

MySQL University

January 2009

MySQL and DTrace

Martin MC Brown

**Technical Writer, MySQL & Software Infrastructure
Group**

Sun Microsystems

Today's Topics

- DTrace and MySQL
 - > Quick DTrace Overview
 - > What you can do without static DTrace Probes
 - > The MySQL Static Probes
 - > What you can do with the DTrace Probes
 - > Upcoming Probes
 - > Future

Quick DTrace Overview

- Dynamic Tracing
- Allows tracing of processes live
- Doesn't require special builds, debugger, or other invasive techniques
- Completely passive
 - > Don't use it, no performance hit
 - > Use it, small performance hit (dependent on detail)
- Trace *anything*
 - > *Application*
 - > *Kernel*
 - > *System libraries*

How to use it

- Tracing is handled either dynamically from the command line
- Or you use a 'D' script
 - > Scripting allows for more advanced probes
 - > Selective processing
 - > Formatting/Calculations

One liners

- Identifying probes:
 - > provider:module:function:name
 - > provider - name of provider (application, kernel, etc.)
 - > module - module within provider
 - > function - name of function call to monitor
 - > name - name of probe (inc. entry, return)
- See queries entering the parser:
 - > `dtrace -qn 'pid$target::*mysql_parse*:entry { printf("%Y %s\n", walltimestamp, copyinstr(arg1)); }' -p `pgrep -x mysqld``
 - > 2009 Feb 12 13:22:52 select * from t4 limit 1
 - > 2009 Feb 12 13:23:00 insert into t4 values (1,'hello')

Getting more detail

```
#pragma D option quiet

pid$target::*mysql_parse*:entry
{
    self->query = copyinstr(arg1);
}

pid$target::*mysql_execute_command*:entry
{
    self->start = timestamp;
}

pid$target::*mysql_execute_command*:return
/self->start/
{
    this->elapsed = timestamp - self->start;
    @time[self->query] = quantize(this->elapsed);
    self->query = 0;
    self->start = 0;
}

dtrace:::END
{
    printf("MySQL Query execution latency (ns):\n");
    printa(@time);
}
```

Results

```
dtrace -s t2.d -p `pgrep -x mysqld`
```

```
^C
```

```
MySQL Query execution latency (ns):
```

```
insert into t4 values (1, 'hello')
```

value	Distribution	count
32768		0
65536	@@	4
131072	@@@@@@@@	1
262144		0

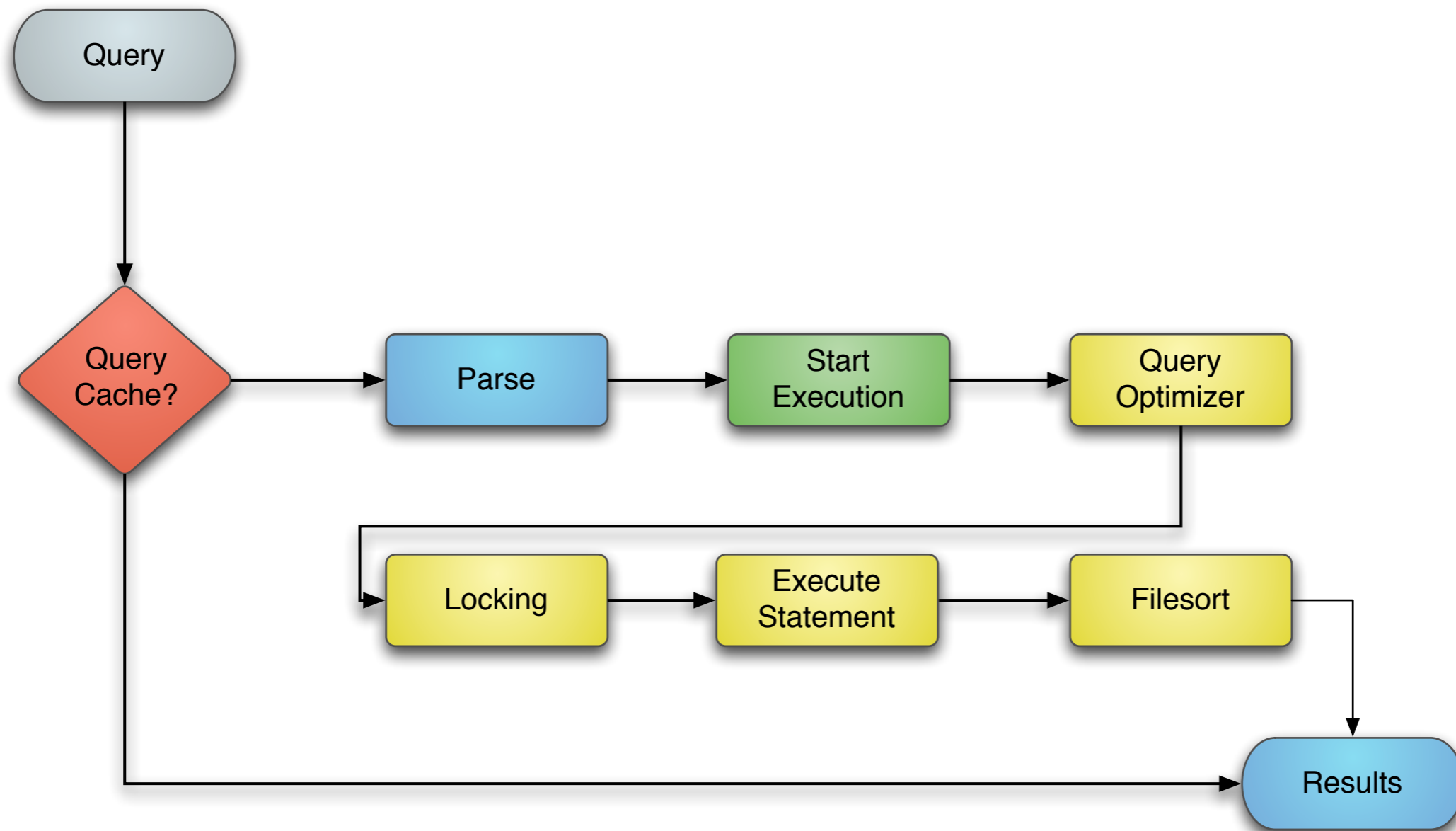
```
select * from t4
```

value	Distribution	count
134217728		0
268435456	@@	1
536870912		0

Becoming a provider

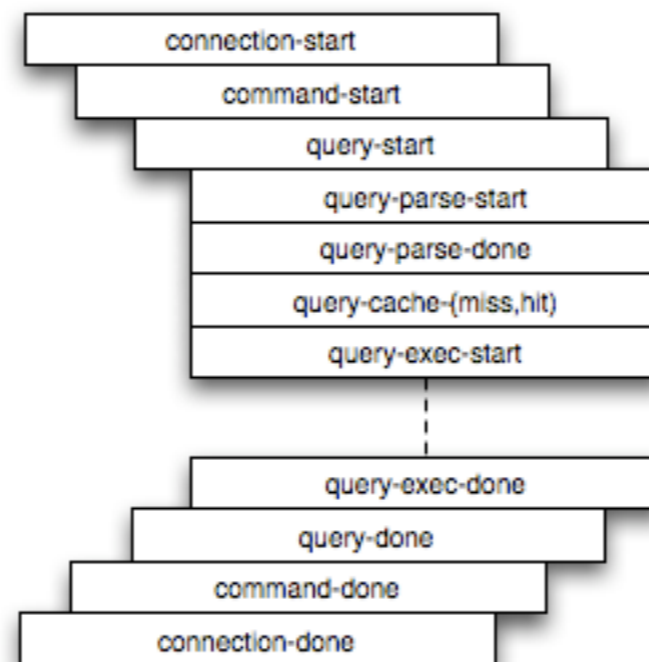
- Tracing without knowing the underlying structure is hard
 - > You need to know functions, names, what they do
 - > Even harder is extracting the info you need
- Static probes are added into the source code
- Provide easier hooks into the information you want
 - > Identify specific operations
 - > Specific information (query, host, client, user)
 - > Much easier to use

How MySQL Executes a Query



Probe Sets

- Probes in MySQL are organized into sets
- Most are part of the execution sequence with the level getting deeper
- But designed so you can go in most levels and get what you want



Probe Sets

- Query
- Query Parsing
- Query Cache
- Query Execution
- Locks
- Statements
- Row-Level
- Filesort
- Network

Key Points

- Providing key information
 - > user, host
 - > connection-start
 - > query-start
 - > query-exec-start
 - > database, query
 - > query-start
 - > query-exec-start
 - > Query cache
 - > query-cache-hit and query-cache-miss
 - > Filesort
 - > filesort-start and filesort-done

Getting Execution Times

- query-start(query, connectionid, database, user, host)
 - > query - query text (arg0)
 - > connectionid - MySQL process ID (arg1)
 - > database - DB name (arg2)
 - > user - user name (arg3)
 - > host - client host (arg4)
- query-done(status)
- Combine with the built-in timestamp to get some execution times

Getting Execution Times Example

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
    printf("%-20s %-20s %-40s %2s %-9s\n", "Who", "Database", "Query", "QC", "Time(ms)");
}
mysql*:::query-start
{
    self->query = copyinstr(arg0);
    self->connid = arg1;
    self->db     = copyinstr(arg2);
    self->who    = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
    self->querystart = timestamp;
    self->qc = 0;
}
mysql*:::query-cache-hit
{
    self->qc = 1;
}
mysql*:::query-cache-miss
{
    self->qc = 0;
}
mysql*:::query-done
{
    printf("%-20s %-20s %-40s %-2s %-9d\n",self->who,self->db,self->query,(self->qc ? "Y" : "N"),
        (timestamp - self->querystart) / 1000000);
}
```

Results

Who	Database	Query	QC	Time (ms)
root@localhost	test	select i,s from t1 order by s,i limit 100	N	47571
root@localhost	test	select i,s from t1 order by s desc,i limit 100	N	53360
root@localhost	test	show create table t1	N	0
root@localhost	test	create index t1b on t1 (s,i)	N	1979127
root@localhost	test	select i,s from t1 order by s desc,i limit 100	N	53776

Getting More Detail

- Find out how much time is spent parsing
- Time spent purely *executing* statement
- Time spent in locks
- Time spent transferring data
- Time spent doing a filesort

Let's look at locks

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

mysql*:::handler-rdlock-start
{
    self->rdlockstart = timestamp;
    this->lockref = strjoin(copyinstr(arg0),strjoin("@",copyinstr(arg1)));
    self->lockmap[this->lockref] = self->rdlockstart;
    printf("Start: Lock->Read   %s.%s\n",copyinstr(arg0),copyinstr(arg1));
}

mysql*:::handler-unlock-start
{
    self->unlockstart = timestamp;
    this->lockref = strjoin(copyinstr(arg0),strjoin("@",copyinstr(arg1)));
    printf("Start: Lock->Unlock %s.%s (%d ms lock duration)\n",
           copyinstr(arg0),copyinstr(arg1),
           (timestamp - self->lockmap[this->lockref])/1000000);
}

mysql*:::handler-rdlock-done
{
    printf("End:   Lock->Read   %d ms\n",
           (timestamp - self->rdlockstart)/1000000);
}
```

Results

```
Start: Lock->Write test.t4
End:   Lock->Write 0 ms
Start: Lock->Read  test.t4
End:   Lock->Read  0 ms
End:   Lock->Unlock 0 ms
Start: Lock->Unlock test.t4 (6320 ms lock duration)
End:   Lock->Unlock 0 ms
Start: Lock->Unlock test.t4 (6319 ms lock duration)
```

Statement Probes

- Get detailed information on the results of different statements
 - > SELECT
 - > INSERT
 - > INSERT ... SELECT
 - > UPDATE
 - > UPDATE t1,t2,t#
 - > DELETE
 - > DELETE ... t1,t2,t#
- Provides rows affected for all probes
- For update, adds rows matching clause

Aggregating time per statement

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
mysql*:::update-start, mysql*:::insert-start,
mysql*:::delete-start, mysql*:::multi-delete-start,
mysql*:::multi-delete-done, mysql*:::select-start,
mysql*:::insert-select-start, mysql*:::multi-update-start
{
    self->querystart = timestamp;
}
mysql*:::select-done
{
    @statements["select"] = sum(((timestamp - self->querystart)/1000000));
}
mysql*:::insert-done, mysql*:::insert-select-done
{
    @statements["insert"] = sum(((timestamp - self->querystart)/1000000));
}
mysql*:::update-done, mysql*:::multi-update-done
{
    @statements["update"] = sum(((timestamp - self->querystart)/1000000));
}
mysql*:::delete-done, mysql*:::multi-delete-done
{
    @statements["delete"] = sum(((timestamp - self->querystart)/1000000));
}
tick-30s
{
    printa(@statements);
}
```

Results

delete	0
update	0
insert	23
select	2484
delete	0
update	0
insert	39
select	10744
delete	0
update	26
insert	56
select	10944
delete	0
update	26
insert	2287
select	15985

Row-level (and Storage Engine)

- Get row-by-row times
 - > Before pushed down to engine
 - > From the engine itself (future release)
- Current probes monitor time spent on row-by-row basis for INSERT/UPDATE/DELETE
- Newer probes ask the storage engine:
 - > read-row-start/done
 - > delete-row-start/done
 - > ...

Example Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
dtrace:::BEGIN
{
    printf("%-2s %-10s %-10s %9s %9s %-s \n",
           "St", "Who", "DB", "ConnID", "Dur ms", "Query");
}
mysql*:::query-start
{
    self->query = copyinstr(arg0);
    self->who   = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
    self->db    = copyinstr(arg2);
    self->connid = arg1;
    self->querystart = timestamp;
    self->rowdur = 0;
}
mysql*:::query-done
{
    this->elapsed = (timestamp - self->querystart) /1000000;
    printf("%2d %-10s %-10s %9d %9d %s\n",
           arg0, self->who, self->db,
           self->connid, this->elapsed, self->query);
}
mysql*:::query-done
/ self->rowdur /
{
    printf("%34s %9d %s\n", "", (self->rowdur/1000000), "-> Row ops");
}
}
```

Results

```
0 root@localhost test          6      3409 insert into t4 (select * from t4)
                                2812 -> Row ops
1 root@localhost test          6          0 update t4 set i = 100 if i>5
0 root@localhost test          6    10465 update t4 set i=100 where i>5
                                9872 -> Row ops
```

Probing everything...

```

St  QC  Who          DB              ConnID  Dur ms  Bytes  Matched  Changed Query
--
 3   0  root@localhost  test            6              0         0         0         Command start
                                     Parsing: insert into t4 (select *
from t4)
 0                                     -> Parsing Complete
 0                                     -> Write lock: t4
 0                                     -> Write lock completed
 0                                     -> Read lock: t4
 0                                     -> Read lock completed
 0                                     -> Unlock completed
 0                                     -> Lock duration for: t4
 0                                     -> Unlock: t4
 0                                     -> Unlock completed
33  0  root@localhost  test            6   36521    0      400944    0  insert into t4 (select * from t4)
                                     -> Net read
                                     -> Net write
                                     -> Row ops
 0                                     -> Row ops
 0                                     Command done
--
                                     -> Lock duration for: t4
                                     -> Unlock: t4

```

Combining Probes

- Static instrumentation currently only goes so far
- By combining static probes and functional probes we can go deeper

Getting byte counts

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
    printf("%-20s %-20s %-40s\n", "Bytes Read", "Bytes Written", "Query");
}

mysql*:::query-start
{
    self->query = copyinstr(arg0);
    self->byteswritten = 0;
    self->bytesread = 0;
}

pid$target::*my_write*:entry
{
    self->byteswritten += arg2;
}

pid$target::*my_read*:entry
{
    self->bytesread += arg2;
}

mysql*:::query-done
{
    printf("%-20d %-20d %-40s\n",
        self->bytesread,
        self->byteswritten,
        self->query);
}
```

Results

```
./io.d -p `pgrep -x mysqld`
```

Bytes Read	Bytes Written
1251	132340061
264662639	201160506
496	1024
0	0
2500857	2500000

Query

```
create table t4 (select * from t2)
create index t4a on t4 (i)
truncate t4
insert into t4 (select * from t2 limit 100000)
insert into t4 (select * from t2 limit 100000)
```

More Info

- Look at DTrace toolkits
 - > iosnoop monitors disk usage
 - > <http://www.context-switch.com/performance/dtrace.htm>
- Brendan Gregg's website:
 - > <http://blogs.sun.com/brendan/category/DTrace>
- OpenSolaris DTrace group:
 - > <http://opensolaris.org/os/community/dtrace/dtracetoolkit/>
- My Blog:
 - > <http://coalface.mcslp.com>
- MySQL Docs:
 - > <http://dev.mysql.com/doc/refman/6.0/en/dba-dtrace-server.html>

Availability

- OpenSolaris/Solaris Compatible Probes in 6.0.8
- Extended set of probes coming in 6.0.10
 - > Including Mac OS X Support
- Extended probes will be in the MySQL 5.1.30 provided with OpenSolaris
- I'm looking into some build issues in FreeBSD

Where Next

- Go deeper into Storage Engines
- Get statistics on global server operations
- Get statistics on general locks and structures
- Get probes into other parts of the Webstack

Questions

Martin 'MC' Brown

**Technical Writer, MySQL & Software Infrastructure
Group**

Sun Microsystems

mc.brown@sun.com