Design and Implementation of a Real-Time Cloud Analytics Platform

OSCON Data 2011

David Pacheco (@dapsays)

Brendan Gregg (@brendangregg)
Agenda

The Problem
Cloud Analytics
Demo
Experiences
Cloud Performance

• The problem: we’ve deployed our software, and now performance sucks.
  • How do we figure out why?

• Focus on source of the pain: latency
  • How long a synchronous operation takes
  • ... while a client is waiting for data
  • ... while a user is waiting for a page to load

• How do you summarize the latency of thousands of operations?
  • ... without losing important details?

• How do you summarize that across a distributed system?

• How do you do this in real time?
**Latency: event-by-event**

```
# ./iosnoop -Dots

<table>
<thead>
<tr>
<th>STIME</th>
<th>TIME</th>
<th>DELTA</th>
<th>DTIME</th>
<th>UID</th>
<th>PID</th>
<th>D</th>
<th>BLOCK</th>
<th>SIZE</th>
<th>COMM</th>
<th>PATHNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>949417936651</td>
<td>949417948636</td>
<td>11984</td>
<td>11999</td>
<td>104</td>
<td>29008</td>
<td>R</td>
<td>99310470</td>
<td>16384</td>
<td>mysqld</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>949418267667</td>
<td>949418275701</td>
<td>8033</td>
<td>8052</td>
<td>104</td>
<td>29008</td>
<td>R</td>
<td>1947809</td>
<td>16384</td>
<td>mysqld</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>949418843669</td>
<td>949418843808</td>
<td>139</td>
<td>156</td>
<td>0</td>
<td>3 W</td>
<td>29024</td>
<td>2048</td>
<td>fsflush</td>
<td>/var/log/...</td>
<td></td>
</tr>
<tr>
<td>949418873385</td>
<td>949418873488</td>
<td>103</td>
<td>121</td>
<td>0</td>
<td>3 W</td>
<td>6695855</td>
<td>2048</td>
<td>fsflush</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949418873564</td>
<td>949418873617</td>
<td>52</td>
<td>57</td>
<td>0</td>
<td>3 W</td>
<td>1829584</td>
<td>512</td>
<td>fsflush</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949418921970</td>
<td>949418932931</td>
<td>10960</td>
<td>10976</td>
<td>104</td>
<td>29008</td>
<td>R</td>
<td>95362430</td>
<td>16384</td>
<td>mysqld</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>949419684613</td>
<td>949419692319</td>
<td>7706</td>
<td>7723</td>
<td>104</td>
<td>29952</td>
<td>R</td>
<td>81475146</td>
<td>16384</td>
<td>mysqld</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>949419693574</td>
<td>949419699461</td>
<td>5886</td>
<td>5906</td>
<td>104</td>
<td>29952</td>
<td>R</td>
<td>60593276</td>
<td>16384</td>
<td>mysqld</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>949422857833</td>
<td>949422857981</td>
<td>148</td>
<td>168</td>
<td>0</td>
<td>3 W</td>
<td>26720</td>
<td>4096</td>
<td>fsflush</td>
<td>/var/adm/...</td>
<td></td>
</tr>
<tr>
<td>949423846191</td>
<td>949423846355</td>
<td>163</td>
<td>181</td>
<td>0</td>
<td>3 W</td>
<td>1990648</td>
<td>4096</td>
<td>fsflush</td>
<td>/var/log/...</td>
<td></td>
</tr>
<tr>
<td>949426420134</td>
<td>949426420265</td>
<td>130</td>
<td>151</td>
<td>0</td>
<td>0 R</td>
<td>400</td>
<td>8192</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420346</td>
<td>949426420423</td>
<td>77</td>
<td>85</td>
<td>0</td>
<td>0 W</td>
<td>65</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420367</td>
<td>949426420459</td>
<td>92</td>
<td>35</td>
<td>0</td>
<td>0 W</td>
<td>129</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420386</td>
<td>949426420490</td>
<td>103</td>
<td>30</td>
<td>0</td>
<td>0 W</td>
<td>146</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420404</td>
<td>949426420566</td>
<td>161</td>
<td>76</td>
<td>0</td>
<td>0 W</td>
<td>193</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420530</td>
<td>949426420604</td>
<td>73</td>
<td>37</td>
<td>0</td>
<td>0 W</td>
<td>206</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
<tr>
<td>949426420547</td>
<td>949426420679</td>
<td>131</td>
<td>75</td>
<td>0</td>
<td>0 W</td>
<td>210</td>
<td>512</td>
<td>sched</td>
<td>&lt;none&gt;</td>
<td></td>
</tr>
</tbody>
</table>

[...thousands of lines...]

- Lots of data to sift through; effective as a last resort
Latency: average

- Some patterns more visible; outlier hidden
- x-axis = time, y-axis = average latency
Great! This example is MySQL query latency

- x-axis = time, y-axis = latency, **z-axis (color saturation) = count of queries**
Latency: heatmap, sliced and diced

- Even better! 4th dimension (color hue) represents different database tables
- x-axis = time, y-axis = latency, **color hue = table**, color saturation = count
Agenda

The Problem

Cloud Analytics

Demo

Experiences
Cloud Analytics

- Key building blocks
  - DTrace
  - OS-level virtualization
  - Node.js
Building blocks: DTrace

• Facility for dynamic instrumentation of production systems
  • Originally developed circa 2003 for Solaris 10, then open-sourced in 2005
  • Available on Solaris-derived OSes (SmartOS, Illumos, etc.)
  • Available on Mac OS X 10.5+, FreeBSD 7.1+, Linux? (http://crtags.blogspot.com)

• Supports arbitrary actions and predicates, in situ data aggregation, dynamic and static tracing of both userland and kernel.

• Designed for safe, ad hoc use in production: concise answers to arbitrary questions
DTrace example: MySQL query latency

MySQL query latency can be measured with a (long) one-liner:

```
# dtrace -n '
mysql*:::query-start { self->start = timestamp; } mysql*:::query-done /self->start/ {
    @["nanoseconds"] = quantize(timestamp - self->start);
    self->start = 0;
}
'
```

```
<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>4096</td>
<td>@</td>
<td>93</td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>16384</td>
<td>@@@</td>
<td>232</td>
</tr>
<tr>
<td>32768</td>
<td>@@@</td>
<td>172</td>
</tr>
<tr>
<td>65536</td>
<td>@@@@@</td>
<td>532</td>
</tr>
<tr>
<td>131072</td>
<td>@@@@@@@@@@@@@</td>
<td>1513</td>
</tr>
<tr>
<td>262144</td>
<td>@@@@@@@@@@@@@</td>
<td>428</td>
</tr>
<tr>
<td>524288</td>
<td>@@@@@@@@@@@@</td>
<td>258</td>
</tr>
<tr>
<td>1048576</td>
<td>@@@@</td>
<td>127</td>
</tr>
<tr>
<td>2097152</td>
<td>@@@@</td>
<td>47</td>
</tr>
<tr>
<td>4194304</td>
<td>@@@@@</td>
<td>20</td>
</tr>
<tr>
<td>8388608</td>
<td>@@@@@</td>
<td>33</td>
</tr>
<tr>
<td>16777216</td>
<td>@@@@@@</td>
<td>9</td>
</tr>
<tr>
<td>33554432</td>
<td>@@@@@@@</td>
<td>0</td>
</tr>
</tbody>
</table>
```
**Building blocks: OS-level Virtualization**

- The Joyent cloud uses OS-level virtualization to achieve high levels of tenancy on a single kernel without sacrificing performance:
  - Allows for transparent instrumentation of all virtual OS instances using DTrace

![Diagram showing OS-level virtualization with SmartOS kernel, virtual OS instances, provisioner, instrumenter, and AMQP agents](image)

**Compute node**
Tens/hundreds per datacenter
Building blocks: Node.js

- node.js is a JavaScript-based framework for building event-oriented servers:

```javascript
var http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(8124, "127.0.0.1");

console.log('Server running at http://127.0.0.1:8124!');
```
The energy behind Node.js

- node.js is a confluence of three ideas:
  - JavaScript’s rich support for asynchrony (i.e. closures)
  - High-performance JavaScript VMs (e.g. V8)
  - Solid system abstractions (i.e. UNIX)
- Because everything is asynchronous, node.js is ideal for delivering scale in the presence of long-latency events
Cloud Analytics

- **configuration service**: manages which metrics are gathered
- **instrumenter**: uses DTrace to gather metric data
  - one per compute node, not per OS instance
  - reports data at 1Hz, summarized in-kernel
- **aggregators**: combine metric data from instrumenters
- **client**: presents metric data retrieved from aggregators
Distributed service

Datacenter headnode

Configuration service

Aggregators
(multiple instances for parallelization)

Compute node

Instrumenter

Compute node

Instrumenter

Compute node

Instrumenter
Step 1: User creates an instrumentation

HTTP user/API request: create instrumentation

Datacenter headnode

Configuration service

AMQP: create

Aggregators
(multiple instances for parallelization)

Compute node

Instrumenter

Compute node

Instrumenter

Compute node

Instrumenter

Wednesday, July 27, 2011
Step 2: Instrumenters report data

Datacenter headnode
- Configuration service
- Aggregators (multiple instances for parallelization)

Compute node
- Instrumenter

AMQP: raw data (repeat @1Hz)

Wednesday, July 27, 2011
Step 3: Users retrieve data

HTTP user/API request: retrieve data

Datacenter headnode

- Configuration service
- HTTP: retrieve data
- Aggregators (multiple instances for parallelization)

Compute node
- Instrumenter

Compute node
- Instrumenter

Compute node
- Instrumenter

Wednesday, July 27, 2011
Inside the instrumenter

- Config service (Node.js)
- Aggregators (Node.js)
- DTrace (kernel)
- AMQP
- Other compute nodes

Instruments:
- "dtrace" backend
- node-libdtrace
- libdtrace

Data flow:
- AMQP to Config service
- Config service to Aggregators
- Data from Aggregators to Instrumenter

Other compute nodes:
- Virtual OS
- Virtual OS

[d] Data flow
**Instrumenter: pluggable backends**

- AMQP daemon, pluggable backends (DTrace, kstat, ZFS, ...)
- Predefined metrics; each plugin registers implementations for each metric
- Backend interface:

  ```
  registerMetric(metric, constructor)  
  ```

  (Invoked by plugin)

  ```
  constructor(metric_info)  
  ```

  Initialize object based on metric, decomposition, predicate, etc.

  ```
  obj.instrument(callback)  
  ```

  Start collecting data (e.g., start DTrace).

  ```
  obj.deinstrument(callback)  
  ```

  Stop collecting data (e.g., stop DTrace).

  ```
  obj.value(callback)  
  ```

  Retrieve current data point (**invoked @ 1Hz**)
• Assemble a D script, compile it, and enable DTrace:

```d
this.cad_prog = mdGenerateDScript(metric, ...)
this.cad_dtr = new mod_dtrace.Consumer();
this.cad_dtr.strcompile(this.cad_prog);
this.cad_dtr.go();
```

• But how do you dynamically generate a D script to support predicates and decompositions?
DTrace backend: simple case

- System calls

```c
syscall:::return
{
        @ = count();
}
```
DTrace backend: with decompositions

• System calls decomposed by application name and latency

```c
syscall:::entry
{
    self->latency0 = timestamp;
}

syscall:::return
/self->latency0 != NULL/
{
    @[execname] =
        llquantize(timestamp - self->latency0, 10, 3, 11, 100);
}

syscall:::return
{
    self->latency0 = 0;
}
```
Generating D scripts

- Number of possible D scripts: exponential with number of possible decompositions
- Need way to automatically generate them
Meta-D uses JSON to describe a family of D scripts differing in predicate and decomposition

```json
{
  module: 'syscall',
  stat: 'syscalls',
  fields: ['hostname', 'zonename', 'execname', 'latency' ... ],
  metad: {
    probedesc: [ {
      probes: ['syscall:::entry'],
      gather: { latency: { gather: 'timestamp', store: 'thread' } }
    }, {
      probes: ['syscall:::return'],
      aggregate: {
        default: 'count()',
        zonename: 'count()',
        hostname: 'count()',
        execname: 'count()',
        latency: 'llquantize($0, 10, 3, 11, 100)'
      }
    }],
    transforms: {
      hostname: '"' + caHostname + '"',
      zonename: 'zonename',
      execname: 'execname',
      latency: 'timestamp - $0',
    }
  }
}
```
DTracing system calls

• Enable: hot-patches system call table entries (redirect into DTrace)
• Disable: revert system call table entries
• Advantages of dynamic tracing:
  • Instruments syscalls in all processes on the system at once
  • The thread is never stopped
  • Zero disabled probe effect
More complex: instrumenting MySQL

• Want to instrument MySQL commands by command name and latency:

```c
mysql:::command-start
{
    self->command0 = lltostr(arg1);
    self->latency0 = timestamp;
}

mysql:::command-done
/((((self->command0 != NULL))) && (((self->latency0 != NULL))))/
{
    @[self->command0] =
        llquantize(timestamp - self->latency0, 10, 3, 11, 100);
}

mysql:::command-done
{
    self->command0 = 0;
    self->latency0 = 0;
}
```
DTracing userland applications

- Userland Statically-Defined Tracing (USDT): developer-defined static probes
  - e.g., mysql*:::command-start
  - maintained as functions and arguments evolve

- How it works:
  - In source, macro expands to DTrace function call
  - During link phase: function calls are replaced with nops and their locations recorded
  - On enable, DTrace replaces nops with trap
  - On disable, revert trap back to nop

- Thread is never actually stopped, but does take a round-trip to the kernel.

- Zero disabled probe effect.

- See also: pid provider
  - extremely powerful, but interface is unstable and requires instrumenting each process
Advantages of DTrace-based analytics

• Combine userland and kernel tracing:
  • heatmap of total time spent in CPU dispatcher queue per HTTP request
  • heatmap of total time spent waiting for filesystem I/O per MySQL query

• Examine activity in all applications at once:
  • heatmap of filesystem latency for all applications on a system, by application name
  • ...and for all systems in a data center

• Zero performance impact when not enabled, small impact when enabled

• No need to restart applications

• Can answer arbitrary performance questions safely in production.
Visualizations

- **Bar charts**: easy
  - Clients request raw data, render a chart
  - e.g., Total number of MySQL queries

- **Stacked bar charts**: easy
  - Clients request raw data for multiple separate data series, render a stacked chart.
  - e.g., Total number of MySQL queries *decomposed by zone name* (each virtual OS instance gets its own set of bars)

- **Heatmaps**: hard(er)
  - Heatmap contains a lot of raw data -- transferring it doesn’t scale.
  - Render the heatmaps server-side
  - Rendering is compute-bound, but generally <40ms per heatmap (often more like 10ms).
    - We use multiple aggregators to parallelize the work.
Visualizations

Client request (HTTP)

Config service
(proxies request to appropriate aggregator)

Raw data request

Raw data

Heatmap request

Heatmap

node-png

node-heatmap

Raw data

Aggregated raw data

Aggregator

Raw data via AMQP
(from all instrumenters)

... (other aggregators)
Agenda

The Problem

Cloud Analytics

Demo

Experiences
Demo

- http://rm.no.de:8001
Agenda

The Problem

Cloud Analytics

Demo

Experiences
Experiences

• Node is solid:
  • All CA components are 100% Node.js (about 85% JavaScript, 15% C++)
  • Although aggregator is compute-bound, scaling it with multiple Node processes was easy

• Weak points:
  • C++ add-ons (no stable ABI and the failure modes are not crisp)
  • Diagnosing failures from the field (no post-mortem debugging)

• Building robust AMQP services is non-trivial (exclusive queue problem)
Filesystem: logical filesystem operations decomposed by latency and application name

<table>
<thead>
<tr>
<th>APPLICATION NAME</th>
<th>COLOR BY:</th>
<th>GRANULARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>pkgin</td>
<td>RANK</td>
<td></td>
</tr>
<tr>
<td>pfexec</td>
<td>LINEAR</td>
<td></td>
</tr>
<tr>
<td>which</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qmgr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uudecode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ttymon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brendan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X-axis: Time, in 1 second increments
Displaying latency from 39.9 microseconds to 322 microseconds
• Thanks!
  • Cloud Analytics: @dapsays, @brendangregg, @bcantrill, @rmustacc
  • Portal and API: @rob_ellis, @notmatt, @kevinykchan, @mcavage
  • OS, Node teams at Joyent

• Check out our blogs at http://dtrace.org/
Resources

- “Instrumenting the real-time web: Node.js, DTrace, and the Robinson Projection”

- “Breaking Down MySQL/Percona Query Latency with DTrace”
  (Brendan Gregg, http://www.percona.com/live/nyc-2011/schedule/sessions/)

- “Visualizing System Latency”
  (Brendan Gregg, http://queue.acm.org/detail.cfm?id=1809426)

- “Visualizations for Performance Analysis”
  (Brendan Gregg, http://www.usenix.org/event/lisa10/tech/tech.html#gregg)

- DTrace Book: http://www.dtracebook.com/

- Our blogs: http://dtrace.org/