CHAPTER 20

# Relational Databases and mod_perl

Nowadays, millions of people surf the Internet. There are millions of terabytes of data lying around, and many new techniques and technologies have been invented to manipulate this data. One of these inventions is the relational database, which makes it possible to search and modify huge stores of data very quickly. The Structured Query Language (SQL) is used to access and manipulate the contents of these databases.

Let's say that you started your web services with a simple, flat-file database. Then with time your data grew big, which made the use of a flat-file database slow and inefficient. So you switched to the next simple solution—using DBM files. But your data set continued to grow, and even the DBM files didn't provide a scalable enough solution. So you finally decided to switch to the most advanced solution, a relational database.

On the other hand, it's quite possible that you had big ambitions in the first place and you decided to go with a relational database right away.

We went through both scenarios, sometimes doing the minimum development using DBM files (when we knew that the data set was small and unlikely to grow big in the short term) and sometimes developing full-blown systems with relational databases at the heart.

As we repeat many times in this book, none of our suggestions and examples should be applied without thinking. But since you're reading this chapter, the chances are that you are doing the right thing, so we are going to concentrate on the extra benefits that mod_perl provides when you use relational databases. We'll also talk about related coding techniques that will help you to improve the performance of your service.

From now on, we assume that you use the `DBI` module to talk to the databases. This in turn uses the unique database driver module for your database, which resides in the `DBD::` namespace (for example, `DBD::Oracle` for Oracle and `DBD::mysql` for MySQL). If you stick to standard SQL, you maximize portability from one database to another. Changing to a new database server should simply be a matter of using a different database driver. You do this just by changing the data set name string (`$dsn`) in the `DBI->connect()` call.

Rather than writing your queries in plain SQL, you should probably use some other abstraction module on top of the DBI module. This can help to make your code more extensible and maintainable. Raw SQL coupled with DBI usually gives you the best machine performance, but sometimes time to market is what counts, so you have to make your choices. An abstraction layer with a well-thought-out API is a pleasure to work with, and future modifications to the code will be less troublesome. Several DBI abstraction solutions are available on CPAN. DBIx::Recordset, Alzabo, and Class::DBI are just a few such modules that you may want to try. Take a look at the other modules in the DBIx:: category—many of them provide some kind of wrapping and abstraction around DBI.

## Persistent Database Connections with Apache::DBI

When people first started to use the Web, they found that they needed to write web interfaces to their databases, or add databases to drive their web interfaces. Whichever way you look at it, they needed to connect to the databases in order to use them.

CGI is the most widely used protocol for building such interfaces, implemented in Apache's mod_cgi and its equivalents. For working with databases, the main limitation of most implementations, including mod_cgi, is that they don't allow persistent connections to the database. For every HTTP request, the CGI script has to connect to the database, and when the request is completed the connection is closed. Depending on the relational database that you use, the time to instantiate a connection may be very fast (for example, MySQL) or very slow (for example, Oracle). If your database provides a very short connection latency, you may get away without having persistent connections. But if not, it's possible that opening a connection may consume a significant slice of the time to serve a request. It may be that if you can cut this overhead you can greatly improve the performance of your service.

Apache::DBI was written to solve this problem. When you use it with mod_perl, you have a database connection that persists for the entire life of a mod_perl process. This is possible because with mod_perl, the child process does not quit when a request has been served. When a mod_perl script needs to use a database, Apache::DBI immediately provides a valid connection (if it was already open) and your script starts doing the real work right away without having to make a database connection first.

Of course, the persistence doesn't help with any latency problems you may encounter during the actual use of the database connections. Oracle, for example, is notorious for generating a network transaction for each row returned. This slows things down if the query execution matches many rows.

You may want to read Tim Bunce's "Advanced DBI" talk, at *http://dbi.perl.org/doc/conferences/tim_1999/index.html*, which covers many techniques to reduce latency.

## Apache::DBI Connections

The DBI module can make use of the Apache::DBI module. When the DBI module loads, it tests whether the environment variable $ENV{MOD_PERL} is set and whether the Apache::DBI module has already been loaded. If so, the DBI module forwards every connect( ) request to the Apache::DBI module.

When Apache::DBI gets a connect( ) request, it checks whether it already has a handle with the same connect( ) arguments. If it finds one, it checks that the connection is still valid using the ping( ) method. If this operation succeeds, the database handle is returned immediately. If there is no appropriate database handle, or if the ping( ) method fails, Apache::DBI establishes a new connection, stores the handle, and then returns the handle to the caller.

It is important to understand that the pool of connections is not shared between the processes. Each process has its own pool of connections.

When you start using Apache::DBI, there is no need to delete all the disconnect( ) statements from your code. They won't do anything, because the Apache::DBI module overloads the disconnect( ) method with an empty one. You shouldn't modify your scripts at all for use with Apache::DBI.

## When to Use Apache::DBI (and When Not to Use It)

You will want to use the Apache::DBI module only if you are opening just a few database connections per process. If there are ten child processes and each opens two different connections (using different connect( ) arguments), in total there will be 20 opened and persistent connections.

This module must *not* be used if (for example) you have many users, and a unique connection (with unique connect( ) arguments) is required for each user.* You cannot ensure that requests from one user will be served by any particular process, and connections are not shared between the child processes, so many child processes will open a separate, persistent connection for each user. In the worst case, if you have 100 users and 50 processes, you could end up with 5,000 persistent connections, which might be largely unused. Since database servers have limitations on the maximum number of opened connections, at some point new connections will not be permitted, and eventually your service will become unavailable.

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 mod_perl-enabled servers, one that uses Apache::DBI and one that does not.

* That is, database user connections. This doesn't mean that if many people register as users on your web site you shouldn't use Apache::DBI; it is only a very special case.

In mod_perl 2.0, a threaded server can be used, and this situation is much improved. Assuming that you have a single process with many threads and each unique open connection is needed by only a single thread, it's possible to have a pool of database connections that are reused by different threads.

## Configuring Apache::DBI

`Apache::DBI` will not work unless mod_perl was built with:

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

or:

```
EVERYTHING=1
```

during the `perl Makefile.PL ...` stage.

After installing this module, configuration is simple—just add a single 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. The best rule is just to load it first of all. You can skip preloading `DBI` at server startup, since `Apache::DBI` does that for you, but there is no harm in leaving it in, as long as `Apache::DBI` is loaded first.

## Debugging Apache::DBI

If you are not sure whether this module is working as advertised and that your connections are actually persistent, you should enable debug mode in the *startup.pl* script, like this:

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

Starting with `Apache::DBI` Version 0.84, the above setting 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* file. Here is a sample of the output with a `DEBUG` level of 1:

```
12851 Apache::DBI new connect to
'test::localhostPrintError=1RaiseError=0AutoCommit=1'

12853 Apache::DBI new connect to
'test::localhostPrintError=1RaiseError=0AutoCommit=1'
```

When a connection is reused, `Apache::DBI` stays silent, so you can see when a real `connect()` is called. If you set the `DEBUG` level to 2, you'll see a more verbose output. This output was generated after two identical requests with a single server running:

```
12885 Apache::DBI need ping: yes
12885 Apache::DBI new connect to
```

```
'test::localhostPrintError=1RaiseError=0AutoCommit=1'
12885 Apache::DBI need ping: yes
12885 Apache::DBI already connected to
'test::localhostPrintError=1RaiseError=0AutoCommit=1'
```

You can see that process 12885 created a new connection on the first request and on the next request reused it, since it was using the same `connect()` argument. Moreover, you can see that the connection was validated each time with the `ping()` method.

## Caveats and Troubleshooting

This section covers some of the risks and things to keep in mind when using `Apache::DBI`.

### Database locking risks

When you use `Apache::DBI` or similar persistent connections, be very careful about locking the database (`LOCK TABLE ...`) or single rows. MySQL threads keep tables locked until the thread ends (i.e., the connection is closed) or until the tables are explicitly unlocked. If your session dies while tables are locked, they will stay locked, as your connection to the database won't be closed. In Chapter 6 we discussed how to terminate the program cleanly if the session is aborted prematurely.

### Transactions

A standard Perl script using `DBI` will automatically perform a rollback whenever the script exits. In the case of persistent database connections, the database handle will not be destroyed and hence no automatic rollback will occur. At first glance it even seems to be possible to handle a transaction over multiple requests, but the temptation should be avoided because different requests are handled by different mod_perl processes, and a mod_perl process does not know the state of a specific transaction that has been started by another mod_perl process.

In general, it is good practice to perform an explicit commit or rollback at the end of every script. To avoid inconsistencies in the database in case `AutoCommit` is `Off` and the script terminates prematurely without an explicit rollback, the `Apache::DBI` module uses a `PerlCleanupHandler` to issue a rollback at the end of every request.

### 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, for example, if you have limited Apache to 40 servers at most, instead of having a maximum of 40 open connections, you may end up with 80.

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) afterward. Let's rewrite the second connect( ) call to do the right thing (i.e., 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 on debug mode.

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 values at the end of database handle usage. As of Apache::DBI Version 0.88, the caller has to do this 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 it causes memory leaks. So the only clean alternative for older Perl versions is to manually restore `$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 database handle
# ...

# 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 a database *bar* with its own table *test*, you can always use:

```
SELECT * FROM foo.test ...
```

or:

```
SELECT * FROM bar.test ...
```

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.

### Cannot find the DBI handler

You must use `DBI->connect()` as in normal `DBI` usage to get your `$dbh` database handle. Using `Apache::DBI` does not eliminate the need to write proper `DBI` code. As the `Apache::DBI` manpage 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()` calls will just be ignored.

### 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`, everyone was bitten by the so-called *morning bug*—

every morning the first user to use the site received a "No Data Returned" message, but after that everything worked fine.

The error was caused by `Apache::DBI` returning an invalid connection handle (the server had 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 work properly until `Apache::DBI` Version 0.82 was released. In that version and after, `ping()` was called inside an `eval` block, which resolved the problem.

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

Another solution is to increase the timeout parameter when starting the database server. We usually start the MySQL server with the script *safe_mysqld*, so we modified it to use this option:

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

The timeout value that we use is 172,800 seconds, or 48 hours. This change solves the problem, but the `ping()` method works properly in `DBD::mysql` as well.

#### Apache:DBI does not work

If `Apache::DBI` doesn't work, first make sure that you have it installed. Then make sure that you configured mod_perl with either:

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

or:

```
EVERYTHING=1
```

Turn on debug mode using the `$Apache::DBI::DEBUG` variable.

#### 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/5.6.1/Apache/DBI.pm line 29.
```

If so, you are trying to open a database connection in the parent *httpd* process. If you do, the 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 (see the next section) in the startup file to preopen a connection before the child processes are spawned.

# Improving Performance

Let's now talk about various techniques that allow you to boost the speed of applications that work with relational databases. A whole book could be devoted to this topic, so here we will concentrate on the techniques that apply specifically to mod_perl servers.

## Preopening DBI Connections

If you are using `Apache::DBI` and you want to make sure that a database connection will already be open when your code is first executed within each child process after a server restart, you should use the `connect_on_init()` method in the startup file to preopen every connection that you are going to use. For example:

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

For this method to work, you need to make sure that you have built mod_perl with `PERL_CHILD_INIT=1` or `EVERYTHING=1`.

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!

## Improving Speed by Skipping ping()

If you use `Apache::DBI` and want to save a little bit of time, you can change how often the `ping()` method is called. The following setting in a startup file:

```
Apache::DBI->setPingTimeOut($data_source, $timeout)
```

will change this behavior. If the value of `$timeout` is `0`, `Apache:DBI` will validate the database connection using the `ping()` method for every database access. This is the default. Setting `$timeout` to a negative value will deactivate the validation of the database handle. This can be used for drivers that do not implement the `ping()` method (but it's generally a bad idea, because you don't know if your database handle really works). Setting `$timeout` to a positive value will *ping* the database on access only if the previous access was more than `$timeout` seconds earlier.

`$data_source` is the same as in the `connect()` method (e.g., `DBI:mysql:...`).

## Efficient Record-Retrieval Techniques

When working with a relational database, you'll often encounter the need to read the retrieved set of records into your program, then format and print them to the browser.

Assuming that you're already connected to the database, let's consider the following code prototype:

```
my $query = "SELECT id,fname,lname FROM test WHERE id < 10";
my $sth = $dbh->prepare($query);
$sth->execute;

my @results = ();
while (my @row_ary  = $sth->fetchrow_array) {
    push @results, [ transform(@row_ary) ];
}
# print the output using the the data returned from the DB
```

In this example, the *httpd* process will grow by the size of the variables that have been allocated for the records that matched the query. Remember that to get the total amount of extra memory required by this technique, this growth should be multiplied by the number of child processes that your server runs—which is probably not a constant.

A better approach is not to accumulate the records, but rather to print them as they are fetched from the DB. You can use the methods `$sth->bind_columns()` and `$sth->fetchrow_arrayref()` (aliased to `$sth->fetch()`) to fetch the data in the fastest possible way. Example 20-1 prints an HTML table with matched data. Now the only additional memory consumed is for an `@cols` array to hold temporary row values.

*Example 20-1. bind_cols.pl*

```
my $query = "SELECT id,fname,lname FROM test WHERE id < 10";
my @fields = qw(id fname lname);

# create a list of cols values
my @cols = ();
@cols[0..$#fields] = ();
$sth = $dbh->prepare($query);
$sth->execute;

# Bind perl variables to columns.
$sth->bind_columns(undef, \(@cols));
print "<table>";
print '<tr bgcolor="grey">',
    map("<th>$_</th>", @fields), "</tr>";
while ($sth->fetch) {
    print "<tr>",
        map("<td>$_</td>", @cols), "</tr>";
}
print "</table>";
```

Note that this approach doesn't tell you how many records have been matched. The workaround is to run an identical query before the code above, using `SELECT count(*)...` instead of `SELECT * ...` to get the number of matched records:

```
my $query = "SELECT count(*) FROM test WHERE id < 10";
```

This should be much faster, since you can remove any `SORT BY` and similar attributes.

You might think that the `DBI` method `$sth->rows` will tell you how many records will be returned, but unfortunately it will not. You can rely on a row count only after a `do` (for some specific operations, such as update and delete), after a non-select `execute`, or after fetching all the rows of a `select` statement.

For `select` statements, it is generally not possible to know how many rows will be returned except by fetching them all. Some `DBD` drivers will return the number of rows the application has fetched so far, but others may return `-1` until all rows have been fetched. Thus, use of the `rows` method with `select` statements is not recommended.

## mysql_use_result Versus mysql_store_result Attributes

Many mod_perl developers use MySQL as their preferred relational database server because of its speed. Depending on the situation, it may be possible to change the way in which the `DBD::mysql` driver delivers data. The two attributes `mysql_use_result` and `mysql_store_result` influence the speed and size of the processes.

You can tell the `DBD::mysql` driver to change the default behavior before you start to fetch the results:

```
my $sth = $dbh->prepare($query);
$sth->{"mysql_use_result"} = 1;
```

This forces the driver to use `mysql_use_result` rather than `mysql_store_result`. The former is faster and uses less memory, but it tends to block other processes, which is 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 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 the situation with `mysql_use_result`.

On the other hand, if you just suck down the whole data set to the client, then the server is free to serve other requests. This improves parallelism, since rather than blocking each other by doing frequent I/O, the server and client are working at the same time. That is the situation with `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.

If you are using some other DBD driver, check its documentation to see if it provides the flexibility of DBD::mysql in this regard.

### Running Two or More Relational Databases

Sometimes you end up running many databases on the same machine. These might have very different needs. For example, one may handle user sessions (updated frequently but with tiny amounts of data), and another may contain large sets of data that are hardly ever updated. You might be able to improve performance by running two differently tuned database servers on one machine. The frequently updated database can gain a lot from fast disk access, whereas the database with mostly static data could benefit from lots of caching.

### Caching prepare( ) Statements

You can also benefit from persistent connections by replacing prepare( ) with prepare_cached( ). That way you will always be sure that you have a good statement handle and you will get some caching benefit. The downside is that you are going to pay for DBI to parse your SQL and do a cache lookup every time you call prepare_cached( ). This will give a big performance boost to database servers that execute prepare( ) quite slowly (e.g., Oracle), but it might add an unnecessary overhead with servers such as MySQL that do this operation very quickly.

Be warned that some databases (e.g., PostgreSQL and Sybase) don't support caches of prepared plans. With Sybase you could open multiple connections to achieve the same result, but this is at the risk of getting deadlocks, depending on what you are trying to do!

Another pitfall to watch out for lies in the fact that prepare_cached( ) actually gives you a reference to the *same* cached statement handle, not just a similar copy. So you can't do this:

```
my $sth1 = $dbh->prepare_cached('SELECT name FROM table WHERE id=?');
my $sth2 = $dbh->prepare_cached('SELECT name FROM table WHERE id=?');
```

because $sth1 and $sth2 are now the same object! If you try to use them independently, your code will fail.

Make sure to read the DBI manpage for the complete documentation of this method and the latest updates.

## DBI Debug Techniques

Sometimes the code that talks to the database server doesn't seem to work. It's important to know how to debug this code at the DBI level. Here is how this debugging can be accomplished.

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* file 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 will probably be interleaved 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 $dbh->trace method.

Using the trace option with a $dbh or $sth handle is useful to limit the trace information to the specific bit of code that you are debugging.

The trace levels are:

*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+* As above, but with more and more obscure information

# References


- "Introduction to Structured Query Language": *http://web.archive.org/web/20011116021648/http://w3.one.net/~jhoffman/sqltut.htm*
- "SQL for Web Nerds," by Philip Greenspun: *http://philip.greenspun.com/sql/*
- DBI-related information: *http://dbi.perl.org/*
- *Programming the Perl DBI*, by Alligator Descartes and Tim Bunce (O'Reilly)
- "DBI Examples and Performance Tuning," by Jeffrey Baker: *http://www.saturn5.com/~jwb/dbi-examples.html*
- *SQL Fundamentals*, by John J Patrick (Prentice Hall)
- *SQL in a Nutshell*, by Kevin Kline with Daniel Kline (O'Reilly)