SOME CONSIDERATIONS FOR SCALING

Bjorn Freeman-Benson

New Relic

@bjorn_fb
Simplified Architecture

Customer's environment

Our datacenter

New Relic

Wednesday, October 3, 12
Our growth

- In 4½ years, zero to 30,000 accounts...
- ... largest account has 17,000 servers
- ... $58 \times 10^9$ metrics per day
  $(40 \times 10^6$ per minute$)$
- ... 5Tb of data a day
  $(3.5\text{Gb per minute})$


Wednesday, October 3, 12
Lean Startup

As a start-up: first prove that we had something, then scale, but plan to scale

Our First System

- PaaS at Engine Yard
- 8 physical machines with multiple VMs
- Everything in Ruby
- Homegrown load balancer
- Separate processes for each activity
- Perfect for the “Search for Business Model”
1. Every app instance of every customer sends us data every minute
2. Only a subset of customers view the data on any given minute
3. Data has a steep half-life: most interesting data is seconds old
4. Accuracy is essential
The Basics (5)
• Reduce the number of connections to the servers
• F5 buffers requests and handles SSL
#2: Bare metal

- VMs didn’t work well for us
- I/O latency problems
- I/O bandwidth jitter
- Ruby is very memory heavy and VMs don’t handle memory mapping as well as native CPUs
#3: Direct Attached Storage

- MySQL depends on really fast write commits
- Thus we need the disk cache as close to the cpu as possible
#4: No App Servers

- Our high throughput collector processes don’t need app servers so they are native Java apps with an embedded Jetty

<table>
<thead>
<tr>
<th>Aggregator</th>
<th>3.1 ms</th>
<th>598,771 rpm</th>
<th>0.01%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beacon</td>
<td>0.17 ms</td>
<td>990,212 rpm</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

#5: Unicorn

- Every worker shares the socket so there’s no need for a dispatcher.
- Also easy to live-deploy new code - helps with our Continuous Deployment.
Our first agent protocol was quick and dirty: Ruby object serialization and multiple round trips.

Refined: reduce round-trips (package more data into the payload); keep-alive.
• If a service is temporarily unavailable, accumulate and retry

```c
recover_from_communication_error:
/*
 * We were unable to contact the collectors, so we need to add all of this data to
 * time unit's pending data.
 */

nr__log (NRL_DEBUG, "[%s] recovering from communication error..", appname);
nr__close_connection_to_daemon (nrdaemon);

nrthread_mutex_lock (&app->lock); {
    /* merge metrics sets the ->replacement pointers of every metric in both from and
     * to table to NULL. if there are only one from table, the values from from table
     */
    nr__metric_table__merge_metrics_from_to (data->metrics, app->pending_harvest->metrics);
    nr__merge_slow_transactions_from_to (&(data->slow_transactions), &(app->pending_h);
    nr__merge_errors_from_to (&(data->errors), &(app->pending_harvest->errors));
}
Our first customers were small.

Later larger customers stretched our assumptions. We added smart sorting, searching, paging, etc.
ORMs (Rails) are nice but can quickly load too many objects. Do a careful audit of slow code.

<table>
<thead>
<tr>
<th>Slow transactions</th>
<th>Resp. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChartData::MetricChartsController#app_breakdown</td>
<td>435 ms</td>
</tr>
<tr>
<td>12:16 — 30 minutes ago</td>
<td></td>
</tr>
<tr>
<td>Api::V1::DataController#multi_app_data</td>
<td>2,230 ms</td>
</tr>
<tr>
<td>12:16 — 30 minutes ago</td>
<td></td>
</tr>
<tr>
<td>ApplicationController#index</td>
<td>527 ms</td>
</tr>
<tr>
<td>12:16 — 30 minutes ago</td>
<td></td>
</tr>
<tr>
<td>Api::V1::DataController#multi_app_data</td>
<td>1,343 ms</td>
</tr>
<tr>
<td>12:16 — 30 minutes ago</td>
<td></td>
</tr>
<tr>
<td>ApplicationController#index</td>
<td>1,272 ms</td>
</tr>
<tr>
<td>12:16 — 30 minutes ago</td>
<td></td>
</tr>
</tbody>
</table>

Show all slow transactions →
The Clever Stuff (6)

UNITED FEDERATION OF AWESOMENESS
#10: Pre-compute

- Pre-compute expensive queries

### Table Example

<table>
<thead>
<tr>
<th>Component</th>
<th>Response Time</th>
<th>RPM</th>
<th>CPU%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beacon</td>
<td>0.2 ms</td>
<td>935,975 rpm</td>
<td>0.01%</td>
</tr>
<tr>
<td>Beacon 1</td>
<td>0.22 ms</td>
<td>724,634 rpm</td>
<td>0.01%</td>
</tr>
<tr>
<td>Beacon 2</td>
<td>0.14 ms</td>
<td>211,331 rpm</td>
<td>0.00%</td>
</tr>
<tr>
<td>Aggregator</td>
<td>8.8 ms</td>
<td>359,875 rpm</td>
<td>0.07%</td>
</tr>
<tr>
<td>Aggregator 0</td>
<td>1.9 ms</td>
<td>712,324 rpm</td>
<td>0.01%</td>
</tr>
<tr>
<td>Aggregator 1</td>
<td>5.3 ms</td>
<td>921,334 rpm</td>
<td>0.00%</td>
</tr>
<tr>
<td>Aggregator 2</td>
<td>3.8 ms</td>
<td>875,975 rpm</td>
<td>0.01%</td>
</tr>
<tr>
<td>Aggregator 3</td>
<td>4.1 ms</td>
<td>789,875 rpm</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

### Diagram Example

```
  +---+    +----------+    +----------+    +----------+
  |  |    |            |    |            |    |            |
  |  |    |  Beacon 1  |    |  Beacon 2  |    |  Beacon    |
  |  |    +----------+    +----------+    +----------+
  |  |        |            |    |            |    |            |
  |  +--------+----------+    |            |    +----------+
  |              |            |    |            |    |  chi-beacon-1|
  |              |            |    |            |    |               |
  |              |            |    |            |    +----------+
  |              |            |    |            |    |  chi-beacon-2|
  |              |            |    |            |    |               |
  +----------+    +---------------------+    +---------------------+
  |            |    +---------------------+    +---------------------+
  |            |    |  Beacon               |    |  Beacon               |
  +----------+    +---------------------+    +---------------------+
```
#11: Real-time BG

- Background job to roll-up timeslice data: minutes to hours, hours to days
Different data has different characteristics

- Account data is classic relational
- Timeslice data is write-once

Use different database instances for each kind of data

- Different tuning parameters (buffer pools, etc)
- Similar to buddy memory allocation
#13: Non-gc gc

- **Problem:** Deleting rows is expensive (due to table-level locking)
- **Solution:** Don’t delete rows
  - Schema has multiple tables (one per account per time period)
  - Use DROP TABLE for gc
- **Similar to the 100-request restart at amazon.com/obidos in 1999**
#14: Computation in DB

- Natural sharding allows us to push computation into the db
  - Supported by schema
  - Limits number of rows returned
  - Thus allows scripting language (Ruby) to do ‘real’ work
- Opposite of the classical advice of doing nothing in the db

## #15: SSDs

- Our data is either random writes and sequential reads, or sequential writes and random reads
  - Choose sequential reads because of UI
  - Use buffers to help random writes, but...
- Switched to SSDs
  - Writes are same or slight slower
  - Reads are fast, random or sequential
Different processes have different performance characteristics: cpu, memory, i/o, time of day, etc.

Allocate processes to machines to balance the resource requirements.

Instead of “all X processes on M1 and Ys on M2” we balance the machines.
Customers have different data characteristics: size, access patterns, ...

Allocate customers to shards to balance the size and loads on the shards

Required an early architectural decision to allow data split between shards

#17: Moving Customers
Lessons Learned
1. Do the basics
2. Design in some scalability
3. Use the unique characteristics of your app to optimize
4. Buzzwords not needed