Backing up MySQL databases – Part 1

Safe and Secure

The popular Open Source database, MySQL, offers various backup methods. Each of them has advantages and disadvantages that can be critical in certain circumstances.

BY THOMAS WÖLFER

Gerd Waloszek, www.visipix.com

MySQL databases can store all kinds of data, from goods for sale to details of your next advertising campaign. All this data needs to be backed up regularly. MySQL [1] has a wide range of backup facilities which are suitable for various applications. Users can choose between quick and extremely safe methods.

Option 1: Select into outfile

If your database consists of just a few tables, and you have shell access to the MySQL server, *select into outfile* is the simplest way of backing up. You can use this to quickly write a table, or part of a table, out to a file. Use the power of the *select* statement to restrict the selection of files to backup. To save a whole table, simply enter *select* *. Listing 1 provides an overview of *select into outfile* options.

File stands for the filename that you will be saving in the table. The *fields terminated by* option specifies the column separating character in the file. This typically defaults to a comma. Note that the output file cannot already exist. The account that runs *select into outfile* on MySQL has to have file permissions for the MySQL server machine. The advantage of this approach is the speed.

Listing 1: Select into outfile

```
select * into outfile 'File.txt'
fields terminated by ','
enclosed by '"'
lines terminated by 'n' from Table;
```

The corresponding import function, *load data infile*, offers similar performance. You can use this option to load a file that you previously stored with the *select into outfile* approach. *load data infile* expects the filename of the file to be parsed as a parameter:

```
load data info 'File' [replace➘
 | ignore] into table Table
```

What this basically does is to load the contents of the file into a table with the specified name. If you want to use the combination of *select into outfile* and *load data info* to back up your tables, you need to make sure that the target table is empty before you *load data*. As an alternative, you can specify the *ignore* parameter. It will drop lines with double primary keys, preventing errors if the target table already contains the same data.

There is a downside to *select into outfile* and *load data infile*. You can only load or save whole tables. This means having to modify your backup scripts each time you create a table to ensure that you backup the new table. If you need to back up a single table, *select into outfile* is a perfect solution.

Option 2: Backup Table

Backup Table works in a similar way to *select into outfile*. Users can formulate a SQL statement with the backup source. The *backup table* options expects one or more tables, and the name of the target file for the tables, as parameters: *backup table Table1 [,MoreTables] to 'File'*. This allows users to backup multiple tables simultaneously, using a comma-separated list of table names.

The MySQL documentation states that backup table is obsolete and that the developers are working on a new variant. You can ignore this statement, as the option works really well. The big advantage it provides in comparison to *select into outfile* is that you can save multiple tables with a single command. On the downside, you will need shell access to the MySQL server.

Option 3: Mysqldump

Mysqldump really takes the work out of creating backup copies. The program can save a complete database, or even multiple databases. You do not need a shell account on the MySQL server machine to do this. The port needs to be accessible to MySQL, otherwise mysqldump will be unable to connect to the server. The mysqldump command is as follows:

```
mysqldump [options] database ↩
[tables]
```

where *[tables]* means a list of all the tables in the database that you need to back up. To back up all the tables in the database, omit this parameter. This command will simply display a mass of data on your screen. To write the database to a file, redirect the output into a file. Again, the file must not already exist:

```
mysqldump database > backup.sql
```

The backup copy *backup.sql* includes both the table structure and the corresponding data. To load both sets of information from the backup copy, enter:

```
mysql < backup.sql
```

If you do not have local access to the MySQL server, admins can tell mysqldump to retrieve the backup data from a remote server. The program needs the *-host=RemoteHost* option. Be aware that this will generate a lot of traffic. This is the biggest obstacle to performing mysqldump backups across a network.

For a local backup, the program is the perfect tool. Mysqldump has a range of options (see Figure 1). For a complete list of command-line flags, check out the MySQL documentation at [2].

Option 4: Manual Backup

If halting the MySQL server for the time it takes to create a backup is one of your options, you can apply an extremely simple backup method. First, stop the MySQL daemon, then copy the database files, and relaunch the daemon. For this to work, you need to copy all the FRM, MYD, and MYI files from the database directory. To restore the data: stop the daemon, copy the database files back, relaunch the server.

Of course, a simple backup of this type has its restrictions. Stopping the MySQL daemon might be an option for smaller databases, websites with low traffic levels, or Intranet sites with defined maintenance windows, but large public sites need to be up 24x7 and will definitely not have the option.

Again, the administrator needs shell access to the database server, so manual backups are not an option for small-scale Web hosting scenarios. If you have shell access, a manual copy is a simple and reliable backup solution. Perl programmers have put together a convenient Perl script called *mysqlhotcopy* [2].

If running Perl scripts on your server is one of your options, mysqlhotcopy is one approach to automating the manual backup process. Again, there is a restriction: the script only works with ISAM and MyISAM databases.

Listing 2: Connecting to the local machine

```
$cL = mysql_connect(
  "Local_Server",
  "Write_Username", "Password");
if( ! $cL)
{
   print 'no local
  connection.';
   return;
}
```

Listing 3: Selecting IDs

```
$qs = "select id from Table
  order by id desc limit 1";
$rLocal = mysql_db_query(
  "Datebase", $qs, $cL);
if( ! $rLocal)
{
   print "query error: $qs
  <br>";
   return;
}
```

Listing 4: Querying a remote server

```
$qsr = "select * from Table
  where id>$num order by id";
$rRemote = mysql_db_query(
  "Database", $qsr, $cR);
if( ! $rRemote)
{
   print "query error: $qsr
  <br>";
   return;
}
```

Option 5: PHP Scripts

You can use a scripting language such as PHP to customize MySQL data backups to reflect your own requirements. The following PHP approach is simple, providing a basic framework on which you can build your own backup script. The full script is available as a download from [3]. A general-purpose solution for transferring data between two MySQL databases is conceivable, such as an extension to PHP MyAdmin [4].

The script expects you to have two MySQL servers up and running. One of the machines also needs to run Apache with PHP support enabled. You also need to install MySQL support for PHP. This is installed by default by the PHP distribution. The backup machine also needs access to the MySQL ports. MySQL typically listens on ports 3306 and 6000 for UDP and TCP.

The way your data are organized on the server is critical to creating a useful backup. If you continually need to modify data, you will need to backup more data than if new records were simply added from time to time. In the first case, you need to store both new data that have been added since your last backup, and any records that have been modified in this time. In the second case, it is quite sufficient to back up the new records. If you intend to use a browser or WGET to launch the script, you need to create a normal PHP page with the instructions required for HTML:

```
<html><head></head><body>
<?php Update(); ?>
</body></html>
```

A minimal HTML page where the PHP statement calls the update function is all you need. In turn, this function will launch all the other update functions one after another. We will be looking at an update function for an incremental update, that is an update that only backs up new records, in the following sections:

```
function UpdateNew()
{
  print "Begin Update New <br>";
```

The function starts by outputting a short message. This is practical if you use a browser to call the update, as it allows the user to see what stage the update has gotten to:

```
$cR = mysql_connect("Remote_↩
Server","Read_Username", ↩
"Password");
```

We now want the script to open up a connection to the remote database server (the remote server is the main server where the source data for our backup are stored). We supply the name or IP address for the *Remote_Server*. Normally, machines of this kind, or the MySQL servers on these machines, are configured to support multiple usernames and passwords for read and write access. The script needs read access and supplies the credentials for an account with read access along with the matching password (*Read_Username*, *Password*) when opening up the connection:

```
if( ! $cR)
{
  print 'No remote connection';
  return;
}
```

As it is common for a connection to fail, the script first checks the return value for the *connect* function. If the connection has failed, the script terminates at this point.

The next step is to open up the connection to the local machine, again with error handling (see Listing 2). The local machine will be storing the data copy. We need to supply the hostname or IP address, along with the user's credentials in the script. This time, we also need a user account with write privileges.

Finding New Records

After setting up both connections to the database servers, we get to the hardest part. As the first sample script only copies new records, it needs to find out which records are in fact new. The script uses the serial number, or a unique ID for the line within the table. It assumes that the highest ID is assigned to the highest numbered record (see Listing 3).

In our example, the SQL query first checks for the highest ID in the table. Users only need to modify the placeholders *Table* and *Database* to reflect their environment, where *Table* refers to the table within the database, and *Database* to the name of the database we will be accessing on the MySQL server. Again, error handling is useful for this query to ensure that the query actually provokes (or even can provoke) a response. If a response is forthcoming, the script outputs the query string. This allows you to use your browser to check where this specific query has caused a problem:

```
$row = mysql_fetch_row↩
( $rLocal);
$num = $row[0];
```

Now we need to parse the number the query returns. PHP has many functions, but as we are interested in a single value, *mysql_fetch_row()* will do the job perfectly. The number is located in the first slot of the returned array, index *[0]*. This tells us the highest local record number. We can then perform a similar query on the remote server (see Listing 4).

In the case of the second query, we are only interested in numbers larger than the one we have already found. The query has been modified slightly to reflect this. The condition is formulated as a *where* statement. The returned number is stored in the results of the query which PHP can again parse:

```
$count = mysql_num_rows↩
( $rRemote);
```

This tells us how many records we need to copy. The next step copies the data (see Listing 5). The step is repeated as often as needed to create a complete copy of the new records. The script calls *mysql _fetch_row()* for each new record. The data are stored in an array.

After creating the copy, we need to insert the data into the table, and replace the *Table* placeholder with the name of our table. Instead of *Fields* we will use a comma-separated list of fields. The values to insert are *$row[0]*, *$row[1]* and so on. Finally, we will submit a query with the query string that the process so far has created. This will take care of actually inserting the data into the table.

Full Update

There are no major differences between the script for a full update, and the script we just looked at:

```
$strDel = "delete from Table";
$r = mysql_db_query("Database",↩
$strDel, $cL);
```

We need to ensure that the table we want to copy has been reset on the target machine. This means we have an empty table, and can use an *insert into* statement to write the new data. We can use *delete from* to empty the table. It is very important to delete the right table from the right database before running the script. The script then selects all the data from the source table and insert it into the target table (see Listing 6). ■

Listing 5: Copying records

```
for( $i=0; $i<$count; $i++)
{
   print "Updating: $i ...
  <br> ";
   $row = mysql_fetch_row(
  $rRemote);
   $qsLocal = "insert into
  Table(Fields) values(Values)";
   $rLocal =
  mysql_db_query("Database",
$qsLocal, $cL);
   if( ! $rLocal)
   {
      print "query error:
  $qsLocal <br>";
      return;
   }
}
```

Listing 6: Inserting data into the target table

```
$qs = "select * from Table";
$r =
  mysql_db_query("Database",
  $qs, $cR);
$num = mysql_num_rows( $r);
for( $i=0; $i<$num; $i++)
```

INFO

[1] MySQL: *http://www.mysql.com*

[2] Mysqldump: *http://www.mysql.com/doc/en/mysqldump.html*

[3] PHP script: *http://www.linux-magazine.com/Magazine/Downloads/44/MySQL*

[4] PHP MyAdmin: *http://www.phpmyadmin.net*