

## Faking Stored Procedures

Zach Thompson

hideo@lastamericanempire.com

Perl's DBI module has become the tool of choice for those squaring off with databases with Perl, and, in my current post, I find myself doing exactly that. I process a lot of data, and to get it down quickly and efficiently, I create dynamic DBI "stored procedures" in my scripts. The stored procedure is just a series of database instructions that I run as a group. I implement these directly in my scripts as Perl closures.

We have legions of people downloading and creating data, filling directories full of hundreds, or sometimes thousands, of files that must be parsed and crammed into an Oracle database. We have several types of files that we must handle slightly differently, although they still belong to the same set of data and we should process them in one shot by the same program. I carve up these unwieldy chunks of data into hundreds of thousands of bite-sized morsels for our customers to consume.

With thousands of files to process, I have to consider efficiency and resource concerns, especially if my program is run from cron with hundreds of other programs waiting for their 15 seconds of fame. The hacking brethren before me resorted to the Oracle "sqlldr" tool, figuring Perl was simply not up to the task of handling large volumes. I've never liked backticking the tools that ship with the various databases when there is a Perl solution that can do it.

Of course, I don't want to process several thousand files into memory only to have the first INSERT statement fail. Even with DBI's "connect-and-prepare once, execute many" model, such a strategy could prove to be a miserable waste of time, and depending on the size of the files, my system or its administrator may not appreciate my gross memory transgressions. Such errors would be a little less devastating by committing a file's worth of data at time.

```
my $dbh = DBI->connect($dsn,
    {
        RaiseError => 1,
        AutoCommit => 0,
        PrintError => 0,
    });

my $insert_stmt = $dbh->prepare(
    q{ insert into table (
        col1,
        col2,
        col3,
        col4,
        col5
    ) values ( ?, ?, ?, ?, ? )
    } );

for my $file ( @files ){
    my $data = parse_file( $file );
    $insert_stmt->execute( @$data ) for @$data;
    archive_file($file);
}

$dbh->disconnect;
```

For simple programs, this approach works. However, when I add statement handles to look up values, multiple **INSERT** statements, transactions, or even more complex SQL statements, the DBI code overwhelms the main flow of my program in a hurry. I have had to modify programs with the main logic embedded in a mess of DBI, and I can say that it is not the most maintainable structure.

It would be much cleaner if my main processing loops could stand on their own, despite the complexity of the DBI code. I can put the DBI stuff somewhere else.

```
for my $file ( @files ) {
    my $data = parse_file( $file );
    insert_data( $data );
    archive( $file );
}

sub insert_data { ...DBI code as before... }
```