Paul DuBois

Fifth Edition

MySQL®

Developer's Library

MySQL

Fifth Edition

Earlier, in Section 7.4.2, "Handling Statements That Return a Result Set," we wrote a version of `process_result_set()` that printed columns from result set rows in tab-delimited format. That's good for certain purposes (such as when you want to import the data into a spreadsheet), but it's not a nice display format for visual inspection or for printouts. Recall that our earlier version of `process_result_set()` produced this output:

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George H.W. Milton  MA
Bush    George W.   New Haven   CT
Carter  James E.    Plains  GA
...
```

Let's write a different version of `process_result_set()` that produces tabular output instead by titling and "boxing" each column. This version displays those same results in a format that's easier to interpret:

```
+------------+--------------+--------------------+-------+
| last_name  | first_name   | city               | state |
+------------+--------------+--------------------+-------+
| Adams      | John         | Braintree          | MA    |
| Adams      | John Quincy  | Braintree          | MA    |
| Arthur     | Chester A.   | Fairfield          | VT    |
| Buchanan   | James        | Mercersburg        | PA    |
| Bush       | George H.W.  | Milton             | MA    |
| Bush       | George W.    | New Haven          | CT    |
| Carter     | James E.     | Plains             | GA    |
...
+------------+--------------+--------------------+-------+
```

The display algorithm performs these steps:

1. Determine the display width of each column.
2. Print a row of boxed column labels (delimited by vertical bars and preceded and followed by rows of dashes).
3. Print the values in each row of the result set, with each column boxed (delimited by vertical bars) and lined up vertically. Print numbers right justified and the word "NULL" for `NULL` values.
4. At the end, print a count of the rows retrieved.

This exercise provides a good demonstration showing how to use result set metadata because it requires knowledge of quite a number of things about the result set other than just the values of the data contained in its rows.

You may be thinking to yourself, "Hmm, that description sounds suspiciously similar to the way `mysql` displays its output." Yes, it does, and you're welcome to compare the source for `mysql` to the code we end up with for `process_result_set()`. They're not the same, and you might find it instructive to compare the two approaches to the same problem.

First, it's necessary to determine the display width of each column. The following listing shows how to do this. Observe that the calculations are based entirely on the result set metadata, and make no reference whatsoever to the row values:

```
MYSQL_FIELD   *field;
unsigned long col_len;
unsigned int  i;

/* determine column display widths; requires result set to be */
/* generated with mysql_store_result(), not mysql_use_result() */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  col_len = strlen (field->name);
  if (col_len < field->max_length)
    col_len = field->max_length;
  if (col_len < 4 && !IS_NOT_NULL (field->flags))
    col_len = 4;  /* 4 = length of the word "NULL" */
  field->max_length = col_len;  /* reset column info */
}
```

This code calculates column widths by iterating through the `MYSQL_FIELD` structures for the columns in the result set. We position to the first structure by calling `mysql_field_seek()`. Subsequent calls to `mysql_fetch_field()` return pointers to the structures for successive columns. The width of a column for display purposes is the maximum of three values, each of which depends on metadata in the column information structure:

- The length of `field->name`, the column title.
- `field->max_length`, the length of the longest data value in the column.
- The length of the string "NULL", if `field->flags` indicates that the column can contain `NULL` values.

Notice that after the display width for a column is known, we assign that value to `max_length`, which is a member of a structure that we obtain from the client library. Is that permitted, or should the contents of the `MYSQL_FIELD` structure be considered read only? Normally, I would say "read only," but some of the client programs in the MySQL distribution change the `max_length` value in a similar way, so I assume that it's okay. (If you prefer an alternative approach that doesn't modify `max_length`, allocate an array of `unsigned long` values and store the calculated widths in that array.)

The display width calculations involve one caveat. Recall that `max_length` has no meaning when you create a result set using `mysql_use_result()`. Because we need `max_length` to determine the display width of the column values, proper operation of the algorithm requires that the result set be generated using `mysql_store_result()`. In programs that use `mysql_use_result()` rather than `mysql_store_result()`, one possible workaround is to use the `length` member of the `MYSQL_FIELD` structure, which tells you the maximum length that column values can be.

When we know the column widths, we're ready to print. Titles are easy to handle. For a given column, use the column information structure pointed to by `field` and print the `name` member, using the width calculated earlier:

```
printf (" %-*s |", (int) field->max_length, field->name);
```

For the data, loop through the rows in the result set, printing column values for the current row during each iteration. Printing column values from the row is a bit tricky because a value might be `NULL`, or it might represent a number (in which case we print it right justified). Column values are printed as follows, where `row[i]` holds the data value and `field` points to the column information:

```
if (row[i] == NULL)       /* print the word "NULL" */
  printf (" %-*s |", (int) field->max_length, "NULL");
else if (IS_NUM (field->type))  /* print value right-justified */
  printf (" %*s |", (int) field->max_length, row[i]);
else              /* print value left-justified */
  printf (" %-*s |", (int) field->max_length, row[i]);
```

The value of the `IS_NUM()` macro is true if the column data type indicated by `field->type` is one of the numeric types, such as `INT`, `FLOAT`, or `DECIMAL`.

The final code to display the result set follows. Because we're printing lines of dashes multiple times, it's easier to write a `print_dashes()` function to do so rather than to repeat the dash-generation code several places:

```
void
print_dashes (MYSQL_RES *res_set)
{
MYSQL_FIELD   *field;
unsigned int  i, j;

  mysql_field_seek (res_set, 0);
  fputc ('+', stdout);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    for (j = 0; j < field->max_length + 2; j++)
      fputc ('-', stdout);
    fputc ('+', stdout);
  }
```

```
  fputc ('\n', stdout);
}

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_ROW     row;
MYSQL_FIELD   *field;
unsigned long col_len;
unsigned int  i;

  /* determine column display widths; requires result set to be */
  /* generated with mysql_store_result(), not mysql_use_result() */
  mysql_field_seek (res_set, 0);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
      col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
      col_len = 4;  /* 4 = length of the word "NULL" */
    field->max_length = col_len;  /* reset column info */
  }

  print_dashes (res_set);
  fputc ('|', stdout);
  mysql_field_seek (res_set, 0);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    printf (" %-*s |", (int) field->max_length, field->name);
  }
  fputc ('\n', stdout);
  print_dashes (res_set);

  while ((row = mysql_fetch_row (res_set)) != NULL)
  {
    mysql_field_seek (res_set, 0);
    fputc ('|', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
      field = mysql_fetch_field (res_set);
      if (row[i] == NULL)       /* print the word "NULL" */
        printf (" %-*s |", (int) field->max_length, "NULL");
      else if (IS_NUM (field->type))  /* print value right-justified */
        printf (" %*s |", (int) field->max_length, row[i]);
```

```
      else              /* print value left-justified */
        printf (" %-*s |", (int) field->max_length, row[i]);
    }
    fputc ('\n', stdout);
  }
  print_dashes (res_set);
  printf ("Number of rows returned: %lu\n",
          (unsigned long) mysql_num_rows (res_set));
}
```

The MySQL client library provides several ways to access the column information structures. For example, the code in the preceding example accesses these structures several times using loops of the following general form:

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  ...
}
```

However, the combination of `mysql_field_seek()` and `mysql_fetch_field()` is only one way of getting `MYSQL_FIELD` structures. For other ways, see the descriptions of the `mysql_fetch_fields()` and `mysql_fetch_field_direct()` functions in Appendix G, "C API Reference."

> **Use the `metadata` Program to Display Result Set Metadata**
>
> The `sampdb` distribution contains the source for a program named `metadata` that you can compile and run to see what metadata different kinds of statements produce. It prompts for and executes SQL statements, but displays result set metadata rather than result set contents. For comparison, invoke `mysql` with the `--column-type-info` option and execute the same statements.

7.4.7 Encoding Special Characters and Binary Data

Programs that execute statements must take care with certain characters. For example, to include a quote character within a quoted string, either double the quote or precede it by a backslash:

```
'O''Malley'
'O\'Malley'
```

This section describes how to handle quoting issues in string values and how to work with binary data.

7.4.7.1 Working with Strings That Contain Special Characters

If inserted literally into a statement, data values containing quotes, null bytes, or backslashes can cause problems when you try to execute the statement. The following discussion describes the nature of the difficulty and how to solve it.

Suppose that you want to construct a `SELECT` statement based on the contents of the null-terminated string pointed to by the `name_val` variable:

```
char stmt_buf[1024];

sprintf (stmt_buf, "SELECT * FROM mytbl WHERE name='%s'", name_val);
```

If the value of `name_val` is something like `O'Malley, Brian`, the resulting statement is illegal because a quote appears inside a quoted string:

```
SELECT * FROM mytbl WHERE name='O'Malley, Brian'
```

You must treat the inner quote specially so that the server doesn't interpret it as the end of the name. The standard SQL convention for doing this is to double the quote within the string. MySQL understands that convention, and also permits the quote to be preceded by a backslash, so you can write the statement using either of the following formats:

```
SELECT * FROM mytbl WHERE name='O''Malley, Brian'
SELECT * FROM mytbl WHERE name='O\'Malley, Brian'
```

To deal with this problem, use `mysql_real_escape_string()`, which encodes special characters to make them usable in quoted strings. Characters that `mysql_real_escape_string()` considers special are the null byte, single quote, double quote, backslash, newline, carriage return, and Control-Z. (The last one is special on Windows, where it sometimes signifies end-of-file.)

When should you use `mysql_real_escape_string()`? The safest answer is "always." However, if you're sure of the format of your data and know that it's okay—perhaps because you have performed some prior validation check on it—you need not encode it. For example, if you are working with strings that you know represent legal phone numbers consisting entirely of digits and dashes, you need not call `mysql_real_escape_string()`. Otherwise, you probably should.

`mysql_real_escape_string()` encodes problematic characters by turning them into two-character sequences that begin with a backslash. For example, a null byte becomes '`\0`', where the '`0`' is a printable ASCII zero, not a null. Backslash, single quote, and double quote become '`\\`', '`\'`', and '`\"`'.

To use `mysql_real_escape_string()`, invoke it like this:

```
to_len = mysql_real_escape_string (conn, to_str, from_str, from_len);
```

`mysql_real_escape_string()` encodes `from_str` and writes the result into `to_str`. It also adds a terminating null, which is convenient because you can use the resulting string with functions such as `strcpy()`, `strlen()`, or `printf()`.