |     1      | 2002-09-16 |    4     | Q    |
| Joseph    |     2      | 2002-09-03 |    1     | Q    |
| Katie     |     4      | 2002-09-23 |    5     | Q    |
| Devri     |     13     | 2002-09-03 |    1     | Q    |
| Devri     |     13     | 2002-10-01 |    6     | T    |
| Will      |     17     | 2002-09-16 |    4     | Q    |
| Avery     |     20     | 2002-09-06 |    2     | Q    |
```

```
| Gregory   |     23     | 2002-10-01 |    6     | T    |
| Sarah     |     24     | 2002-09-23 |    5     | Q    |
| Carter    |     27     | 2002-09-16 |    4     | Q    |
| Carter    |     27     | 2002-09-23 |    5     | Q    |
| Gabrielle |     29     | 2002-09-16 |    4     | Q    |
| Grace     |     30     | 2002-09-23 |    5     | Q    |
+-----------+------------+------------+----------+------+
```

Here's a subtle point. The output displays the student IDs and the event IDs. The `student_id` column appears in both the `student` and `score` tables, so at first you might think that the selection list could name either `student. student_id` or `score.student_id`. That's not the case because the entire basis for being able to find the records we're interested in is that all the `score` table fields are returned as `NULL`. Selecting `score.student_id` would produce only a column of `NULL` values in the output. The same principle applies to deciding which `event_id` column to display. It appears in both the `event` and `score` tables, but the query selects `event.event_id` because the `score.event_id` values will always be `NULL`.

Using Subselects

One of the features that MySQL 4.1 introduces is subselect support, which is a long-awaited capability that allows one `SELECT` query to be nested inside other. The following is an example that looks up the IDs for event records corresponding to tests (`'T'`) and uses them to select scores for those tests:

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM event WHERE type = 'T');
```

In some cases, subselects can be rewritten as joins. I'll show how to do that later in this section. You may find subselect rewriting techniques useful if your version of MySQL precedes 4.1.

A related feature that MySQL supports is the ability to delete or update records in one table based on the contents of another. For example, you might want to remove records in one table that aren't matched by any record in another, or copy values from columns in one table to columns in another. These types of operations are discussed in the "Multiple-Table Deletes and Updates" section later in this chapter.

There are several forms you can use to write subselects; this section surveys just a few of them.

- **Using a subselect to produce a reference value.** In this case, you want the inner `SELECT` to identify a single value to be used in comparisons with the outer `SELECT`. For example, to identify the scores for the quiz that took place on `'2002-09-23'`, use an inner `SELECT` to determine the quiz event ID, and then match score records against it in the outer `SELECT`:

  ```
  SELECT * FROM score
  WHERE event_id =
  (SELECT event_id FROM event WHERE date = '2002-09-23' AND
  ```

  With this form of subselect, where the inner query is preceded by a comparison operator, it's necessary that the inner join produce no more than a single value (that is, one row, one column). If it produces multiple values, the query will fail. (In some cases, it may be appropriate to satisfy this constraint by limiting the inner query result with `LIMIT 1`.)

  This form of subselect can be handy for situations where you'd be tempted to use an aggregate function in a `WHERE` clause. For example, to determine which president was born first, you might try the following:

  ```
  SELECT * FROM president WHERE birth = MIN(birth);
  ```

  That doesn't work because you can't use aggregates in `WHERE` clauses. (The `WHERE`

clause determines which records to select, but the value of `MIN()` isn't known until after the records have already been selected.) However, you can use a subselect to produce the minimum birth date as follows:

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

- **EXISTS and NOT EXISTS subselects.** These forms of subselects work by passing values from the outer query to the inner one to see whether they match the conditions specified in the inner query. For this reason, you'll need to qualify column names with table names if they are ambiguous (appear in more than one table). `EXISTS` and `NOT EXISTS` subselects are useful for finding records in one table that match or don't match records in another.
  Refer once again to our `t1` and `t2` tables:

```
Table t1:   Table t2:
+----+----+  +----+----+
| i1 | c1 |  | i2 | c2 |
+----+----+  +----+----+
| 1  | a  |  | 2  | c  |
| 2  | b  |  | 3  | b  |
| 3  | c  |  | 4  | a  |
+----+----+  +----+----+
```

  The following query identifies matches between the tables—that is, values that are present in both:

```
mysql> SELECT i1 FROM t1
  -> WHERE EXISTS (SELECT * FROM t2 WHERE t1.i1 = t2.i2);
+----+
| i1 |
+----+
| 2  |
| 3  |
+----+
```

  `NOT EXISTS` identifies non-matches—values in one table that are not present in the other:

```
mysql> SELECT i1 FROM t1
  -> WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.i1 = t2.
+----+
| i1 |
+----+
| 1  |
+----+
```

  With these forms of subselect, the inner query uses `*` as the output column list. There's no need to name columns explicitly because the inner query is assessed as true or false based on whether or not it returns rows, not based on the particular values that the rows may contain. In MySQL, you can actually write pretty much anything for the column selection list, but if you want to make it explicit that you're returning a true value when the inner `SELECT` succeeds, you might write the queries like this:

```
SELECT i1 FROM t1
WHERE EXISTS (SELECT 1 FROM t2 WHERE t1.i1 = t2.i2);
```

```
SELECT i1 FROM t1
WHERE NOT EXISTS (SELECT 1 FROM t2 WHERE t1.i1 = t2.i2);
```

- **`IN` and `NOT IN` subselects.** The `IN` and `NOT IN` forms of subselect should return a single column of values from the inner `SELECT` to be evaluated in a comparison in the outer `SELECT`. For example, the preceding `EXISTS` and `NOT EXISTS` queries can be written using `IN` and `NOT IN` syntax as follows:

```
mysql> SELECT i1 FROM t1 WHERE i1 IN (SELECT i2 FROM t2);
+----+
| i1 |
+----+
| 2  |
| 3  |
+----+
mysql> SELECT i1 FROM t1 WHERE i1 NOT IN (SELECT i2 FROM
+----+
| i1 |
+----+
| 1  |
+----+
```

Rewriting Subselects as Joins

For versions of MySQL prior to 4.1, subselects are not available. However, it's often possible to rephrase a query that uses a subselect in terms of a join. In fact, even if you have MySQL 4.1 or later, it's not a bad idea to examine queries that you might be inclined to write in terms of subselects; a join is sometimes more efficient than a subselect.

Rewriting Subselects That Select Matching Values

The following is an example query containing a subselect; it selects scores from the `score` table for all tests (that is, it ignores quiz scores):

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM event WHERE type = 'T');
```

The same query can be written without a subselect by converting it to a simple join:

```
SELECT score.* FROM score, event
WHERE score.event_id = event.event_id AND event.type = 'T';
```

As another example, the following query selects scores for female students:

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = '
```

This can be converted to a join as follows:

```
SELECT score.* FROM score, student
WHERE score.student_id = student.student_id AND student.sex = 'F'
```

There is a pattern here. The subselect queries follow this form:

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2a FROM table2 WHERE column2b = va
```

Such queries can be converted to a join using the following form:

```
SELECT table1.* FROM table1, table2
WHERE table1.column1 = table2.column2a AND table2.column2b = value
```

Rewriting Subselects That Select Non-Matching (Missing) Values

Another common type of subselect query searches for values in one table that are not present in another table. As we've seen before, the "which values are not present" type of problem is a clue that a `LEFT JOIN` may be helpful. The following is a query with a subselect that tests for students who are *not* listed in the `absence` table (it finds those students with perfect attendance):

```
SELECT * FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

This query can be rewritten using a `LEFT JOIN` as follows:

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.stu
WHERE absence.student_id IS NULL;
```

In general terms, the subselect query form is as follows:

```
SELECT * FROM table1
WHERE column1 NOT IN (SELECT column2 FROM table2);
```

A query having that form can be rewritten like this:

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL;
```

This assumes that `table2.column2` is declared as `NOT NULL`.

Retrieving from Multiple Tables with UNION

If you want to create a result set by selecting records from multiple tables one after the other, you can do that using a `UNION` statement. `UNION` is available as of MySQL 4, although prior to that you can use a couple of workarounds (shown later).

For the following examples, assume you have three tables, `t1`, `t2`, and `t3` that look like this:

```
mysql> SELECT * FROM t1;
+------+-------+
| i    | c     |
+------+-------+
|  1   | red   |
|  2   | blue  |
|  3   | green |
+------+-------+
mysql> SELECT * FROM t2;
+------+------+
| i    | c    |
+------+------+
|  -1  | tan  |
|  1   | red  |
+------+------+
mysql> SELECT * FROM t3;
+------------+------+
| d          | i    |
+------------+------+
| 1904-01-01 | 100  |
```

```
| 2004-01-01 | 200  |
| 2004-01-01 | 200  |
+------------+------+
```

Tables `t1` and `t2` have integer and character columns, and `t3` has date and integer columns. To write a `UNION` statement that combines multiple retrievals, just write several `SELECT` statements and put the keyword `UNION` between them. For example, to select the integer column from each table, do this:

```
mysql> SELECT i FROM t1 UNION SELECT i FROM t2 UNION SELECT i FRO
+------+
| i    |
+------+
|  1   |
|  2   |
|  3   |
|  -1  |
| 100  |
| 200  |
+------+
```

`UNION` has the following properties:

- The names and data types for the columns of the `UNION` result come from the names and types of the columns in the first `SELECT`. The second and subsequent `SELECT` statements in the `UNION` must select the same number of columns, but they need not have the same names or types. Columns are matched by position (not by name), which is why these two queries return different results:

  ```
  mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3;
  +------+------------+
  | i  | c     |
  +------+------------+
  |  1 | red    |
  |  2 | blue    |
  |  3 | green   |
  | 100 | 1904-01-01 |
  | 200 | 2004-01-01 |
  +------+------------+
  mysql> SELECT i, c FROM t1 UNION SELECT d, i FROM t3;
  +------+-------+
  | i  | c   |
  +------+-------+
  |  1 | red  |
  |  2 | blue |
  |  3 | green |
  | 1904 | 100  |
  | 2004 | 200  |
  +------+-------+
  ```

  In both cases, the columns selected from `t1` (`i` and `c`) determine the types used in the `UNION` result. These columns have integer and string types, so type conversion takes place when selecting values from `t3`. For the first query, `d` is converted from date to string. That happens to result in no loss of information. For the second query, `d` is converted from date to integer (which *does* lose information), and `i` is converted from integer to string.

- By default, `UNION` eliminates duplicate rows from the result set:

  ```
  mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELE
  +------+-------+
  | i    | c     |
  +------+-------+
  |  1   | red   |
  ```

```
|  2   | blue  |
|  3   | green |
|  -1  | tan   |
| 1904 | 100   |
| 2004 | 200   |
+------+-------+
```

`t1` and `t2` both have a row containing values of `1` and `'red'`, but only one such row appears in the output. Also, `t3` has two rows containing `'2004-01-01'` and `200`, one of which has been eliminated.

- If you want to preserve duplicates, follow the first `UNION` keyword with `ALL`:

```
mysql> SELECT * FROM t1 UNION ALL SELECT * FROM t2 UNION
+------+-------+
| i    | c     |
+------+-------+
|  1   | red   |
|  2   | blue  |
|  3   | green |
|  -1  | tan   |
|  1   | red   |
| 1904 | 100   |
| 2004 | 200   |
| 2004 | 200   |
+------+-------+
```

- To sort a `UNION` result, add an `ORDER BY` clause after the last `SELECT`; it applies to the query result as a whole. However, because the `UNION` uses column names from the first `SELECT`, the `ORDER BY` should refer to those names, not the column names from the last `SELECT`, if they differ.

```
mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3
  -> ORDER BY c;
+------+------------+
| i    | c          |
+------+------------+
| 100  | 1904-01-01 |
| 200  | 2004-01-01 |
|  2   | blue       |
|  3   | green      |
|  1   | red        |
+------+------------+
```

- You can also specify an `ORDER BY` clause for an individual `SELECT` statement within the `UNION`. To do this, enclose the `SELECT` (including its `ORDER BY`) within parentheses:

```
mysql> (SELECT i, c FROM t1 ORDER BY i DESC)
  -> UNION (SELECT i, c FROM t2 ORDER BY i);
+------+-------+
| i    | c     |
+------+-------+
|  3   | green |
|  2   | blue  |
|  1   | red   |
|  -1  | tan   |
+------+-------+
```

- `LIMIT` can be used in a `UNION` in a manner similar to that for `ORDER BY`. If added to the end of the statement, it applies to the `UNION` result as a whole:

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELE
  -> LIMIT 1;
+------+------+
| i    | c    |
+------+------+
|  1   | red  |
+------+------+
```

- If enclosed within parentheses as part of an individual SELECT statement, it applies only to that SELECT:

```
mysql> (SELECT * FROM t1 LIMIT 1)
  -> UNION (SELECT * FROM t2 LIMIT 1)
  -> UNION (SELECT * FROM t3 LIMIT 1);
+------+------+
| i    | c    |
+------+------+
|  1   | red  |
|  -1  | tan  |
| 1904 | 100  |
+------+------+
```

  You need not select from different tables. You can select different subsets of the same table using different conditions. This can be useful as an alternative to running several different SELECT queries, because you get all the rows in a single result set rather than as several result sets.

Prior to MySQL 4, UNION is unavailable, but you can work around this difficulty by selecting rows from each table into a temporary table and then selecting the contents of that table. In MySQL 3.23 and later, you can handle this problem easily by allowing the server to create the holding table for you. Also, you can make the table a temporary table so that it will be dropped automatically when your session with the server terminates. For quicker performance, use a HEAP (in-memory) table.

```
CREATE TEMPORARY TABLE tmp TYPE = HEAP SELECT ... FROM t1 WHERE .
INSERT INTO tmp SELECT ... FROM t2 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t3 WHERE ... ;
...
SELECT * FROM tmp ORDER BY ... ;
```

Because tmp is a TEMPORARY table, the server will drop it automatically when your client session ends. (Of course, you can drop the table explicitly as soon as you're done with it to allow the server to free resources associated with it. This is a good idea if you will continue to perform further queries, particularly for HEAP tables.)

For versions of MySQL older than 3.23, the concept is similar, but the details differ because the HEAP table type and TEMPORARY tables are unavailable, as is CREATE TABLE ... SELECT. To adapt the preceding procedure, it's necessary to explicitly create the table first before retrieving any rows into it. (The only table type available will be ISAM, so you cannot use a TYPE option.) Then retrieve the records into the table. When you're done with it, you must use DROP TABLE explicitly because the server will not drop it automatically.

```
CREATE TABLE tmp (column1, column2, ...);
INSERT INTO tmp SELECT ... FROM t1 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t2 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t3 WHERE ... ;
SELECT * FROM tmp ORDER BY ... ;
DROP TABLE tmp;
```

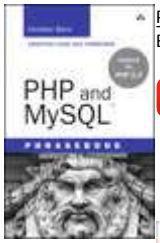
If you want to run a UNION-type query on MyISAM tables that have identical structure, you may be able to set up a MERGE table and query that as a workaround for lack of UNION. (In fact, this can be useful even if you do have UNION, because a query on a MERGE table will be simpler than the corresponding UNION query.) A query on the MERGE table is similar to a UNION that selects

corresponding columns from the individual tables that make up the MERGE table. That is, `SELECT` on a MERGE table is like `UNION ALL` (duplicates are not removed), and `SELECT DISTINCT` is like `UNION` (duplicates are removed).

You might also like:

MySQL, 5th Edition
By Paul Dubois

Learn More



PHP and MySQL Phrasebook
By Christian Wenz

Learn More