

Application Performance using DBI and mod_perl

Contents

- [Analysis of the Problem](#)
- [Stage 1: Optimizing Database Connections](#)
- [Stage 2: Utilizing the Database Server's Cache](#)
- [Stage 3: Eliminating SQL Statement Parsing](#)
- [Conclusion](#)
- Tables
 - [Table 1: Expense Analysis of Initial Implementation](#)
 - [Table 2: Expense Analysis after Stage 1 Optimization](#)
 - [Table 3: Expense Analysis after Stage 2 Optimization](#)
 - [Table 4: Expense Analysis after Stage 3 Optimization](#)

Analysis of the Problem

A common web application architecture is one or more application servers which handle requests from client browsers by consulting one or more database servers and performing a transform on the data. When an application must consult the database on every request, the interaction with the database server becomes the central performance issue. Spending a bit of time optimizing your database access can result in significant application performance improvements. In this analysis, a system using Apache, mod_perl, DBI, and Oracle will be considered. The application server uses Apache and mod_perl to service client requests, and DBI to communicate with a remote Oracle database.

In the course of servicing a typical client request, the application server must retrieve some data from the database and execute a stored procedure. There are several steps that need to be done by the application and the database to complete a request:

1. Connect to the database server
2. Prepare a SQL SELECT statement
3. Execute the SELECT statement
4. Retrieve the results of the SELECT statement
5. Release the SELECT statement handle
6. Prepare a PL/SQL stored procedure call
7. Execute the stored procedure
8. Release the stored procedure statement handle
9. Commit or rollback
10. Disconnect from the database server

In this document, an application will be described which achieves maximum performance by eliminating some of the steps above and optimizing others. The expense of each each implementation will be analyzed after each stage of optimization.

Stage 1: Optimizing Database Connections

A naive implementation would perform steps 1 through 10 from above on every request. A portion of the source code might look like this:

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
    || die $DBI::errstr;

my $baz = $r->param('baz');

eval {
    my $sth = $dbh->prepare(qq{
        SELECT foo
        FROM bar
        WHERE baz = $baz
    });
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    $sth->finish;

    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => $baz
            );
        END;
    });
    $sph->execute;
    $sph->finish;
```

```

        $dbh->commit;
    };
    if ($@) {
        $dbh->rollback;
    }

    $dbh->disconnect;

    # ...

```

In practice, such an implementation would have hideous performance problems. The majority of the execution time of this program would likely be spent connecting to the database. Table 1 shows an analysis of what is happening during each request.

Table 1: Expense Analysis of Initial Implementation

Step/Actor	Application	Database	Relative Expense
1	Create client-side database connection context		high
2	Open socket to database server		high
3		Create server-side database connection context	high
4	Parse SQL SELECT statement		low
5	Create client-side SQL statement context		low
6		Lookup (and not find) SQL statement in cache	low
7		Create execution plan for SQL statement	moderate
8		Execute the SQL statement	low
9	Fetch result rows		low
10	Destroy client-side SQL statement context		low
11		Destroy server-side SQL statement context	low
12	Parse PL/SQL statement		low

13	Create client-side PL/SQL statement context		low
14		Lookup (and not find) PL/SQL statement in cache	low
15		Create execution plan for PL/SQL statement	moderate
16		Execute the PL/SQL statement	low
17	Destroy client-side PL/SQL statement context		low
18		Destroy server-side PL/SQL statement context	low
19		Commit or rollback	moderate
20	Destory client-side connection context		high
21		Destory server-side connection context	high

The naive implementation waits for all of these steps to happen, and then throws away the database connection when it is done! This is obviously wasteful, as the database connection and disconnection steps are the most expensive. The best solution is to hoist the expensive database connection step out of the per-request lifecycle so the cost of connecting to the database is amortized over many requests. This can be done by connecting to the database server once, and then not disconnecting until the Apache child process exits. The Apache::DBI module does this transparently and automatically with little effort on the part of the programmer.

Apache::DBI intercepts calls to DBI's `connect` and `disconnect` methods and replaces them with its own. Apache::DBI caches database connections when they are first opened, and it ignores disconnect commands. When an application tries to connect to the same database, Apache::DBI returns a cached connection, thus saving the significant time penalty of repeatedly connecting to the database. A full treatment of Apache::DBI doesn't belong in this document, but you can find more information in Stas Bekman's `mod_perl` guide at <http://perl.apache.org/guide/>.

Table 2: Expense Analysis after Stage 1 Optimization

Step/Actor	Application	Database	Relative Expense
1	Successfully retrieve connection from Apache::DBI's cache		low
2	Parse SQL SELECT statement		low

3	Create client-side SQL statement context		low
4		Lookup (and not find) SQL statement in cache	low
5		Create execution plan for SQL statement	moderate
6		Execute the SQL statement	low
7	Fetch result rows		low
8	Destroy client-side SQL statement context		low
9		Destroy server-side SQL statement context	low
10	Parse PL/SQL statement		low
11	Create client-side PL/SQL statement context		low
12		Lookup (and not find) PL/SQL statement in cache	low
13		Create execution plan for PL/SQL statement	moderate
14		Execute the PL/SQL statement	low
15	Destroy client-side PL/SQL statement context		low
16		Destroy server-side PL/SQL statement context	low
17		Commit or rollback	moderate

Table 2 shows the actions performed during the request after the first optimization. All of the most expensive steps have been removed from the per-request cycle. Happily, When Apache::DBI is in use, none of the code in the example needs to change. The code is upgraded from naive to respectable with the use of a simple module! The first and biggest database performance problem is quickly dispensed with.

Stage 2: Utilizing the Database Server's Cache

Most database servers, including Oracle, utilize a cache to improve the performance of recently seen queries. The cache is keyed on the SQL statement. If a statement is identical to a previously seen statement, the execution plan for the previous statement is reused. This can be a considerable

improvement over building a new statement execution plan.

Our respectable implementation from the last section is not making use of this caching ability. It is preparing the statement `SELECT foo FROM bar WHERE baz = $baz`. The problem is that `$baz` is being read from an HTML form, and is therefore likely to change on every request. When the database server sees this statement, it is going to look like `SELECT foo FROM bar WHERE baz = 1`, and on the next request, the SQL will be `SELECT foo FROM bar WHERE baz = 42`. Since the statements are different, the database server will not be able to reuse its execution plan, and will proceed to make another one. This defeats the purpose of the SQL statement cache.

The application server needs to make sure that SQL statements which are the same look the same. The way to achieve this is to use placeholders and bound parameters. The placeholder is a blank in the SQL statement, which tells the database server that the value will be filled in later. The bound parameter is the value which is inserted into the blank before the statement is executed.

With placeholders, the SQL statement looks like `SELECT foo FROM bar WHERE baz = :baz`. Regardless of whether `baz` is 1 or 42, the SQL always looks the same, and the database server can reuse its cached execution plan for this statement. This technique has eliminated the execution plan generation penalty from the per-request runtime. The potential performance improvement from this optimization could range from modest to very significant.

Here is the updated code fragment which employs this optimization, and Table 3 shows effect of this optimization on performance.

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
    || die $DBI::errstr;

my $baz = $r->param('baz');

eval {
    my $sth = $dbh->prepare(qq{
        SELECT foo
        FROM bar
        WHERE baz = :baz
    });
    $sth->bind_param(':baz', $baz);
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
```

```

        # do HTML stuff
    }

    $sth->finish;

    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => :baz
            );
        END;
    });
    $sph->bind_param(':baz', $baz);
    $sph->execute;
    $sph->finish;

    $dbh->commit;
};
if ($@) {
    $dbh->rollback;
}

# ...

```

Table 3: Expense Analysis after Stage 2 Optimization

Step/Actor	Application	Database	Relative Expense
1	Successfully retrieve connection from Apache::DBI's cache		low
2	Parse SQL SELECT statement		low
3	Create client-side SQL statement context		low
4		Successfully lookup SQL statement in cache	low
5		Bind SQL parameters	low
6		Execute the SQL statement	low
7	Fetch result rows		low

8	Destroy client-side SQL statement context		low
9		Destroy server-side SQL statement context	low
10	Parse PL/SQL statement		low
11	Create client-side PL/SQL statement context		low
12		Successfully lookup PL/SQL statement in cache	low
13		Bind PL/SQL parameters	low
14		Execute the PL/SQL statement	low
15	Destroy client-side PL/SQL statement context		low
16		Destroy server-side PL/SQL statement context	low
17		Commit or rollback	moderate

Stage 3: Eliminating SQL Statement Parsing

The example program has certainly come a long way and the performance is now probably much better than that of the first revision. However, there is still more speed that can be wrung out of this server architecture. The last bottleneck is in SQL statement parsing. Every time DBI's `prepare` method is called, DBI parses the SQL command looking for placeholder strings, and does some housekeeping work. Worse, a context has to be built on the client and server sides of the connection which the database will use to refer to the statement. These things take time, and by eliminating these steps the time can be saved.

To get rid of the statement handle construction and statement parsing penalties, we could use DBI's `prepare_cached` method. This method compares the SQL statement to others that have already been executed. If there is a match, the cached statement handle is returned. But the application server is still spending time calling an object method (very expensive in Perl), and doing a hash lookup. Both of these steps are unnecessary, since the SQL is very likely to be static and known at compile time. The smart programmer can take advantage of these two attributes to gain better database performance. In this example, the database statements will be prepared immediately after the connection to the database is made, and they will be cached in package scalars to eliminate the method call.

What is needed is a routine that will connect to the database and prepare the statements. Since the statements are dependent upon the connection, the integrity of the connection needs to be checked

before using the statements, and a reconnection should be attempted if needed. Since the routine presented here does everything that Apache::DBI does, it does not use Apache::DBI and therefore has the added benefit of eliminating a cache lookup on the connection.

Here is an example of such a package:

```
package My::DB;

use strict;
use DBI;

sub connect {
    if (defined $My::DB::conn) {
        eval {
            $My::DB::conn->ping;
        };
        if (!$@) {
            return $My::DB::conn;
        }
    }

    $My::DB::conn = DBI->connect(
        'dbi:Oracle:server', 'user', 'pass', {
            PrintError => 1,
            RaiseError => 1,
            AutoCommit => 0
        }
    ) || die $DBI::errstr; #Assume application handles this

    $My::DB::select = $My::DB::conn->prepare(q{
        SELECT foo
        FROM bar
        WHERE baz = :baz
    });

    $My::DB::procedure = $My::DB::conn->prepare(q{
        BEGIN
            my_procedure(
                arg_in => :baz
            );
        END;
    });
}
```

```

    });

    return $My::DB::conn;
}

1;

```

Now the example program needs to be modified to use this package.

```

# ...
my $dbh = My::DB->connect;

my $baz = $r->param('baz');

eval {
    my $sth = $My::DB::select;
    $sth->bind_param(':baz', $baz);
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    my $sph = $My::DB::procedure;
    $sph->bind_param(':baz', $baz);
    $sph->execute;

    $dbh->commit;
};
if ($?) {
    $dbh->rollback;
}

# ...

```

Notice that several improvements have been made. Since the statement handles have a longer life than the request, there is no need for each request to prepare the statement, and no need to call the statement

handle's `finish` method. Since `Apache::DBI` and the `prepare_cached` method are not used, no cache lookups are needed.

Table 4: Expense Analysis after Stage 3 Optimization

Step/Actor	Application	Database	Relative Expense
1	Validate existing database connection		low
2		Bind SQL parameters	low
3		Execute the SQL statement	low
4	Fetch result rows		low
5		Bind PL/SQL parameters	low
6		Execute the PL/SQL statement	low
7		Commit or rollback	moderate

Conclusion

The number of steps needed to service the request in the example system has been reduced significantly. All of the highly expensive steps have been eliminated, and only one of the moderately expensive steps remains. The remaining steps exercise highly optimized code in the database server. Overall, the time needed to service the application request has been dramatically slashed.

It is probably possible to optimize this example even further, but I have not tried. It is very likely that the time could be better spent improving your database indexing scheme or web server buffering and load balancing. If there are any suggestions for further optimization of the application-database interaction, please mail them to me at jwb@cp.net.

This document Copyright [Jeffrey William Baker](#). Last modified 14 October 1999



Other documents on this site include:

- [Perl DBI Examples](#)
- [Resumé of Jeffrey William Baker](#)