

## Using MySQL with XML

Paul DuBois – November, 2001

### TABLE OF CONTENTS

Introduction

Writing Query Results  
with XML

- Writing XML by Adding Element Tags Yourself
- Writing XML by Using a Utility Mode

Reading XML Documents  
into MySQL

- Reading XML with XML::Parser
- Reading XML with XML::XPath

Delivering XML Over the  
Web

More Information

About NuSphere

A growing number of applications today use data represented in the form of XML documents. XML use is on the rise because it is a simple ASCII format that has a well-defined yet extensible structure. As a result, many standardized XML processing tools have been written. What is the impact of these developments for users of the MySQL database system? MySQL has no native facilities for dealing with XML—does this mean we are left out of the XML movement? By no means. Many of the most popular languages for writing MySQL applications also have XML support, so these languages provide a natural bridge for spanning the gap between XML and relational databases. The following list indicates just some of the possibilities open to you for employing XML processing techniques to make more productive use of your MySQL server:

- *XML as a data transfer medium.* Writing a query result as an XML document results in a platform-neutral ASCII file that can be used by other applications, even those that are not necessarily database-oriented. The recipient of such a document can employ standard XML tools to parse it and recover the original data values. Used this way, XML serves as an interface between your MySQL database and other applications that can read XML but may know nothing about MySQL. This works in the other direction, too. If an application can produce XML-formatted documents, you can read them and store the information contained therein into MySQL by using simple XML parsing techniques.

- *XML as a web delivery format.* XML's simple, well-defined structure makes it useful for information delivery in a web environment. For example, you can set up an information feed that clients can use on an automated basis: Define XML formatting conventions with which to express the information and provide access to it through a web script. Clients can send requests to the script, which connects to MySQL, retrieves the desired information, and formats it as an XML document that is returned to the client. The client then extracts information from the document using standard XML tools.
- *Using XML to write web pages.* As the limitations of HTML for writing web pages become more keenly felt, web developers turn increasingly to the more expressive capabilities of XML. HTML serves primarily as a destination format, whereas XML is useful both as source and destination formats. For example, an XML document can incorporate the results of database queries and then, with the help of a rendering engine such as AxKit, be transformed into a format that matches the type of client you wish to serve. You can send HTML, WML, or plain text to web browsers, wireless devices, or printers. (Or, as indicated in the previous item, you can serve the document directly to clients that understand XML.) Contrast this with HTML, which does not render well into other formats.
- *Storing XML directly.* You can of course store XML itself in your database. You might store templates for documents such as form letters that you combine with customer records to produce mailings, for example.

To help you get started, this article focuses on the first two items in the preceding list. It shows how to create XML documents from query results, how to create new database records from information contained in XML documents, and how to set up a web-based XML delivery service.

You can use XML from within any language that has the appropriate processing tools available. For example, XML APIs exist for languages such as PHP, Python, Java, and Tcl, all of which also have MySQL capabilities. This article uses Perl, another language that enjoys strong XML and MySQL API support. The examples use the Perl DBI module to interact with MySQL, in

conjunction with a variety of XML processing modules. Some NuSphere products include MySQL, Perl, and the DBI module, and the XML modules can be obtained from the CPAN (the Perl archive at [cpan.perl.org](http://cpan.perl.org)).

The examples shown here assume that you have a MySQL database named `test` on the local host, accessed through a MySQL account with a user name and password of `testuser` and `testpass`. The article also assumes that you have a basic working knowledge of XML and the DBI module.

## Writing Query Results with XML

Suppose that you want to produce XML output from a table named `animal` that has two string columns:

```
+-----+-----+
| name  | category |
+-----+-----+
| snake | reptile  |
| frog  | amphibian|
| tuna  | fish     |
| racoon| mammal   |
+-----+-----+
```

The information in the table can be retrieved easily using a simple SQL statement:

```
SELECT name, category FROM animal
```

The question is how to convert the information residing in MySQL to an XML representation.

The two methods shown in the following discussion demonstrate how to write an XML document “manually” by adding the XML tags yourself, and how to use one of the available Perl modules to do most of the work.

## Writing XML By Adding Element Tags Yourself

One way to generate XML from the contents of the `animal` table is to write all the document tags explicitly using `print` statements. Connect to MySQL, issue the query, fetch the results, and wrap them within the appropriate tags required to produce a properly formatted XML document:

```
use strict;
use DBI;

print "<?xml version=\"1.0\"?>\n";
print "<dataset>\n";
my $dbh = DBI->connect ("DBI:mysql:test",
                      "testuser", "testpass",
                      { RaiseError => 1, PrintError => 0});
my $sth = $dbh->prepare ("SELECT name, category FROM animal");
$sth->execute ();
while (my ($name, $category) = $sth->fetchrow_array ())
{
    print " <row>\n";
    print "  <name>$name</name>\n";
    print "  <category>$category</category>\n";
    print " </row>\n";
}
$dbh->disconnect ();
print "</dataset>\n";
```

This script produces the following XML representation of the data set, where the root element `<dataset>` contains a `<row>` element for each row in the table and each row contains an element per column:

```
<?xml version="1.0"?>
<dataset>
  <row>
    <name>snake</name>
```

```

    <category>reptile</category>
</row>
<row>
  <name>frog</name>
  <category>amphibian</category>
</row>
<row>
  <name>tuna</name>
  <category>fish</category>
</row>
<row>
  <name>raccoon</name>
  <category>mammal</category>
</row>
</dataset>

```

This script has the advantage of being simple to write, and thus can be implemented relatively quickly. However, it also has some specific disadvantages:

- Knowledge of the table structure is built in, such as the column names to use for element tags within the rows. This means the code could not be used with a different query without modifying the processing loop.
- The script does not encode any special characters that might occur within data values (such as `<` or `&`). (The `animal` table does not have any, but another table might.)

The script could be rewritten to be less query-specific and to perform encoding, but an easier approach is to use existing Perl modules that do the work for you.

## Writing XML By Using a Utility Module

A number of Perl modules are available for writing XML documents; the example shown in this section uses `XML::Generator::DBI`. This module is designed to work in concert with DBI, which

makes it especially convenient for writing scripts that fetch information from MySQL or other databases. The API for `XML::Generator::DBI` consists of two methods:

- `$gen = XML::Generator::DBI->new(arguments);`

`new()` creates a new generator object. This method requires that you first open a connection to MySQL to get a database handle, and that you supply a handler object that understands the SAX (Simple API for XML) protocol. Pass the database handle and the SAX object to `new()` to obtain an object to use for executing queries.

- `$gen->execute(query);`

The `execute()` method issues the query and writes the results as an XML document. The generator uses the database handle to read information from the database, and it posts SAX events to the SAX object to write the information in XML format.

The following script shows how to use `XML::Generator::DBI` to convert the contents of the `animal` table to XML. The SAX handler is obtained from the `XML::Handler::YAWriter` (yet another writer) module.

```
use strict;
use DBI;
use XML::Generator::DBI;
use XML::Handler::YAWriter;

my $dbh = DBI->connect ("DBI:mysql:test",
                      "testuser", "testpass",
                      { RaiseError => 1, PrintError => 0});
my $out = XML::Handler::YAWriter->new (AsFile => "-");
my $gen = XML::Generator::DBI->new (
                                Handler => $out,
                                dbh => $dbh
                                );

$gen->execute ("SELECT name, category FROM animal");
```

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

This example involves about the same amount of code as the previous one, but it's more general. For example, to generate XML for a different query, all you have to do is change the argument to the `execute()` call. The XML that this second script generates (shown below) is somewhat different than for the preceding example. Compare it the XML document shown earlier:

```
<?xml version="1.0" encoding="UTF-8"?>
<database>
  <select query="SELECT name, category FROM animal">
    <row>
      <name>snake</name>
      <category>reptile</category>
    </row>
    <row>
      <name>frog</name>
      <category>amphibian</category>
    </row>
    <row>
      <name>tuna</name>
      <category>fish</category>
    </row>
    <row>
      <name>racoon</name>
      <category>mammal</category>
    </row>
  </select>
</database>
```

This output differs in the following ways:

- The `<?xml?>` tag includes an `encoding` attribute specifying the UTF-8 character set. If the table had contained any characters that lie outside this set, `XML::Generator::DBI` would convert them to base64 encoding automatically.

- The document root element is `<database>` rather than `<dataset>`. You could change this if desired by providing a `RootElement` argument when creating the generator object:

```
my $gen = XML::Generator::DBI->new (
    Handler => $out,
    dbh => $dbh,
    RootElement => "dataset"
);
```

- The `<row>` elements are placed within a `<select>` element that shows the query used to produce the data set. Thus, they are at the third level within the document, not the second. This could be significant to applications that act as consumers of the document, depending on the method they use to read it.

The `XML::Generator::DBI` and `XML::Handler::YAWriter` modules have other options you can use to modify their behavior. For more information, read their documentation using the **perldoc** command:

```
% perldoc XML::Generator::DBI
% perldoc XML::Handler::YAWriter
```

## Reading XML Documents into MySQL

The previous section showed how to convert query results to XML for use by other applications. This section illustrates how to go in the opposite direction, that is, how to extract records from an XML document and insert them into MySQL. This task generally requires that you know something about the structure of the document and the table, so that you can determine the correspondence between document elements and table columns. The examples assume that you want to process an XML document, `animal.xml`, that contains new records to be added to the `animal` table. The records are contained within `<row>` elements, each of which includes elements for the columns in the record.

You also need a module that implements some kind of parsing mechanism. The most popular Perl parser is `XML::Parser`, a module that you can use directly, or indirectly through one of the higher-level parser modules that are built on top of it. Higher-level parsers implement a variety of approaches. Some modules convert the entire document to an in-memory structure. For example, `XML::DOM` produces a structure that conforms to the Document Object Module standard. If you use such a module, you can iterate through the structure to access each row's contents. Other modules, such as those that use SAX-based parsers, implement a streaming approach that returns elements of the document as they are encountered. Using an in-memory structure may be more convenient in some ways (particularly if you want to check relationships between column values), but if you're working with large amounts of data, a streaming approach that does not require holding the entire document in memory at once may be preferable.

The examples in this section show how to read `animal.xml` two ways, first by using `XML::Parser` directly, then by using `XML::XPath`, a module that (like `XML::DOM`) holds the document in memory.

## Reading XML with `XML::Parser`

`XML::Parser` supports a variety of ways to parse XML documents. The following example uses its `Handler` interface, in which you create a parser object and register callback functions to be invoked when the parser encounters opening and closing element tags, or text within the body of elements. The main part of the parsing script sets up a `%row` hash that contains a member for each column name in the `animal` table, connects to MySQL, and creates a parser object. Then it parses the input file, which causes the handler functions to be called at appropriate places in the file:

```
use strict;
use DBI;
use XML::Parser;

# create hash to hold values for expected column names
my %row = ("name" => undef, "category" => undef);
```

```

# connect to database and create parser object
my $dbh = DBI->connect ("DBI:mysql:test",
                      "testuser", "testpass",
                      { RaiseError => 1, PrintError => 0});

my $parser = new XML::Parser (
    Handlers => {
        Start => \&handle_start,
        End   => \&handle_end,
        Char  => \&handle_text
    }
);

# parse file and disconnect
$parser->parsefile ("animal.xml");
$dbh->disconnect ();

```

The callback functions coordinate to recognize the start and end of rows and to extract column values contained within each row. They work together using the following logic:

- When a `<row>` element begins, `handle_start()` empties the `%row` hash in preparation for collecting a new set of column values:

```

sub handle_start
{
    my ($p, $tag) = @_;    # parser, tag name

    if ($tag eq "row")
    {
        foreach my $key (keys (%row))
        {
            $row{$key} = undef;
        }
    }
}

```

- When the parser finds text data, `handle_text()` saves it in a `%row` hash member if the current element corresponds to one of the column names in the `animal` table (`name`, `category`):

```
sub handle_text
{
my ($p, $data) = @_;      # parser, text

    my $tag = $p->current_element ();
    $row{$tag} .= $data if exists ($row{$tag});
}

```

The reason that `handle_text()` concatenates the text to the current value of the hash member is that XML parsers do not necessarily return all the text for an XML element at once.

- When a `<row>` element ends, `handle_end()` uses the values collected in the `%row` hash to construct and issue an `INSERT` statement that creates a new record:

```
sub handle_end
{
my ($p, $tag) = @_;      # parser, tag name

    if ($tag eq "row")
    {
        my $str;
        # construct column assignments for INSERT statement
        foreach my $key (keys (%row))
        {
            $str .= "," if $str;
            $str .= "$key=" . $dbh->quote($row{$key});
        }
        $dbh->do ("INSERT INTO animal SET $str");
    }
}

```

`handle_end()` creates `INSERT` statements that look like this:

```
INSERT INTO animal SET name='name_val',category='category_val'
```

The `quote()` function escapes any special characters in the data values, to make them legal for inclusion in a SQL query string.

The preceding example only scratches the surface of the ways you can interact with the `XML::Parser` module. For more information, read its documentation with the **perldoc** command.

## Reading XML with XML::XPath

The `XML::Parser` example just shown represents a low-level approach to XML input processing that has the advantage of placing a low memory burden on your script. But it isn't necessarily very easy to understand, due to the fragmentation of different parsing tasks into separate handler functions. If you're willing to hold the document in memory in exchange for being able to use a method that is conceptually more intuitive, consider `XML::XPath`. This module implements the XPath specification, which allows you to specify absolute or relative element paths to the parts of the document you're interested in. For example, the absolute path

`/database/select/row` selects only `<row>` elements that are reached by traveling through `<database>` and `<select>` elements, whereas the relative path `//row` selects `<row>` elements reachable by any path from the document root.

The following example creates a new `XPath` object that contains an in-memory representation of the `animal.xml` file, then gets a pointer to a list of the `<row>` elements within the document. For each of these, it extracts the text for the `<name>` and `<category>` elements and inserts them into the `animal` table:

```
use strict;
use DBI;
use XML::XPath;
use XML::XPath::XMLParser;

my $dbh = DBI->connect ("DBI:mysql:test",
                      "testuser", "testpass",
                      { RaiseError => 1, PrintError => 0});
```

```

my $xp = XML::XPath->new (filename => "animal.xml");
my $nodelist = $xp->find ("//row");
foreach my $row ($nodelist->get_nodelist ())
{
    $dbh->do (
        "INSERT INTO animal (name, category) VALUES (?,?)",
        undef,
        $row->find ("name")->string_value (),
        $row->find ("category")->string_value ()
    );
}
$dbh->disconnect ();

```

The code uses a relative path (`//row`) to locate the `<row>` elements within the document. If you wanted to, you could specify an absolute path by changing the line that returns the node list. For example, to read rows from the document produced earlier in “Writing XML Using a Utility Module,” the node list would be obtained using an absolute path like this:

```
my $nodelist = $xp->find ("/database/select/row");
```

However, this is not as general because it requires a specific relationship between elements in the document. For example, it would not work for the document produced in “Writing XML Tags Yourself,” for which the absolute path would be written like this:

```
my $nodelist = $xp->find ("/dataset/row");
```

## Delivering XML Over the Web

XML has come to be used heavily on the web, although often in terms of documents that are transformed into HTML before delivery to clients that do not understand XML. This section illustrates an application for direct delivery of XML to clients that are presumed capable of understanding it. The application is named `animserv.pl` and functions as a simple information server: Given an animal name supplied by the client, it looks up the corresponding record from the `animal` table and returns the result to the client as an XML document. To invoke this application, a

client sends a request to your web server host that looks something like this, assuming that the script is installed in the `cgi-bin` directory:

```
http://your.server.host/cgi-bin/animserv.pl?name=raccoon
```

In response to the request, `animserv.pl` returns the following response:

```
Content-Length: 221
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<database>
  <select query="SELECT name, category FROM animal
                WHERE name = 'raccoon'">
    <row>
      <name>raccoon</name>
      <category>mammal</category>
    </row>
  </select>
</database>
```

The first couple of lines are the HTTP response headers that indicate the length of the response and the response type. This is followed by a blank line to separate the headers from the body, and then the body itself, which is an XML document containing the requested information. We'll assume that clients know that they should look for `<name>` and `<category>` tags within a `<row>` element to get the information. For a more sophisticated application, you might want to define and publish a DTD that describes how clients should interpret documents that the application produces.

The `animserv.pl` script is implemented as shown below. It obtains the `name` parameter supplied by the client, constructs a query to look for the appropriate record, and generates a web response that includes the appropriate headers:

```
use strict;
use CGI qw(param header);
use DBI;
```

```

use XML::Generator::DBI;
use XML::Handler::YAWriter;

# get animal name from client; exit if none found
my $name = param ("name");
exit (0) unless defined ($name);

# run query to look for given animal, generating result
# as a string so the length can be determined
my $dbh = DBI->connect ("DBI:mysql:test",
                      "testuser", "testpass",
                      { RaiseError => 1, PrintError => 0});
my $out = XML::Handler::YAWriter->new (AsString => 1);
my $gen = XML::Generator::DBI->new (
                                Handler => $out,
                                dbh => $dbh
                                );
$name = $dbh->quote ($name);
my $doc = $gen->execute (
    "SELECT name, category FROM animal
    WHERE name = $name"
    );
$dbh->disconnect ();

# generate type and length headers, then print document
print header (-Content_Type => "text/xml",
             -charset => "UTF-8",
             -Content_Length => length ($doc));
print $doc;

```

The `animserv.pl` script is very similar to the earlier example that used `XML::Generate::DBI`. The primary differences are:

- `animserv.pl` must determine what information the client wants, based on the value of the `name` parameter that indicates the desired animal. The script uses the `param()` function from the `CGI.pm` module to get this value.
- The script returns a data set that contains only part of the `animal` table, so the `execute()` call issues a query that includes a `WHERE` clause to indicate which record to retrieve. Note that the `$name` value is converted using `quote()` before it's inserted into the query string. It's dangerous to include client input directly into queries (someone may attempt to break the script by passing something nasty), so `animserv.pl` sanitizes the animal name value by escaping any special characters in the value.
- The XML document that is sent to the client must be preceded by HTTP headers that tell the client the length and type of the response. The length is obtained by passing the SAX handler an `AsString` argument to cause it to return the document as a string rather than printing it immediately. The length of the string becomes the value used in the `Content-Length:` header. The script prints the HTTP headers by invoking `header()`, another function from the `CGI.pm` module.

The `animserv.pl` application is very simple, but could be modified to act as the basis for a variety of information servers. For example, if you have a dictionary of words and meanings stored in MySQL, a few minor changes to the application would allow you to set up a dictionary server to which clients submit words and from which they receive definitions in response.

## More Information

This article illustrates some of the ways that MySQL can be used in applications that process XML documents, but the XML modules used in the example scripts represent only a few of the many available to you. To see others, visit the CPAN at [cpan.perl.org](http://cpan.perl.org). For more information on using XML from within Perl scripts, check out the Perl & XML column at the [xml.com](http://xml.com) web site; it contains a very helpful series of articles.

## About NuSphere Corporation

NuSphere delivers the first Internet Application Platform (IAP) based on open source components, providing an integrated foundation that allows companies to deploy reliable, cost-effective, enterprise-class applications across Windows, UNIX and Linux environments. NuSphere® Advantage is an integrated software suite that pairs the reliability and cost-effectiveness of PHP, Apache, Perl and open source databases with new technology for building business-critical web applications and web services. Based in Bedford, Mass., the company's commercial software services include technical support, consulting and training. For more information, visit [www.nusphere.com](http://www.nusphere.com) or call +1-781-280-4600.

NuSphere is a registered trademark in Australia, Norway, Hong Kong, Switzerland, and the European Community; NuSphere and PHPed are trademarks of NuSphere Corporation in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.

MySQL AB distributes the MySQL database pursuant to the applicable GNU General Public License that is available as of the date of this publication at <http://www.fsf.org/licenses/gpl.txt> and all of the terms and disclaimers contained therein. NuSphere Corporation is not affiliated with MySQL AB. The products and services of NuSphere Corporation are not sponsored or endorsed by MySQL AB.