Deep-inspecting MySQL with DTrace

Domas Mituzas, Sun Microsystems

Me

- MySQL Senior Support Engineer @ Sun
- Doing performance engineering for Wikipedia, develop performance accounting tools
- Don’t like waste

DTrace

- Lovely basic use (with tools and toolkits)
- Advanced capabilities:
  - Debugger
  - Profiler
  - Tracer
  - Database :-)
  - Memory editor

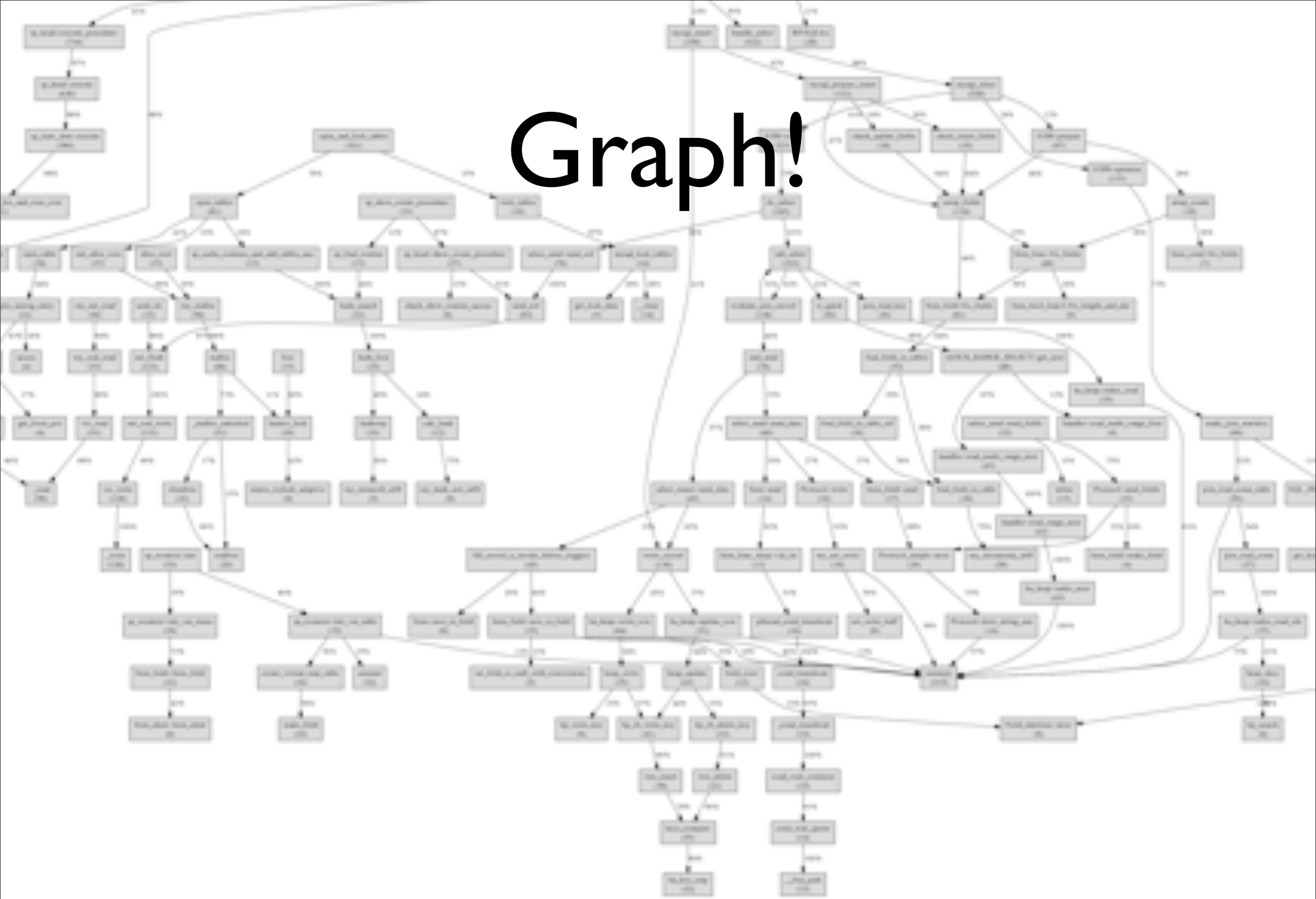
Basics

- dtrace -p $(pidof mysqld) -s script.d
- $target gets replaced by -p value
- probe /predicate/ { action }

Easy stuff

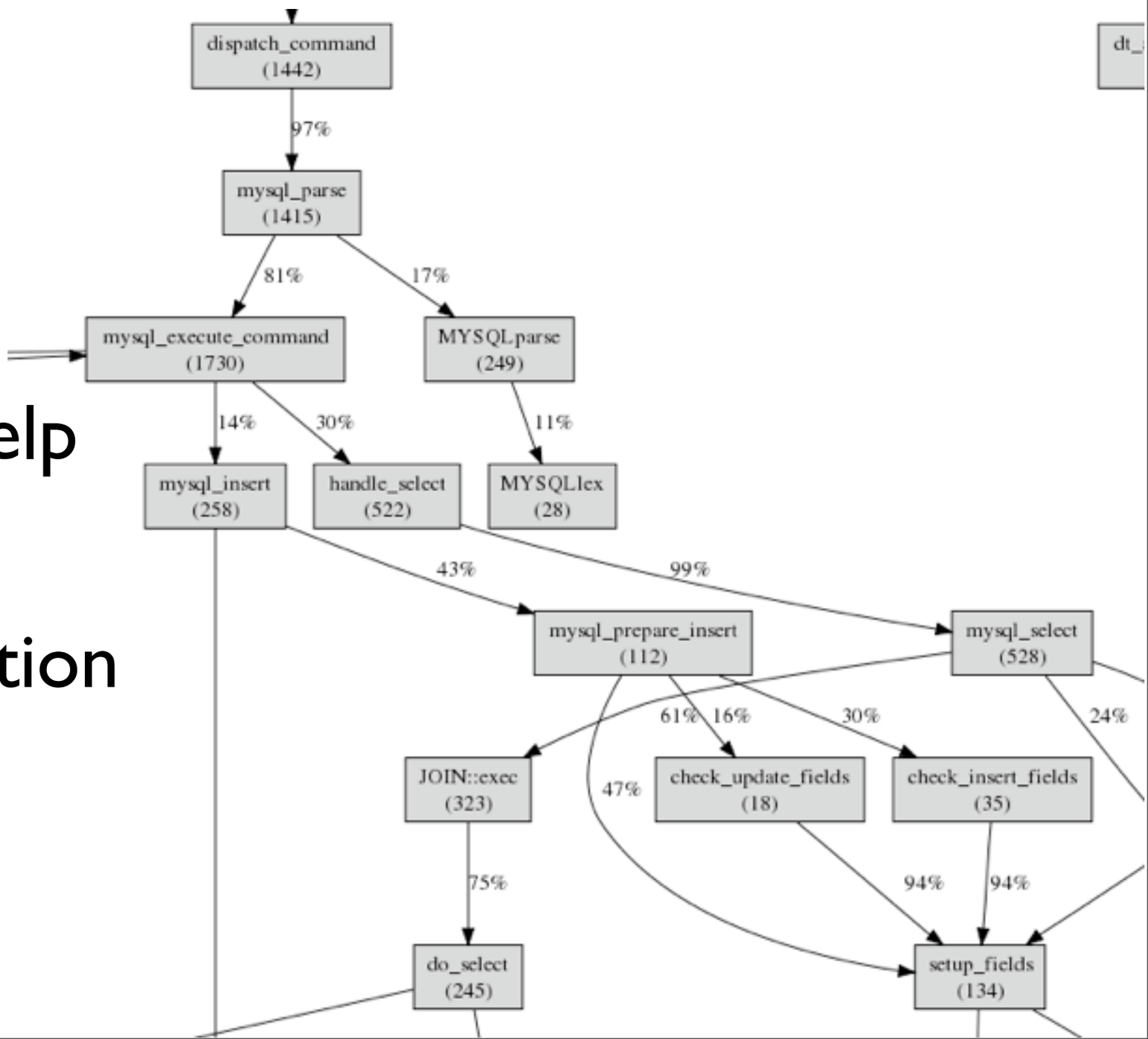
- Profiler
  - profile-997 { a[ustack() = count() }
  - Simple sampler, useful output
  - Gives lots of stacks
  - Fails on MacOSX (DRM protection)
  - http://dammit.lt/tech/dtrace/callgraph

Graph!



Graph!!!

- Function names
- # of samples
- % of total
- Colors might help
- Can be any DTrace aggregation



Probes

- Have very limited scripting abilities
- Can be fired depending on environment or other probe actions
- Can access and set thread local or global variables

Data

- Simple scalars are easy
- Timestamps are in nanoseconds
- Structures and strings have to be copied in (need definitions to do anything useful with structures)
- Aggregations are trivial (but fast)

plockstat

- Provider, telling about pthread locks
- Userland utility visualizing that data
- plockstat -V:

```
plockstat$target:::mutex-spun
/self->mtxspin[arg0] && arg1 != 0/
{
        @mtx_spin[arg0, ustack()] =
            quantize(timestamp - self->mtxspin[arg0]);
        self->mtxspin[arg0] = 0;
        mtx_spin_found = 1;
}
```

- Doesn't account for internal application-implemented locks (like scalability patches)

pid provider

- Ultimate deep inspection
- Ability to access function arguments and CPU register values
- Probes at function entry/return points or anywhere in between (hex offsets)

MySQL!!! !! !

- Open source - and we are going to use that
- Threaded - needs some more care
- High performance - needs even more care
- Has DTrace probes in newer versions, but one can avoid them

Tracing, first steps

- dtrace -F -n pid123:mysqld:*:*r* | c++filt

```
...
        -> free_root
          -> my_no_flags_free
          <- my_no_flags_free
         <- free_root
       <- mysql_reset_errors(THD*, bool)
       -> check_table_access(THD*, unsigned long, TABLE_LIST*, unsigned int, bool)
         -> get_schema_table_idx(st_schema_table*)
         <- get_schema_table_idx(st_schema_table*)
         -> check_grant(THD*, unsigned long, TABLE_LIST*, unsigned int, unsigned int, bool)
         <- check_grant(THD*, unsigned long, TABLE_LIST*, unsigned int, unsigned int, bool)
       <- check_table_access(THD*, unsigned long, TABLE_LIST*, unsigned int, bool)
       -> execute_sqlcom_select(THD*, TABLE_LIST*)
...
```

Limited tracing

- Show InnoDB handler calls:
- dtrace -F -n pid$(pidof mysqld):mysqld:*ha_inno*:*r* | c++filt
- Trace InnoDB row fetches:
- dtrace -p $(pidof mysqld) -s trace.d

```
pid$target::*ha_innobase*general_fetch*:entry { self->trace=1; }
pid$target::*ha_innobase*general_fetch*:return { self->trace=0; }
pid$target::*:*r* /self->trace/  {}
```

Static probes

- Static probes can be mixed with dynamic ones
- MySQL probes since 6.0
- Provide user, host, query and other information at various points
- http://dev.mysql.com/doc/refman/6.0/en/dba-dtrace-mysqld-ref.html
- Getting some of the data is difficult otherwise (not impossible)

Extracting the query

- Full query

```
pid$target::*dispatch_command*:entry
{ trace(copyinstr(arg2)) }
```

  - Does not “canonize” anything
- Application (API/abstraction) can help
  - Prepend queries with /* code::point */
  - copyinstr(arg2,30)

THD, user context

- Unfortunately in this case, is a class
- Need manual offset checks to see if one can get values

```
(gdb) print thd.
$20 = (class THD *) 0x1038400
(gdb) print &thd->main_security_ctx->host
$21 = (char **) 0x1038d14
(gdb) print 0x1038d14-0x1038400
$22 = 2324
(gdb) print &thd->main_security_ctx->user
$23 = (char **) 0x1038d18
```

- We can copy in some pointers

Dark arts: Copying in!

```
pid$target::*do_command*:entry
{
   /* Save pointers for clarity */
   this->hostp = (uintptr_t *)copyin(arg0+2324,4);
   this->userp = (uintptr_t *)copyin(arg0+2328,4);
   /* save for later re-use */

   self->user=copyinstr(*this->userp);
   self->host=copyinstr(*this->hostp);
}
```

Dark arts: copying out!

```
#!/usr/sbin/dtrace -Fws
int *zero;
BEGIN { zero=alloca(4); *zero=0; }
pid$target:mysqld:add_to_status*:1b {
 copyout(zero,uregs[R_ECX],4);
 copyout(zero,uregs[R_EDX],4);
}
```

(had to disassemble a_t_s for this one, sorry)

Lots of data

- Aggregations:
  - count() = useful for sampling or pure overview
  - sum() = sum up bytes, time, etc
  - quantize(), lquantize() = distribution analysis
  - by function, stack, user, combinations, anything
  - Often need frontends

Getting interesting data

- Start with libc call (or a syscall):
  - pid$target::read:entry
    { @a[ustack()]=count() }
- Pick interesting function
  - fil_io - InnoDB file ops
- Check time at entry and return
- Aggregate!

InnoDB iotop

```
pid$target::fil_io:entry {
  self->io = timestamp; }

pid$target::fil_io:return /self->io/ {
  @thrs[tid] = sum(timestamp - self->io); self->io=0; }

pid$target::read:entry, pid$target::pread:entry /self->io/ { @reads[tid] = sum(arg2);}
pid$target::pwrite64:entry /self->io/ { @writes[tid] = sum(arg2); }

/* This can be done in profile-1, would have to reset aggregations then and clear screen */
END {
printf ("\n");
printf ("Time spent:\n "); printa(@thrs);
printf ("Bytes read:\n "); printa(@reads);
printf ("Bytes written:\n "); printa(@writes);
}
```

More data!

- Aggregate by any data one has at hands - tables, users, queries, etc
- Look at times, data amount, distribution, distribution by stack

```
pid$target::fil_io:entry { self->io = timestamp;}
pid$target::fil_io:return /self->io/ {
  @thrs[ustack(5)] = quantize(timestamp - self->io); self->io=0; }
```

```
              mysqld`fil_io+0x428
              mysqld`log_write_up_to+0x42a
              mysqld`trx_commit_off_kernel+0x328
              mysqld`trx_commit_for_mysql+0xd4
              mysqld`_Z19innobase_commit_lowP10trx_struct+0x1c

           value  ------------- Distribution ------------- count
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
           65536 |@@@@@@@@@@@@@@@                          1
          131072 |                                         0
```

Sightseeing

- Handler interfaces are easy to trace
- Network code (vio_*)
- Mutexes - plockstat & application layers

Follow the white rabbit

- No ultimate probes are written at once
- Keep process iterative - follow the clues, get new ones

What takes it so long

- timestamp - real time
- vtimestamp - CPU time
- hold events - understand the work being done within
- wait events - understand why one has to wait - scope, who is holding, etc
- some long calls don't cost (like network), some do (like IO), some are critical sections (with mutexes attached)

mysqlshowslowprofile

- Initialize profile variables for every do_command entry (start time, lock time, handler, network, etc)
- Set self-> variables for each profiled area
- Print the values if query took long enough

Recording areas

```
pid$target::_Z11lock_tables*:entry /self->in_query==1/
{ self->lock_start_time = timestamp; }

pid$target::_Z11lock_tables*:return / self->in_query==1 / {
self->lock_time += (timestamp - self->lock_start_time ) / 1000;
self->lock_start_time = 0; }

pid$target::*net_real_write*:entry / self->in_query==1 /
{ self->network_write_start_time = timestamp; }
pid$target::*net_real_write*:return / self->in_query==1 /
{
        self->network_write_time += (timestamp - self->network_write_start_time ) / 1000;
        self->network_write_start_time = 0;
}
```

Are we slow? Print!

```
pid$target::*do_command*:return
/ (timestamp - self->start_time) > long_query_time && self->in_query==1 /
{
        self->total_time = timestamp - self->start_time;
        @total["Total slow queries"] = count();
        printf("Date/time: %Y\n", walltimestamp);
        printf("Query: %s\n",self->query);

        printf("Lock time: %d microseconds\n", self->lock_time);
        printf("Network write time: %d microseconds\n", self->network_write_time);
        printf("Total time: %d microseconds\n", self->total_time / 1000);

        printa("Other threads running: %@d\n\n", @queries);
}
```

Of course...

- DTrace is platform specific (for now):
  - Solaris, OpenSolaris, Nexenta, MacOSX, FreeBSD?
- Google patches add easier long-term accounting
  - DTrace is system inspection, not accounting framework
- performance schema will be interesting too
- Still, none as powerful and flexible

Limitations

- No loops
- No conditional execution (except ?:)
- No array ‘flattening’, only way to see arrays is using aggregations:
  - sum(-1) to deallocate
- Slow to attach to debug builds (

Getting big

- #pragma D option dynvarsize=512m
- bufsize, aggsize - if drops occur, raise up
- http://wikis.sun.com/display/DTrace/Options+and+Tunables
- http://wikis.sun.com/display/DTrace/Buffers+and+Buffering

DTraceToolkit

- Lots of useful examples
- More about system inspection
- Need DTraceMySQLToolkit (especially for dynamic structures)

Questions?

- domas at sun dot com
- http://dammit.lt/
- http://sun.com/