What’s new in the DBI
(since the book)

DBI-1.14-1.52.diff

by Tim Bunce
Profiling DBI Performance

Time flies like an arrow
(fruit flies like a banana)
How fast was that?

- The DBI has performance profiling built in

- Overall summary:

  ```
  $ DBI_PROFILE=1 ex/profile.pl
  DBI::Profile: 0.190639s 20.92% (219 calls) profile.pl @ 2006-07-24 15:47:07
  ```

- Breakdown by statement:

  ```
  $ DBI_PROFILE='!Statement' ex/profile.pl
  DBI::Profile: 0.206872s 20.69% (219 calls) profile.pl @ 2006-07-24 15:44:37

  '' =>
    0.001403s / 9 = 0.000156s avg (first 0.001343s, min 0.000002s, max 0.001343s)
  'CREATE TABLE ex_profile (a int)' =>
    0.002503s
  'INSERT INTO ex_profile (a) VALUES (?)' =>
    0.193871s / 100 = 0.001939s avg (first 0.002119s, min 0.001676s, max 0.002251s)
  'SELECT a FROM ex_profile' =>
    0.004776s / 108 = 0.000044s avg (first 0.000700s, min 0.000004s, max 0.003129s)
  ```
$ DBI_PROFILE='!Statement:!MethodName' ex/profile.pl

DBI::Profile: 0.203922s (219 calls) profile.pl @ 2006-07-24 15:29:29

'' =>
  'FETCH' =>
    0.000002s
  'STORE' =>
    0.000039s / 5 = 0.000008s avg (first 0.000019s, min 0.000002s, max 0.000019s)
  'connect' =>
    0.001336s

'CREATE TABLE ex_profile (a int)' =>
  'do' =>
    0.002324s

'INSERT INTO ex_profile (a) VALUES (?)' =>
  'do' =>
    0.192104s / 100 = 0.001921s avg (first 0.001929s, min 0.001520s, max 0.002699s)

'SELECT a FROM ex_profile' =>
  'execute' =>
    0.000082s
  'fetchrow_array' =>
    0.000667s / 101 = 0.000007s avg (first 0.000010s, min 0.000006s, max 0.000018s)
  'prepare' =>
    0.000122s
  'selectall_arrayref' =>
    0.000676s
  'selectall_hashref' =>
    0.003452s
Profile of a Profile

- Profiles ‘top level’ calls from application into DBI

- Profiling is controlled by, and collected into, $h->{Profile} attribute

- Child handles inherit reference to parent $h->{Profile}
  - So child handle activity is aggregated into parent

- When enabled by DBI_PROFILE env var
  - uses a single $h->{Profile} is shared by all handles
  - so all activity is aggregated into a single data tree

- Data is dumped when the $h->{Profile} object is destroyed
Profile Path ⇒ Profile Data

- The Path determines where each sample is accumulated within the Data

```perl
$h->{Profile}->{Path} = [ ]
$h->{Profile}->{Data} = [ ...accumulated sample data... ]

$h->{Profile}->{Path} = [ "MethodName" ]
$h->{Profile}->{Data} = { "prepare" } -> [ ... ]
                      { "execute" } -> [ ... ]
                      { ... }    -> [ ... ]

$h->{Profile}->{Path} = [ "Statement", "MethodName" ]
$h->{Profile}->{Data} = { "INSERT ..." } -> { "prepare" } -> [ ... ]
                      -> { "execute" } -> [ ... ]
                      { "SELECT ..." } -> { "prepare" } -> [ ... ]
                      -> { "execute" } -> [ ... ]
```
# Profile Path Elements

<table>
<thead>
<tr>
<th>Kind</th>
<th>Examples</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{AttributeName}</code></td>
<td><code>{Statement}</code></td>
<td>&quot;SELECT ...&quot;</td>
</tr>
<tr>
<td><code>{Username}</code></td>
<td>&quot;timbunce&quot;</td>
<td></td>
</tr>
<tr>
<td><code>{AutoCommit}</code></td>
<td>&quot;1&quot;</td>
<td>&quot;the value of private_attr&quot;</td>
</tr>
<tr>
<td><code>{private_attr}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;!Magic&quot;</td>
<td>&quot;!Statement&quot;</td>
<td>&quot;SELECT ...&quot;</td>
</tr>
<tr>
<td>&quot;!MethodName&quot;</td>
<td>&quot;selectrow_array&quot;</td>
<td>&quot;DBD::Pg::db::selectrow_array&quot;</td>
</tr>
<tr>
<td>&quot;!MethodClass&quot;</td>
<td>&quot;MyFoo.pm&quot;</td>
<td>&quot;MyFoo.pm&quot;</td>
</tr>
<tr>
<td>&quot;!File&quot;</td>
<td>&quot;MyFoo.pm line 23 via Bar.pm line 9&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;!Caller2&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;code_ref</td>
<td>sub { &quot;bar&quot; }</td>
<td>&quot;bar&quot;</td>
</tr>
<tr>
<td>&amp;subname</td>
<td></td>
<td></td>
</tr>
<tr>
<td>anything else</td>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
</tr>
</tbody>
</table>
“!Statement” vs “{Statement}”

- “{Statement}” is always the value of the Statement attribute
  - Fine for statement handle
  - For database handles it’s the last statement executed
  - That’s often not useful, or even misleading, for profiling

- “!Statement” is smarter
  - Is an empty string for methods that are unrelated to current statement
    - ping, commit, rollback, quote, dbh attribute FETCH & STORE, etc.
  - so you get more accurate separation of profile data using “!Statement”

- Statement tracking can’t be perfect
  - but is certainly good enough for profiling
Profile Leaf Node Data

• Each leaf node is a ref to an array:

```
[  
  106,                  # 0: count of samples at this node
  0.0312958955764771,   # 1: total duration
  0.000490069389343262, # 2: first duration
  0.000176072120666504, # 3: shortest duration
  0.00140702724456787,  # 4: longest duration
  1023115819.83019,     # 5: time of first sample
  1023115819.86576,     # 6: time of last sample
]
```

- First sample to create the leaf node populates all values
- Later samples reaching that node always update elements 0, 1, and 6
- and may update 3 or 4 depending on the duration of the sampled call
Working with profile data

- To aggregate sample data for any part of the tree
  - to get total time spent inside the DBI
  - and return a merge all those leaf nodes

\[
\text{$time\_in\_dbi = \text{dbi\_profile\_merge}(my \ \$totals=[], @\$leaves);}
\]

- To aggregate time in DBI since last measured
  - For example per-httpd request

```perl
my $time_in_dbi = 0;
if (my $Profile = $dbh->{Profile}) { # if profiling enabled
    $time_in_dbi = dbi_profile_merge([], $Profile->{Data});
    $Profile->{Data} = undef; # reset the profile Data
}
# add $time_in_dbi to httpd log
```
Profile something else

- Adding your own samples

```perl
use DBI::Profile (dbi_profile dbi_time);

my $t1 = dbi_time(); # floating point high-resolution time

... execute code you want to profile here ...

my $t2 = dbi_time();
dbi_profile($h, $statement, $method, $t1, $t2);
```
Profile specification

- Profile specification
  - `<path> / <class> / <args>`
  - `DBI_PROFILE='!/Statement:!MethodName/DBI::ProfileDumper::Apache/arg1:arg2:arg3'`
  - `$h->{Profile} = '...same...';`

- Class
  - Currently only controls output formatting
  - Other classes should subclass DBI::Profile

- DBI::Profile is the default
  - provides a basic summary for humans
  - large outputs are not easy to read
  - can’t be filtered or sorted
dbiprof

- DBI::ProfileDumper
  - writes profile data to dbi.prof file for analysis

- DBI::ProfileDumper::Apache
  - for mod_perl, writes a file per httpd process/thread

- DBI::ProfileData
  - reads and aggregates dbi.prof files
  - can remap and merge nodes in the tree

- dbiprof
  - reads, summarizes, and reports on dbi.prof files
  - by default prints nodes sorted by total time
  - has options for filtering and sorting
Managing statement variations

- For when placeholders aren’t being used or there are tables with numeric suffixes.
- A `&norm_std_n3’ in the Path maps to ‘!Statement’ edited in this way:

  ```
  s/\b\d+\b/<N>/g;             # 42 -> <N>
  s/\b0x[0-9A-Fa-f]+\b/<N>/g;  # 0xFE -> <N>
  s/'.*?'/'<S>'/g;             # single quoted strings (doesn't handle escapes)
  s/".*?"/"<S>"/g;             # double quoted strings (doesn't handle escapes)

  # convert names like log20001231 into log<N>
  s/([a-z_.]+)(\d{3,})\b/${1}<N>/ieg;

  # abbreviate massive "in (...)" statements and similar
  s!((\s*<[NS]\s*,\s*){100,})!sprintf("$2,<repeated %d times>",length($1)/2)!eg;
  ```

- It’s aggressive and simplistic but usually very effective.
- You can define your own subs in the DBI::ProfileSubs namespace
Other stuff...

a random assortment
Unicode Tools

- Unicode problems can have many causes
- The DBI provides some simple tools to help:
  - **neat($value)**
    - Unicode strings are shown double quoted, else single
  
  - **data_string_desc($value)**
    - Returns ‘physical’ description of a string, for example:
      - UTF8 on but INVALID ENCODING, non-ASCII, 4 chars, 9 bytes

  - **data_string_diff($value1, $value2)**
    - Compares the logical characters not physical bytes
    - Returns description of logical differences, else an empty string

  - **data_diff($value1, $value2)**
    - Calls `data_string_desc` and `data_string_diff`
    - Returns description of logical and physical differences, else an empty string
Keep track of your kids!

- Handles now keep (weak) references to their children

```perl
$kids = $dbh->{ChildHandles};
for my $sth (@$kids) {
    next unless $sth; # ignore destroyed handles
    print "$sth->{Statement}\n";
}
```
Brain Surgery

- Swap the inner handle of two DBI handles

```perl
$h1->swap_inner_handle($h2)
```

- Enables a dead handle to effectively be resuscitated
- Used by DBIx::HA module

- Cryogenics for handle brains

```perl
$frozen = $dbh1->take_imp_data();

$dbh2 = DBI->connect(..., { dbi_imp_data => $frozen });
```

- Powerful voodoo. Needed for DBI::Pool
Fetching one row in one call

- Extra do-it-all-in-one-call utility methods:
  
  ```perl
  $aref = $dbh->selectrow_arrayref($select, \%attr, @bind)
  $href = $dbh->selectrow_hashref($select, \%attr, @bind)
  ```

- The `$select` parameter can be a prepared statement handle for extra speed
Fetching all rows in one call

- Want all the rows in a single hash?

```perl
$href = $dbh->selectall_hashref(
    "select id, name, country from ...", "id"
); {
    42 => { id=>42, name=>'Tim', country=>'Ireland' },
    43 => { id=>43, name=>'Jim', country=>'USA' },
    ...
}
```

- There's also a `$sth->fetchall_hashref($keyfield)` method.
Fetching Multiple Keys

- `fetchall_hashref()` now supports multiple key columns

```perl
$sth = $dbh->prepare("select state, city, ... ");
$sth->execute;
$data = $sth->fetchall_hashref( [ 'state', 'city' ] );

$data = {
    CA => {
        LA => { state=>'CA', city=>'LA', ... },
        SF => { state=>'CA', city=>'SF', ... },
    },
    NY => {
        NY => { ... },
    }
};
```

- Also works for `selectall_hashref()`
Batch fetching

- How to bulk fetch more rows than fit in memory?

```perl
while ( $rows = $sth->fetchall_arrayref(undef, 10_000) && @$rows) {
    while ( $row = shift @$rows ) {
        ...
    }
}
```

- Or

```perl
while ( $row = shift(@$cache)
    || shift @{$cache=$sth->fetchall_arrayref(undef, 10_000)}
) {
    ...
}
```
Do it in bulk...

```perl
$sth = $dbh->prepare("insert into foo (a, b) values (?, ?)");

$sth->execute_array( { ArrayTupleStatus => \@tuple_status },
     \@array_a,
     \@array_b,
);

$sth->execute_for_fetch( sub { ... }, \@tuple_status );
```

- Works for all drivers now
- Some drivers implement optimized methods (DBD::ODBC, DBD::Oracle,...)
Do it in parallel...

- DBI supports iThreads

- But...
  - Like all extensions using tied magic, handles can't be cloned or shared

- So...
  - Each thread/interpreter needs to make it's own connection

- However...
  - DBI::Pool module is partly implemented, needs a volunteer
Information and Warnings

- Drivers can indicate Information and Warning states in addition to Error states
  - Uses *false-but-defined* values of $h->err and $DBI::err
  - Zero "0" indicates a "warning"
  - Empty "" indicates "success with information" or other messages from database
- Drivers should use $h->set_err(...) method to record info/warn/error states
  - implements logic to correctly merge multiple info/warn/error states
  - info/warn/error messages are appended to errstr with a newline
  - $h->{ErrCount} attribute is incremented whenever an error is recorded
- The $h->{HandleSetErr} attribute can be used to influence $h->set_err()
  - A code reference that's called by set_err and can edit its parameters
  - So can promote warnings/info to errors or demote/hide errors etc.
  - Called at point of error from within driver, unlike $h->{HandleError}
- The $h->{PrintWarn} attribute acts like $h->{PrintError} but for warnings
  - Default is on
Error Handling

- $dbh->{Statement} is copy of most recent $sth->{Statement}

- $h->{ShowErrorStatement} = 1;

  appends Statement text to the RaiseError/PrintError message:

  DBD::foo::db do failed: errstr [for statement "..."]

- $sth->{ParamValues} ==> { hash of bound placeholder values };

  if driver supports ParamValues then it'll be included in ShowErrorStatement:

  DBD::foo::db ... [for statement "..." with 1='foo', 2='bar']
Custom Error Handling

- Don’t want to just Print or Raise an Error?
  
  ```perl
  $h->{HandleError} = sub { ... };
  ```

- The `HandleError` code
  - is called just before `PrintError/RaiseError` are handled
  - is passed the error message, handle, and return value
  - if it returns `false` then `RaiseError/PrintError` are checked and acted upon as normal

- The hander code can
  - alter the error message text by changing `$_[0]`
  - use `caller()` or `Carp::confess()` or similar to get a full stack trace
  - use `Exception` or a similar module to throw a formal exception object
Tweaked Tracing

- Trace level 1 made more useful
  - doesn’t show nested DBI calls
  - shows just the first and last fetch calls
  - shows first two parameters of all methods

- Trace for `fetch` methods now shows row number

- Can now set/get trace level via handle attribute
  ```perl
  local $h->{TraceLevel} = N;
  $dsn = "dbi:Driver(TraceLevel=2):dbname=foo";
  ```

- Trace level 3 and over includes some extra call stack information
  ```perl
  <- prepare= DBI::st=HASH(0x8367760) at DBI.pm line 1287 via test.pl line 11
  ```
More Metadata

- $sth = $dbh->column_info(...)
- $sth = $dbh->primary_key_info(...)
- @ary = $dbh->primary_key(...)
- $sth = $dbh->foreign_key_info(...)
- $sth = $dbh->statistics_info(...)
- $foo = $dbh->get_info(...)
- $id = $dbh->quote_identifier(...)
Other Stuff

- $dbh->last_insert_id()
- $dbh2 = $dbh1->clone()
- %drhs = DBI->installed_drivers()
- DBI->installed_versions()
- ($scheme, $driver, $attr_string, $attr_hash, $driver_dsn) = DBI->parse_dsn($dsn)
DBD::PurePerl

- Need to use the DBI somewhere where you can’t compile extensions?

- The DBI::PurePerl module is an emulation of the DBI written in Perl
  - Works with pure-perl drivers, including: AnyData, Excel, LDAP, mysqlPP, etc.
  - plus DBD::Proxy

- Enabled via the $DBI_PUREPERL environment variable:
  1 = Automatically fall-back to DBI::PurePerl if DBI extension can’t be bootstrapped
  2 = Force use of DBI::PurePerl

- Reasonably complete emulation - enough for the drivers to work well
  - See DBI::PurePerl documentation for the small-print if you want to use it
The DBI now includes an SQL parser module: DBI::SQL::Nano
  - Has an API compatible with SQL::Statement

If SQL::Statement is installed
  - then DBI::SQL::Nano becomes an empty subclass of SQL::Statement

Existing DBD::File module is now shipped with the DBI
  - base class for simple DBI drivers
  - modified to use DBI::SQL::Nano.

New DBD::DBM driver now shipped with the DBI
  - An SQL interface to DBM and MLDBM files using DBD::File and DBI::SQL::Nano.

Thanks to Jeff Zucker
DBI::SQL::Nano

• Supported syntax

  DROP TABLE [IF EXISTS] <table_name>
  CREATE TABLE <table_name> <col_def_list>
  INSERT INTO <table_name> [<insert_col_list>] VALUES <val_list>
  DELETE FROM <table_name> [<where_clause>]
  UPDATE <table_name> SET <set_clause> [<where_clause>]
  SELECT <select_col_list> FROM <table_name> [<where_clause>] [<order_clause>]

• Where clause

  ■ a single "[NOT] column/value <op> column/value" predicate
  ■ multiple predicates combined with ORs or ANDs are not supported
  ■ op may be one of: < > >= <= = <> LIKE CLIKE IS

• If you need more functionality...

  ■ Just install the SQL::Statement module
New Attributes for Fieldnames

• Control case of key (field) names returned by `fetchrow_hashref`

```perl
$h->{FetchHashKeyName} = 'NAME_lc'; # or 'NAME_uc'
```

• Fieldname-to-column-index mapping:

```perl
$h->{NAME_lc_hash}  ==>  { id => 0, name => 1, country => 2 };
```

• Also `NAME_uc_hash`, `NAME_hash`
Intercepting DBI Method Calls

• An alternative to subclassing
  ■ Added in DBI 1.49 - Nov 2005
  ■ but not yet documented and subject to change

• Example:
  $dbh->{Callbacks}->{prepare} = sub { ... }

  ■ Arguments to original method are passed in.
  ■ The name of the method is in $_(localized).
  ■ The Callbacks attribute is not inherited by child handle

• Some special ‘method names’ are supported:
  connect_cached.new
  connect_cached.reused
The end

for now.