

DTrace support in MySQL: guide to identifying common performance problems

Alexey.Kopytov@Sun.com

May 12, 2009

- A dynamic tracing framework
- No need to restart application
- No special builds
- Used for debugging, troubleshooting and performance analysis
- Support in Solaris, OpenSolaris, Mac OS X, FreeBSD and Linux (in development).

How does it work?

- Probe is an instrumentation point
- Probe “fires” when execution flow reaches probe location
- May have optional arguments describing the context
- No overhead for inactive probes
 - Solaris kernel: ~36,000 probes
 - Solaris kernel + libraries: ~50,000 probes
 - Applications: Java VM, Apache, PHP, Ruby, **MySQL**, ...

Probe naming conventions

- `probe_id := provider:module:function:name`

`provider`: Collection of related probes (syscall, io, mysql)

`module`: Executable or library name

`function`: Function name in the source code

`name`: Probe name (query-start)

- Example:

- `syscall::read:entry`

Using probes

Scripts in **D** language:

- specify probes we want to trace
- describe actions to be run when a probe fires

Example

reads.d

```
syscall::read:entry
/execname == "mysqld"/
{
    printf("%s: read(%d, 0x%x, %d)\n",
           execname, arg0, arg1, arg2);
}
```

```
# dtrace -qs ./reads.d
mysqld: read(31, 0x4ef1018, 16)
mysqld: read(31, 0x4ef1018, 4)
mysqld: read(31, 0x4ef1018, 4)
mysqld: read(31, 0x4ef1018, 20)
^C
```

Dynamic probes

- Dynamic probes:
 - created on-the-fly
- 'pid' provider:
 - does not require any support from an application
 - dynamic entry/return probes for every function
 - function arguments are available in a script as `arg0`, `arg1`, ...

Example

```
mysql_parse.d
```

```
pid$target::*mysql_parse*:entry
{
    printf("%s\n", copyinstr(arg1))
}
```

```
# dtrace -p 'pgrep mysqld' -qs ./mysql_parse.d
BEGIN
SELECT c from sbtest where id=5042
...
```

Static probes

Static probes:

- must be added into application source code
- may have arguments not available from dynamic ones

Example

sql/sql_parse.cc

```
bool dispatch_command(...)
{
    switch (command) {
        case COM_QUERY:
            MYSQL_QUERY_START(query, thread_id, db, user, host);
            /* Query processing starts here */
            ...
    }
}
```

```
# dtrace -qn 'mysql*:::query-start \
{ printf("%s\n", copyinstr(arg0)) }'
BEGIN
SELECT c from sbtest where id=5042
...
```

Comparison of static and dynamic probes

Pros and cons of dynamic and static probes		
	Dynamic	Static
Work with any MySQL version	yes	no
Stable interface	no	yes
Require knowledge of the source code	yes	no
Coverage	high	low

MySQL static probes

~60 probes, 200 locations in MySQL 5.4/6.0:

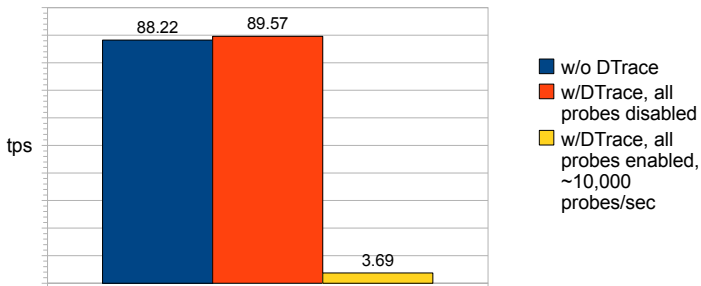
- connections
- Query Cache
- MyISAM key cache
- network I/O
- SQL query execution:
 - parsing
 - table locks
 - row operations in storage engines
 - filesort
- individual query types (SELECT, INSERT, UPDATE, ...)

Overhead

sysbench, OLTP read-only, 1 thread, all-in-memory

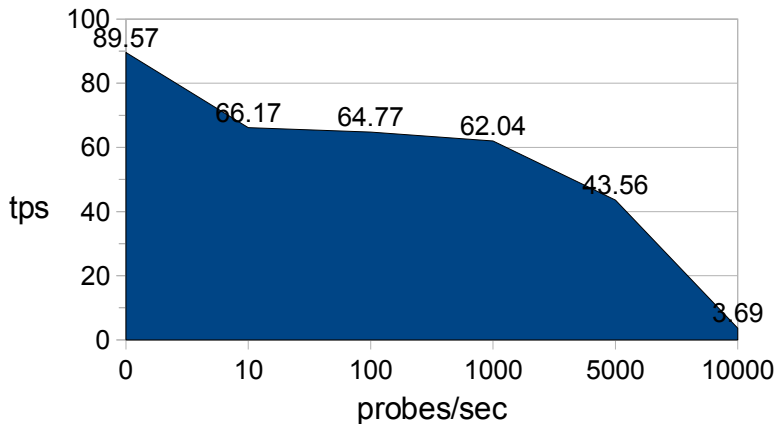
D script

```
mysql*:::  
{  
  @nprobes = count();  
}
```



- inactive probes do have zero overhead
- 24x slowdown for 10,000 probes/sec

Overhead



- ~25% slowdown with rates < 1000 probes/sec
- sharp performance drop with rates > 1000 probes/sec

Examples

Query tracing: storage engine operations

Measuring time spent in a storage engine for each query.

Row operation probes:

```
read-row-start/done  
index-read-row-start/done  
update-row-start/done  
delete-row-start/done  
insert-row-start/done
```

Query tracing: storage engine operations

D script

```
mysql$target:::query-start,
{
    self->query = copyinstr(arg0);
    self->query_start = timestamp;
    self->engine = 0;
}

mysql$target:::*-row-start
{
    self->engine_start = timestamp;
}

mysql$target:::*-row-done
{
    self->engine = self->engine + timestamp - self->engine_start;
}

mysql$target:::query-done,
/ self->query != 0 /
{
    printf("%s\n", self->query);
    printf("Total: %dus Engine: %dus\n",
           (timestamp - self->query_start) / 1000,
           self->engine / 1000,
           self->query = 0;
}
```

Query tracing: network I/O

Measuring time spent on network operations.

Network I/O probes:

```
net-read-start/done  
net-write-start/done
```

D script

```
...  
mysql$target:::net-*-start  
{  
    self->net_start = timestamp;  
}  
  
mysql$target:::net-*-done  
{  
    self->network = self->network + timestamp - self->net_start;  
}  
...
```

Query tracing: table locks and filesort

Measuring time spent on table locks and filesort.

Table locks probes:

```
handler-rdlock-start/done  
handler-wrlock-start/done  
handler-unlock-start/done
```

Filesort probes:

```
filesort-start/done
```

D script

```
...  
mysql$target::handler-*lock-start  
{  
    self->lock_start = timestamp;  
}  
  
mysql$target::handler-*lock-done  
{  
    self->locks = self->locks + timestamp - self->lock_start;  
}  
  
mysql$target::filesort-start  
{  
    self->filesort_start = timestamp;  
}  
  
mysql$target::filesort-done  
{  
    self->filesort = timestamp - self->filesort_start;  
}  
...
```

Putting it all together

Results

```
# dtrace -qp 'pgrep mysqld' -s ./query_snoop.d
```

```
INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number,  
                        ol_i_id, ol_supply_w_id, ol_delivery_d,  
                        ol_quantity, ol_amount, ol_dist_info)
```

```
VALUES (3302, 3, 8, 4, 32055, 8, NULL, 5, 106.500000, 'anYwJuccgMvV
```

```
*** Total: 595us Locks: 63us Engine: 378us Network: 101us Filesort: 0us
```

```
SELECT i_price, i_name, i_data
```

```
FROM item
```

```
WHERE i_id = 28416
```

```
*** Total: 753us Locks: 106us Engine: 462us Network: 123us Filesort: 0us
```

Hot tables: table reads

D script

table_reads.d

```
mysql$target:::index-read-row-start,  
mysql$target:::read-row-start  
{  
    @tables[copyinstr(arg0), copyinstr(arg1)] = count();  
}  
  
END  
{  
    printa("%s.%s %u\n", @tables);  
}
```

Probes used:

```
read-row-start(db, table, scan_flag)  
index-read-row-start(db, table)
```

Hot tables: table reads

Results

```
# dtrace -qp 'pgrep mysqld' -s ./table_reads.d  
^C  
dbt2w10.orders 15490  
dbt2w10.warehouse 15990  
dbt2w10.district 33152  
dbt2w10.item 57559  
dbt2w10.customer 78256  
dbt2w10.order_line 219413  
dbt2w10.stock 277777  
dbt2w10.new_order 4421952
```

Hot tables: table updates

D script

table_updates.d

```
mysql$target:::insert-row-start ,
mysql$target:::update-row-start ,
mysql$target:::delete-row-start
{
    @tables[copyinstr(arg0), copyinstr(arg1)] = count();
}

END
{
    printa("%s.%s %@u\n", @tables);
}
```

Probes used:

```
insert-row-start(db, table)
update-row-start(db, table)
delete-row-start(db, table)
```

Hot tables: table updates

Results

```
# dtrace -qp 'pgrep mysqld' -s ./table_updates.d
```

```
^C
```

```
dbt2w10.orders 12247
```

```
dbt2w10.warehouse 12676
```

```
dbt2w10.district 26291
```

```
dbt2w10.item 45762
```

```
dbt2w10.customer 60011
```

```
dbt2w10.order_line 159498
```

```
dbt2w10.stock 211085
```

```
dbt2w10.new_order 3479681
```

Queries accessing a particular table

D script

table_snoop.d

```
mysql$target:::query-exec-start
{
    self->query = copyinstr(arg0);
    self->used = 0;
}

mysql$target:::handler-*lock-start
/ copyinstr(arg0) == $$1 && copyinstr(arg1) == $$2 /
{
    self->used = 1;
}

mysql$target:::query-exec-done
/ self->query != 0 && self->used == 1 /
{
    printf("%s;\n", self->query);
    self->query = 0;
    self->used = 0;
}
```

Probes used

```
query-exec-start(query, connid, db_name, user, host, exec_type)
query-exec-done(status)
handler-rdlock-start(db, table)
handler-wrlock-start(db, table)
handler-unlock-start(db, table)
```

Queries accessing a particular table

Results

```
# dtrace -qp 'pgrep mysqld' -s ./table_snoop.d \  
  dbt2w10 new_order  
SELECT no_o_id  
FROM new_order  
WHERE no_w_id = 9  
  AND no_d_id = 10;  
DELETE FROM new_order  
WHERE no_o_id = 2197  
  AND no_w_id = 9  
  AND no_d_id = 10;  
INSERT INTO new_order (no_o_id, no_w_id, no_d_id)  
VALUES (3251, 2, 2);  
...
```

Stored procedures tracing

- Need to find slow/ineffective queries within a store routine
- Slow query log only shows the CALL statement for the stored procedure, not the individual queries

D script

sp_trace.d

```
mysql$target:::query-exec-start
/ arg5 == 3 /
{
  self->query_start = timestamp;
  self->query = copyinstr(arg0);
}

mysql$target:::query-exec-done
/ self->query != 0 /
{
  printf("%uus %s\n", (timestamp - self->query_start) / 1000,
        self->query);
  self->query = 0;
}
```

Probes used

```
query-exec-start(query, connid, db_name, user, host, exec_type)
/* exec_type = 3 for stored routines queries */
query-exec-done(status)
```

Stored procedures tracing

Results:

```
# dtrace -qp 'pgrep mysqld' -s ./sp_trace.d
523us SELECT no_o_id
      INTO tmp_o_id
      FROM new_order
      WHERE no_w_id = NAME_CONST('in_w_id',1) AND no_d_id =
      NAME_CONST('tmp_d_id',1) limit 1
77us DELETE FROM new_order
      WHERE no_o_id = NAME_CONST('tmp_o_id',2381)
      AND no_w_id = NAME_CONST('in_w_id',1)
      AND no_d_id = NAME_CONST('tmp_d_id',1)
...
```

What's next?

- more engine-specific probes
- I/O probes (`syscall` and `io` are too limited with background I/O threads)
- optimizer
- replication

- **Section in the Reference Manual:**
“5.7. Tracing mysqld Using DTrace”.
<http://dev.mysql.com/doc/refman/6.0/en/dba-dtrace-server.htm>
- **DTrace documentation in Wiki:**
<http://wikis.sun.com/display/DTrace/>
- **Feature preview tree:**
`bzr branch lp:~akopytov/mysql-server/mysql-6.0-dtrace`
- Alexey.Kopytov@Sun.com

Thank you!