1 mod_perl and Relational Databases

1.1 Description

Creating dynamic websites with mod_perl often involves using relational databases. `Apache::DBI`, which provides a database connections persistence which boosts the mod_perl performance, is explained in this chapter.

1.2 Why Relational (SQL) Databases

Nowadays millions of people surf the Internet. There are millions of Terabytes of data lying around. To manipulate the data new smart techniques and technologies were invented. One of the major inventions was the relational database, which allows us to search and modify huge stores of data very quickly. We use **SQL** (Structured Query Language) to access and manipulate the contents of these databases.

1.3 Apache::DBI - Initiate a persistent database connection

When people started to use the web, they found that they needed to write web interfaces to their databases. CGI is the most widely used technology for building such interfaces. The main limitation of a CGI script driving a database is that its database connection is not persistent - on every request the CGI script has to re-connect to the database, and when the request is completed the connection is closed.

`Apache::DBI` was written to remove this limitation. When you use it, you have a database connection which persists for the process' entire life. So when your mod_perl script needs to use a database, `Apache::DBI` provides a valid connection immediately and your script starts work right away without having to initiate a database connection first.

This is possible only with CGI running under a mod_perl enabled server, since in this model the child process does not quit when the request has been served.

It's almost as straightforward as is it sounds; there are just a few things to know about and we will cover them in this section.

1.3.1 Introduction

The DBI module can make use of the `Apache::DBI` module. When it loads, the DBI module tests if the environment variable `$ENV{MOD_PERL}` is set, and if the `Apache::DBI` module has already been loaded. If so, the DBI module will forward every connect() request to the `Apache::DBI` module. `Apache::DBI` uses the ping() method to look for a database handle from a previous connect() request, and tests if this handle is still valid. If these two conditions are fulfilled it just returns the database handle.

If there is no appropriate database handle or if the ping() method fails, `Apache::DBI` establishes a new connection and stores the handle for later re-use. When the script is run again by a child that is still connected, `Apache::DBI` just checks the cache of open connections by matching the *host*, *username* and *password* parameters against it. A matching connection is returned if available or a new one is initiated and then returned.

There is no need to delete the disconnect() statements from your code. They won't do anything because the `Apache::DBI` module overloads the disconnect() method with an empty one.

1.3.2 When should this module be used and when shouldn't it be used?

You will want to use this module if you are opening several database connections to the server. `Apache::DBI` will make them persistent per child, so if you have ten children and each opens two different connections (with different connect() arguments) you will have in total twenty opened and persistent connections. After the initial connect() you will save the connection time for every connect() request from your `DBI` module. This can be a huge benefit for a server with a high volume of database traffic.

You must **not** use this module if you are opening a special connection for each of your users (meaning that the login arguments are different for each user). Each connection will stay persistent and after a certain period the number of open connections will reach the allowed limit (configured by the database server) and new database connection opening requests will be refused, rendering your service unusable for some of your users.

If you want to use `Apache::DBI` but you have both situations on one machine, at the time of writing the only solution is to run two Apache/mod_perl servers, one which uses `Apache::DBI` and one which does not.

1.3.3 Configuration

After installing this module, the configuration is simple - add the following directive to `httpd.conf`

```
PerlModule Apache::DBI
```

Note that it is important to load this module before any other `Apache*DBI` module and before the `DBI` module itself!

You can skip preloading `DBI`, since `Apache::DBI` does that. But there is no harm in leaving it in, as long as it is loaded after `Apache::DBI`.

1.3.4 Preopening DBI connections

If you want to make sure that a connection will already be opened when your script is first executed after a server restart, then you should use the `connect_on_init()` method in the startup file to preload every connection you are going to use. For example:

```
Apache::DBI->connect_on_init
("DBI:mysql:myDB:myserver",
 "username",
 "passwd",
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);
```

As noted above, use this method only if you want all of apache to be able to connect to the database server as one user (or as a very few users), i.e. if your user(s) can effectively share the connection. Do **not** use this method if you want for example one unique connection per user.

Be warned though, that if you call `connect_on_init()` and your database is down, Apache children will be delayed at server startup, trying to connect. They won't begin serving requests until either they are connected, or the connection attempt fails. Depending on your DBD driver, this can take several minutes!

1.3.5 Debugging Apache::DBI

If you are not sure if this module is working as advertised, you should enable Debug mode in the startup script by:

```
$Apache::DBI::DEBUG = 1;
```

Starting with `ApacheDBI-0.84`, setting `$Apache::DBI::DEBUG = 1` will produce only minimal output. For a full trace you should set `$Apache::DBI::DEBUG = 2`.

After setting the DEBUG level you will see entries in the `error_log` both when `Apache::DBI` initializes a connection and when it returns one from its cache. Use the following command to view the log in real time (your `error_log` might be located at a different path, it is set in the Apache configuration files):

```
tail -f /usr/local/apache/logs/error_log
```

I use `alias` (in `tcsh`) so I do not have to remember the path:

```
alias err "tail -f /usr/local/apache/logs/error_log"
```

1.3.6 Database Locking Risks

Be very careful when locking the database (`LOCK TABLE ...`) or singular rows if you use `Apache::DBI` or similar persistent connections. MySQL threads keep tables locked until the thread ends (connection is closed) or the tables are unlocked. If your session die()'s while tables are locked, they will stay neatly locked as your connection won't be closed either.

See the section Handling the 'User pressed Stop button' case for more information on prevention.

1.3.7 Troubleshooting

1.3.7.1 The Morning Bug

The SQL server keeps a connection to the client open for a limited period of time. In the early days of `Apache::DBI` developers were bitten by so called *Morning bug*, when every morning the first users to use the site received a `No Data Returned` message, but after that everything worked fine.

The error was caused by `Apache::DBI` returning a handle of the invalid connection (the server closed it because of a timeout), and the script was dying on that error. The `ping()` method was introduced to solve this problem, but it didn't worked properly till `Apache::DBI` version 0.82 was released. In that version and afterwards ping() was called inside the `eval` block, which resolved the problem.

It's possible that some `DBD::` drivers don't have the ping() method implemented. The `Apache::DBI` manpage explains how to write one.

Another solution was found - to increase the timeout parameter when starting the database server. Currently we startup `MySQL` server with a script `safe_mysql`, so we have modified it to use this option:

```
nohup $ledir/mysqld [snipped other options] -O wait_timeout=172800
```

172800 seconds is equal to 48 hours. This change solves the problem, but the ping() method works properly in `DBD::mysql` as well.

1.3.7.2 Opening Connections With Different Parameters

When `Apache::DBI` receives a connection request, before it decides to use an existing cached connection it insists that the new connection be opened in exactly the same way as the cached connection. If you have one script that sets `AutoCommit` and one that does not, `Apache::DBI` will make two different connections. So if for example you have limited Apache to 40 servers at most, instead of having a maximum of 40 open connections you may end up with 80.

So these two connect() calls will create two different connections:

```
my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 1, # commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";

my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 0, # don't commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";
```

Notice that the only difference is in the value of `AutoCommit`.

However, you are free to modify the handle immediately after you get it from the cache. So always initiate connections using the same parameters and set `AutoCommit` (or whatever) afterwards. Let's rewrite the second connect call to do the right thing (not to create a new connection):

```
my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 1, # commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";
$dbh->{AutoCommit} = 0; # don't commit if not asked to
```

When you aren't sure whether you're doing the right thing, turn debug mode on.

However, when the `$dbh` attribute is altered after connect() it affects all other handlers retrieving this database handle. Therefore it's best to restore the modified attributes to their original value at the end of database handle usage. As of `Apache::DBI` version 0.88 the caller has to do it manually. The simplest way to handle this is to localize the attributes when modifying them:

```
my $dbh = DBI->connect(...) ...
{
  local $dbh->{LongReadLen} = 40;
}
```

Here the `LongReadLen` attribute overrides the value set in the connect() call or its default value only within the enclosing block.

The problem with this approach is that prior to Perl version 5.8.0 this causes memory leaks. So the only clean alternative for older Perl versions is to manually restore the `dbh`'s values:

```
my @attrs = qw(LongReadLen PrintError);
my %orig = ();

my $dbh = DBI->connect(...) ...
# store the values away
$orig{$_} = $dbh->{$_} for @attrs;
# do local modifications
$dbh->{LongReadLen} = 40;
$dbh->{PrintError}  = 1;
# do something with the filehandle
# ...
# now restore the values
$dbh->{$_} = $orig{$_} for @attrs;
```

Another thing to remember is that with some database servers it's possible to access more than one database using the same database connection. MySQL is one of those servers. It allows you to use a fully qualified table specification notation. So if there is a database *foo* with a table *test* and database *bar* with its own table *test*, you can always use:

```
SELECT from foo.test ...
```

or:

```
SELECT from bar.test ...
```

So no matter what database you have used in the database name string in the connect() call (e.g.: `DBI:mysql:foo:localhost`) you can still access both tables by using a fully qualified syntax.

Alternatively you can switch databases with `USE foo` and `USE bar`, but this approach seems less convenient, and therefore error-prone.

1.3.7.3 Cannot find the DBI handler

You must use `DBI::connect()` as in normal DBI usage to get your $dbh database handler. Using the `Apache::DBI` does not eliminate the need to write proper `DBI` code. As the `Apache::DBI` man page states, you should program as if you are not using `Apache::DBI` at all. `Apache::DBI` will override the DBI methods where necessary and return your cached connection. Any `disconnect()` call will be just ignored.

1.3.7.4 Apache:DBI does not work

Make sure you have it installed.

Make sure you configured mod_perl with either:

```
PERL_CHILD_INIT=1 PERL_STACKED_HANDLERS=1
```

or

```
EVERYTHING=1
```

Use the example script eg/startup.pl (in the mod_perl distribution). Remove the comment from the line.

```
# use Apache::DebugDBI;
```

and adapt the connect string. Do not change anything in your scripts for use with `Apache::DBI`.

1.3.7.5 Skipping connection cache during server startup

Does your error_log look like this?

```
10169 Apache::DBI PerlChildInitHandler
10169 Apache::DBI skipping connection cache during server startup
Database handle destroyed without explicit disconnect at
/usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 29.
```

If so you are trying to open a database connection in the parent httpd process. If you do, children will each get a copy of this handle, causing clashes when the handle is used by two processes at the same time. Each child must have its own, unique, connection handle.

To avoid this problem, `Apache::DBI` checks whether it is called during server startup. If so the module skips the connection cache and returns immediately without a database handle.

You must use the `Apache::DBI->connect_on_init()` method in the startup file.

1.3.7.6 Debugging code which deploys DBI

To log a trace of `DBI` statement execution, you must set the `DBI_TRACE` environment variable. The `PerlSetEnv DBI_TRACE` directive must appear before you load `Apache::DBI` and `DBI`.

For example if you use `Apache::DBI`, modify your `httpd.conf` with:

```
PerlSetEnv DBI_TRACE "3=/tmp/dbitrace.log"
PerlModule Apache::DBI
```

Replace `3` with the TRACE level you want. The traces from each request will be appended to `/tmp/dbitrace.log`. Note that the logs might interleave if requests are processed concurrently.

Within your code you can control trace generation with the trace() method:

```
DBI->trace($trace_level)
DBI->trace($trace_level, $trace_filename)
```

DBI trace information can be enabled for all handles using this DBI class method. To enable trace information for a specific handle use the similar `$h->trace` method.

Using the handle trace option with a `$dbh` or `$sth` is handy for limiting the trace info to the specific bit of code that you are interested in.

Trace Levels:

- **0 - trace disabled.**
- **1 - trace DBI method calls returning with results.**
- **2 - trace method entry with parameters and exit with results.**
- **3 - as above, adding some high-level information from the driver and also adding some internal information from the DBI.**
- **4 - as above, adding more detailed information from the driver and also including DBI mutex information when using threaded perl.**
- **5 and above - as above but with more and more obscure information.**

1.4 mysql_use_result vs. mysql_store_result.

Since many mod_perl developers use mysql as their preferred SQL engine, these notes explain the difference between `mysql_use_result()` and `mysql_store_result()`. The two influence the speed and size of the processes.

The `DBD::mysql` (version 2.0217) documentation includes the following snippet:

```
mysql_use_result attribute: This forces the driver to use
mysql_use_result rather than mysql_store_result. The former is
faster and less memory consuming, but tends to block other
processes. (That's why mysql_store_result is the default.)
```

Think about it in client/server terms. When you ask the server to spoon-feed you the data as you use it, the server process must buffer the data, tie up that thread, and possibly keep any database locks open for a long time. So if you read a row of data and ponder it for a while, the tables you have locked are still locked, and the server is busy talking to you every so often. That is `mysql_use_result()`.

If you just suck down the whole dataset to the client, then the server is free to go about its business serving other requests. This results in parallelism since the server and client are doing work at the same time, rather than blocking on each other doing frequent I/O. That is `mysql_store_result()`.

As the mysql manual suggests: you should not use `mysql_use_result()` if you are doing a lot of processing for each row on the client side. This can tie up the server and prevent other threads from updating the tables.

1.5 Transactions Not Committed with MySQL InnoDB Tables

Sometimes, when using MySQL's InnoDB table type, you may notice that changes you committed in one process don't seem to be visible to other processes. You may not be aware that InnoDB tables use a default approach to transactions that is actually more cautious than PostgreSQL or Oracle's default. It's called "repeatable read", and the gist of it is that you don't see updates made in other processes since your last commit. There is an explanation of this here: http://dev.mysql.com/doc/mysql/en/InnoDB_Consistent_read_example.html

This is actually not directly related to mod_perl, but you wouldn't notice this issue when using CGI because reconecting to the database on each request resets things just as doing a commit does. It is the persistent connections used with mod_perl that make this issue visible.

If you suspect this is causing you problems, the simplest way to deal with it is to change the isolation level to "read committed" -- which is more like what PostgreSQL and Oracle do by default -- with the "set transaction" command, described here: http://dev.mysql.com/doc/mysql/en/InnoDB_transaction_isolation.html

1.6 Optimize: Run Two SQL Engine Servers

Sometimes you end up running many databases on the same machine. These might have very varying database needs (such as one db with sessions, very frequently updated but tiny amounts of data, and another with large sets of data that's hardly ever updated) you might be able to gain a lot by running two differently configured databases on one server. One would benefit from lots of caching, the other would probably reduce the efficiency of the cache but would gain from fast disk access. Different usage profiles require vastly different performance needs.

This is basically a similar idea to having two Apache servers, each optimized for its specific requirements.

1.7 Some useful code snippets to be used with relational Databases

In this section you will find scripts, modules and code snippets to help you get started using relational Databases with mod_perl scripts. Note that I work with `mysql` (http://www.mysql.com), so the code you find here will work out of box with mysql. If you use some other SQL engine, it might work for you or it might need some changes. YMMV.

1.7.1 Turning SQL query writing into a short and simple task

Having to write many queries in my CGI scripts, persuaded me to write a stand alone module that saves me a lot of time in coding and debugging my code. It also makes my scripts much smaller and easier to read. I will present the module here, with examples following:

Notice the `DESTROY` block at the end of the module, which makes various cleanups and allows this module to be used under mod_perl and `mod_cgi` as well. Note that you will not get the benefit of persistent database handles with mod_cgi.

1.7.2 The My::DB module

The *code/My-DB.pm*:

```
package My::DB;

use strict;
use 5.004;

use DBI;

use vars qw(%c);
use constant DEBUG => 0;

%c =
  (
   db => {
          DB_NAME      => 'foo',
          SERVER       => 'localhost',
          USER         => 'put_username_here',
          USER_PASSWD  => 'put_passwd_here',
         },

  );

use Carp qw(croak verbose);
#local $SIG{__WARN__} = \&Carp::cluck;

# untaint the path by explicit setting
local $ENV{PATH} = '/bin:/usr/bin';
```

```
#######
sub new {
  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $self  = {};

    # connect to the DB, Apache::DBI takes care of caching the connections
    # save into a dbh - Database handle object
  $self->{dbh} = DBI->connect("DBI:mysql:$c{db}{DB_NAME}::$c{db}{SERVER}",
                              $c{db}{USER},
                              $c{db}{USER_PASSWD},
                              {
                               PrintError => 1, # warn() on errors
                               RaiseError => 0, # don't die on error
                               AutoCommit => 1, # commit executes immediately
                              }
                             )
    or die "Cannot connect to database: $DBI::errstr";

    # we want to die on errors if in debug mode
  $self->{dbh}->{RaiseError} = 1 if DEBUG;

    # init the sth - Statement handle object
  $self->{sth} = '';

  bless ($self, $class);

  $self;

} # end of sub new



################################################################################
               ####################################
               ###                              ###
               ###       SQL Functions          ###
               ###                              ###
               ####################################
################################################################################

# print debug messages
sub d{
   # we want to print the trace in debug mode
  print "".join("", @_)."\n" if DEBUG;

} # end of sub d


################################################################################
# return a count of matched rows, by conditions
#
#  $count = sql_count_matched($table_name,\@conditions,\@restrictions);
#
# conditions must be an array so we can pass more than one column with
# the same name.
#
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
```

```
#                  foo    => ['<',30],
#                );
#
# The sub knows automatically to detect and quote strings
#
# Restrictions are the list of restrictions like ('order by email')
#
##########################
sub sql_count_matched{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT COUNT(*) FROM $table ";
  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= "WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();
  my ($count) = $self->{sth}->fetchrow_array;

  d("Result: $count") if DEBUG;

  $self->{sth}->finish;

  return $count;

} # end of sub sql_count_matched


#############################################################################
# return a count of matched distinct rows, by conditions
#
#  $count = sql_count_matched_distinct($table_name,\@conditions,\@restrictions);
#
# conditions must be an array so we can path more than one column with
# the same name.
#
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
```

```
#                );
#
# The sub knows automatically to detect and quote strings
#
# Restrictions are the list of restrictions like ('order by email')
#
# This a slow implementation - because it cannot use select(*), but
# brings all the records in first and then counts them. In the next
# version of mysql there will be an operator 'select (distinct *)'
# which will make things much faster, so we will just change the
# internals of this sub, without changing the code itself.
#
################################
sub sql_count_matched_distinct{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT DISTINCT * FROM $table ";
  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= "WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
#  $self->{sth} = $self->{dbh}->prepare($do_sql);
#  $self->{sth}->execute();

  my $count = @{$self->{dbh}->selectall_arrayref($do_sql)};

#  my ($count) = $self->{sth}->fetchrow_array;

  d("Result: $count") if DEBUG;

#  $self->{sth}->finish;

  return $count;

} # end of sub sql_count_matched_distinct



#########################################################################
# return a single (first) matched value or undef, by conditions and
```

```
# restrictions
#
# sql_get_matched_value($table_name,$column,\@conditions,\@restrictions);
#
# column is a name of the column
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_value{
  my $self    = shift;
  my $table   = shift || '';
  my $column  = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT $column FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  return $self->{dbh}->selectrow_array($do_sql);

} # end of sql_get_matched_value




#############################################################################
# return a single row of first matched rows, by conditions and
# restrictions. The row is being inserted into @results_row array
# (value1,value2,...)  or empty () if none matched
#
# sql_get_matched_row(\@results_row,$table_name,\@columns,\@conditions,\@restrictions);
```

```
#
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_row{
  my $self    = shift;
  my $r_row   = shift || {};
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  @{$r_row} = $self->{dbh}->selectrow_array($do_sql);

} # end of sql_get_matched_row



#############################################################################
# return a ref to hash of single matched row, by conditions
# and restrictions. return undef if nothing matched.
# (column1 => value1, column2 => value2) or empty () if non matched
#
# sql_get_hash_ref($table_name,\@columns,\@conditions,\@restrictions);
#
```

```
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_hash_ref{
  my $self    = shift;
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

  return $self->{sth}->fetchrow_hashref;

} # end of sub sql_get_hash_ref




#############################################################################
# returns a reference to an array, matched by conditions and
# restrictions, which contains one reference to array per row. If
# there are no rows to return, returns a reference to an empty array:
```

```
# [
#  [array1],
#   ......
#  [arrayN],
# ];
#
# $ref = sql_get_matched_rows_ary_ref($table_name,\@columns,\@conditions,\@restrictions);
#
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name. @conditions are being cancatenated with AND
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# results in
# WHERE foo > 15 AND foo < 30
#
#  to make an OR logic use (then ANDed )
#  @conditions =  ( column => ['comp_sign',['value1','value2']],
#                   foo    => ['=',[15,24] ],
#                   bar    => ['=',[16,21] ],
#                 );
# results in
# WHERE (foo = 15 OR foo = 24) AND (bar = 16 OR bar = 21)
#
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_rows_ary_ref{
  my $self    = shift;
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {

    if (ref $$r_conds[$i+1][1] eq 'ARRAY') {
        # multi condition for the same field/comparator to be ORed
      push @where, map {"($_)"} join " OR ",
        map { join " ",
                $r_conds->[$i],
                $r_conds->[$i+1][0],
                sql_quote(sql_escape($_));
            } @{$r_conds->[$i+1][1]};
    } else {
```

```
        # single condition for the same field/comparator
       push @where, join " ",
         $r_conds->[$i],
         $r_conds->[$i+1][0],
         sql_quote(sql_escape($r_conds->[$i+1][1]));
     }
   } # end of for(my $i=0;$i<@{$r_conds};$i=$i+2

     # Add the where clause if we have one
   $do_sql .= " WHERE ". join " AND ", @where if @where;

     # restrictions (DONT put commas!)
   $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

   d("SQL: $do_sql") if DEBUG;

     # do query
   return $self->{dbh}->selectall_arrayref($do_sql);

} # end of sub sql_get_matched_rows_ary_ref



###########################################################################
# insert a single row into a DB
#
#  sql_insert_row($table_name,\%data,$delayed);
#
# data is hash of type (column1 => value1 ,column2 => value2 , )
#
# $delayed: 1 => do delayed insert, 0 or none passed => immediate
#
# * The sub knows automatically to detect and quote strings
#
# * The insert id delayed, so the user will not wait untill the insert
# will be completed, if many select queries are running
#
############################
sub sql_insert_row{
  my $self    = shift;
  my $table   = shift || '';
  my $r_data = shift || {};
  my $delayed = (shift) ? 'DELAYED' : '';

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "INSERT $delayed INTO $table ";
  $do_sql   .= "(".join(",",keys %{$r_data}).")";
  $do_sql   .= " VALUES (";
  $do_sql   .= join ",", sql_quote(sql_escape( values %{$r_data} ) );
  $do_sql   .= ")";

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
```

```
  $self->{sth}->execute();

} # end of sub sql_insert_row


#########################################################################
# update rows in a DB by condition
#
#  sql_update_rows($table_name,\%data,\@conditions,$delayed);
#
# data is hash of type (column1 => value1 ,column2 => value2 , )
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
#
# $delayed: 1 => do delayed insert, 0 or none passed => immediate
#
# * The sub knows automatically to detect and quote strings
#
#
############################
sub sql_update_rows{
  my $self    = shift;
  my $table   = shift || '';
  my $r_data = shift || {};
  my $r_conds = shift || [];
  my $delayed = (shift) ? 'LOW_PRIORITY' : '';

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "UPDATE $delayed $table SET ";
  $do_sql   .= join ",",
    map { "$_=".join "",sql_quote(sql_escape($$r_data{$_})) } keys %{$r_data};

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;


  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);

  $self->{sth}->execute();

#  my ($count) = $self->{sth}->fetchrow_array;
```

```
#
#  d("Result: $count") if DEBUG;

} # end of sub sql_update_rows


#############################################################################
# delete rows from DB by condition
#
# sql_delete_rows($table_name,\@conditions);
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
#
# * The sub knows automatically to detect and quote strings
#
#
############################
sub sql_delete_rows{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "DELETE FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }

    # Must be very careful with deletes, imagine somehow @where is
    # not getting set, "DELETE FROM NAME" deletes the contents of the table
  warn("Attempt to delete a whole table $table from DB\n!!!"),return unless @where;

    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where;

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

} # end of sub sql_delete_rows


#############################################################################
# executes the passed query and returns a reference to an array which
```

```
# contains one reference per row. If there are no rows to return,
# returns a reference to an empty array.
#
# $r_array = sql_execute_and_get_r_array($query);
#
#
##########################
sub sql_execute_and_get_r_array{
  my $self     = shift;
  my $do_sql   = shift || '';

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  d("SQL: $do_sql") if DEBUG;

  $self->{dbh}->selectall_arrayref($do_sql);

} # end of sub sql_execute_and_get_r_array


#############################################################################
# lock the passed tables in the requested mode (READ|WRITE) and set
# internal flag to handle possible user abortions, so the tables will
# be unlocked thru the END{} block
#
# sql_lock_tables('table1','lockmode',..,'tableN','lockmode'
# lockmode = (READ | WRITE)
#
# _side_effect_ $self->{lock} = 'On';
#
##########################
sub sql_lock_tables{
  my $self   = shift;
  my %modes = @_;

  return unless %modes;

  my $do_sql = 'LOCK TABLES ';
  $do_sql .= join ",", map {"$_ $modes{$_}"} keys %modes;

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  d("SQL: $do_sql") if DEBUG;

  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

    # Enough to set only one lock, unlock will remove them all
  $self->{lock} = 'On';

} # end of sub sql_lock_tables



#############################################################################
# unlock all tables, unset internal flag to handle possible user
# abortions, so the tables will be unlocked thru the END{} block
```

```
#
# sql_unlock_tables()
#
# _side_effect_: delete $self->{lock}
#
##########################
sub sql_unlock_tables{
  my $self   = shift;

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  $self->{dbh}->do("UNLOCK TABLES");

    # Enough to set only one lock, unlock will remove them all
  delete $self->{lock};

} # end of sub sql_unlock_tables

#
#
# return current date formatted for a DATE field type
# YYYYMMDD
#
# Note: since this function actually doesn't need an object it's being
# called without parameter as well as procedural call
#############
sub sql_date{
  my $self     = shift;

  my ($mday,$mon,$year) = (localtime)[3..5];
  return sprintf "%0.4d%0.2d%0.2d",1900+$year,++$mon,$mday;

} # end of sub sql_date

#
#
# return current date formatted for a DATE field type
# YYYYMMDDHHMMSS
#
# Note: since this function actually doesn't need an object it's being
# called without parameter as well as procedural call
#############
sub sql_datetime{
  my $self     = shift;

  my ($sec,$min,$hour,$mday,$mon,$year) = localtime();
  return sprintf "%0.4d%0.2d%0.2d%0.2d%0.2d%0.2d",1900+$year,++$mon,$mday,$hour,$min,$sec;

} # end of sub sql_datetime


# Quote the list of parameters.  Parameters consisting entirely of
# digits (i.e. integers) are unquoted.
# print sql_quote("one",2,"three"); => 'one', 2, 'three'
##############
sub sql_quote{ map{ /^(\d+|NULL)$/ ? $_ : "\'$_\'" } @_ }

# Escape the list of parameters (all unsafe chars like ",' are escaped)
```

```
# We make a copy of @_ since we might try to change the passed values,
# producing an error when modification of a read-only value is attempted
###############
sub sql_escape{ my @a = @_; map { s/([\'\\])/\\$1/g;$_} @a }


# DESTROY makes all kinds of cleanups if the fuctions were interuppted
# before their completion and haven't had a chance to make a clean up.
############
sub DESTROY{
  my $self = shift;

  $self->sql_unlock_tables() if $self->{lock};
  $self->{sth}->finish       if $self->{sth};
  $self->{dbh}->disconnect   if $self->{dbh};

} # end of sub DESTROY

# Don't remove
1;
```

module

(Note that you will not find this on CPAN. at least not yet :)

1.7.3 My::DB Module's Usage Examples

To use `My::DB` in your script, you first have to create a `My::DB` object:

```
  use vars qw($db_obj);
  my $db_obj = new My::DB or croak "Can't initialize My::DB object: $!\n";
```

Now you can use any of `My::DB`'s methods. Assume that we have a table called *tracker* where we store the names of the users and what they are doing at each and every moment (think about an online community program).

I will start with a very simple query--I want to know where the users are and produce statistics. `tracker` is the name of the table.

```
    # fetch the statistics of where users are
  my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
    ("tracker",
     [qw(where_user_are)],
    );

  my %stats = ();
  my $total = 0;
  foreach my $r_row (@$r_ary){
    $stats{$r_row->[0]}++;
    $total++;
  }
```

Now let's count how many users we have (in table `users`):

```
my $count = $db_obj->sql_count_matched("users");
```

Check whether a user exists:

```
my $username = 'stas';
my $exists = $db_obj->sql_count_matched
("users",
 [username => ["=",$username]]
);
```

Check whether a user is online, and get the time since she went online (`since` is a column in the `tracker` table, it tells us when a user went online):

```
my @row = ();
$db_obj->sql_get_matched_row
(\@row,
 "tracker",
 ['UNIX_TIMESTAMP(since)'],
 [username => ["=",$username]]
);

if (@row) {
  my $idle = int( (time() - $row[0]) / 60);
  return "Current status: Is Online and idle for $idle minutes.";
}
```

A complex query. I join two tables, and I want a reference to an array which will store a slice of the matched query (`LIMIT $offset,$hits`) sorted by `username`. Each row in the array is to include the fields from the `users` table, but only those listed in `@verbose_cols`. Then we print it out.

```
my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
  (
   "tracker STRAIGHT_JOIN users",
   [map {"users.$_"} @verbose_cols],
   [],
   ["WHERE tracker.username=users.username",
    "ORDER BY users.username",
    "LIMIT $offset,$hits"],
  );

foreach my $r_row (@$r_ary){
  print ...
}
```

Another complex query. The user checks checkboxes to be queried by, selects from lists and types in match strings, we process input and build the `@where` array. Then we want to get the number of matches and the matched rows as well.

```
my @search_keys = qw(choice1 choice2);
my @where = ();
  # Process the checkboxes - we turn them into a regular expression
foreach (@search_keys) {
  next unless defined $q->param($_) and $q->param($_);
```

```
  my $regexp = "[".join("",$q->param($_))."]";
  push @where, ($_ => ['REGEXP',$regexp]);
}

  # Add the items selected by the user from our lists
  # selected => exact match
push @where,(country => ['=',$q->param('country')]) if $q->param('country');

  # Add the parameters typed by the user
foreach (qw(city state)) {
  push @where,($_ => ['LIKE',$q->param($_)]) if $q->param($_);
}

   # Count all that matched the query
my $total_matched_users =  $db_obj->sql_count_matched
  (
   "users",
   \@where,
  );

  # Now process the orderby
my $orderby = $q->param('orderby') || 'username';

   # Do the query and fetch the data
my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
(
 "users",
 \@display_columns,
 \@where,
 ["ORDER BY $orderby",
  "LIMIT $offset,$hits"],
);
```

`sql_get_matched_rows_ary_ref` knows to handle both `OR`ed and `AND`ed params. This example shows how to use `OR` on parameters:

This snippet is an implementation of a watchdog. Our users want to know when their colleagues go online. They register the usernames of the people they want to know about. We have to make two queries: one to get a list of usernames, the second to find out whether any of these users is online. In the second query we use the `OR` keyword.

```
# check who we are looking for
$r_ary = $db_obj->sql_get_matched_rows_ary_ref
  ("watchdog",
   [qw(watched)],
   [username => ['=',$username)],
   ],
  );

  # put them into an array
my @watched = map {$_->[0]} @{$r_ary};

my %matched = ();
  # Does the user have some registered usernames?
if (@watched) {
```

```
  # Try to fetch all the users who match the usernames exactly.
  # Put it into an array and compare it with a hash!
    $r_ary = $db_obj->sql_get_matched_rows_ary_ref
      ("tracker",
       [qw(username)],
       [username => ['=',\@watched],
       ]
      );

    map {$matched{$_->[0]} = 1} @{$r_ary};
  }

  # Now %matched includes the usernames of the users who are being
  # watched by $username and currently are online.
```

1.8 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

1.9 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: