NAME

DBI::Profile - Performance profiling and benchmarking for the DBI

SYNOPSIS

The easiest way to enable DBI profiling is to set the DBI_PROFILE environment variable to 2 and then run your code as usual:

```
DBI_PROFILE=2 prog.pl
```

This will profile your program and then output a textual summary grouped by query when the program exits. You can also enable profiling by setting the Profile attribute of any DBI handle:

```
$dbh->{Profile} = 2;
```

Then the summary will be printed when the handle is destroyed.

Many other values apart from are possible - see "ENABLING A PROFILE" below.

DESCRIPTION

The DBI::Profile module provides a simple interface to collect and report performance and benchmarking data from the DBI.

For a more elaborate interface, suitable for larger programs, see DBI::ProfileDumper and dbiprof. For Apache/mod_perl applications see DBI::ProfileDumper::Apache.

OVERVIEW

Performance data collection for the DBI is built around several concepts which are important to understand clearly.

Method Dispatch
: Every method call on a DBI handle passes through a single 'dispatch' function which manages all the common aspects of DBI method calls, such as handling the RaiseError attribute.

Data Collection
: If profiling is enabled for a handle then the dispatch code takes a high-resolution timestamp soon after it is entered. Then, after calling the appropriate method and just before returning, it takes another high-resolution timestamp and calls a function to record the information. That function is passed the two timestamps plus the DBI handle and the name of the method that was called. That data about a single DBI method call is called a *profile sample*.

Data Filtering
: If the method call was invoked by the DBI or by a driver then the call is ignored for profiling because the time spent will be accounted for by the original 'outermost' call for your code.

  For example, the calls that the *selectrow_arrayref()* method makes to *prepare()* and *execute()* etc. are not counted individually because the time spent in those methods is going to be allocated to the *selectrow_arrayref()* method when it returns. If this was not done then it would be very easy to double count time spent inside the DBI.

Data Storage Tree
: The profile data is accumulated as 'leaves on a tree'. The 'path' through the branches of the tree to a particular leaf is determined dynamically for each sample. This is a key feature of DBI profiling.

  For each profiled method call the DBI walks along the Path and uses each value in the Path to step into and grow the Data tree.

  For example, if the Path is

  ```
  [ 'foo', 'bar', 'baz' ]
  ```

  then the new profile sample data will be *merged* into the tree at

  ```
  $h->{Profile}->{Data}->{foo}->{bar}->{baz}
  ```

  But it's not very useful to merge all the call data into one leaf node (except to get an overall

'time spent inside the DBI' total). It's more common to want the Path to include dynamic values such as the current statement text and/or the name of the method called to show what the time spent inside the DBI was for.

The Path can contain some 'magic cookie' values that are automatically replaced by corresponding dynamic values when they're used. These magic cookies always start with a punctuation character.

For example a value of '`!MethodName`' in the Path causes the corresponding entry in the Data to be the name of the method that was called. For example, if the Path was:

```
[ 'foo', '!MethodName', 'bar' ]
```

and the *selectall_arrayref()* method was called, then the profile sample data for that call will be merged into the tree at:

```
$h->{Profile}->{Data}->{foo}->{selectall_arrayref}->{bar}
```

Profile Data

Profile data is stored at the 'leaves' of the tree as references to an array of numeric values. For example:

```
[
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
]
```

After the first sample, later samples always update elements 0, 1, and 6, and may update 3 or 4 depending on the duration of the sampled call.

ENABLING A PROFILE

Profiling is enabled for a handle by assigning to the Profile attribute. For example:

```
$h->{Profile} = DBI::Profile->new();
```

The Profile attribute holds a blessed reference to a hash object that contains the profile data and attributes relating to it.

The class the Profile object is blessed into is expected to provide at least a DESTROY method which will dump the profile data to the DBI trace file handle (STDERR by default).

All these examples have the same effect as each other:

```
$h->{Profile} = 0;
$h->{Profile} = "/DBI::Profile";
$h->{Profile} = DBI::Profile->new();
$h->{Profile} = {};
$h->{Profile} = { Path => [] };
```

Similarly, these examples have the same effect as each other:

```
$h->{Profile} = 6;
$h->{Profile} = "6/DBI::Profile";
$h->{Profile} = "!Statement:!MethodName/DBI::Profile";
$h->{Profile} = { Path => [ '!Statement', '!MethodName' ] };
```

If a non-blessed hash reference is given then the DBI::Profile module is automatically `require`'d and the reference is blessed into that class.

If a string is given then it is processed like this:

```
($path, $module, $args) = split /\//, $string, 3

@path = split /:/, $path
@args = split /:/, $args

eval "require $module" if $module
$module ||= "DBI::Profile"

$module->new( Path => \@Path, @args )
```

So the first value is used to select the Path to be used (see below). The second value, if present, is used as the name of a module which will be loaded and it's `new` method called. If not present it defaults to DBI::Profile. Any other values are passed as arguments to the `new` method. For example: `2/DBIx::OtherProfile/Foo:42`.

Numbers can be used as a shorthand way to enable common Path values. The simplest way to explain how the values are interpreted is to show the code:

```
push @Path, "DBI" if $path_elem & 0x01;
push @Path, "!Statement" if $path_elem & 0x02;
push @Path, "!MethodName" if $path_elem & 0x04;
push @Path, "!MethodClass" if $path_elem & 0x08;
push @Path, "!Caller2" if $path_elem & 0x10;
```

So "2" is the same as "!Statement" and "6" (2+4) is the same as "!Statement:!Method". Those are the two most commonly used values. Using a negative number will reverse the path. Thus "-6" will group by method name then statement.

The splitting and parsing of string values assigned to the Profile attribute may seem a little odd, but there's a good reason for it. Remember that attributes can be embedded in the Data Source Name string which can be passed in to a script as a parameter. For example:

```
dbi:DriverName(Profile=>2):dbname
dbi:DriverName(Profile=>{Username}:!Statement/MyProfiler/Foo:42):dbname
```

And also, if the `DBI_PROFILE` environment variable is set then The DBI arranges for every driver handle to share the same profile object. When perl exits a single profile summary will be generated that reflects (as nearly as practical) the total use of the DBI by the application.

THE PROFILE OBJECT

The DBI core expects the Profile attribute value to be a hash reference and if the following values don't exist it will create them as needed:

Data

A reference to a hash containing the collected profile data.

Path

The Path value is a reference to an array. Each element controls the value to use at the corresponding level of the profile Data tree.

If the value of Path is anything other than an array reference, it is treated as if it was:

```
[ '!Statement' ]
```

The elements of Path array can be one of the following types:

Special Constant

!Statement

Use the current Statement text. Typically that's the value of the Statement attribute for the handle the method was called with. Some methods, like *commit()* and *rollback()*, are unrelated to a particular statement. For those methods !Statement records an empty string.

For statement handles this is always simply the string that was given to *prepare()* when the

handle was created. For database handles this is the statement that was last prepared or executed on that database handle. That can lead to a little 'fuzzyness' because, for example, calls to the *quote()* method to build a new statement will typically be associated with the previous statement. In practice this isn't a significant issue and the dynamic Path mechanism can be used to setup your own rules.

!MethodName

Use the name of the DBI method that the profile sample relates to.

!MethodClass

Use the fully qualified name of the DBI method, including the package, that the profile sample relates to. This shows you where the method was implemented. For example:

```
'DBD::_::db::selectrow_arrayref' =>
0.022902s
'DBD::mysql::db::selectrow_arrayref' =>
2.244521s / 99 = 0.022445s avg (first 0.022813s, min 0.022051s, max 0.028932s)
```

The ''DBD::_::db::selectrow_arrayref'' shows that the driver has inherited the selectrow_arrayref method provided by the DBI.

But you'll note that there is only one call to DBD::_::db::selectrow_arrayref but another 99 to DBD::mysql::db::selectrow_arrayref. Currently the first call doesn't record the true location. That may change.

!Caller

Use a string showing the filename and line number of the code calling the method.

!Caller2

Use a string showing the filename and line number of the code calling the method, as for !Caller, but also include filename and line number of the code that called that. Calls from DBI:: and DBD:: packages are skipped.

!File

Same as !Caller above except that only the filename is included, not the line number.

!File2

Same as !Caller2 above except that only the filenames are included, not the line number.

!Time

Use the current value of *time()*. Rarely used. See the more useful `!TimeN` below.

!Time~N

Where `N` is an integer. Use the current value of *time()* but with reduced precision. The value used is determined in this way:

```
int( time() / N ) * N
```

This is a useful way to segregate a profile into time slots. For example:

```
[ '!Time60', '!Statement' ]
```

Code Reference

The subroutine is passed the handle it was called on and the DBI method name. The current Statement is in `$_`. The statement string should not be modified, so most subs start with `local $_ = $_;`.

The list of values it returns is used at that point in the Profile Path.

The sub can 'veto' (reject) a profile sample by including a reference to undef in the returned list. That can be useful when you want to only profile statements that match a certain pattern, or

only profile certain methods.

Subroutine Specifier

A Path element that begins with '`&`' is treated as the name of a subroutine in the DBI::ProfileSubs namespace and replaced with the corresponding code reference.

Currently this only works when the Path is specified by the `DBI_PROFILE` environment variable.

Also, currently, the only subroutine in the DBI::ProfileSubs namespace is `'&norm_std_n3'`. That's a very handy subroutine when profiling code that doesn't use placeholders. See DBI::ProfileSubs for more information.

Attribute Specifier

A string enclosed in braces, such as '`{Username}`', specifies that the current value of the corresponding database handle attribute should be used at that point in the Path.

Reference to a Scalar

Specifies that the current value of the referenced scalar be used at that point in the Path. This provides an efficient way to get 'contextual' values into your profile.

Other Values

Any other values are stringified and used literally.

(References, and values that begin with punctuation characters are reserved.)

REPORTING

Report Format

The current accumulated profile data can be formatted and output using

```
print $h->{Profile}->format;
```

To discard the profile data and start collecting fresh data you can do:

```
$h->{Profile}->{Data} = undef;
```

The default results format looks like this:

```
DBI::Profile: 0.001015s 42.7% (5 calls) programname @ YYYY-MM-DD HH:MM:SS
'' =>
0.000024s / 2 = 0.000012s avg (first 0.000015s, min 0.000009s, max 0.000015s)
'SELECT mode,size,name FROM table' =>
0.000991s / 3 = 0.000330s avg (first 0.000678s, min 0.000009s, max 0.000678s)
```

Which shows the total time spent inside the DBI, with a count of the total number of method calls and the name of the script being run, then a formatted version of the profile data tree.

If the results are being formatted when the perl process is exiting (which is usually the case when the DBI_PROFILE environment variable is used) then the percentage of time the process spent inside the DBI is also shown. If the process is not exiting then the percentage is calculated using the time between the first and last call to the DBI.

In the example above the paths in the tree are only one level deep and use the Statement text as the value (that's the default behaviour).

The merged profile data at the 'leaves' of the tree are presented as total time spent, count, average time spent (which is simply total time divided by the count), then the time spent on the first call, the time spent on the fastest call, and finally the time spent on the slowest call.

The 'avg', 'first', 'min' and 'max' times are not particularly useful when the profile data path only contains the statement text. Here's an extract of a more detailed example using both statement text and method name in the path:

```
'SELECT mode,size,name FROM table' =>
'FETCH' =>
0.000076s
'fetchrow_hashref' =>
0.036203s / 108 = 0.000335s avg (first 0.000490s, min 0.000152s, max 0.002786s)
```

Here you can see the 'avg', 'first', 'min' and 'max' for the 108 calls to *fetchrow_hashref()* become rather more interesting. Also the data for FETCH just shows a time value because it was only called once.

Currently the profile data is output sorted by branch names. That may change in a later version so the leaf nodes are sorted by total time per leaf node.

Report Destination

The default method of reporting is for the DESTROY method of the Profile object to format the results and write them using:

```
DBI->trace_msg($results, 0); # see $ON_DESTROY_DUMP below
```

to write them to the DBI *trace()* filehandle (which defaults to STDERR). To direct the DBI trace filehandle to write to a file without enabling tracing the *trace()* method can be called with a trace level of 0. For example:

```
DBI->trace(0, $filename);
```

The same effect can be achieved without changing the code by setting the `DBI_TRACE` environment variable to `0=filename`.

The `$DBI::Profile::ON_DESTROY_DUMP` variable holds a code ref that's called to perform the output of the formatted results. The default value is:

```
$ON_DESTROY_DUMP = sub { DBI->trace_msg($results, 0) };
```

Apart from making it easy to send the dump elsewhere, it can also be useful as a simple way to disable dumping results.

CHILD HANDLES

Child handles inherit a reference to the Profile attribute value of their parent. So if profiling is enabled for a database handle then by default the statement handles created from it all contribute to the same merged profile data tree.

PROFILE OBJECT METHODS

format

See "REPORTING".

as_node_path_list

```
@ary = $dbh->{Profile}->as_node_path_list();
@ary = $dbh->{Profile}->as_node_path_list($node, $path);
```

Returns the collected data ($dbh->{Profile}{Data}) restructured into a list of array refs, one for each leaf node in the Data tree. This 'flat' structure is often much simpler for applications to work with.

The first element of each array ref is a reference to the leaf node. The remaining elements are the 'path' through the data tree to that node.

For example, given a data tree like this:

```
{key1a}{key2a}[node1]
{key1a}{key2b}[node2]
{key1b}{key2a}{key3a}[node3]
```

The *as_node_path_list()* method will return this list:

```
[ [node1], 'key1a', 'key2a' ]
[ [node2], 'key1a', 'key2b' ]
[ [node3], 'key1b', 'key2a', 'key3a' ]
```

The nodes are ordered by key, depth-first.

The `$node` argument can be used to focus on a sub-tree. If not specified it defaults to `$dbh->{Profile}{Data}`.

The `$path` argument can be used to specify a list of path elements that will be added to each element of the returned list. If not specified it defaults to a ref to an empty array.

as_text

```
@txt = $dbh->{Profile}->as_text();
$txt = $dbh->{Profile}->as_text({
node => undef,
path => [],
separator => " > ",
format => '%1$s: %11$fs / %10$d = %2$fs avg (first %12$fs, min %13$fs, max %14$fs)'."\n";
sortsub => sub { ... },
);
```

Returns the collected data ($dbh->{Profile}{Data}) reformatted into a list of formatted strings. In scalar context the list is returned as a single concatenated string.

A hashref can be used to pass in arguments, the default values are shown in the example above.

The `node` and <path> arguments are passed to *as_node_path_list()*.

The `separator` argument is used to join the elements of the path for each leaf node.

The `sortsub` argument is used to pass in a ref to a sub that will order the list. The subroutine will be passed a reference to the array returned by *as_node_path_list()* and should sort the contents of the array in place. The return value from the sub is ignored. For example, to sort the nodes by the second level key you could use:

```
sortsub => sub { my $ary=shift; @$ary = sort { $a->[2] cmp $b->[2] } @$ary }
```

The `format` argument is a `sprintf` format string that specifies the format to use for each leaf node. It uses the explicit format parameter index mechanism to specify which of the arguments should appear where in the string. The arguments to sprintf are:

```
1: path to node, joined with the separator
2: average duration (total duration/count)
(3 thru 9 are currently unused)
10: count
11: total duration
12: first duration
13: smallest duration
14: largest duration
15: time of first call
16: time of first call
```

CUSTOM DATA MANIPULATION

Recall that `$h->{Profile}->{Data}` is a reference to the collected data. Either to a 'leaf' array (when the Path is empty, i.e., DBI_PROFILE env var is 1), or a reference to hash containing values that are either further hash references or leaf array references.

Sometimes it's useful to be able to summarise some or all of the collected data. The *dbi_profile_merge_nodes()* function can be used to merge leaf node values.

dbi_profile_merge_nodes

```
use DBI qw(dbi_profile_merge_nodes);

$time_in_dbi = dbi_profile_merge_nodes(my $totals=[], @$leaves);
```

Merges profile data node. Given a reference to a destination array, and zero or more references to profile data, merges the profile data into the destination array. For example:

```
$time_in_dbi = dbi_profile_merge_nodes(
my $totals=[],
[ 10, 0.51, 0.11, 0.01, 0.22, 1023110000, 1023110010 ],
[ 15, 0.42, 0.12, 0.02, 0.23, 1023110005, 1023110009 ],
);
```

`$totals` will then contain

```
[ 25, 0.93, 0.11, 0.01, 0.23, 1023110000, 1023110010 ]
```

and `$time_in_dbi` will be 0.93;

The second argument need not be just leaf nodes. If given a reference to a hash then the hash is recursively searched for leaf nodes and all those found are merged.

For example, to get the time spent 'inside' the DBI during an http request, your logging code run at the end of the request (i.e. mod_perl LogHandler) could use:

```
my $time_in_dbi = 0;
if (my $Profile = $dbh->{Profile}) { # if DBI profiling is enabled
$time_in_dbi = dbi_profile_merge_nodes(my $total=[], $Profile->{Data});
$Profile->{Data} = {}; # reset the profile data
}
```

If profiling has been enabled then `$time_in_dbi` will hold the time spent inside the DBI for that handle (and any other handles that share the same profile data) since the last request.

Prior to DBI 1.56 the *dbi_profile_merge_nodes()* function was called *dbi_profile_merge()*. That name still exists as an alias.

CUSTOM DATA COLLECTION

Using The Path Attribute

```
XXX example to be added later using a selectall_arrayref call
XXX nested inside a fetch loop where the first column of the
XXX outer loop is bound to the profile Path using
XXX bind_column(1, \${ $dbh->{Profile}->{Path}->[0] })
XXX so you end up with separate profiles for each loop
XXX (patches welcome to add this to the docs :)
```

Adding Your Own Samples

The *dbi_profile()* function can be used to add extra sample data into the profile data tree. For example:

```
use DBI;
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);
```

The `$h` parameter is the handle the extra profile sample should be associated with. The `$statement` parameter is the string to use where the Path specifies !Statement. If `$statement` is

undef then `$h->{Statement}` will be used. Similarly `$method` is the string to use if the Path specifies !MethodName. There is no default value for `$method`.

The `$h->{Profile}{Path}` attribute is processed by *dbi_profile()* in the usual way.

The `$h` parameter is usually a DBI handle but it can also be a reference to a hash, in which case the *dbi_profile()* acts on each defined value in the hash. This is an efficient way to update multiple profiles with a single sample, and is used by the DashProfiler module.

SUBCLASSING

Alternate profile modules must subclass DBI::Profile to help ensure they work with future versions of the DBI.

CAVEATS

Applications which generate many different statement strings (typically because they don't use placeholders) and profile with !Statement in the Path (the default) will consume memory in the Profile Data structure for each statement. Use a code ref in the Path to return an edited (simplified) form of the statement.

If a method throws an exception itself (not via RaiseError) then it won't be counted in the profile.

If a HandleError subroutine throws an exception (rather than returning 0 and letting RaiseError do it) then the method call won't be counted in the profile.

Time spent in DESTROY is added to the profile of the parent handle.

Time spent in DBI->*() methods is not counted. The time spent in the driver connect method, `$drh->`*connect()*, when it's called by DBI->connect is counted if the DBI_PROFILE environment variable is set.

Time spent fetching tied variables, `$DBI::errstr`, is counted.

Time spent in FETCH for `$h->{Profile}` is not counted, so getting the profile data doesn't alter it.

DBI::PurePerl does not support profiling (though it could in theory).

For asynchronous queries, time spent while the query is running on the backend is not counted.

A few platforms don't support the *gettimeofday()* high resolution time function used by the DBI (and available via the *dbi_time()* function). In which case you'll get integer resolution time which is mostly useless.

On Windows platforms the *dbi_time()* function is limited to millisecond resolution. Which isn't sufficiently fine for our needs, but still much better than integer resolution. This limited resolution means that fast method calls will often register as taking 0 time. And timings in general will have much more 'jitter' depending on where within the 'current millisecond' the start and end timing was taken.

This documentation could be more clear. Probably needs to be reordered to start with several examples and build from there. Trying to explain the concepts first seems painful and to lead to just as many forward references. (Patches welcome!)