# Seventeen Things We Did To Scale Our System


**Bjorn Freeman-Benson**

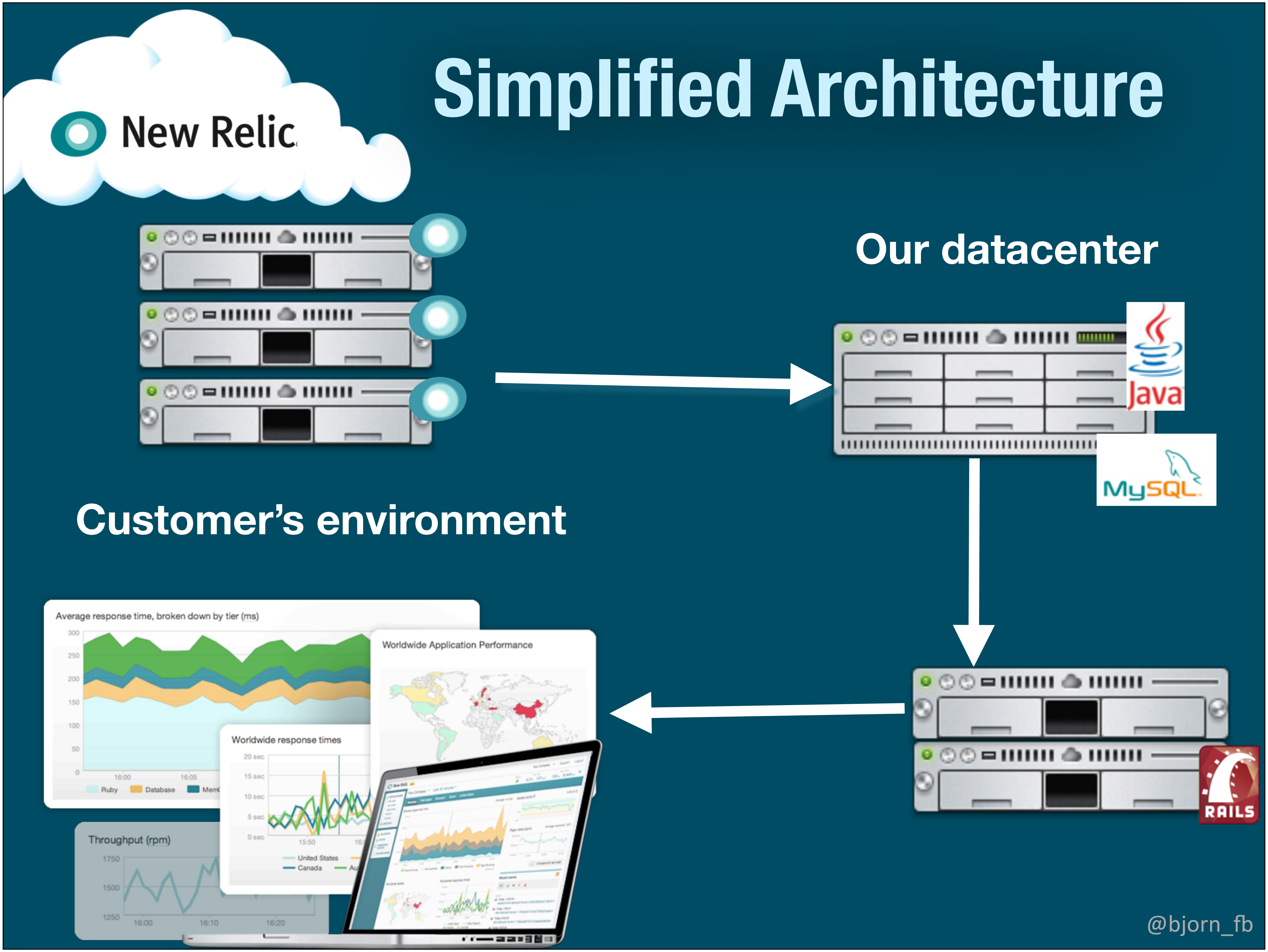
***New Relic***

@bjorn_fb


http://bit.ly/c_sources

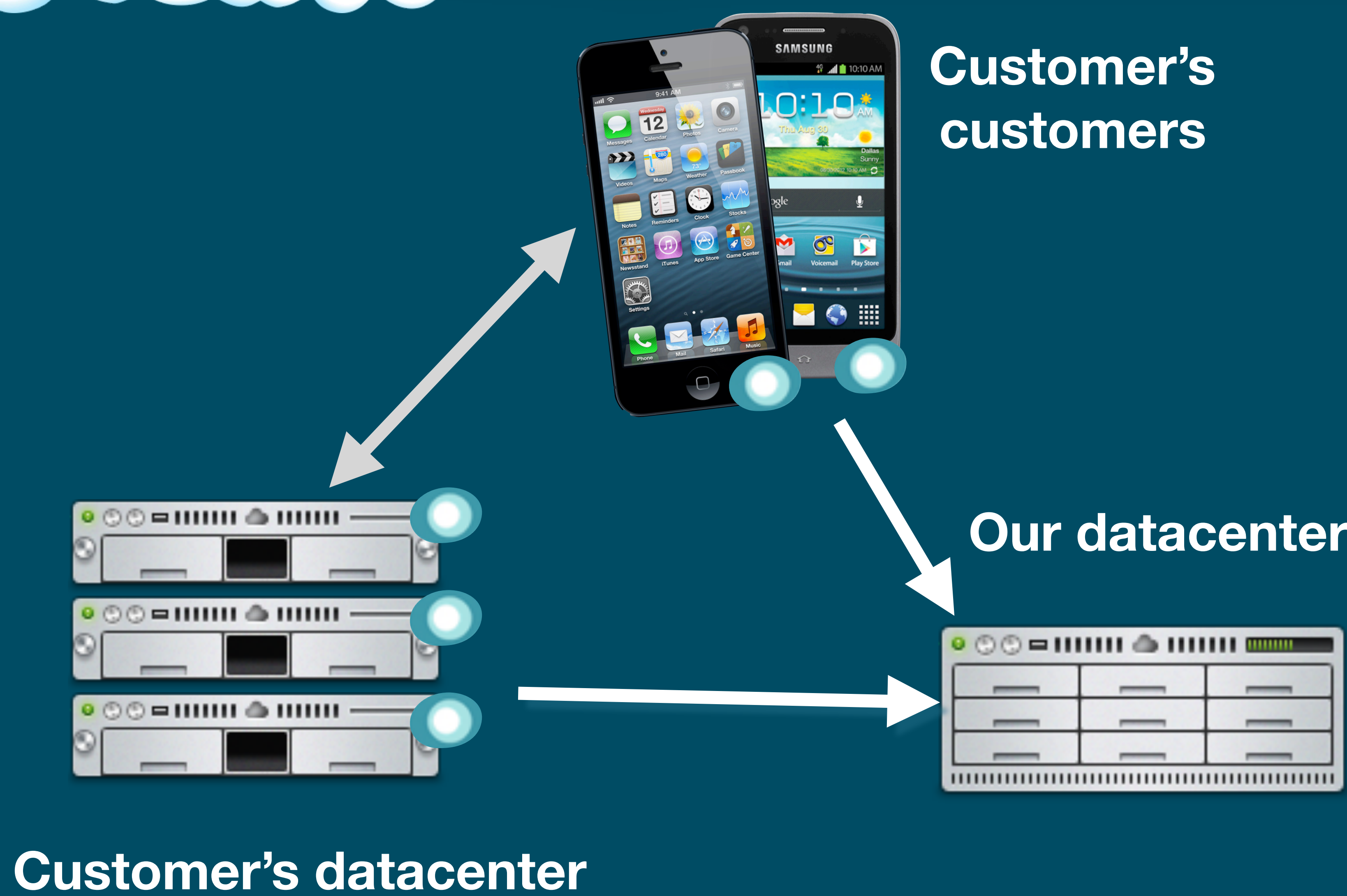
# New Relic


End User 1.9 sec
0.89 [2.4] 346 cpm
RPM UI 226 ms
2 servers
0.91 [0.4] 3,223 cpm
Database 2.9 ms
27.0 : 1 arr 87,074 cpm
Memcache 0.37 ms
1.3 : 1 arr 4,062 cpm
api.heroku.com 325 ms
1 cpm
Sort by Average page load time
Hide < 1% throughput
Countries by average page load time
Germany 4.7 sec
Australia 4.7 sec
Brazil 2.5 sec
Canada 1.9 sec
Russian Federation 1.4 sec
United States 1.4 sec
Hungary 1.3 sec
United Kingdom 0.88 sec
Average page load time
1.2 sec 7.2 sec
Top countries by average page load time
nr-chi-app-1
Overview Processes Network Disks
30 minutes ending at Today, 19:13:26
CPU usage
IO Wait System User
Load average
Physical memory
Swap Used
nr-chi-app-1.newrelic.com
24 CPUs
47 GB RAM
Intel Xeon
Red Hat Enterprise Linux Server release 5.5 (Tikanga)
Linux 2.6.18-194.17.1.el5 x86_64
New Relic agent 1.1.2.124
I/O utilization
Network I/O (Mb/s)
VolGroup00-LogVol00
Transmitted Received
Apps
Response time Throughput Errors
RPM UI (nr-chi-app-1.newrelic.com) 230 ms 1,627 rpm 0.03%
Collector Proxy (nr-chi-app-1.newrelic.com) 3.5 ms 1,835 rpm 0.00%
Processes
User Count CPU Memory
ruby rpmui 24 20.2% 9.8 GB
memcached memcached 1 0.0% 1 GB


New Relic
Simplified Architecture
Our datacenter
Java
MySQL
Customer's environment
Average response time, broken down by tier (ms)
Ruby
Database
Worldwide Application Performance
Worldwide response times
United States
Canada
Throughput (rpm)
RAILS
@bjorn_fb

New Relic

# Simplified Architecture



# Our growth

- In 5 years, zero to 40,000 accounts...
- ... largest account has 17,000 servers
- ... 108 x $10^9$ metrics per day
  (75 x $10^6$ per minute)
- ... 8Tb of data a day
  (5.5Gb per minute)

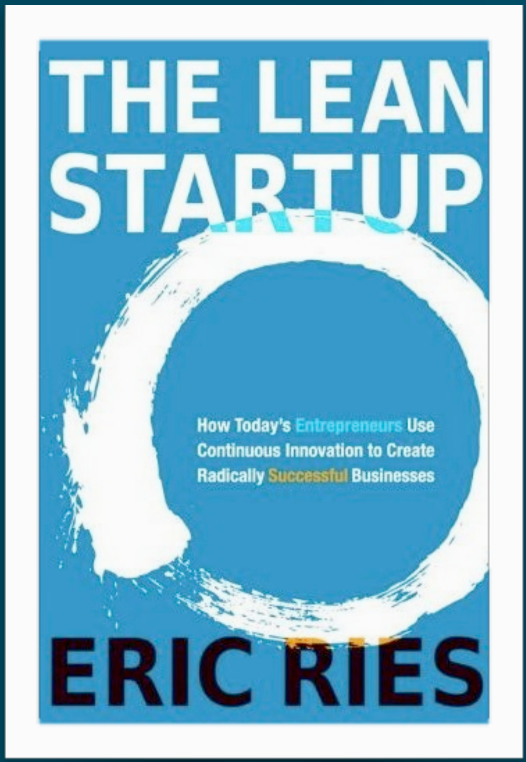
http://bit.ly/newrelic_stats

@bjorn_fb

New Relic

# Lean Startup

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

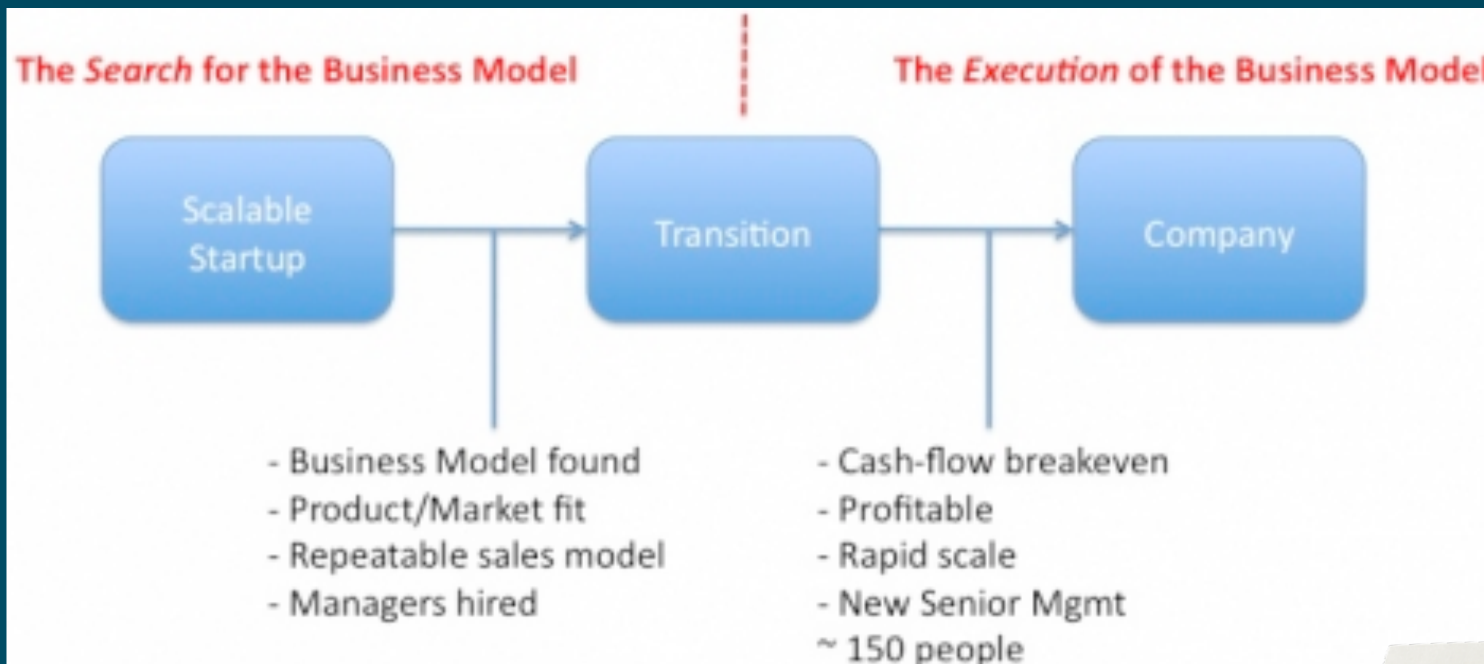



http://amzn.to/V86o4v

@bjorn_fb

# Lean Startup

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




http://bit.ly/PPj3Yi

New Relic

# 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 system for the "Search for Business Model"

http://bit.ly/TuOibr

@bjorn_fb

New Relic

# System Characteristics

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

@bjorn_fb

# The Basics (5)


@bjorn_fb

New Relic

# #1: F5

- Reduce the number of connections to the servers
  - F5 buffers requests and handles SSL


@bjorn_fb

New Relic

# #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

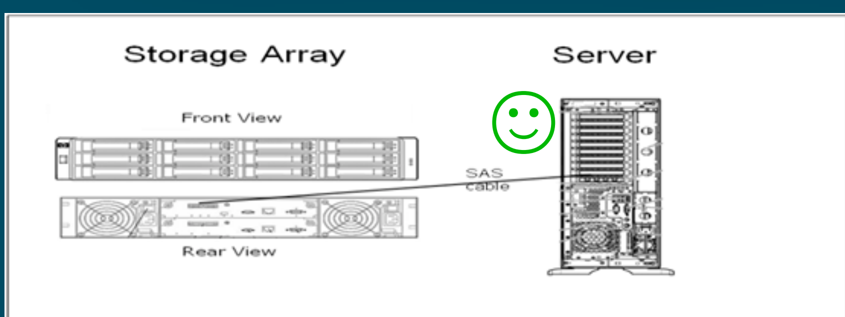
http://bit.ly/Stmu5t

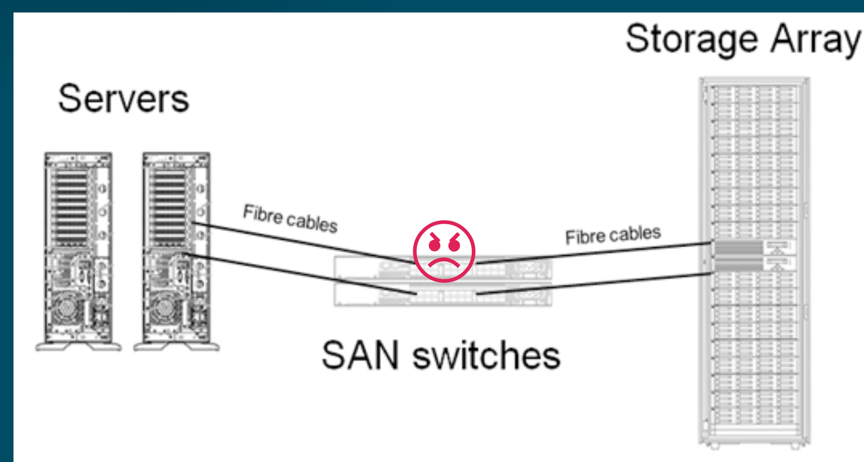
@bjorn_fb

New Relic

# #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

| | | | | |
|---|---|---|---|---|
| Beacon | | 0.144 ms | 2.16M rpm | 0.0023 % |
| Aggregator | | 3.37 ms | 1.39M rpm | 0.0494 % |

http://bit.ly/QrOExM

@bjorn_fb

New Relic

# #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

http://bit.ly/UAPARX

# The Usual Suspects (4)


@bjorn_fb

# #6: Agent Protocol

- Our first agent protocol was quick and dirty: Ruby object serialization and multiple round trips

```ruby
def marshal_data(data)
  NewRelic::LanguageSupport.with_cautious_gc do
    Marshal.dump(data)
  end
rescue => e
  log.debug("#{e.class.name} : #{e.message} when marshalling #{object}")
  raise
end
```

- Refined: reduce round-trips (package more data into the payload); keep-alive

# #7: Accumulate & Resend

- If a service is temporarily unavailable, accumulate and retry

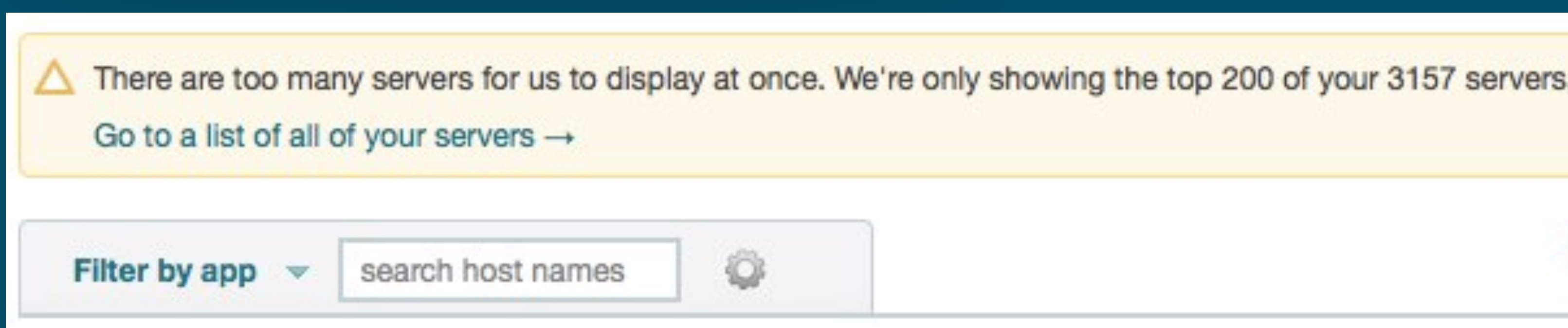
```
recover_from_communication_error:
        /*
         * We were unable to contact the collectors, so we need to add all of this data to
         * time unit's pending data.
         */
        nr__log (NRL_DEBUG, "[%s] recovering from communication error..", appname);
        nr__close_connection_to_daemon (nrdaemon);

        nrthread_mutex_lock (&app->lock); {
          /* merge metrics sets the ->replacement pointers of every metric in both from and
          nr_metric_table__merge_metrics_from_to (data->metrics, app->pending_harvest->metr
          nr__merge_slow_transactions_from_to (&(data->slow_transactions), &(app->pending_h
          nr__merge_errors_from_to (&(data->errors), &(app->pending_harvest->errors));
```

New Relic

# #8: Large Accounts

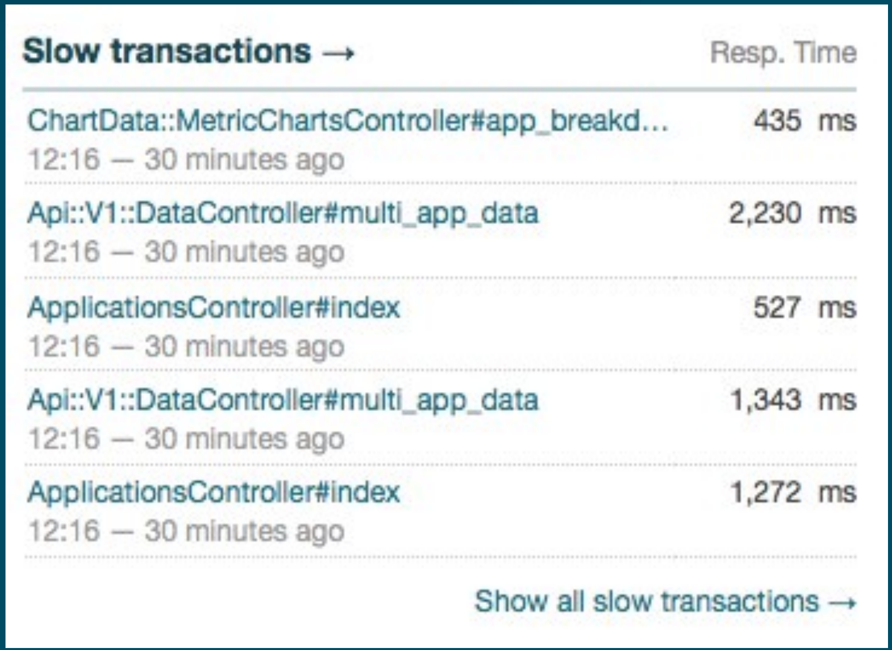
- Our first customers were small.
- Later larger customers stretched our assumptions. We added smart sorting, searching, paging, etc.



New Relic

# #9: ORM Issues

- ORMs (Rails) are nice but can quickly load too many objects. Do a careful audit of slow code.



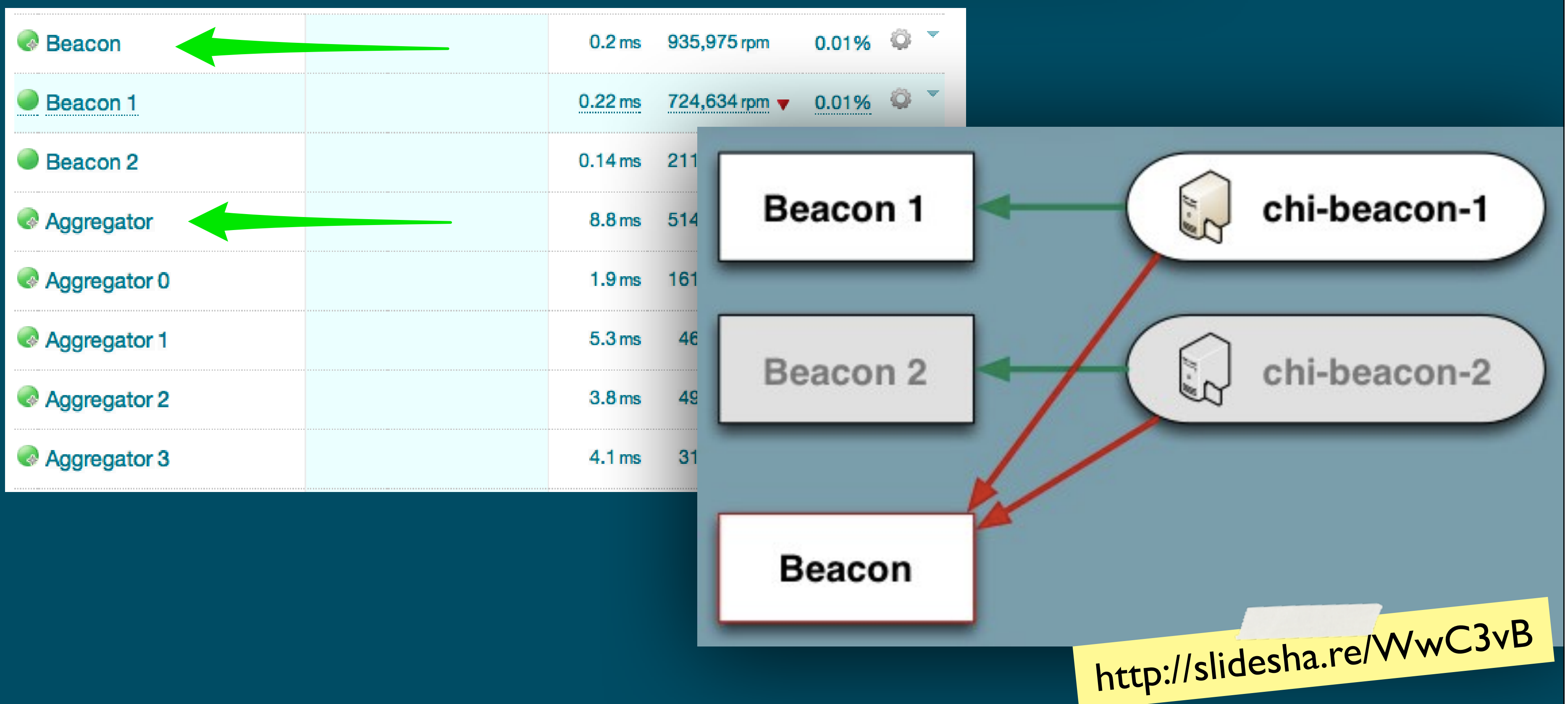

@bjorn_fb

# The Clever Stuff (6)


@bjorn_fb

# #10: Pre-compute

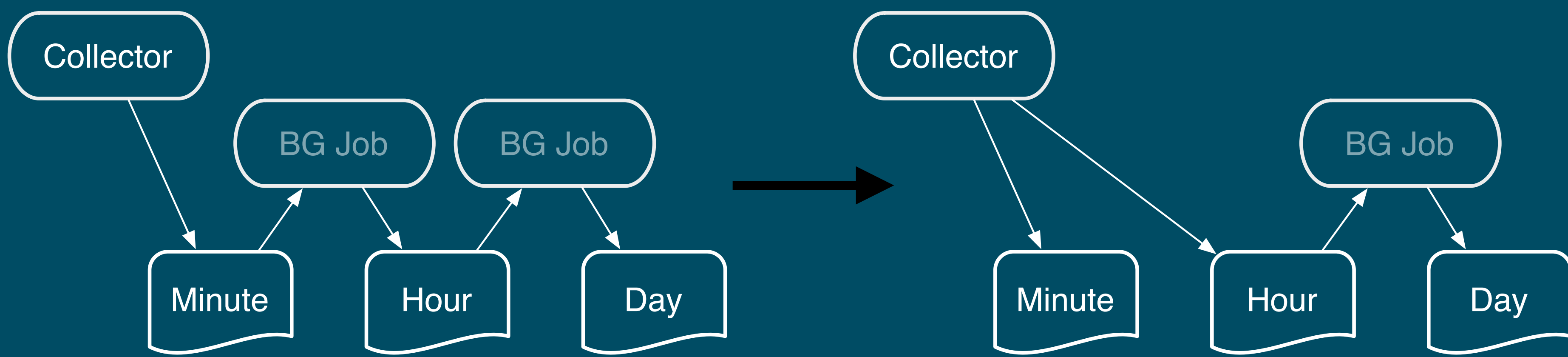
- Pre-compute expensive queries




@bjorn_fb

# #11: Real-time BG

- Background job to roll-up timeslice data: minutes to hours, hours to days



# #12: Different DBs

- 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

http://bit.ly/VfQG8R

@bjorn_fb

# #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

@bjorn_fb

# #14: Computation in DB

New Relic

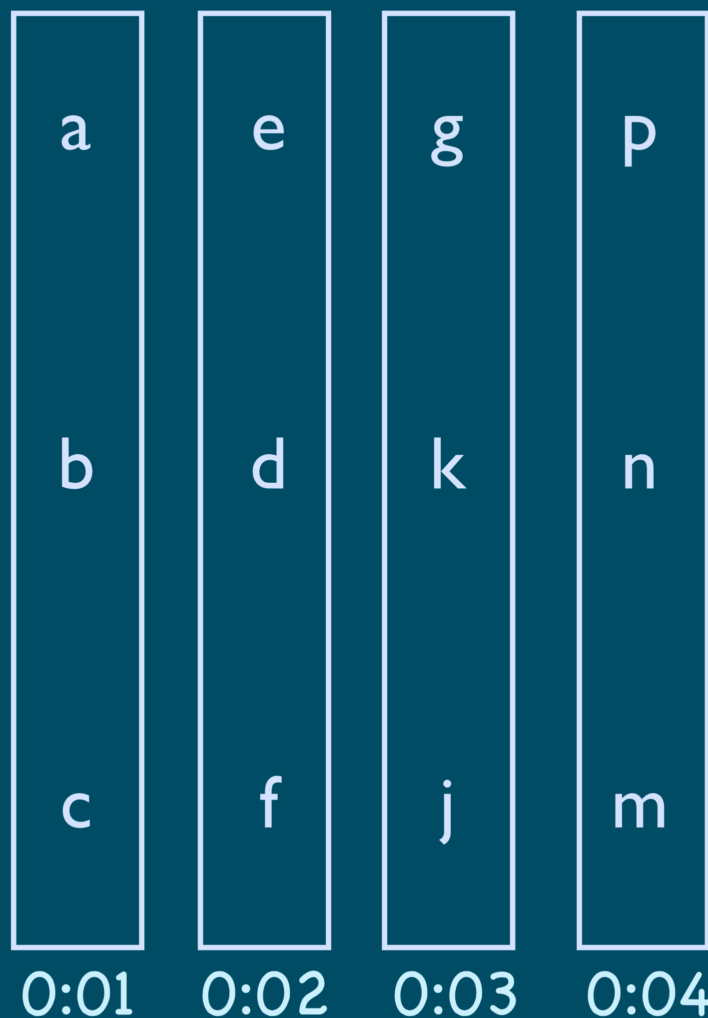
- Natural sharding allows us to push computation into the db
  - Supported by schema
  - Limits number of rows returned
  - Thus allows scripting language (Ruby) to do ‘real’ work
- Opposite of the classical advice of doing nothing in the db

http://bit.ly/PFppZh

@bjorn_fb

New Relic

# #15: SSDs




- sequential writes
- random reads

New Relic

# #15: SSDs

| | | | |
|---|---|---|---|
| a | e | g | p |
| b | d | k | n |
| c | f | j | m |
| 0:01 | 0:02 | 0:03 | 0:04 |

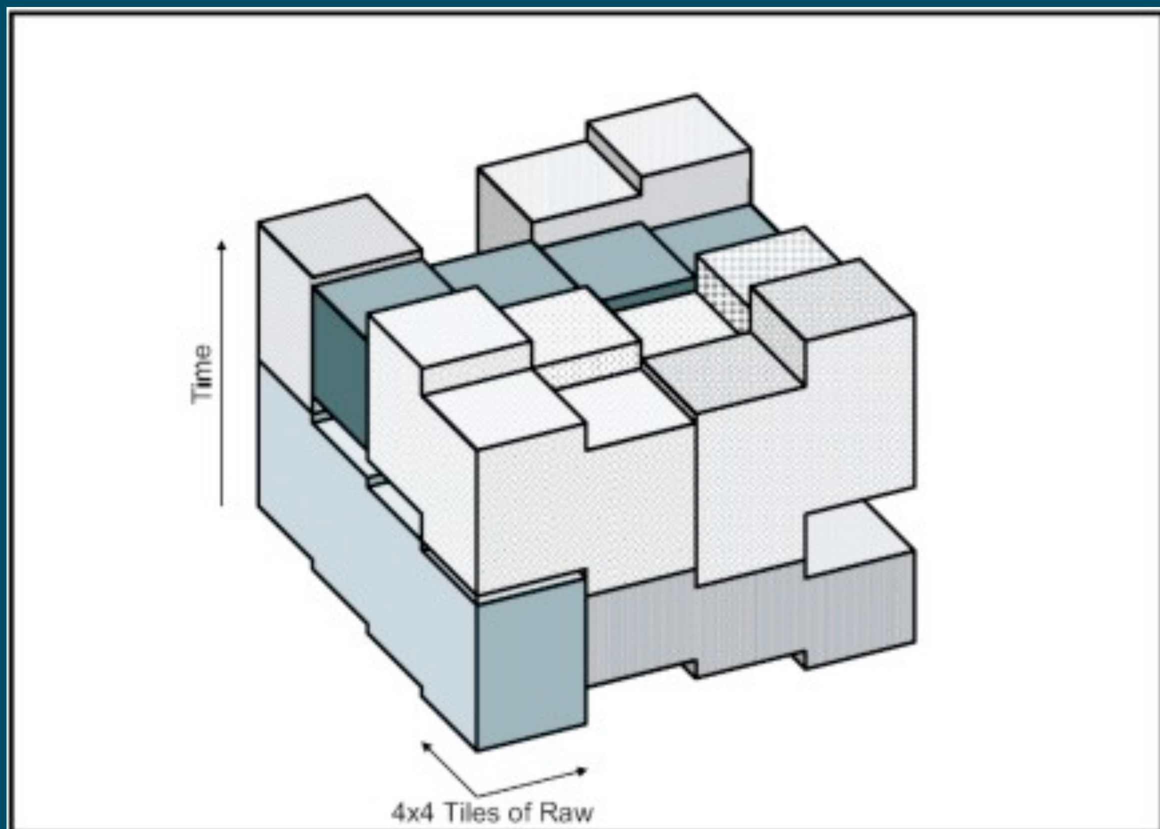
- random writes
- sequential reads

# #15: SSDs

- Choose sequential reads because of UI
  - Use buffers to help random writes, but...
- Switched to SSDs
  - writes are same or slightly slower
  - reads are fast, random or sequential
  - write limits not a problem due to non-rewrite nature of our data tables

@bjorn_fb

# The Optimizations (2)




@bjorn_fb

# #16: Moving Processes

- 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 type X processes on M1 and Ys on M2” we balance the machines

New Relic

# #17: Moving Customers

- 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

@bjorn_fb

New Relic

# Take-away


@bjorn_fb

# Take-away

1. Do the basics
2. Design in some scalability
3. Use the unique characteristics of your app to optimize
4. Buzzwords not needed


@bjorn_fb