



PERCONA  
Performance Consulting Experts

---

# Instrumenting PHP and Memcache applications for performance optimization.

Peter Zaitsev  
Justin Swanhart

# Introduction

---

- Peter Zaitsev
  - CEO, Percona Inc.
- Justin Swanhart
  - Principal Consultant, Percona Inc.

# What are we going to talk about?

---

- What does instrumentation mean?
- Why is it important?
- What do you instrument?
- How do you do it?
- How to make effective use of the gathered information.

# What does instrumentation mean?

Instrumentation is the branch of mechanical engineering that deals with measurement and control.

All cars give you some basic information

- How fast am I going?
- How far have I gone?
- At what rate am I consuming fuel?
- What is the engine temperature?
- Do I need oil?

# Software Instrumentation

Instrumenting an application means adding the ability to quantitatively measure the time it takes to perform some operation in your application.

Your app can provide basic information too

- CPU usage
- Memory usage
- Total wall-clock time
- Database access time and number of requests
- Cache layer access time and number of requests
- Other wait times such as IO time
- Session information and a unique request identifier

# Instrument for optimization

## Goal driven performance optimization

- You must observe and measure your application if you want to establish goals regarding its performance.
- Services like MySQL and Memcache impact your application in addition to the resources used on the actual web server.
- Choose optimization targets which most help you reach your goals. Prioritize optimizations which will help meet more than one goal.
- Optimization rarely decreases complexity. Avoid adding complexity by only optimizing what is necessary to meet your goals.

# Instrument for capacity planning

---

## If you understand your resource utilization levels

- It is easier to estimate how many resources you will use as application usage grows.
- Some resources are probably more utilized than others. Focus on making sure that the real bottlenecks are being addressed. You may need to increase capacity for only one portion of your application.
- Focusing on the right resources means more value for your customers, your shareholders and your employees.

# What do we measure?

## The response time for every request

- To the end user, response time is everything.
- Using good logging we can analyze the data effectively

## When possible measure the response time of all sub requests

- Database
- Cache
- Expensive calculations (uuid)
- Other: template rendering time, session create/resume time



# Where do I get started?

## PHP

- Implement resource counters and augment SQL queries
- Use PHP features to track CPU and memory usage
- Populate Apache environment with counters

## Why start here?

- Because without the information from the application, the other pieces won't be useful.
- Instrumenting the application is the most time consuming part.

## Combine the information from PHP with other systems

- MySQL slow query log contains augmented SQL queries

# Use an Instrumentation class

We've created a demonstration class to build from:

- It follows the singleton pattern:
  - `$instance = Instrumentation::get_instance();`
- It should be started at the beginning of the request:
  - `Instrumentation::get_instance()->start_request();`
- It provides counters (set, increment, append,get):
  - `Instrumentation::get_instance()->increment('ctr_name');`
- It augments (adds comments to) queries:
  - `$query = $instance->instrument_query($query);`
- And it exports counters to the Apache environment automatically.

# The Instrumentation singleton

```
Class Instrumentation {  
    /* Intrumentation follows the singleton  
       pattern  
    */  
  
    private static $instance;  
  
    public static function get_instance() {  
        if (!isset(self::$instance)) {  
            $c = __CLASS__;  
            self::$instance = new $c;  
        }  
        return self::$instance;  
    }  
  
    /* Private constructor */  
    private function __construct() {  
        ...  
    }  
}
```

# Call start\_request at the very start

## The start\_request public method:

- Captures a snapshot of CPU utilization with getrusage()
- Captures a snapshot of memory with memory\_get\_usage()
- Sets up counters for MySQL and memcache
- Registers a shutdown function
  - » Calculates total memory and cpu usage
  - » Exports counters with apache\_setenv

# If you are using PHP Sessions

You can ask `start_request` to call `session_start()`

- `Instrumentation::get_instance()->start_request(true);`
- This will ensure that time to create the session is accounted for in your application:

You can automatically capture session variables into counters

- `::get_instance()->start_request(true, array('uname'));`
- A `session_uname` counter will be populated

Both calls populate the `session_create_time` counter.

# Capture MySQL Information

How you capture the MySQL information depends on how you are executing queries

- If you are using the OO mysqli interface, extend it and replace `$obj=new mysqli(...);` with `$obj=new mysqli_x(...);`
- If you are using the functional MySQL interfaces (`mysqli_query`) then create a new custom function and call it instead.
- If you have another data access layer then modify it, or extend it to increment the appropriate counters.
- Use a tool like Eclipse for easy refactoring.

# Example MySQLi Extension

/\* Additional code for ->query(), ::mysqli\_query, ->multi\_query(), ::mysqli\_multi\_query()  
can be found in the example class. \*/

```
class MySQLi_perf extends MySQLi {  
    /* Object interface constructor */  
    public function __construct($host=NULL, $user=NULL, $pass=NULL, $db=NULL){  
        if($host === NULL) ini_get("mysql.default_host");  
        if($user === NULL) ini_get("mysql.default_user");  
        if($pass === NULL) ini_get("mysql.default_password");  
        Instrumentation::get_instance()->increment('mysql_connection_count');  
        parent::__construct($host, $user, $pass, $db);  
    }  
  
    /* emulate functional mysqli_connect interface */  
    public static function mysqli_connect($host=NULL, $user=NULL, $pass=NULL,  
                                         $db=NULL, $port=NULL, $socket=NULL)  
    {  
        Instrumentation::get_instance()->increment('mysql_connection_count');  
        return mysqli_connect($host, $user, $pass, $db, $port, $socket);  
    }  
}
```

# Using the example

Using the OO interface:

OLD: \$obj = new MySQLi(...);

NEW: \$obj = new MySQLi\_perf(...);

Using the functional interface:

OLD: \$conn = mysqli\_connect(...)

NEW: \$conn = MySQLi\_perf::mysqli\_connect(...);

Using the old mysql interface:

\$conn = MySQL\_perf::mysql\_connect(...);



# Augmenting SQL queries

Have you ever looked at SHOW PROCESSLIST and wondered

- What page is sending that query?
- What user of my application initiated this query?
- Which server issued this query (if using a proxy)?

# debug\_backtrace

```
1. <?php
2.
3. function my_function($arg) {
4.   print_r(debug_backtrace());
5. }
6.
7. my_function(1);
```

```
Array
(
    [0] => Array
        (
            [file] => /tmp/test.php
            [line] => 7
            [function] => my_function
            [args] => Array
                (
                    [0] => 1
                )
        )
    )
)
```

# Augmenting SQL queries

```
public function instrument_query(  
$query_sql="", $keys = array('request_id', 'session_id', 'SESSION_uname'))  
{  
    $query_header = "";  
    if ($query_sql) {  
        /* the first frame is the original caller */  
        $frame = array_pop(debug_backtrace());  
        #Add the PHP source location  
        $query_header = "-- File: {$frame['file']}\tLine:  
{ $frame['line'] }\tFunction: { $frame['function'] }\t";  
        foreach($keys as $x => $key) {  
            $val = $this->get($key);  
            if($val) {  
                $val = str_replace(array("\t", "\n", "\0"), "", $val); /* all other  
chars are safe in comments */  
                $key = strtolower(str_replace(array(": ", "\t", "\n", "\0"), "", $key));  
                #Add the requested instrumentation keys  
                $query_header .= "\t{$key}: {$val}";  
            }  
        }  
    }  
    return $query_header . "\n" . $query_sql;  
}
```

# Example Augmented SQL

The query comment consists of key value pairs:

```
-- File: index.php Line: 118 Function: fullCachePage request_id: ABC session_id: XYZ
select Socialeventid,
       Title,
       Summary,
       Imagethumburl,
       Createdtimestamp,
       Eventdate,
       Submitterusername
from SOCIALEVENT
where eventtimestamp>=CURRENT_TIMESTAMP
ORDER BY eventdate ASC limit 0,10;
```

mk-query-digest can use the pairs as attributes:

```
--embedded-attributes '^-- [^\n]+'(\w+): ([^\t]+)'
```

# Apache

- MOD\_LOG\_CONFIG is very flexible
- Environment variables can be written to logs
- Apache 2 provides %D to record time in microseconds
- If total time (%D) is high, and PHP counters are low, then a slow link may be responsible, or there is possibly CPU resource contention on your web server (long run queue).

# What do we want in the logs?

## Some examples of useful information

- Session ID
- A unique identifier per request
- The number of MySQL queries/connections per request
- Count of Memcache operations/connections per request
- The time in microseconds to service the entire request
- The user identifier of the user issuing the request
- The total amount of time spent servicing memcache and SQL requests

# Get information into Apache logs

## MOD\_LOG\_CONFIG is used to set up custom logs

- It can record information from the Apache environment
- You can log specific information to different log files based on the information contained in the Apache environment
- Apache 2 can include the total response time for the request in microseconds, which is very important.
- It can write custom formatted timestamps, like mysql timestamps

# An example Apache configuration

```
SetEnvIf Request_URI \.php instrumented # SET instrumented to ON for PHP reqs
#
# Instrumented application logging
# May be loaded with LOAD DATA INFILE
# Use %D for request time instead of %T because %D is microseconds
# mod_logio is required for %I %O
# memcache counters omitted for brevity
LogFormat "\"%{%Y-%m-%d %H:%M:%S}t\" %a %I %O %D %f %H %m \"%q\" %>s %V
%{CTR_total_cpu_time}e %{CTR_cpu_user}e %{CTR_cpu_system}e %
{CTR_memory_usage}e \"%{CTR_request_id}e\" \"%{CTR_SESSION_uname}e\" \"%
{CTR_session_id}e\" %{CTR_mysql_query_count}e %{CTR_mysql_prepare_count}e %
{CTR_mysql_prepare_time}e %{CTR_mysql_connection_count}e %
{CTR_mysql_query_exec_time}e performance

#regular access log for all requests
CustomLog logs/access_log common

#performance data goes in this log ONLY for PHP apps (see SetEnvIf above)
CustomLog logs/performance_log performance env=instrumented
```



# MySQL

## Percona extensions to MySQL slow log

- `long_query_time=N` #zero=all, .001, .01,.1, 5, 10
- `slow_query_log=on` #on|off
- `slow_query_log_file=slow.log`
- `log_slow_verbosity=full` #full, innodb,microsecond,query\_plan
- `slow_query_rate_limit=N` #only log every Nth session

Maatkit's `mk-query-digest` can be used to analyze the slow query log

- It supports embedded attributes in SQL queries, which the example query augments provides.
- It can record stats from the slow log into review tables

`LOAD DATA INFILE` can be used to load the Apache performance log into a table

# Store the Apache log in a table

```
CREATE TABLE `performance_log` (  
  `access_time` datetime DEFAULT NULL,  
  `remote_address` varchar(25) DEFAULT NULL,  
  `bytes_in` bigint(20) unsigned DEFAULT NULL,  
  `bytes_out` bigint(20) unsigned DEFAULT NULL,  
  `service_time` bigint(20) DEFAULT NULL,  
  `file` varchar(100) DEFAULT NULL,  
  `protocol` char(10) DEFAULT NULL,  
  `action` char(10) DEFAULT NULL,  
  `query_string` text,  
  `status` smallint(5) unsigned DEFAULT NULL,  
  `virtualhost` varchar(50) DEFAULT NULL,  
  `total_cpu_time` float DEFAULT NULL,
```

# Why you want to use a log file

---

Don't use a logging mechanism which can significantly impact the performance of your application.

Databases are great for history and ad-hoc analysis, but when you want to see what is going on “right now” `tail -f` gets results faster

# LOAD DATA INFILE

```
mysql> load data
      infile '/var/log/httpd/performance_log'
      into table performance_log
      fields terminated by ' '
      optionally enclosed by '"';
Query OK, 5471 rows affected, 1946 warnings (0.12 sec)
Records: 5471  Deleted: 0  Skipped: 0  Warnings: 0
```

```
mysql> show warnings;
```

Level	Code	Message
Warning	1366	Incorrect integer value: '-' for column 'memory_usage' at row 366

....

# Create a view for ease of use

```
CREATE ALGORITHM=MERGE VIEW `performance_view`  
AS SELECT  
    `mysql_query_count` + `mysql_prepare_count` AS `mysql_ops`,  
from `performance_log`  
mysql> desc performance_view;
```

Field	Type	Null	Key	Default	Extra
access_time	datetime	YES		NULL	
remote_address	varchar(25)	YES		NULL	
bytes_in	bigint(20) unsigned	YES		NULL	
bytes_out	bigint(20) unsigned	YES		NULL	
service_time	bigint(20)	YES		NULL	
file	varchar(100)	YES		NULL	
protocol	char(10)	YES		NULL	
action	char(10)	YES		NULL	
query_string	text	YES		NULL	
status	smallint(5) unsigned	YES		NULL	
virtualhost	varchar(50)	YES		NULL	
total_cpu_time	float	YES		NULL	
cpu_user	float	YES		NULL	
cpu_system	float	YES		NULL	
memory_usage	int(10) unsigned	YES		NULL	
request_id	char(40)	YES		NULL	
SESSION_uname	varchar(25)	YES		NULL	
session_id	char(32)	YES		NULL	
mysql_ops	int(9)	YES		NULL	
mysql_connection_count	mediumint(9)	YES		NULL	
mysql_time	double	YES		NULL	
memcache_connection_count	mediumint(9)	YES		NULL	
memcache_ops	bigint(15)	YES		NULL	
memcache_time	double	YES		NULL	

24 rows in set (0.00 sec)

# Look at the data

```
create or replace view file_performance_day
as
select date(access_time) day,
       file,
       count(*) cnt,
       sum(bytes_in) / 1024 kb_in,
       sum(bytes_out) / 1024 kb_out,
       sum(bytes_in) / 1024 / 1024 mb_in,
       sum(bytes_out) / 1024 / 1024 mb_out,
       round(sum(service_time / 1e6),4) service_time,
       round(sum(total_cpu_time),4) total_cpu_time,
       round(sum(cpu_user),4) cpu_user,
       round(sum(cpu_system),4) cpu_system,
       sum(memory_usage) / 1024 / 1024 memory_usage_MB,
       round(sum(mysql_time),4) mysql_time,
       sum(mysql_ops) mysql_ops,
       sum(memcache_ops) memcache_ops,
       round(sum(memcache_time),4) memcache_time,
       sum(mysql_connection_count) mysql_connection_count,
       sum(memcache_connection_count) memcache_connection_count,
       round( sum(mysql_time) / sum(service_time/1e6) * 100, 2) mysql_pct,
       round( sum(memcache_time) / sum(service_time/1e6) * 100, 2) memcache_pct,
       round( sum(cpu_user + cpu_system) / sum(service_time/1e6) * 100, 2) cpu_pct,
       round( ( sum(service_time/1e6) - ( sum(mysql_time) + sum(memcache_time) + sum(cpu_user +
cpu_system) ) ) / sum(service_time / 1e6),4) * 100 other_pct
from performance_view
group by day,file
```

# What is the slowest page?

```
mysql> select * from file_performance_day order by service_time desc limit 1\G
```

```
***** 1. row *****
```

```
      day: 2010-04-11
```

```
      file:
```

```
      /var/www/html/oliophp/public_html/taggedEvents.php
```

```
      cnt: 858
```

```
      kb_in: 203.4248
```

```
      kb_out: 14249.8896
```

```
      mb_in: 0.19865704
```

```
      mb_out: 13.91590786
```

```
      service_time: 265.7764
```

```
      total_cpu_time: 13.5320
```

```
      cpu_user: 7.6168
```

```
      cpu_system: 5.9151
```

```
      memory_usage_MB: 161.22107697
```

```
      mysql_time: 252.2444
```

```
      mysql_ops: 4805
```

```
      memcache_ops: 0
```

```
      memcache_time: 0.0000
```

```
      mysql_connection_count: 1566
```

```
      memcache_connection_count: 0
```

```
      mysql_pct: 94.91
```

```
      memcache_pct: 0.00
```

```
      cpu_pct: 5.09
```

```
      other_pct: 0.0000
```

```
1 row in set (0.09 sec)
```

# MySQL slow? Lets see why

Mk-query-digest can help us find queries from only the file we are looking before *because we augmented the queries.*

```
mk-query-digest /var/lib/mysql/slow --embedded-attributes '^-- [^\n]+' '(\w+): ([^\t]+)'
```

```
-- File: index.php Line: 118 Function: fullCachePage request_id: ABC session_id: XYZ
```



# Use mk-query-digest!

Use mk-query-digest to filter on only the file in question:

```
mk-query-digest /var/lib/mysql/slow --embedded-attributes '^-- [^\n]+','(\w+): ([^\t]+)' --filter '$event->{File} && $event->{File} =~ m/taggedEvent/'
```

Only four queries:

#	Rank	Query ID	Response time	Calls	R/Call	Item
#	1	0x27A1E0E70C50976A	29.1573	51.7%	708	0.0412 SELECT SOCIALEVENT PERSON_SOCIALEVENT
#	2	0xFB3C3A24510F4D17	14.4064	25.5%	858	0.0168 SELECT SOCIALEVENTTAG SOCIALEVENTTAG_SOCIALEVENT
#	3	0xD5E65E7DEAA2876A	6.5342	11.6%	429	0.0152 SELECT SOCIALEVENTTAG
#	4	0xE00FB796CC8A5183	6.3538	11.3%	2810	0.0023 SELECT SOCIALEVENT