

*Solutions and Examples for MySQL
Database Developers.*

Covers
MySQL 4.01



MySQL Cookbook

O'REILLY®

Paul DuBois

MySQL Cookbook

Paul DuBois

Record Selection Techniques

3.0 Introduction

This chapter focuses on the `SELECT` statement that is used for retrieving information from a database. It provides some essential background that shows various ways you can use `SELECT` to tell MySQL what you want to see. You should find the chapter helpful if your SQL background is limited or if you want to find out about the MySQL-specific extensions to `SELECT` syntax. However, there are so many ways to write `SELECT` queries that we'll necessarily touch on just a few. You may wish to consult the MySQL Reference Manual or a MySQL text for more information about the syntax of `SELECT`, as well as the functions and operators that you can use for extracting and manipulating data.

`SELECT` gives you control over several aspects of record retrieval:

- Which table to use
- Which columns to display from the table
- What names to give the columns
- Which rows to retrieve from the table
- How to sort the rows

Many useful queries are quite simple and don't specify all those things. For example, some forms of `SELECT` don't even name a table—a fact used in Recipe 1.31, which discusses how to use *mysql* as a calculator. Other non-table-based queries are useful for purposes such as checking what version of the server you're running or the name of the current database:

```
mysql> SELECT VERSION(), DATABASE();
+-----+-----+
| VERSION() | DATABASE() |
+-----+-----+
| 3.23.51-log | cookbook |
+-----+-----+
```

However, to answer more involved questions, normally you'll need to pull information from one or more tables. Many of the examples in this chapter use a table named `mail`, which contains columns used to maintain a log of mail message traffic between users on a set of hosts. Its definition looks like this:

```
CREATE TABLE mail
(
    t          DATETIME,    # when message was sent
    srcuser   CHAR(8),     # sender (source user and host)
    srchost   CHAR(20),
    dstuser   CHAR(8),     # recipient (destination user and host)
    dsthost   CHAR(20),
    size      BIGINT,      # message size in bytes
    INDEX    (t)
);
```

And its contents look like this:

t	srcuser	srchost	dstuser	dsthost	size
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-12 15:02:49	phil	mars	phil	saturn	1048
2001-05-13 13:59:18	barb	saturn	tricia	venus	271
2001-05-14 09:31:37	gene	venus	barb	mars	2291
2001-05-14 11:52:17	phil	mars	tricia	saturn	5781
2001-05-14 14:42:21	barb	venus	barb	venus	98151
2001-05-14 17:03:01	tricia	saturn	phil	venus	2394482
2001-05-15 07:17:48	gene	mars	gene	saturn	3824
2001-05-15 08:50:57	phil	venus	phil	venus	978
2001-05-15 10:25:52	gene	mars	tricia	saturn	998532
2001-05-15 17:35:31	gene	saturn	gene	mars	3856
2001-05-16 09:00:28	gene	venus	barb	mars	613
2001-05-16 23:04:19	phil	venus	barb	venus	10294
2001-05-17 12:49:23	phil	mars	tricia	saturn	873
2001-05-19 22:21:51	gene	saturn	gene	venus	23992

To create the `mail` table and load its contents, change location into the `tables` directory of the recipes distribution and run this command:

```
% mysql cookbook < mail.sql
```

This chapter also uses other tables from time to time. Some of these were used in previous chapters, while others are new. For any that you need to create, do so the same way as for the `mail` table, using scripts in the `tables` directory. In addition, the text for many of the scripts and programs used in the chapter may be found in the `select` directory. You can use the files there to try out the examples more easily.

Many of the queries shown here can be tried out with `mysql`, which you can read about in Chapter 1. Some of the examples issue queries from within the context of a programming language. See Chapter 2 for background on programming techniques.

3.1 Specifying Which Columns to Display

Problem

You want to display some or all of the columns from a table.

Solution

Use `*` as a shortcut that selects all columns. Or name the columns you want to see explicitly.

Discussion

To indicate what kind of information you want to see from a table, name a column or a list of columns and the table to use. The easiest way to select output columns is to use the `*` specifier, which is a shortcut for naming all the columns in a table:

```
mysql> SELECT * FROM mail;
+-----+-----+-----+-----+-----+-----+
| t           | srcuser | srchost | dstuser | dsthost | size |
+-----+-----+-----+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | saturn | tricia  | mars   | 58274 |
| 2001-05-12 12:48:13 | tricia | mars   | gene    | venus  | 194925 |
| 2001-05-12 15:02:49 | phil   | mars   | phil    | saturn | 1048 |
| 2001-05-13 13:59:18 | barb   | saturn | tricia  | venus  | 271 |
...

```

Alternatively, you can list the columns explicitly:

```
mysql> SELECT t, srcuser, srchost, dstuser, dsthost, size FROM mail;
+-----+-----+-----+-----+-----+-----+
| t           | srcuser | srchost | dstuser | dsthost | size |
+-----+-----+-----+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | saturn | tricia  | mars   | 58274 |
| 2001-05-12 12:48:13 | tricia | mars   | gene    | venus  | 194925 |
| 2001-05-12 15:02:49 | phil   | mars   | phil    | saturn | 1048 |
| 2001-05-13 13:59:18 | barb   | saturn | tricia  | venus  | 271 |
...

```

It's certainly easier to use `*` than to write out a list of column names. However, with `*`, there is no guarantee about the order in which columns will be returned. (The server returns them in the order they are listed in the table definition, but this may change if you change the definition. See Chapter 8.) Thus, one advantage of naming the columns explicitly is that you can place them in whatever order you want. Suppose you want hostnames to appear before usernames, rather than after. To accomplish this, name the columns as follows:

```
mysql> SELECT t, srchost, srcuser, dsthost, dstuser, size FROM mail;
+-----+-----+-----+-----+-----+-----+
| t           | srchost | srcuser | dsthost | dstuser | size |
+-----+-----+-----+-----+-----+-----+

```

```

| 2001-05-11 10:15:08 | saturn | barb | mars | tricia | 58274 |
| 2001-05-12 12:48:13 | mars   | tricia | venus | gene   | 194925 |
| 2001-05-12 15:02:49 | mars   | phil  | saturn | phil   | 1048   |
| 2001-05-13 13:59:18 | saturn | barb | venus | tricia | 271    |
...

```

Another advantage of naming the columns compared to using `*` is that you can name just those columns you want to see and omit those in which you have no interest:

```

mysql> SELECT size FROM mail;
+-----+
| size |
+-----+
| 58274 |
| 194925 |
| 1048 |
| 271 |
...
mysql> SELECT t, srcuser, srchost, size FROM mail;
+-----+-----+-----+-----+
| t | srcuser | srchost | size |
+-----+-----+-----+-----+
| 2001-05-11 10:15:08 | barb | saturn | 58274 |
| 2001-05-12 12:48:13 | tricia | mars | 194925 |
| 2001-05-12 15:02:49 | phil | mars | 1048 |
| 2001-05-13 13:59:18 | barb | saturn | 271 |
...

```

3.2 Avoiding Output Column Order Problems When Writing Programs

Problem

You're issuing a `SELECT *` query from within a program, and the columns don't come back in the order you expect.

Solution

When you use `*` to select columns, all bets are off; you can't assume anything about the order in which they'll be returned. Either name the columns explicitly in the order you want, or retrieve them into a data structure that makes their order irrelevant.

Discussion

The examples in the previous section illustrate the differences between using `*` versus a list of names to specify output columns when issuing `SELECT` statements from within the *mysql* program. The difference between approaches also may be

significant when issuing queries through an API from within your own programs, depending on how you fetch result set rows. If you select output columns using `*`, the server returns them using the order in which they are listed in the table definition—an order that may change if the table structure is modified. If you fetch rows into an array, this non-determinacy of output column order makes it impossible to know which column each array element corresponds to. By naming output columns explicitly, you can fetch rows into an array with confidence that the columns will appear in the array in the same order that you named them in the query.

On the other hand, your API may allow you to fetch rows into a structure containing elements that are accessed by name. (For example, in Perl you can use a hash; in PHP you can use an associative array or an object.) If you do this, you can issue a `SELECT *` query and then access structure members by referring to the column names in any order you want. In this case, there is effectively no difference between selecting columns with `*` or by naming them explicitly: If you can access values by name within your program, their order within result set rows is irrelevant. This fact makes it tempting to take the easy way out by using `SELECT *` for all your queries, even if you're not actually going to use every column. Nevertheless, it's more efficient to name specifically only the columns you want so that the server doesn't send you information you're just going to ignore. (An example that explains in more detail why you may want to avoid retrieving certain columns is given in Recipe 9.8, in the section "Selecting All Except Certain Columns.")

3.3 Giving Names to Output Columns

Problem

You don't like the names of the columns in your query result.

Solution

Supply names of your own choosing using column aliases.

Discussion

Whenever you retrieve a result set, MySQL gives every output column a name. (That's how the `mysql` program gets the names that you see displayed as the initial row of column headers in result set output.) MySQL assigns default names to output columns, but if the defaults are not suitable, you can use column aliases to specify your own names.

This section explains aliases and shows how to use them to assign column names in queries. If you're writing a program that needs to retrieve information about column names, see Recipe 9.2.

If an output column in a result set comes directly from a table, MySQL uses the table column name for the result set column name. For example, the following statement selects three table columns, the names of which become the corresponding output column names:

```
mysql> SELECT t, srcuser, size FROM mail;
+-----+-----+-----+
| t           | srcuser | size  |
+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | 58274 |
| 2001-05-12 12:48:13 | tricia | 194925 |
| 2001-05-12 15:02:49 | phil   | 1048  |
| 2001-05-13 13:59:18 | barb   | 271   |
...

```

If you generate a column by evaluating an expression, the expression itself is the column name. This can produce rather long and unwieldy names in result sets, as illustrated by the following query that uses an expression to reformat the t column of the mail table:

```
mysql> SELECT
  -> CONCAT(MONTHNAME(t),' ',DAYOFMONTH(t),' ',YEAR(t)),
  -> srcuser, size FROM mail;
+-----+-----+-----+
| CONCAT(MONTHNAME(t),' ',DAYOFMONTH(t),' ',YEAR(t)) | srcuser | size  |
+-----+-----+-----+
| May 11, 2001                                     | barb   | 58274 |
| May 12, 2001                                     | tricia | 194925 |
| May 12, 2001                                     | phil   | 1048  |
| May 13, 2001                                     | barb   | 271   |
...

```

The preceding example uses a query that is specifically contrived to illustrate how awful-looking column names can be. The reason it's contrived is that you probably wouldn't really write the query that way—the same result can be produced more easily using MySQL's DATE_FORMAT() function. But even with DATE_FORMAT(), the column header is still ugly:

```
mysql> SELECT
  -> DATE_FORMAT(t,'%M %e, %Y'),
  -> srcuser, size FROM mail;
+-----+-----+-----+
| DATE_FORMAT(t,'%M %e, %Y') | srcuser | size  |
+-----+-----+-----+
| May 11, 2001               | barb   | 58274 |
| May 12, 2001               | tricia | 194925 |
| May 12, 2001               | phil   | 1048  |
| May 13, 2001               | barb   | 271   |
...

```

To give a result set column a name of your own choosing, use *AS name* to specify a column alias. The following query retrieves the same result as the previous one, but renames the first column to `date_sent`:

```
mysql> SELECT
  -> DATE_FORMAT(t,'%M %e, %Y') AS date_sent,
  -> srcuser, size FROM mail;
```

date_sent	srcuser	size
May 11, 2001	barb	58274
May 12, 2001	tricia	194925
May 12, 2001	phil	1048
May 13, 2001	barb	271

...

You can see that the alias makes the column name more concise, easier to read, and more meaningful. If you want to use a descriptive phrase, an alias can consist of several words. (Aliases can be fairly arbitrary, although they are subject to a few restrictions such as that they must be quoted if they are SQL keywords, contain spaces or other special characters, or are entirely numeric.) The following query retrieves the same data values as the preceding one but uses phrases to name the output columns:

```
mysql> SELECT
  -> DATE_FORMAT(t,'%M %e, %Y') AS 'Date of message',
  -> srcuser AS 'Message sender', size AS 'Number of bytes' FROM mail;
```

Date of message	Message sender	Number of bytes
May 11, 2001	barb	58274
May 12, 2001	tricia	194925
May 12, 2001	phil	1048
May 13, 2001	barb	271

...

Aliases can be applied to any result set column, not just those that come from tables:

```
mysql> SELECT '1+1+1' AS 'The expression', 1+1+1 AS 'The result';
```

The expression	The result
1+1+1	3

Here, the value of the first column is '1+1+1' (quoted so that it is treated as a string), and the value of the second column is 1+1+1 (without quotes so that MySQL treats it as an expression and evaluates it). The aliases are descriptive phrases that help to make clear the relationship between the two column values.

If you try using a single-word alias and MySQL complains about it, the alias probably is a reserved word. Quoting it should make it legal:

```
mysql> SELECT 1 AS INTEGER;
You have an error in your SQL syntax near 'INTEGER' at line 1
mysql> SELECT 1 AS 'INTEGER';
+-----+
| INTEGER |
+-----+
|        1 |
+-----+
```

3.4 Using Column Aliases to Make Programs Easier to Write

Problem

You're trying to refer to a column by name from within a program, but the column is calculated from an expression. Consequently, it's difficult to use.

Solution

Use an alias to give the column a simpler name.

Discussion

If you're writing a program that fetches rows into an array and accesses them by numeric column indexes, the presence or absence of column aliases makes no difference, because aliases don't change the positions of columns within the result set. However, aliases make a big difference if you're accessing output columns by name, because aliases change those names. You can exploit this fact to give your program easier names to work with. For example, if your query displays reformatted message time values from the `mail` table using the expression `DATE_FORMAT(t, '%M %e, %Y')`, that expression is also the name you'd have to use when referring to the output column. That's not very convenient. If you use `AS date_sent` to give the column an alias, you can refer to it a lot more easily using the name `date_sent`. Here's an example that shows how a Perl DBI script might process such values:

```
$sth = $dbh->prepare (
    "SELECT srcuser,
       DATE_FORMAT(t, '%M %e, %Y') AS date_sent
    FROM mail");
$sth->execute ();
while (my $ref = $sth->fetchrow_hashref ())
{
    printf "user: %s, date sent: %s\n", $ref->{srcuser}, $ref->{date_sent};
}
```

In Java, you'd do something like this:

```
Statement s = conn.createStatement ();
s.executeQuery ("SELECT srcuser,"
               + " DATE_FORMAT(t,'%M %e, %Y') AS date_sent"
               + " FROM mail");
ResultSet rs = s.getResultSet ();
while (rs.next ()) // loop through rows of result set
{
    String name = rs.getString ("srcuser");
    String dateSent = rs.getString ("date_sent");
    System.out.println ("user: " + name
                       + ", date sent: " + dateSent);
}
rs.close ();
s.close ();
```

In PHP, retrieve result set rows using `mysql_fetch_array()` or `mysql_fetch_object()` to fetch rows into a data structure that contains named elements. With Python, use a cursor class that causes rows to be returned as dictionaries containing key/value pairs where the keys are the column names. (See Recipe 2.4.)

3.5 Combining Columns to Construct Composite Values

Problem

You want to display values that are constructed from multiple table columns.

Solution

One way to do this is to use `CONCAT()`. You might also want to give the column a nicer name by using an alias.

Discussion

Column values may be combined to produce composite output values. For example, this expression concatenates `srcuser` and `srchost` values into email address format:

```
CONCAT(srcuser,'@',srchost)
```

Such expressions tend to produce ugly column names, which is yet another reason why column aliases are useful. The following query uses the aliases `sender` and `recipient` to name output columns that are constructed by combining usernames and hostnames into email addresses:

```
mysql> SELECT
-> DATE_FORMAT(t,'%M %e, %Y') AS date_sent,
-> CONCAT(srcuser,'@',srchost) AS sender,
-> CONCAT(dstuser,'@',dsthost) AS recipient,
```

```
-> size FROM mail;
```

date_sent	sender	recipient	size
May 11, 2001	barb@saturn	tricia@mars	58274
May 12, 2001	tricia@mars	gene@venus	194925
May 12, 2001	phil@mars	phil@saturn	1048
May 13, 2001	barb@saturn	tricia@venus	271

...

3.6 Specifying Which Rows to Select

Problem

You don't want to see all the rows from a table, just some of them.

Solution

Add a `WHERE` clause to the query that indicates to the server which rows to return.

Discussion

Unless you qualify or restrict a `SELECT` query in some way, it retrieves every row in your table, which may be a lot more information than you really want to see. To be more precise about the rows to select, provide a `WHERE` clause that specifies one or more conditions that rows must match.

Conditions can perform tests for equality, inequality, or relative ordering. For some column types such as strings, you can use pattern matches. The following queries select columns from rows containing `srchost` values that are exactly equal to the string `'venus'`, that are lexically less than the string `'pluto'`, or that begin with the letter `'s'`:

```
mysql> SELECT t, srcuser, srchost FROM mail WHERE srchost = 'venus';
```

t	srcuser	srchost
2001-05-14 09:31:37	gene	venus
2001-05-14 14:42:21	barb	venus
2001-05-15 08:50:57	phil	venus
2001-05-16 09:00:28	gene	venus
2001-05-16 23:04:19	phil	venus

```
mysql> SELECT t, srcuser, srchost FROM mail WHERE srchost < 'pluto';
```

t	srcuser	srchost
2001-05-12 12:48:13	tricia	mars
2001-05-12 15:02:49	phil	mars
2001-05-14 11:52:17	phil	mars

```

| 2001-05-15 07:17:48 | gene   | mars |
| 2001-05-15 10:25:52 | gene   | mars |
| 2001-05-17 12:49:23 | phil   | mars |
+-----+-----+-----+
mysql> SELECT t, srcuser, srchost FROM mail WHERE srchost LIKE 's%';
+-----+-----+-----+
| t           | srcuser | srchost |
+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | saturn |
| 2001-05-13 13:59:18 | barb   | saturn |
| 2001-05-14 17:03:01 | tricia | saturn |
| 2001-05-15 17:35:31 | gene   | saturn |
| 2001-05-19 22:21:51 | gene   | saturn |
+-----+-----+-----+

```

WHERE clauses can test multiple conditions. The following statement looks for rows where the `srcuser` column has any of three different values. (It asks the question, “When did gene, barb, or phil send mail?”):

```

mysql> SELECT t, srcuser, dstuser FROM mail
-> WHERE srcuser = 'gene' OR srcuser = 'barb' OR srcuser = 'phil';
+-----+-----+-----+
| t           | srcuser | dstuser |
+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | tricia |
| 2001-05-12 15:02:49 | phil   | phil   |
| 2001-05-13 13:59:18 | barb   | tricia |
| 2001-05-14 09:31:37 | gene   | barb   |
...

```

Queries such as the preceding one that test a given column to see if it has any of several different values often can be written more easily by using the `IN()` operator. `IN()` is true if the column is equal to any value in its argument list:

```

mysql> SELECT t, srcuser, dstuser FROM mail
-> WHERE srcuser IN ('gene','barb','phil');
+-----+-----+-----+
| t           | srcuser | dstuser |
+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | tricia |
| 2001-05-12 15:02:49 | phil   | phil   |
| 2001-05-13 13:59:18 | barb   | tricia |
| 2001-05-14 09:31:37 | gene   | barb   |
...

```

Different conditions can test different columns. This query finds messages sent by barb to tricia:

```

mysql> SELECT * FROM mail WHERE srcuser = 'barb' AND dstuser = 'tricia';
+-----+-----+-----+-----+-----+-----+
| t           | srcuser | srchost | dstuser | dsthost | size |
+-----+-----+-----+-----+-----+-----+
| 2001-05-11 10:15:08 | barb   | saturn | tricia | mars   | 58274 |
| 2001-05-13 13:59:18 | barb   | saturn | tricia | venus  | 271   |
+-----+-----+-----+-----+-----+-----+

```

Comparisons need only be legal syntactically; they need not make any sense semantically. The comparison in the following query doesn't have a particularly obvious meaning, but MySQL will happily execute it:*

```
SELECT * FROM mail WHERE srcuser + dsthost < size
```

Are Queries That Return No Rows Failed Queries?

If you issue a SELECT statement and get no rows back, has the query failed? It depends. If the lack of a result set is due to a problem such as that the statement is syntactically invalid or refers to nonexistent tables or columns, the query did indeed fail, because it could not even be executed. In this case, some sort of error condition should occur and you should investigate why your program is attempting to issue a malformed statement. If the query executes without error but returns nothing, it simply means that the query's WHERE clause matched no rows:

```
mysql> SELECT * FROM mail WHERE srcuser = 'no-such-user';  
Empty set (0.01 sec)
```

This is *not* a failed query. It ran successfully and produced a result; the result just happens to be empty because no rows have a srcuser value of no-such-user.

Columns need not be compared to literal values. You can test a column against other columns. Suppose you have a cd table lying around that contains year, artist, and title columns:†

```
mysql> SELECT year, artist, title FROM cd;  
+-----+-----+-----+  
| year | artist      | title      |  
+-----+-----+-----+  
| 1990 | Iona        | Iona       |  
| 1992 | Charlie Peacock | Lie Down in the Grass |  
| 1993 | Iona        | Beyond These Shores |  
| 1987 | The 77s     | The 77s    |  
| 1990 | Michael Gettel | Return     |  
| 1989 | Richard Souther | Cross Currents |  
| 1996 | Charlie Peacock | strangelanguage |  
| 1982 | Undercover   | Undercover  |  
...  
+-----+-----+-----+
```

If so, you can find all your eponymous CDs (those with artist and title the same) by performing a comparison of one column within the table to another:

* If you try issuing the query to see what it returns, how do you account for the result?

† It's not unlikely you'll have such a table if you've been reading other database books. Many of these have you go through the exercise of creating a database to keep track of your CD collection, a scenario that seems to rank second in popularity only to parts-and-suppliers examples.

```
mysql> SELECT year, artist, title FROM cd WHERE artist = title;
```

year	artist	title
1990	Iona	Iona
1987	The 77s	The 77s
1982	Undercover	Undercover

A special case of within-table column comparison occurs when you want to compare a column to itself rather than to a different column. Suppose you collect stamps and list your collection in a `stamp` table that contains columns for each stamp's ID number and the year it was issued. If you know that a particular stamp has an ID number 42 and want to use the value in its year column to find the other stamps in your collection that were issued in the same year, you'd do so by using year-to-year comparison—in effect, comparing the year column to itself:

```
mysql> SELECT stamp.* FROM stamp, stamp AS stamp2
-> WHERE stamp.year = stamp2.year AND stamp2.id = 42 AND stamp.id != 42;
```

id	year	description
97	1987	1-cent transition stamp
161	1987	aviation stamp

This kind of query involves a self-join, table aliases, and column references that are qualified using the table name. But that's more than I want to go into here. Those topics are covered in Chapter 12.

3.7 WHERE Clauses and Column Aliases

Problem

You want to refer to a column alias in a `WHERE` clause.

Solution

Sorry, you cannot.

Discussion

You cannot refer to column aliases in a `WHERE` clause. Thus, the following query is illegal:

```
mysql> SELECT t, srcuser, dstuser, size/1024 AS kilobytes
-> FROM mail WHERE kilobytes > 500;
ERROR 1054 at line 1: Unknown column 'kilobytes' in 'where clause'
```

The error occurs because aliases name *output* columns, whereas a WHERE clause operates on *input* columns to determine which rows to select for output. To make the query legal, replace the alias in the WHERE clause with the column or expression that the alias represents:

```
mysql> SELECT t, srcuser, dstuser, size/1024 AS kilobytes
-> FROM mail WHERE size/1024 > 500;
```

t	srcuser	dstuser	kilobytes
2001-05-14 17:03:01	tricia	phil	2338.36
2001-05-15 10:25:52	gene	tricia	975.13

3.8 Displaying Comparisons to Find Out How Something Works

Problem

You're curious about how a comparison in a WHERE clause works. Or perhaps about why it doesn't seem to be working.

Solution

Display the result of the comparison to get more information about it. This is a useful diagnostic or debugging technique.

Discussion

Normally you put comparison operations in the WHERE clause of a query and use them to determine which records to display:

```
mysql> SELECT * FROM mail WHERE srcuser < 'c' AND size > 5000;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-14 14:42:21	barb	venus	barb	venus	98151

But sometimes it's desirable to see the result of the comparison itself (for example, if you're not sure that the comparison is working the way you expect it to). To do this, just put the comparison expression in the output column list, perhaps including the values that you're comparing as well:

```
mysql> SELECT srcuser, srcuser < 'c', size, size > 5000 FROM mail;
+-----+-----+-----+-----+
| srcuser | srcuser < 'c' | size | size > 5000 |
+-----+-----+-----+-----+
| barb   | 1             | 58274 | 1           |
| tricia | 0             | 194925 | 1           |
| phil   | 0             | 1048 | 0           |
| barb   | 1             | 271 | 0           |
...

```

This technique of displaying comparison results is particularly useful for writing queries that check how a test works without using a table:

```
mysql> SELECT 'a' = 'A';
+-----+
| 'a' = 'A' |
+-----+
| 1         |
+-----+

```

This query result tells you that string comparisons are not by default case sensitive, which is a useful thing to know.

3.9 Reversing or Negating Query Conditions

Problem

You know how to write a query to answer a given question; now you want to ask the opposite question.

Solution

Reverse the conditions in the WHERE clause by using negation operators.

Discussion

The WHERE conditions in a query can be negated to ask the opposite questions. The following query determines when users sent mail to themselves:

```
mysql> SELECT * FROM mail WHERE srcuser = dstuser;
+-----+-----+-----+-----+-----+-----+
| t          | srcuser | srchost | dstuser | dsthost | size |
+-----+-----+-----+-----+-----+-----+
| 2001-05-12 15:02:49 | phil   | mars   | phil   | saturn  | 1048 |
| 2001-05-14 14:42:21 | barb   | venus  | barb   | venus   | 98151 |
| 2001-05-15 07:17:48 | gene   | mars   | gene   | saturn  | 3824 |
| 2001-05-15 08:50:57 | phil   | venus  | phil   | venus   | 978 |
| 2001-05-15 17:35:31 | gene   | saturn | gene   | mars    | 3856 |
| 2001-05-19 22:21:51 | gene   | saturn | gene   | venus   | 23992 |
+-----+-----+-----+-----+-----+-----+

```

To reverse this query, to find records where users sent mail to someone *other* than themselves, change the comparison operator from = (equal to) to != (not equal to):

```
mysql> SELECT * FROM mail WHERE srcuser != dstuser;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-13 13:59:18	barb	saturn	tricia	venus	271
2001-05-14 09:31:37	gene	venus	barb	mars	2291

...

A more complex query using two conditions might ask when people sent mail to themselves on the same machine:

```
mysql> SELECT * FROM mail WHERE srcuser = dstuser AND srchost = dsthost;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-14 14:42:21	barb	venus	barb	venus	98151
2001-05-15 08:50:57	phil	venus	phil	venus	978

Reversing the conditions for this query involves not only changing the = operators to !=, but changing the AND to OR:

```
mysql> SELECT * FROM mail WHERE srcuser != dstuser OR srchost != dsthost;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-12 15:02:49	phil	mars	phil	saturn	1048
2001-05-13 13:59:18	barb	saturn	tricia	venus	271

...

You may find it easier just to put the entire original expression in parentheses and negate the whole thing with NOT:

```
mysql> SELECT * FROM mail WHERE NOT (srcuser = dstuser AND srchost = dsthost);
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-12 15:02:49	phil	mars	phil	saturn	1048
2001-05-13 13:59:18	barb	saturn	tricia	venus	271

...

See Also

If a column involved in a condition may contain NULL values, reversing the condition is a little trickier. See Recipe 3.12 for details.

3.10 Removing Duplicate Rows

Problem

Output from a query contains duplicate records. You want to eliminate them.

Solution

Use `DISTINCT`.

Discussion

Some queries produce results containing duplicate records. For example, to see who sent mail, you could query the `mail` table like this:

```
mysql> SELECT srcuser FROM mail;
+-----+
| srcuser |
+-----+
| barb   |
| tricia |
| phil   |
| barb   |
| gene   |
| phil   |
| barb   |
| tricia |
| gene   |
| phil   |
| gene   |
| gene   |
| phil   |
| phil   |
| gene   |
+-----+
```

But that result is heavily redundant. Adding `DISTINCT` to the query removes the duplicate records, producing a set of unique values:

```
mysql> SELECT DISTINCT srcuser FROM mail;
+-----+
| srcuser |
+-----+
| barb   |
| tricia |
| phil   |
| gene   |
+-----+
```

DISTINCT works with multiple-column output, too. The following query shows which dates are represented in the mail table:

```
mysql> SELECT DISTINCT YEAR(t), MONTH(t), DAYOFMONTH(t) FROM mail;
+-----+-----+-----+
| YEAR(t) | MONTH(t) | DAYOFMONTH(t) |
+-----+-----+-----+
| 2001 | 5 | 11 |
| 2001 | 5 | 12 |
| 2001 | 5 | 13 |
| 2001 | 5 | 14 |
| 2001 | 5 | 15 |
| 2001 | 5 | 16 |
| 2001 | 5 | 17 |
| 2001 | 5 | 19 |
+-----+-----+-----+
```

To count the number of unique values, do this:

```
mysql> SELECT COUNT(DISTINCT srcuser) FROM mail;
+-----+
| COUNT(DISTINCT srcuser) |
+-----+
| 4 |
+-----+
```

COUNT(DISTINCT) requires MySQL 3.23.2 or higher.

See Also

DISTINCT is revisited in Chapter 7. Duplicate removal is discussed in more detail in Chapter 14.

3.11 Working with NULL Values

Problem

You're trying to compare column values to NULL, but it isn't working.

Solution

You have to use the proper comparison operators: IS NULL, IS NOT NULL, or <=>.

Discussion

Conditions involving NULL are special. You cannot use = NULL or != NULL to look for NULL values in columns. Such comparisons always fail because it's impossible to tell whether or not they are true. Even NULL = NULL fails. (Why? Because you can't determine whether one unknown value is the same as another unknown value.)

To look for columns that are or are not NULL, use IS NULL or IS NOT NULL. Suppose a table taxpayer contains taxpayer names and ID numbers, where a NULL ID indicates that the value is unknown:

```
mysql> SELECT * FROM taxpayer;
+-----+-----+
| name  | id    |
+-----+-----+
| bernina | 198-48 |
| bertha  | NULL  |
| ben    | NULL  |
| bill   | 475-83 |
+-----+-----+
```

You can see that = and != do not work with NULL values as follows:

```
mysql> SELECT * FROM taxpayer WHERE id = NULL;
Empty set (0.00 sec)
mysql> SELECT * FROM taxpayer WHERE id != NULL;
Empty set (0.01 sec)
```

To find records where the id column is or is not NULL, the queries should be written like this:

```
mysql> SELECT * FROM taxpayer WHERE id IS NULL;
+-----+-----+
| name  | id    |
+-----+-----+
| bertha | NULL  |
| ben   | NULL  |
+-----+-----+
mysql> SELECT * FROM taxpayer WHERE id IS NOT NULL;
+-----+-----+
| name  | id    |
+-----+-----+
| bernina | 198-48 |
| bill   | 475-83 |
+-----+-----+
```

As of MySQL 3.23, you can also use <=> to compare values, which (unlike the = operator) is true even for two NULL values:

```
mysql> SELECT NULL = NULL, NULL <=> NULL;
+-----+-----+
| NULL = NULL | NULL <=> NULL |
+-----+-----+
|          NULL |                1 |
+-----+-----+
```

See Also

NULL values also behave specially with respect to sorting and summary operations. See Recipe 6.5 and Recipe 7.8.

3.12 Negating a Condition on a Column That Contains NULL Values

Problem

You're trying to negate a condition that involves NULL, but it's not working.

Solution

NULL is special in negations, just like it is otherwise. Perhaps even more so.

Discussion

Recipe 3.9 pointed out that you can reverse query conditions, either by changing comparison operators and Boolean operators, or by using NOT. These techniques may not work if a column can contain NULL. Recall that the taxpayer table from Recipe 3.11 looks like this:

```
+-----+-----+
| name  | id    |
+-----+-----+
| bernina | 198-48 |
| bertha  | NULL  |
| ben     | NULL  |
| bill    | 475-83 |
+-----+-----+
```

Now suppose you have a query that finds records with taxpayer ID values that are lexically less than 200-00:

```
mysql> SELECT * FROM taxpayer WHERE id < '200-00';
+-----+-----+
| name  | id    |
+-----+-----+
| bernina | 198-48 |
+-----+-----+
```

Reversing this condition by using `>=` rather than `<` may not give you the results you want. It depends on what information you want to obtain. If you want to select only records with non-NULL ID values, `>=` is indeed the proper test:

```
mysql> SELECT * FROM taxpayer WHERE id >= '200-00';
+-----+-----+
| name | id    |
+-----+-----+
| bill | 475-83 |
+-----+-----+
```

But if you want all the records not selected by the original query, simply reversing the operator will not work. NULL values fail comparisons both with `<` and with `>=`, so you must add an additional clause specifically for them:

```
mysql> SELECT * FROM taxpayer WHERE id >= '200-00' OR id IS NULL;
+-----+-----+
| name  | id    |
+-----+-----+
| berth | NULL  |
| ben   | NULL  |
| bill  | 475-83 |
+-----+-----+
```

3.13 Writing Comparisons Involving NULL in Programs

Problem

You're writing a program that issues a query, but it fails for NULL values.

Solution

Try writing the comparison selectively for NULL and non-NULL values.

Discussion

The need to use different comparison operators for NULL values than for non-NULL values leads to a subtle danger when constructing query strings within programs. If you have a value stored in a variable that might represent a NULL value, you must account for that if you use the value in comparisons. For example, in Perl, `undef` represents a NULL value, so to construct a statement that finds records in the `taxpayer` table matching some arbitrary value in an `$id` variable, you cannot do this:

```
$sth = $dbh->prepare ("SELECT * FROM taxpayer WHERE id = ?");
$sth->execute ($id);
```

The statement fails when `$id` is `undef`, because the resulting query becomes:

```
SELECT * FROM taxpayer WHERE id = NULL
```

That statement returns no records—a comparison of `= NULL` always fails. To take into account the possibility that `$id` may be `undef`, construct the query using the appropriate comparison operator like this:

```
$operator = (defined ($id) ? "=" : "IS");
$sth = $dbh->prepare ("SELECT * FROM taxpayer WHERE id $operator ?");
$sth->execute ($id);
```

This results in queries as follows for `$id` values of `undef` (NULL) or `43` (not NULL):

```
SELECT * FROM taxpayer WHERE id IS NULL
SELECT * FROM taxpayer WHERE id = 43
```

For inequality tests, set `$operator` like this instead:

```
$operator = (defined ($id) ? "!=" : "IS NOT");
```

3.14 Mapping NULL Values to Other Values for Display

Problem

A query's output includes NULL values, but you'd rather see something more meaningful, like "Unknown."

Solution

Convert NULL values selectively to another value when displaying them. You can also use this technique to catch divide-by-zero errors.

Discussion

Sometimes it's useful to display NULL values using some other distinctive value that has more meaning in the context of your application. If NULL id values in the taxpayer table mean "unknown," you can display that label by using IF() to map them onto the string Unknown:

```
mysql> SELECT name, IF(id IS NULL, 'Unknown', id) AS 'id' FROM taxpayer;
+-----+-----+
| name  | id      |
+-----+-----+
| bernina | 198-48 |
| bertha  | Unknown |
| ben     | Unknown |
| bill    | 475-83 |
+-----+-----+
```

Actually, this technique works for any kind of value, but it's especially useful with NULL values because they tend to be given a variety of meanings: unknown, missing, not yet determined, out of range, and so forth.

The query can be written more concisely using IFNULL(), which tests its first argument and returns it if it's not NULL, or returns its second argument otherwise:

```
mysql> SELECT name, IFNULL(id, 'Unknown') AS 'id' FROM taxpayer;
+-----+-----+
| name  | id      |
+-----+-----+
| bernina | 198-48 |
| bertha  | Unknown |
| ben     | Unknown |
| bill    | 475-83 |
+-----+-----+
```

In other words, these two tests are equivalent:

```
IF(expr1 IS NOT NULL, expr1, expr2)
IFNULL(expr1, expr2)
```

From a readability standpoint, IF() often is easier to understand than IFNULL(). From a computational perspective, IFNULL() is more efficient because *expr1* never need be evaluated twice, as sometimes happens with IF().

IF() and IFNULL() are especially useful for catching divide-by-zero operations and mapping them onto something else. For example, batting averages for baseball players are calculated as the ratio of hits to at-bats. But if a player has no at-bats, the ratio is undefined:

```
mysql> SET @hits = 0, @atbats = 0;
mysql> SELECT @hits, @atbats, @hits/@atbats AS 'batting average';
+-----+-----+-----+
| @hits | @atbats | batting average |
+-----+-----+-----+
| 0     | 0       | NULL           |
+-----+-----+-----+
```

To handle that case by displaying zero, do this:

```
mysql> SET @hits = 0, @atbats = 0;
mysql> SELECT @hits, @atbats, IFNULL(@hits/@atbats,0) AS 'batting average';
+-----+-----+-----+
| @hits | @atbats | batting average |
+-----+-----+-----+
| 0     | 0       | 0               |
+-----+-----+-----+
```

Earned run average calculations for a pitcher with no innings pitched can be treated the same way. Other common uses for this idiom are as follows:

```
IFNULL(expr, 'Missing')
IFNULL(expr, 'N/A')
IFNULL(expr, 'Unknown')
```

3.15 Sorting a Result Set

Problem

Your query results aren't sorted the way you want.

Solution

MySQL can't read your mind. Add an ORDER BY clause to tell it exactly how you want things sorted.

Discussion

When you select rows, the MySQL server is free to return them in any order, unless you instruct it otherwise by saying how to sort the result. There are lots of ways to use

sorting techniques. Chapter 6 explores this topic further. Briefly, you sort a result set by adding an ORDER BY clause that names the column or columns you want to sort by:

```
mysql> SELECT * FROM mail WHERE size > 100000 ORDER BY size;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-15 10:25:52	gene	mars	tricia	saturn	998532
2001-05-14 17:03:01	tricia	saturn	phil	venus	2394482

```
mysql> SELECT * FROM mail WHERE dstuser = 'tricia'
-> ORDER BY srchost, srcuser;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-15 10:25:52	gene	mars	tricia	saturn	998532
2001-05-14 11:52:17	phil	mars	tricia	saturn	5781
2001-05-17 12:49:23	phil	mars	tricia	saturn	873
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274
2001-05-13 13:59:18	barb	saturn	tricia	venus	271

To sort a column in reverse (descending) order, add the keyword DESC after its name in the ORDER BY clause:

```
mysql> SELECT * FROM mail WHERE size > 50000 ORDER BY size DESC;
```

t	srcuser	srchost	dstuser	dsthost	size
2001-05-14 17:03:01	tricia	saturn	phil	venus	2394482
2001-05-15 10:25:52	gene	mars	tricia	saturn	998532
2001-05-12 12:48:13	tricia	mars	gene	venus	194925
2001-05-14 14:42:21	barb	venus	barb	venus	98151
2001-05-11 10:15:08	barb	saturn	tricia	mars	58274

3.16 Selecting Records from the Beginning or End of a Result Set

Problem

You want to see only certain rows from a result set, like the first one or the last five.

Solution

Use a LIMIT clause, perhaps in conjunction with an ORDER BY clause.

Discussion

MySQL supports a LIMIT clause that tells the server to return only part of a result set. LIMIT is a MySQL-specific extension to SQL that is extremely valuable when your result set contains more rows than you want to see at a time. It allows you to retrieve just the first part of a result set or an arbitrary section of the set. Typically, LIMIT is used for the following kinds of problems:

- Answering questions about first or last, largest or smallest, newest or oldest, least or more expensive, and so forth.
- Splitting a result set into sections so that you can process it one piece at a time. This technique is common in web applications for displaying a large search result across several pages. Showing the result in sections allows display of smaller pages that are easier to understand.

The following examples use the profile table that was introduced in Chapter 2. Its contents look like this:

```
mysql> SELECT * FROM profile;
+----+-----+-----+-----+-----+-----+
| id | name  | birth   | color | foods                | cats |
+----+-----+-----+-----+-----+-----+
| 1  | Fred  | 1970-04-13 | black | lutefisk,fadge,pizza | 0    |
| 2  | Mort  | 1969-09-30 | white | burrito,curry,eggroll | 3    |
| 3  | Brit  | 1957-12-01 | red   | burrito,curry,pizza  | 1    |
| 4  | Carl  | 1973-11-02 | red   | eggroll,pizza        | 4    |
| 5  | Sean  | 1963-07-04 | blue  | burrito,curry        | 5    |
| 6  | Alan  | 1965-02-14 | red   | curry,fadge          | 1    |
| 7  | Mara  | 1968-09-17 | green | lutefisk,fadge       | 1    |
| 8  | Shepard | 1975-09-02 | black | curry,pizza          | 2    |
| 9  | Dick  | 1952-08-20 | green | lutefisk,fadge       | 0    |
| 10 | Tony  | 1960-05-01 | white | burrito,pizza        | 0    |
+----+-----+-----+-----+-----+-----+
```

To select the first n records of a query result, add LIMIT n to the end of your SELECT statement:

```
mysql> SELECT * FROM profile LIMIT 1;
+----+-----+-----+-----+-----+-----+
| id | name  | birth   | color | foods                | cats |
+----+-----+-----+-----+-----+-----+
| 1  | Fred  | 1970-04-13 | black | lutefisk,fadge,pizza | 0    |
+----+-----+-----+-----+-----+-----+
mysql> SELECT * FROM profile LIMIT 5;
+----+-----+-----+-----+-----+-----+
| id | name  | birth   | color | foods                | cats |
+----+-----+-----+-----+-----+-----+
| 1  | Fred  | 1970-04-13 | black | lutefisk,fadge,pizza | 0    |
| 2  | Mort  | 1969-09-30 | white | burrito,curry,eggroll | 3    |
| 3  | Brit  | 1957-12-01 | red   | burrito,curry,pizza  | 1    |
| 4  | Carl  | 1973-11-02 | red   | eggroll,pizza        | 4    |
| 5  | Sean  | 1963-07-04 | blue  | burrito,curry        | 5    |
+----+-----+-----+-----+-----+-----+
```

However, because the rows in these query results aren't sorted into any particular order, they may not be very meaningful. A more common technique is to use `ORDER BY` to sort the result set. Then you can use `LIMIT` to find smallest and largest values. For example, to find the row with the minimum (earliest) birth date, sort by the birth column, then add `LIMIT 1` to retrieve the first row:

```
mysql> SELECT * FROM profile ORDER BY birth LIMIT 1;
+-----+-----+-----+-----+-----+-----+
| id | name | birth      | color | foods          | cats |
+-----+-----+-----+-----+-----+-----+
| 9  | Dick | 1952-08-20 | green | lutefisk,fadge | 0    |
+-----+-----+-----+-----+-----+-----+
```

This works because MySQL processes the `ORDER BY` clause to sort the rows first, then applies `LIMIT`. To find the row with the most recent birth date, the query is similar, except that you sort in descending order:

```
mysql> SELECT * FROM profile ORDER BY birth DESC LIMIT 1;
+-----+-----+-----+-----+-----+-----+
| id | name  | birth      | color | foods          | cats |
+-----+-----+-----+-----+-----+-----+
| 8  | Shepard | 1975-09-02 | black | curry,pizza   | 2    |
+-----+-----+-----+-----+-----+-----+
```

You can obtain the same information by running these queries without `LIMIT` and ignoring everything but the first row. The advantage of using `LIMIT` is that the server returns just the first record and the extra rows don't travel over the network at all. This is much more efficient than retrieving an entire result set, only to discard all but one row.

The sort column or columns can be whatever you like. To find the row for the person with the most cats, sort by the cats column:

```
mysql> SELECT * FROM profile ORDER BY cats DESC LIMIT 1;
+-----+-----+-----+-----+-----+-----+
| id | name | birth      | color | foods          | cats |
+-----+-----+-----+-----+-----+-----+
| 5  | Sean | 1963-07-04 | blue  | burrito,curry | 5    |
+-----+-----+-----+-----+-----+-----+
```

However, be aware that using `LIMIT n` to select the “*n* smallest” or “*n* largest” values may not yield quite the results you expect. See Recipe 3.18 for some discussion on framing `LIMIT` queries appropriately.

To find the earliest birthday within the calendar year, sort by the month and day of the birth values:

```
mysql> SELECT name, DATE_FORMAT(birth, '%m-%e') AS birthday
-> FROM profile ORDER BY birthday LIMIT 1;
+-----+-----+
| name | birthday |
+-----+-----+
| Alan | 02-14    |
+-----+-----+
```

Note that `LIMIT n` really means “return at most n rows.” If you specify `LIMIT 10` and the result set has only 3 rows, the server returns 3 rows.

See Also

You can use `LIMIT` in combination with `RAND()` to make random selections from a set of items. See Chapter 13.

As of MySQL 3.22.7, you can use `LIMIT` to restrict the effect of a `DELETE` statement to a subset of the rows that would otherwise be deleted. As of MySQL 3.23.3, the same is true for `UPDATE`. This can be useful in conjunction with a `WHERE` clause. For example, if a table contains five instances of a record, you can select them in a `DELETE` statement with an appropriate `WHERE` clause, then remove the duplicates by adding `LIMIT 4` to the end of the statement. This leaves only one copy of the record. For more information about uses of `LIMIT` in duplicate record removal, see Chapter 14.

3.17 Pulling a Section from the Middle of a Result Set

Problem

You don’t want the first or last rows of a result set. Instead, you want to pull a section of rows out of the middle of the set, such as rows 21 through 40.

Solution

That’s still a job for `LIMIT`. But you need to tell it the starting position within the result set in addition to the number of rows you want.

Discussion

`LIMIT n` tells the server to return the first n rows of a result set. `LIMIT` also has a two-argument form that allows you to pick out any arbitrary section of rows from a result. The arguments indicate how many rows to skip and how many to return. This means that you can use `LIMIT` to do such things as skip two rows and return the next, thus answering questions such as “what is the *third*-smallest or *third*-largest value?,” something that’s more difficult with `MIN()` or `MAX()`:

```
mysql> SELECT * FROM profile ORDER BY birth LIMIT 2,1;
+-----+-----+-----+-----+-----+
| id | name | birth      | color | foods          | cats |
+-----+-----+-----+-----+-----+
| 10 | Tony | 1960-05-01 | white | burrito,pizza | 0 |
+-----+-----+-----+-----+-----+
```

```
mysql> SELECT * FROM profile ORDER BY birth DESC LIMIT 2,1;
+-----+-----+-----+-----+-----+
| id | name | birth      | color | foods                | cats |
+-----+-----+-----+-----+-----+
| 1  | Fred | 1970-04-13 | black | lutefisk,fadge,pizza | 0    |
+-----+-----+-----+-----+-----+
```

The two-argument form of LIMIT also makes it possible to partition a result set into smaller sections. For example, to retrieve 20 rows at a time from a result, issue the same SELECT statement repeatedly, but vary the LIMIT clauses like so:

```
SELECT ... FROM ... ORDER BY ... LIMIT 0, 20;  retrieve first 20 rows
SELECT ... FROM ... ORDER BY ... LIMIT 20, 20; skip 20 rows, retrieve next 20
SELECT ... FROM ... ORDER BY ... LIMIT 40, 20; skip 40 rows, retrieve next 20
etc.
```

Web developers often use LIMIT this way to split a large search result into smaller, more manageable pieces so that it can be presented over several pages. We'll discuss this technique further in Recipe 18.10.

If you want to know how large a result set is so that you can determine how many sections there are, you can issue a COUNT() query first. Use a WHERE clause that is the same as for the queries you'll use to retrieve the rows. For example, if you want to display profile table records in name order four at a time, you can find out how many there are with the following query:

```
mysql> SELECT COUNT(*) FROM profile;
+-----+
| COUNT(*) |
+-----+
|          10 |
+-----+
```

That tells you you'll have three sets of rows (although the last one will have fewer than four records), which you can retrieve as follows:

```
SELECT * FROM profile ORDER BY name LIMIT 0, 4;
SELECT * FROM profile ORDER BY name LIMIT 4, 4;
SELECT * FROM profile ORDER BY name LIMIT 8, 4;
```

Beginning with MySQL 4.0, you can fetch a part of a result set, but also find out how big the result would have been without the LIMIT clause. For example, to fetch the first four records from the profile table and then obtain the size of the full result, run these queries:

```
SELECT SQL_CALC_FOUND_ROWS * FROM profile ORDER BY name LIMIT 4;
SELECT FOUND_ROWS();
```

The keyword SQL_CALC_FOUND_ROWS in the first query tells MySQL to calculate the size of the entire result set even though the query requests that only part of it be returned. The row count is available by calling FOUND_ROWS(). If that function returns a value greater than four, there are other records yet to be retrieved.

3.18 Choosing Appropriate LIMIT Values

Problem

LIMIT doesn't seem to do what you want it to.

Solution

Be sure you understand what question you're asking. It may be that LIMIT is exposing some interesting subtleties in your data that you have not considered or are not aware of.

Discussion

LIMIT *n* is useful in conjunction with ORDER BY for selecting smallest or largest values from a result set. But does that actually give you the rows with the *n* smallest or largest values? Not necessarily! It does if your rows contain unique values, but not if there are duplicates. You may find it necessary to run a preliminary query first to help you choose the proper LIMIT value.

To see why this is, consider the following dataset, which shows the American League pitchers who won 15 or more games during the 2001 baseball season:

```
mysql> SELECT name, wins FROM al_winner
      -> ORDER BY wins DESC, name;
```

name	wins
Mulder, Mark	21
Clemens, Roger	20
Moyer, Jamie	20
Garcia, Freddy	18
Hudson, Tim	18
Abbott, Paul	17
Mays, Joe	17
Mussina, Mike	17
Sabathia, C.C.	17
Zito, Barry	17
Buehrle, Mark	16
Milton, Eric	15
Pettitte, Andy	15
Radke, Brad	15
Sele, Aaron	15

If you want to know who won the most games, adding LIMIT 1 to the preceding query will give you the correct answer, because the maximum value is 21 and there is

only one pitcher with that value (Mark Mulder). But what if you want the four highest game winners? The proper queries depend on what you mean by that, which can have various interpretations:

- If you just want the first four rows, sort the records and add LIMIT 4:

```
mysql> SELECT name, wins FROM al_winner
-> ORDER BY wins DESC, name
-> LIMIT 4;
```

```
+-----+-----+
| name          | wins |
+-----+-----+
| Mulder, Mark  |    21 |
| Clemens, Roger |    20 |
| Moyer, Jamie  |    20 |
| Garcia, Freddy |    18 |
+-----+-----+
```

That may not suit your purposes because LIMIT imposes a cutoff that occurs in the middle of a set of pitchers with the same number of wins (Tim Hudson also won 18 games).

- To avoid making a cutoff in the middle of a set of rows with the same value, select rows with values greater than or equal to the value in the fourth row. Find out what that value is with LIMIT, then use it in the WHERE clause of a second query to select rows:

```
mysql> SELECT wins FROM al_winner
-> ORDER BY wins DESC, name
-> LIMIT 3, 1;
```

```
+-----+
| wins |
+-----+
|    18 |
+-----+
```

```
mysql> SELECT name, wins FROM al_winner
-> WHERE wins >= 18
-> ORDER BY wins DESC, name;
```

```
+-----+-----+
| name          | wins |
+-----+-----+
| Mulder, Mark  |    21 |
| Clemens, Roger |    20 |
| Moyer, Jamie  |    20 |
| Garcia, Freddy |    18 |
| Hudson, Tim   |    18 |
+-----+-----+
```

- If you want to know all the pitchers with the four largest wins values, another approach is needed. Determine the fourth-largest value with DISTINCT and LIMIT, then use it to select rows:

```
mysql> SELECT DISTINCT wins FROM al_winner
-> ORDER BY wins DESC, name
-> LIMIT 3, 1;
```

```

+-----+
| wins |
+-----+
|  17  |
+-----+
mysql> SELECT name, wins FROM a1_winner
      -> WHERE wins >= 17
      -> ORDER BY wins DESC, name;
+-----+-----+
| name          | wins |
+-----+-----+
| Mulder, Mark  |  21  |
| Clemens, Roger |  20  |
| Moyer, Jamie  |  20  |
| Garcia, Freddy |  18  |
| Hudson, Tim   |  18  |
| Abbott, Paul  |  17  |
| Mays, Joe     |  17  |
| Mussina, Mike |  17  |
| Sabathia, C.C. |  17  |
| Zito, Barry   |  17  |
+-----+-----+

```

For this dataset, each method yields a different result. The moral is that the way you use LIMIT may require some thought about what you really want to know.

3.19 Calculating LIMIT Values from Expressions

Problem

You want to use expressions to specify the arguments for LIMIT.

Solution

Sadly, you cannot. You can use only literal integers—unless you issue the query from within a program, in which case you can evaluate the expressions yourself and stick the resulting values into the query string.

Discussion

Arguments to LIMIT must be literal integers, not expressions. Statements such as the following are illegal:

```

SELECT * FROM profile LIMIT 5+5;
SELECT * FROM profile LIMIT @skip_count, @show_count;

```

The same “no expressions allowed” principle applies if you’re using an expression to calculate a LIMIT value in a program that constructs a query string. You must

evaluate the expression first, then place the resulting value in the query. For example, if you produce a query string in Perl (or PHP) as follows, an error will result when you attempt to execute the query:

```
$str = "SELECT * FROM profile LIMIT $x + $y";
```

To avoid the problem, evaluate the expression first:

```
$z = $x + $y;  
$str = "SELECT * FROM profile LIMIT $z";
```

Or do this (but don't omit the parentheses or the expression won't evaluate properly):

```
$str = "SELECT * FROM profile LIMIT " . ($x + $y);
```

If you're constructing a two-argument LIMIT clause, evaluate both expressions before placing them into the query string.

3.20 What to Do When LIMIT Requires the “Wrong” Sort Order

Problem

LIMIT usually works best in conjunction with an ORDER BY clause that sorts rows. But sometimes the sort order is the opposite of what you want for the final result.

Solution

Rewrite the query, or write a program that retrieves the rows and sorts them into the order you want.

Discussion

If you want the last four records of a result set, you can obtain them easily by sorting the set in reverse order and using LIMIT 4. For example, the following query returns the names and birth dates for the four people in the profile table who were born most recently:

```
mysql> SELECT name, birth FROM profile ORDER BY birth DESC LIMIT 4;  
+-----+-----+  
| name   | birth   |  
+-----+-----+  
| Shepard | 1975-09-02 |  
| Carl   | 1973-11-02 |  
| Fred   | 1970-04-13 |  
| Mort   | 1969-09-30 |  
+-----+-----+
```

But that requires sorting the birth values in descending order to place them at the head of the result set. What if you want them in ascending order instead? One way to solve this problem is to use two queries. First, use `COUNT()` to find out how many rows are in the table:

```
mysql> SELECT COUNT(*) FROM profile;
+-----+
| COUNT(*) |
+-----+
|        10 |
+-----+
```

Then, sort the values in ascending order and use the two-argument form of `LIMIT` to skip all but the last four records:

```
mysql> SELECT name, birth FROM profile ORDER BY birth LIMIT 6, 4;
+-----+-----+
| name  | birth      |
+-----+-----+
| Mort  | 1969-09-30 |
| Fred  | 1970-04-13 |
| Carl  | 1973-11-02 |
| Shepard | 1975-09-02 |
+-----+-----+
```

Single-query solutions to this problem may be available if you're issuing queries from within a program and can manipulate the query result. For example, if you fetch the values into a data structure, you can reverse the order of the values in the structure. Here is some Perl code that demonstrates this approach:

```
my $stmt = "SELECT name, birth FROM profile ORDER BY birth DESC LIMIT 4";
# fetch values into a data structure
my $ref = $dbh->selectall_arrayref($stmt);
# reverse the order of the items in the structure
my @val = reverse(@{$ref});
# use $val[$i] to get a reference to row $i, then use
# $val[$i]->[0] and $val[$i]->[1] to access column values
```

Alternatively, you can simply iterate through the structure in reverse order:

```
my $stmt = "SELECT name, birth FROM profile ORDER BY birth DESC LIMIT 4";
# fetch values into a data structure
my $ref = $dbh->selectall_arrayref($stmt);
# iterate through the structure in reverse order
my $row_count = @{$ref};
for (my $i = $row_count - 1; $i >= 0; $i--)
{
    # use $ref->[$i]->[0] and $ref->[$i]->[1] here...
}
```

3.21 Selecting a Result Set into an Existing Table

Problem

You want to run a `SELECT` query but save the results into another table rather than displaying them.

Solution

If the other table exists, use `INSERT INTO ... SELECT`, described here. If the table doesn't exist, skip ahead to Recipe 3.22.

Discussion

The MySQL server normally returns the result of a `SELECT` statement to the client that issued the statement. For example, when you run a query from within *mysql*, the server returns the result to *mysql*, which in turn displays it to you on the screen. It's also possible to send the results of a `SELECT` statement directly into another table. Copying records from one table to another is useful in a number of ways:

- If you're developing an algorithm that modifies a table, it's safer to work with a copy of a table so that you need not worry about the consequences of mistakes. Also, if the original table is large, creating a partial copy can speed the development process because queries run against it will take less time.
- For data-loading operations that work with information that might be malformed, you can load new records into a temporary table, perform some preliminary checks, and correct the records as necessary. When you're satisfied the new records are okay, copy them from the temporary table into your main table.
- Some applications maintain a large repository table and a smaller working table into which records are inserted on a regular basis, copying the working table records to the repository periodically and clearing the working table.
- If you're performing a number of similar summary operations on a large table, it may be more efficient to select summary information once into a second table and use that for further analysis, rather than running expensive summary operations repeatedly on the original table.

This section shows how to use `INSERT ... SELECT` to retrieve a result set for insertion into an existing table. The next section discusses `CREATE TABLE ... SELECT`, a statement available as of MySQL 3.23 that allows you to create a table on the fly directly from a query result. The table names `src_tbl` and `dst_tbl` in the examples refer to the source table from which rows are selected and the destination table into which they are stored.

If the destination table already exists, use `INSERT ... SELECT` to copy the result set into it. For example, if `dst_tbl` contains an integer column `i` and a string column `s`, the following statement copies rows from `src_tbl` into `dst_tbl`, assigning column `val` to `i` and column `name` to `s`:

```
INSERT INTO dst_tbl (i, s) SELECT val, name FROM src_tbl;
```

The number of columns to be inserted must match the number of selected columns, and the correspondence between sets of columns is established by position rather than name. In the special case that you want to copy all columns from one table to another, you can shorten the statement to this form:

```
INSERT INTO dst_tbl SELECT * FROM src_tbl;
```

To copy only certain rows, add a `WHERE` clause that selects the rows you want:

```
INSERT INTO dst_tbl SELECT * FROM src_tbl WHERE val > 100 AND name LIKE 'A%';
```

It's not necessary to copy column values without modification from the source table into the destination table. The `SELECT` statement can produce values from expressions, too. For example, the following query counts the number of times each name occurs in `src_tbl` and stores both the counts and the names in `dst_tbl`:

```
INSERT INTO dst_tbl (i, s) SELECT COUNT(*), name FROM src_tbl GROUP BY name;
```



When you use `INSERT ... SELECT`, you cannot use the same table both as a source and a destination.

3.22 Creating a Destination Table on the Fly from a Result Set

Problem

You want to run a `SELECT` query and save the result set into another table, but that table doesn't exist yet.

Solution

Create the destination table first, or create it directly from the result of the `SELECT`.

Discussion

If the destination table does not exist, you can create it first with a `CREATE TABLE` statement, then copy rows into it with `INSERT ... SELECT` as described in Recipe 3.21. This technique works for any version of MySQL.

In MySQL 3.23 and up, a second option is to use `CREATE TABLE ... SELECT`, which creates the destination table directly from the result of a `SELECT`. For example, to create `dst_tbl` and copy the entire contents of `src_tbl` into it, do this:

```
CREATE TABLE dst_tbl SELECT * FROM src_tbl;
```

MySQL creates the columns in `dst_tbl` based on the name, number, and type of the columns in `src_tbl`. Add an appropriate `WHERE` clause, should you wish to copy only certain rows. If you want to create an empty table, use a `WHERE` clause that is always `false`:

```
CREATE TABLE dst_tbl SELECT * FROM src_tbl WHERE 0;
```

To copy only some of the columns, name the ones you want in the `SELECT` part of the statement. For example, if `src_tbl` contains columns `a`, `b`, `c`, and `d`, you can copy just `b` and `d` like this:

```
CREATE TABLE dst_tbl SELECT b, d FROM src_tbl;
```

To create columns in a different order than that in which they appear in the source table, just name them in the desired order. If the source table contains columns `a`, `b`, and `c`, but you want them to appear in the destination table in the order `c`, `a`, and `b`, do this:

```
CREATE TABLE dst_tbl SELECT c, a, b FROM src_tbl;
```

To create additional columns in the destination table besides those selected from the source table, provide appropriate column definitions in the `CREATE TABLE` part of the statement. The following statement creates `id` as an `AUTO_INCREMENT` column in `dst_tbl`, and adds columns `a`, `b`, and `c` from `src_tbl`:

```
CREATE TABLE dst_tbl
(
  id INT NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (id)
)
SELECT a, b, c FROM src_tbl;
```

The resulting table contains four columns in the order `id`, `a`, `b`, `c`. Defined columns are assigned their default values. (This means that `id`, being an `AUTO_INCREMENT` column, will be assigned successive sequence numbers starting from one. See Recipe 11.1.)

If you derive a column's values from an expression, it's prudent to provide an alias to give the column a name. Suppose `src_tbl` contains invoice information listing items in each invoice. Then the following statement generates a summary of each invoice named in the table, along with the total cost of its items. The second column includes an alias because the default name for an expression is the expression itself, which is difficult to work with:

```
CREATE TABLE dst_tbl
SELECT inv_no, SUM(unit_cost*quantity) AS total_cost
FROM src_tbl
GROUP BY inv_no;
```

In fact, prior to MySQL 3.23.6, the alias is required, not just advisable; column naming rules are stricter and an expression is not a legal name for a column in a table.

CREATE TABLE ... SELECT is extremely convenient, but does have some limitations. These stem primarily from the fact that the information available from a result set is not as extensive as what you can specify in a CREATE TABLE statement. If you derive a table column from an expression, for example, MySQL has no idea whether or not the column should be indexed or what its default value is. If it's important to include this information in the destination table, use the following techniques:

- If you want indexes in the destination table, you can specify them explicitly. For example, if `src_tbl` has a PRIMARY KEY on the `id` column, and a multiple-column index on `state` and `city`, you can specify them for `dst_tbl` as well:

```
CREATE TABLE dst_tbl (PRIMARY KEY (id), INDEX(state,city))
SELECT * FROM src_tbl;
```

- Column attributes such as AUTO_INCREMENT and a column's default value are not copied to the destination table. To preserve these attributes, create the table, then use ALTER TABLE to apply the appropriate modifications to the column definition. For example, if `src_tbl` has an `id` column that is not only a PRIMARY KEY but an AUTO_INCREMENT column, copy the table, then modify it:

```
CREATE TABLE dst_tbl (PRIMARY KEY (id)) SELECT * FROM src_tbl;
ALTER TABLE dst_tbl MODIFY id INT UNSIGNED NOT NULL AUTO_INCREMENT;
```

- If you want to make the destination table an exact copy of the source table, use the cloning technique described in Recipe 3.25.

3.23 Moving Records Between Tables Safely

Problem

You're moving records by copying them from one table to another and then deleting them from the original table. But some records seem to be getting lost.

Solution

Be careful to delete exactly the same set of records from the source table that you copied to the destination table.

Discussion

Applications that copy rows from one table to another can do so with a single operation, such as INSERT ... SELECT to retrieve the relevant rows from the source table and add them to the destination table. If an application needs to *move* (rather than copy) rows, the procedure is a little more complicated: After copying the rows to the destination table, you must remove them from the source table. Conceptually, this is

nothing more than `INSERT ... SELECT` followed by `DELETE`. In practice, the operation may require more care, because it's necessary to select exactly the same set of rows in the source table for both the `INSERT` and `DELETE` statements. If other clients insert new rows into the source table after you issue the `INSERT` and before you issue the `DELETE`, this can be tricky.

To illustrate, suppose you have an application that uses a working log table `worklog` into which records are entered on a continual basis, and a long-term repository log table `repolog`. Periodically, you move `worklog` records into `repolog` to keep the size of the working log small, and so that clients can issue possibly long-running log analysis queries on the repository without blocking processes that create new records in the working log.*

How do you properly move records from `worklog` to `repolog` in this situation, given that `worklog` is subject to ongoing insert activity? The obvious (but incorrect) way is to issue an `INSERT ... SELECT` statement to copy all the `worklog` records into `repolog`, followed by a `DELETE` to remove them from `worklog`:

```
INSERT INTO repolog SELECT * FROM worklog;
DELETE FROM worklog;
```

This is a perfectly workable strategy when you're certain nobody else will insert any records into `worklog` during the time between the two statements. But if other clients insert new records in that period, they'll be deleted without ever having been copied, and you'll lose records. If the tables hold logs of web page requests, that may not be such a big deal, but if they're logs of financial transactions, you could have a serious problem.

What can you do to keep from losing records? Two possibilities are to issue both statements within a transaction, or to lock both tables while you're using them. These techniques are covered in Chapter 15. However, either one might block other clients longer than you'd prefer, because you tie up the tables for the duration of both queries. An alternative strategy is to move only those records that are older than some cutoff point. For example, if the log records have a column `t` containing a timestamp, you can limit the scope of the selected records to all those created before today. Then it won't matter whether new records are added to `worklog` between the copy and delete operations. Be sure to specify the cutoff properly, though. Here's a method that fails under some circumstances:

```
INSERT INTO repolog SELECT * FROM worklog WHERE t < CURDATE();
DELETE FROM worklog WHERE t < CURDATE();
```

This won't work if you happen to issue the `INSERT` statement at one second before midnight and the `SELECT` statement one second later. The value of `CURDATE()` will differ for the two statements, and the `DELETE` operation may remove too many records.

* If you use a MyISAM log table that you only insert into and never delete from or modify, you can run queries on the table without preventing other clients from inserting new log records at the end of the table.

If you're going to use a cutoff, make sure it has a fixed value, not one that may change between statements. For example, a SQL variable can be used to save the value of `CURDATE()` in a form that won't change as time passes:

```
SET @cutoff = CURDATE();
INSERT INTO repolog SELECT * FROM worklog WHERE t < @cutoff;
DELETE FROM worklog WHERE t < @cutoff;
```

This ensures that both statements use the same cutoff value so that the `DELETE` operation doesn't remove records that it shouldn't.

3.24 Creating Temporary Tables

Problem

You need a table only for a short time, then you want it to disappear automatically.

Solution

Create a `TEMPORARY` table and let MySQL take care of clobbering it.

Discussion

Some operations require a table that exists only temporarily and that should disappear when it's no longer needed. You can of course issue a `DROP TABLE` statement explicitly to remove a table when you're done with it. Another option, available in MySQL 3.23.2 and up, is to use `CREATE TEMPORARY TABLE`. This statement is just like `CREATE TABLE` except that it creates a transient table that disappears when your connection to the server closes, if you haven't already removed it yourself. This is extremely useful behavior because you need not remember to remove the table. MySQL drops it for you automatically.

Temporary tables are connection-specific, so several clients each can create a temporary table having the same name without interfering with each other. This makes it easier to write applications that use transient tables, because you need not ensure that the tables have unique names for each client. (See Recipe 3.26 for further discussion of this issue.)

Another property of temporary tables is that they can be created with the same name as a permanent table. In this case, the temporary table "hides" the permanent table for the duration of its existence, which can be useful for making a copy of a table that you can modify without affecting the original by mistake. The `DELETE` statement in the following set of queries removes records from a temporary `mail` table, leaving the original permanent one unaffected:

```
mysql> CREATE TEMPORARY TABLE mail SELECT * FROM mail;
mysql> SELECT COUNT(*) FROM mail;
```

```

+-----+
| COUNT(*) |
+-----+
|      16 |
+-----+
mysql> DELETE FROM mail;
mysql> SELECT COUNT(*) FROM mail;
+-----+
| COUNT(*) |
+-----+
|      0 |
+-----+
mysql> DROP TABLE mail;
mysql> SELECT COUNT(*) FROM mail;
+-----+
| COUNT(*) |
+-----+
|      16 |
+-----+

```

Although temporary tables created with `CREATE TEMPORARY TABLE` have the preceding benefits, keep the following caveats in mind:

- If you want to reuse the temporary table within a given session, you'll still need to drop it explicitly before recreating it. It's only the *last* use within a session that you need no explicit `DROP TABLE` for. (If you've already created a temporary table with a given name, attempting to create a second one with that name results in an error.)
- Some APIs support persistent connections in a web environment. Use of these prevents temporary tables from being dropped as you expect when your script ends, because the web server keeps the connection open for reuse by other scripts. (The server may close the connection eventually, but you have no control over when that happens.) This means it can be prudent to issue the following statement prior to creating a temporary table, just in case it's still hanging around from the previous execution of the script:

```
DROP TABLE IF EXISTS tbl_name
```

- If you modify a temporary table that “hides” a permanent table with the same name, be sure to test for errors resulting from dropped connections. If a client program automatically reconnects after a dropped connection, you'll be modifying the *original* table after the reconnect.

3.25 Cloning a Table Exactly

Problem

You need an exact copy of a table, and `CREATE TABLE ... SELECT` doesn't suit your purposes because the copy must include the same indexes, default values, and so forth.

Solution

Use `SHOW CREATE TABLE` to get a `CREATE TABLE` statement that specifies the source table's structure, indexes and all. Then modify the statement to change the table name to that of the clone table and execute the statement. If you need the table contents copied as well, issue an `INSERT INTO ... SELECT` statement, too.

Discussion

Because `CREATE TABLE ... SELECT` does not copy indexes or the full set of column attributes, it doesn't necessarily create a destination table as an exact copy of the source table. Because of that, you might find it more useful to issue a `SHOW CREATE TABLE` query for the source table. This statement is available as of MySQL 3.23.20; it returns a row containing the table name and a `CREATE TABLE` statement that corresponds to the table's structure—including its indexes (keys), column attributes, and table type:

```
mysql> SHOW CREATE TABLE mail\G
***** 1. row *****
      Table: mail
Create Table: CREATE TABLE `mail` (
  `t` datetime default NULL,
  `srcuser` char(8) default NULL,
  `srchost` char(20) default NULL,
  `dstuser` char(8) default NULL,
  `dsthost` char(20) default NULL,
  `size` bigint(20) default NULL,
  KEY `t` (`t`)
) TYPE=MyISAM
```

By issuing a `SHOW CREATE TABLE` statement from within a program and performing a string replacement to change the table name, you obtain a statement that can be executed to create a new table with the same structure as the original. The following Python function takes three arguments (a connection object, and the names of the source and destination tables). It retrieves the `CREATE TABLE` statement for the source table, modifies it to name the destination table, and returns the result:

```
# Generate a CREATE TABLE statement to create dst_tbl with the same
# structure as the existing table src_tbl. Return None if an error
# occurs. Requires the re module.

def gen_clone_query (conn, src_tbl, dst_tbl):
    try:
        cursor = conn.cursor ()
        cursor.execute ("SHOW CREATE TABLE " + src_tbl)
        row = cursor.fetchone ()
        cursor.close ()
        if row == None:
            query = None
        else:
```

```

# Replace src_tbl with dst_tbl in the CREATE TABLE statement
query = re.sub ("CREATE TABLE .*`" + src_tbl + "`",
               "CREATE TABLE `" + dst_tbl + "`",
               row[1])

except:
    query = None
return query

```

You can execute the resulting statement as is to create the new table if you like:

```

query = gen_clone_query (conn, old_tbl, new_tbl)
cursor = conn.cursor ()
cursor.execute (query)
cursor.close ()

```

Or you can get more creative. For example, to create a temporary table rather than a permanent one, change CREATE to CREATE TEMPORARY before executing the statement:

```

query = gen_clone_query (conn, old_tbl, new_tbl)
query = re.sub ("CREATE ", "CREATE TEMPORARY ", query)
cursor = conn.cursor ()
cursor.execute (query)
cursor.close ()

```

Executing the statement returned by `gen_clone_query()` creates an empty copy of the source table. To copy the contents as well, do something like this after creating the copy:

```

cursor = conn.cursor ()
cursor.execute ("INSERT INTO " + new_tbl + " SELECT * FROM " + old_tbl)
cursor.close ()

```



Prior to MySQL 3.23.50, there are a few attributes that you can specify in a CREATE TABLE statement that SHOW CREATE TABLE does not display. If your source table was created with any of these attributes, the cloning technique shown here will create a destination table that does not have quite the same structure.

3.26 Generating Unique Table Names

Problem

You need to create a table with a name that is guaranteed not to exist already.

Solution

If you can create a TEMPORARY table, it doesn't matter if the name exists already. Otherwise, try to generate a value that is unique to your client program and incorporate it into the table name.

Discussion

MySQL is a multiple-client database server, so if a given script that creates a transient table might be invoked by several clients simultaneously, you must take care to keep multiple invocations of the script from fighting over the same table name. If the script creates tables using `CREATE TEMPORARY TABLE`, there is no problem because different clients can create temporary tables having the same name without clashing.

If you can't use `CREATE TEMPORARY TABLE` because the server version is older than 3.23.2, you should make sure that each invocation of the script creates a uniquely named table. To do this, incorporate into the name some value that is guaranteed to be unique per invocation. A timestamp won't work, because it's easily possible for two instances of a script to be invoked within the same second. A random number may be somewhat better. For example, in Java, you can use the `java.util.Random()` class to create a table name like this:

```
import java.util.Random;
import java.lang.Math;

Random rand = new Random ();
int n = rand.nextInt ();           // generate random number
n = Math.abs (n);                 // take absolute value
String tblName = "tmp_tbl_" + n;
```

Unfortunately, random numbers only reduce the possibility of name clashes, they do not eliminate it. Process ID (PID) values are a better source of unique values. PIDs are reused over time, but never for two processes at the same time, so a given PID is guaranteed to be unique among the set of currently executing processes. You can use this fact to create unique table names as follows:

Perl:

```
my $tbl_name = "tmp_tbl_$$";
```

PHP:

```
$tbl_name = "tmp_tbl_" . posix_getpid ();
```

Python:

```
import os
tbl_name = "tmp_tbl_%d" % os.getpid ()
```

Note that even if you create a table name using a value like a PID that is guaranteed to be unique to a given script invocation, there may still be a chance that the table will exist. This can happen if a previous invocation of the script with the same PID created a table with the same name, but crashed before removing the table. On the other hand, any such table cannot still be in use because it will have been created by a process that is no longer running. Under these circumstances, it's safe to remove the table if it does exist by issuing the following statement:

```
DROP TABLE IF EXISTS tbl_name
```

Then you can go ahead and create the new table.