"Stored Procedures" on the Fly with DBI

Zach Thompson

DBI has become the weapon of choice for those of us squaring off with databases with Perl, and, in my current post, I find myself doing exactly that. We have legions of people downloading and creating data, filling directories full of hundreds, sometimes thousands, of files that must be parsed and crammed into an Oracle database. There are several types of files that must be handled slightly differently, although they still belong to the same *set* of data and should probably be processed in one shot by the same program. So, it is my task to 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, efficiency and resource concerns must be considered, especially if your program is being cron'ed with hundreds of other programs waiting for their 15 seconds of fame. My hacking brethren before me resorted to the Oracle tool "sqldr" in such situations, figuring Perl was simply not up to the task of handling any type of volume. I've never liked the idea of backticking the tools that ship with the various databases — or any external program for that matter — especially when there is a Perl solution that can sufficiently cope.

Of course, we 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" paradigm, such a strategy could prove to be a miserable waste of time. And, depending on the size of the files, your system or its administrator may not appreciate your gross memory transgressions. Perhaps such errors would be a little less devastating by committing a file's worth of data at time. A skeletal version of such a program might look like this:

```
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);
  $insert_stmt->execute(@$_) for @$data;
  archive_file($file);
}
$dbh->disconnect;
```

For simple programs, this approach is probably sufficient. However, when you add statement handles to look up values, multiple insert statements, transaction handling, and when the SQL itself becomes more complex, the DBI code can overwhelm the main flow of your 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 our main processing loops could stand on their own, despite the complexity of the DBI code lurking elsewhere:

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

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

Here we have exiled the DBI code to the `insert_date()` subroutine. This has two advantages:

1. The flow of the program is cleaner.
2. The bulk of the database code can now be found in a single spot, making it easier to maintain in the future.

However, even if the connect to the database were done somewhere earlier in the program, we now have the problem of preparing whatever statement handles we put in `insert_data()` for every file.

We need a way to save our prepared statements between calls to `insert_date()` without putting them in the main flow of the program. Read on.

What is a Closure, Anyway?

Not long ago, in a company far, far away I mucked around on MS SQL servers creating stored procedures at will to handle the raw data my programs had coaxed from the ether. Stored procedures were an effective way to greatly simplify the interaction between my programs and the database. In fact, after the connection to the database was made with Win32::ODBC, the interaction usually boiled down to a single “if” clause:

```
if($con->Sql('exec stored_procedure list of data')) { ..handle error.. }
```

As long as the “list of data” passed in was in the expected order, you might never need to know what monstrous collection of T-SQL statements “stored_procedure” was hiding.

There is still an effective way to play database developer by using closures with DBI. If you’ve never used or even heard of a closure before, here is an example:

```
my $incrementer = print_increment();

for my $word (qw/what will happen?/)
{
  $incrementer->($word);
}

sub print_increment
{
  # initialization #
  my $call_count;

  # closure #
  return sub { print ++$call_count, ': ', $_[0], "\n" };
}

# OUTPUT:
# 1: what
# 2: will
# 3: happen
```

As seen from the output, the call to the anonymous subroutine referenced by `$incrementer` prints the value of `$call_count` as well as the string passed in as a parameter. Normally, lexical variables such as `$call_count` would be swept into the abyss at the end of their scope — in this case the end of `print_increment()`. However, because the returned subroutine reference has access to `$call_count`, `$call_count` will be accessible until `$incrementer` goes out of scope.

This very same technique, albeit a little more complex, can be used to effectively turn your DBI code into a stored procedure for the duration of your program. Note in particular that the general structure of `print_increment()` comprises two sections: initialization and the declaration of the closure itself. The `initialization` section will house all of the variables that the `closure` will need to access.

Using Closures with DBI

Suppose we have directory files containing stock exchange information. Each file contains the data for a specific date with one line per company. From each line in the file, we extract 10 values: a company name, date, day high, day low, open price, closing price, volume, 52-week high, 52-week low, and the percentage change for the day. In other words, from the `for` loop above, `parse_files()` will return an array reference (i.e., `$data`), where each element itself is an array reference of the form:

```
[company, date, day high, day low, open price, closing price, \
  volume, 52-week high, 52-week low, change]
```

Each array reference will be inserted into the table "stock_prices" as a single row. However, we also need to look up the appropriate ticker symbol for each company in the table "ticker_map".

The bare bones structure will be very similar to the previous closure example. However, the initialization section will contain our prepared statement handles, and the closure will loop through the rows in `$data` while providing transaction handling by using `eval`:

```
sub init_stock_inserter {
  my ($con) = @_; # Pass in connection

  # Declare and initialize lexical variables that we want our
  # anon sub to access

  my $get_ticker = $con->prepare(q-select ticker from ticker_map
                                  where company = ?-);

  my $insert_stock_prices = $con->prepare(q-insert into
stock_prices (ticker,
                                      company,
                                    date,
                                    day_high,
                                    day_low,
                                    open,
                                    close,
                                    volume,
                                    year_high,
                                    year_low,
                                    change)
                              values (?,
                                    TO_DATE(?, 'YYYY/MM/DD'),
                                    ?,
                            ?,
                            ?,
                            ?,
                            ?,
                            ?,
                            ?,
                            ?)-);

    return sub {
         my ($data) = @_;

         eval {
       for my $row (@$data) {
         $get_ticker->execute($row->[0]);
         my $ticker = ($get_ticker->fetchall_array())[0];
         $ticker or die join(' ', 'No ticker for', $row->[0],
                             'found');
         $insert_stock_prices->execute($ticker, @$row);
       }
       $con->commit;
         };

         if($@) {
           $con->rollback;
           $con->disconnect;
           notify('Insert error: ' . $@);
         }
    };
}
```

Despite the simplicity of the subroutine, there is actually a lot going on here, so let's look at it from start to finish.

Initialization

The first half of `init_stock_inserter` is a typical setup, except for the fact that we pass in the database handle. This is really the only loose end preventing us from quarantining all of our database code in `init_stock_inserter`. So, why have we created it elsewhere and passed it in? There are really two reasons.

First, if we wanted to create multiple inserters by calling a *similar* inserter subroutine several times, we would need to create a connection each time. I say "similar" because in this case there's really no need to call `init_stock_inserter` more than once. However, what if we were processing two different types of files, say, NASDAQ and NYSE? The data from each type might be inserted into its own table with its own set of statement handles, though still using the same database connection. So, passing in the database handle saves us from having to connect again and again for every inserter we create.

The second reason we pass in the database connection is because we really have no *good* way to explicitly disconnect when we're finished with the handle. You *could* have a second parameter to the subroutine reference, which would indicate that the `$data` coming in is from the last file in the queue:

```
return sub {
    my ($data, $last_file) = @_;

      ...later
      $con->disconnect if $last_file;
}
```

Of course, you would also need to keep track of the number of files outside of `init_stock_inserter()`, in addition to making sure you don't re-create or delete the connection when calling the inserter the second time. And that's just plain ugly.

Besides, we can easily hide the connection code away with the other subroutines if we wish:

```
sub make_DB_connection {
      return DBI->connect(..., {RaiseError => 1,
                              AutoCommit => 0,
                              PrintError => 0});
}
```

Usually, I put all of the command-line arguments, including the data source, username, and password, into an `our` variable so they will be accessible from `make_DB_connection` and won't need to be passed in.

The next step is to prepare two statement handles using placeholders for the data we expect to be passed in: one to obtain the ticker for each company and the other to insert the new found ticker and associated data. Nothing special here.

The Closure

The return statement is where things start to get interesting. The only parameter our sub will need is the data itself, which will be passed in for each file. Furthermore, since we are using references to arrays to store our individual rows of data, we know that the element order will be preserved and invariably map to that of our placeholders. This is assuming, of course, that our `parse_file()` subroutine was designed to populate `$data` in the same order as the placeholders (*hint, hint*).

The `eval` block is the way to implement transactions using DBI, so everything the sub will do during normal execution is enclosed within the block. For `eval` to manage the transaction, we must make sure that our DB connection specifies `RaiseError` to 1 and

AutoCommit to 0. With RaiseError set to 1, any database error encountered while selecting a ticker or inserting the data will cause die to be called with the specific database error message. The eval block will exit at the point of failure, setting $@ with the error message associated with the call to die.

We also have set AutoCommit to 0, so any rows that may have already been inserted can be rolled back. Thus, the scope of our transaction will be at the level of an entire file: either all of the data from a file will be inserted or none at all. This will prevent us from having to determine whether a particular company's data has already been inserted for a specific date if our program chokes midway through a file.

The for loop simply iterates through the rows of data that will eventually be passed in. Both $get_ticker and $insert_stock_prices are lexical variables (as was $call_count) and will be accessible through the variable to which the returned sub reference is assigned.

For each row of data, we fetch the associated ticker using the company value and ensure that it's not undefined or empty. The final statement in the loop simply calls our insert handle with the ticker and the array to which $row refers. Because we have assured the order of elements for each $row in our parse routine, de-referencing $row will expand it into a list that maps exactly to our placeholders. Our call to $insert_stock_prices will *effectively* become:

```
$insert_stock_prices->execute($ticker,
                       company,
                  date,
                  day high,
                  day low,
                  open price,
                  closing price,
                  volume,
                  52-week high,
                  52-week low,
                  change
                  );
```

Once we leave the for loop, we can be reasonably certain that all of the rows passed in were inserted successfully, and we can commit our transaction to the database. $con, too, is just a lexical variable and will be preserved between invocations of the subroutine reference. So, all of the database handle methods (e.g., commit and rollback) are available to use as well.

The final component in our inserter provides any error handling we wish to implement for a failed transaction. In this case, we:

1. Rollback any rows that may have already been inserted.
2. Close our database connection.
3. Call a general error handling routine that perhaps notifies interested parties of the error or simply logs it to a file.

After closing the database connection, notify() should eventually die also, because the closure will no longer have valid database and statement handle references.

The Closure in Action

Using init_stock_inserter(), we can arrange the main flow of our program in a way that will make our grandchildren giddy when it comes time to maintain our broken gem:

```
my $con = make_DB_connection();              # 1. Connect
my $stock_inserter = init_stock_inserter($con);# 2. Initialize

for my $file (@files){
      my $data = parse_file($file);           # 3. Parse
      $stock_inserter->($data);               # 4. Insert
      archive($file);                         # 5. Archive
}
$con->disconnect;                             # 6. Disconnect
```

We’ve reduced the main flow to less than ten lines that accurately represent the primary steps in our program. Our simulated stored procedure boils down to two lines of code. On line 2, we call `init_stock_inserter()` to create our closure. As we have seen, `init_stock_inserter` returns a subroutine reference, which we store in `$stock_inserter` and execute on line 6, passing in the `$data` freshly extracted from one of the files. Once `$stock_inserter` gets a reference to some data, all of its primed statement handles will be brought to bear on conducting the data to its new home in the database.

Further Fine Tuning

Although the “initialization” section in `init_stock_inserter` was fairly simple, it doesn’t necessarily have to be. All the database related code — other than the connection itself — that your closure requires to function properly can and should be placed here.

For example, if the stock exchange files we were processing always contained the same 10 companies in each file, we could look up their ticker symbols during initialization and store them in a hash. The company field for each row of data would then be used as the key to retrieve the ticker from the hash:

```
sub init_stock_inserter{

my ($con) = @_;

    my $tickers = get_tickers($con);  # $tickers->{company} = ticker

    ...

}
```

`get_tickers()` would contain the code to perform the lookups from ticker_map. When executing `$insert_stock_prices` in the closure, we simply retrieve the ticker from the hash:

```
$insert_stock_prices->execute($tickers->{$row->[0]}, @$row);
```

Even in our original implementation of `init_stock_inserter()`, we could dramatically reduce the number of lookups from ticker_map by storing the ticker symbols as we go:

```
sub init_stock_inserter{

    my ($con) = @_;
    my %tickers;     # hash to hold the tickers we've already
                     # retrieved

    ...statements as before

    return sub{
        my ($data) = @_;
        eval{
                for my $row (@data){
                my $company = $row->[0];
                unless(exists $tickers{$company}){
                        # check if we've already looked it up
                    $get_ticker->execute($company);
                    my $ticker = ($get_ticker->
                                 fetchall_array())[0];
                    $ticker or die join(' ', 'No ticker for',
                                      $company, 'found.');
                    $tickers{$company} = $ticker;
                }
                $insert_stock_prices->execute($tickers{$company},
                                          @$row);
            }
        };
    };

    ...error checking as before
}
```

Now the lookup code, lines 14-17, is only executed once for each company, and the ticker is stored in `%tickers` for future reference. As with our statement handles, the closure will preserve the state of `%tickers` between invocations. If we were processing a year's worth of files, this might save us a few hundred lookups for each company, depending on how often data appears for that company in the files.

Creating Multiple Inserters

Creating multiple inserters is easy enough, we just call our init subroutine for each inserter we wish to create. With a little modification to our main loop and the initialization section of `init_stock_inserter`, we could have our program handle several file types with different database requirements. We might iterate through each file type as follows in our main loop:

```
my $con = make_DB_connection();

for my $type (qw/NYSE NASDAQ/){

    my $stock_inserter = init_stock_inserter($con, $type);

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

Now we loop through both NYSE and NASDAQ type files. These may differ in file structure, lookup table referenced, and the table that ultimately stores the data. Nevertheless, the steps in processing the data remain relatively unchanged from our initial six. Notice, however, that we now pass the type of file being processed to the init routine in addition to the database connection. This allows our init routine to distinguish which type of file it is expected to handle and to initialize the closure appropriately:

```
sub init_stock_inserter{

    my ($con, $type) = @_;

    my (%tickers, $insert_stmt, $ticker_query);

    if($type eq 'NYSE'){
    $insert_stmt = 'insert into NYSE_prices...';
    $ticker_query = 'select ticker from NYSE_ticker_map...';
}
elsif($type eq 'NASDAQ'){
    $insert_stmt = 'insert into NASDAQ_prices...';
    $ticker_query = 'select ticker from NASDAQ_ticker_map...';
}
else{
    notify('Invalid file type specified: ' . $type);
}

    my $get_ticker = $con->prepare($ticker_query);
my $insert_stock_prices = $con->prepare($insert_stmt);

....as before
}
```

The major difference in this new version is that we use a chain of conditionals to determine the specific SQL to be used according to the file type. Once the SQL statements have been assigned, the same two statement handles as before are prepared for use by the closure. Our closure won't even mind if the data passed in differs in the order or number of columns in each row. As long as the dereferenced `$row` maps to the placeholders in `$insert_stmt`, the data will be inserted.

We could even further simplify `init_stock_inserter` by delegating the SQL generation to another subroutine entirely:

```
sub init_stock_inserter{
```

```
my ($con, $type) = @_;
my (%tickers, $get_ticker, $insert_stock_prices);

    {
    my ($insert_stmt, $ticker_query) = get_sql($type);
    $get_ticker = $con->prepare($ticker_query);
    $insert_stock_prices = $con->prepare($insert_stmt);
}

    ...create and return closure
}
```

The initialization section is now compact and clean as a whistle. Our new subroutine `get_sql()` doesn't even need to contain the SQL itself. It could suck the statements in from a prescribed file, thus eliminating the SQL from our program altogether. Having declared the two SQL statement variables — `$insert_stmt` and `$ticker_query` — inside a superficial block, we ensure that they are returned to obscurity as quickly as they were generated. After all, they are only needed to prepare our statement handles, and we don't need our closure dragging around dead weight. However, the statement handle variables must now be declared outside of the block so that our closure can still access them.

Final Musings

Using closures with DBI is an effective way to introduce some order into complex database code while still taking advantage of the efficiency that DBI provides with prepared statement handles. DBI closures also allow you to simulate, to some degree, calls to stored procedures native to the database. One advantage to the closure strategy that shouldn't be underestimated is that database-specific code — the SQL in particular — can be roped off and prevented from dominating the flow of your programs. In Perl, SQL is a second-class citizen and shouldn't be allowed to run the show.

Give it a spin the next time the mailman drops 10,000 files on your doorstep!

Zach Thompson can be reached at: `cublai@lastamericanempire.com`.