Performance Debugging: Finding Bottlenecks in Distributed Systems

Christian Grabowski - Backend Engineer

Github: cpg1111 Twitter: @christiang0817

NS1.

About Me

- Backend Engineer @ NS1
- Total Linux Nerd
- <3 open source

About NS1

- We run a global network of authoritative DNS servers
- This network is made up of many different services that each provide a unique function in order to deliver our product
- These services are distributed globally across 25 POPs
- We’re always receiving traffic, and spikes in traffic will of course happen, so performance and reliability are important to us

In a nutshell...

- Distributed systems are complicated, and as the system scales to handle the growth of a company, the complexity grows along with it.
- As a system scales, bottlenecks are bound to occur, but with this complexity, it becomes more and more difficult to identify these bottlenecks.
- This can be frustrating, because logically speaking, your system is working, but it’s just not cutting it.

In A Little More Detail

- Where do you start?
  - There are so many metrics to look at
  - There are so many tools out there for performance debugging
- How do I identify the bottleneck in a multiple services of a distributed system?
  - Is it my database? Is it my service? Is it how I use my database? Is it another service?
- How do I identify the bottleneck in a single process?
  - Is it my code? Is it a Library? Is it the OS?

There are many ways to approach the identification of said issues. Based on my experience, here are a couple cases that will aim to answer these questions.

Case 1: Metrics Aggregation Problems

NS1.

Case 1

Initial Issue

Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster

Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process

Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process
- Throughput wasn't ideal

Case 1

Initial Issue

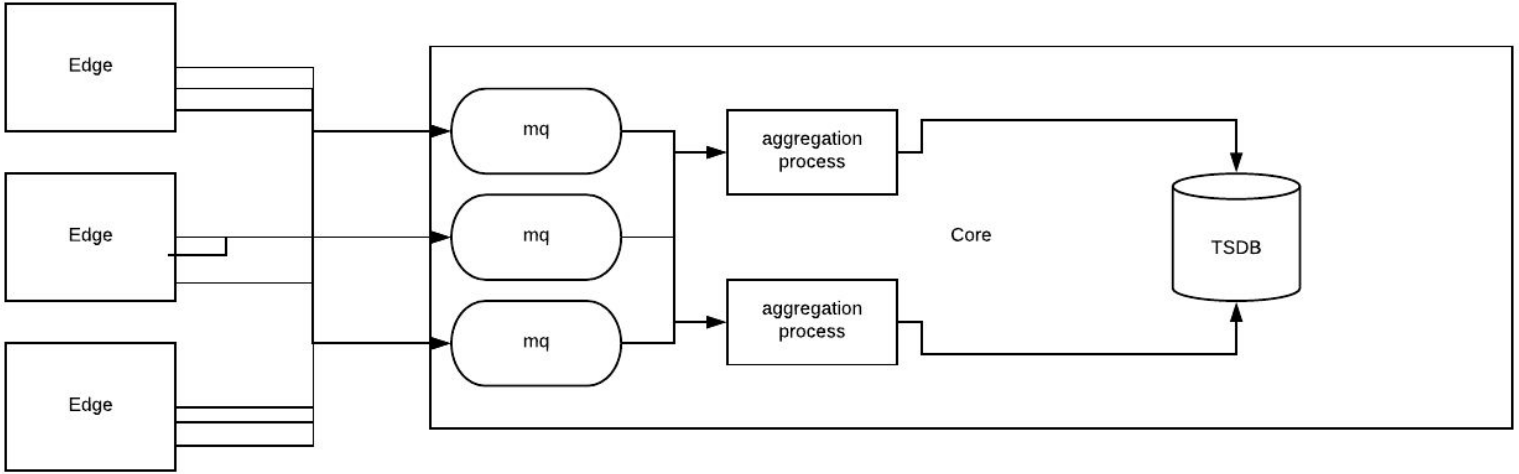
- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process
- Throughput wasn't ideal
- It consisted of too many Python processes

Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process
- Throughput wasn't ideal
- It consisted of too many Python processes
- More timeseries, more problems

Case 1


Edge
Edge
Edge
mq
mq
mq
aggregation process
aggregation process
Core
TSDB

Case 1

Hypothesised Solution (came up with a few, but ultimately)

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory
  - First approach was a COW btree, but that turned out to cause a lot of memory issues

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory
  - First approach was a COW btree, but that turned out to cause a lot of memory issues
  - Second approach was a sorted array of metrics sorted by last update that coincided with a map of metric name = aggregated value

Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory
  - First approach was a COW btree, but that turned out to cause a lot of memory issues
  - Second approach was a sorted array of metrics sorted by last update that coincided with a map of metric name = aggregated value
- Sounds great right?

```
[4075623.871205] outboundd invoked oom-killer: gfp_mask=0x24000c0, order=0, oom_score_adj=0
[4075623.871210] outboundd cpuset=749b1d12085c451c0714abd85b6112795eafc8356d77083aaa0e578db8353143 mems_allowed=0-1
[4075623.871218] CPU: 3 PID: 104311 Comm: outboundd Not tainted 4.4.0-83-generic #106~14.04.1-Ubuntu
[4075623.871220] Hardware name: Dell Inc. PowerEdge C6320/082F9M, BIOS 2.4.2 01/09/2017
[4075623.871222]  0000000000000000 ffff883f95f87c80 ffffffff813ddefc ffff883f95f87d80
[4075623.871226]  ffff8839e3664400 ffff883f95f87d10 ffffffff811fb976 0000000000000000
[4075623.871228]  0000000000000003 ffff883e85c5c1c0 ffff883fed469c60 ffff883ffde56e00
[4075623.871231] Call Trace:
[4075623.871240]  [<ffffffff813ddefc>] dump_stack+0x63/0x87
[4075623.871246]  [<ffffffff811fb976>] dump_header+0x5b/0x1d5
[4075623.871254]  [<ffffffff81188b7b>] ? find_lock_task_mm+0x3b/0x80
[4075623.871257]  [<ffffffff81189115>] oom_kill_process+0x205/0x3d0
[4075623.871260]  [<ffffffff811f1abf>] mem_cgroup_out_of_memory+0x26f/0x2c0
[4075623.871263]  [<ffffffff811f27dd>] mem_cgroup_oom_synchronize+0x30d/0x330
[4075623.871268]  [<ffffffff811edb20>] ? mem_cgroup_events.constprop.50+0x30/0x30
[4075623.871271]  [<ffffffff811897d5>] pagefault_out_of_memory+0x35/0x90
[4075623.871277]  [<ffffffff8107ad1f>] mm_fault_error+0x67/0x140
[4075623.871282]  [<ffffffff81068ef8>] __do_page_fault+0x3f8/0x430
[4075623.871284]  [<ffffffff81068f52>] do_page_fault+0x22/0x30
[4075623.871290]  [<ffffffff8180e9f8>] page_fault+0x28/0x30
[4075623.871292] Task in /docker/749b1d12085c451c0714abd85b6112795eafc8356d77083aaa0e578db8353143 killed as a result of limit of /docker/749b1d12085c451c0714abd85b6112795eafc83
56d77083aaa0e578db8353143
[4075623.871297] memory: usage 20971520kB, limit 20971520kB, failcnt 102
[4075623.871299] memory+swap: usage 0kB, limit 9007199254740988kB, failcnt 0
[4075623.871300] kmem: usage 0kB, limit 9007199254740988kB, failcnt 0
[4075623.871301] Memory cgroup stats for /docker/749b1d12085c451c0714abd85b6112795eafc8356d77083aaa0e578db8353143: cache:40KB rss:20971480KB rss_huge:804864KB mapped_file:0KB d
irty:0KB writeback:0KB inactive_anon:16KB active_anon:20970912KB inactive_file:0KB active_file:0KB unevictable:0KB
[4075623.871340] [ pid ]   uid  tgid total_vm      rss nr_ptes nr_pmds swapents oom_score_adj name
[4075623.871496] [104302]     0 104302  5358032  5244843   10308      25        0             0 outboundd
[4075623.871506] Memory cgroup out of memory: Kill process 104302 (outboundd) score 1002 or sacrifice child
[4075623.882195] Killed process 104302 (outboundd) total-vm:21432128kB, anon-rss:20971236kB, file-rss:8136kB
[4075625.094650] init: outboundd_go main process ended, respawning ]
```

NS1.

Case 1

What was wrong?

Case 1

What was wrong?

- There was a memory leak directly related to each metric received in the aggregation process

Case 1

What was wrong?

- There was a memory leak directly related to each metric received in the aggregation process
- The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced

Case 1

What was wrong?

- There was a memory leak directly related to each metric received in the aggregation process
- The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced
- At this level of throughput, this becomes a problem quickly, with rapidly increasing memory utilization.

Case 1

What was wrong?

- There was a memory leak directly related to each metric received in the aggregation process
- The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced
- At this level of throughput, this becomes a problem quickly, with rapidly increasing memory utilization.
- So what caused this?

Case 1

Steps taken

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
    - In our case, we created in-memory counters that would be sampled off of an http endpoint

```
GET /metrics?snapshot=true

[
    {
        "name": "msgs.recv",
        "time": 1257894000,
        "stags": {"host": "a.host"},
        "utags": ["bulk_size"],
        "T": [20],
        "V": 18500,
        "N": 8
    }, {
        "name": "metrics.recv",
        "time": 1257894000,
        "stags": {"host": "a.host"},
        "utags": ["type"],
        "T": ["by_customer", "by_zone", "by_record"],
        "V": 370000,
        "N": 160
    }
]
```

NS1.

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
    - In our case, we created in-memory counters that would be sampled off of an http endpoint
    - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment

Case 1

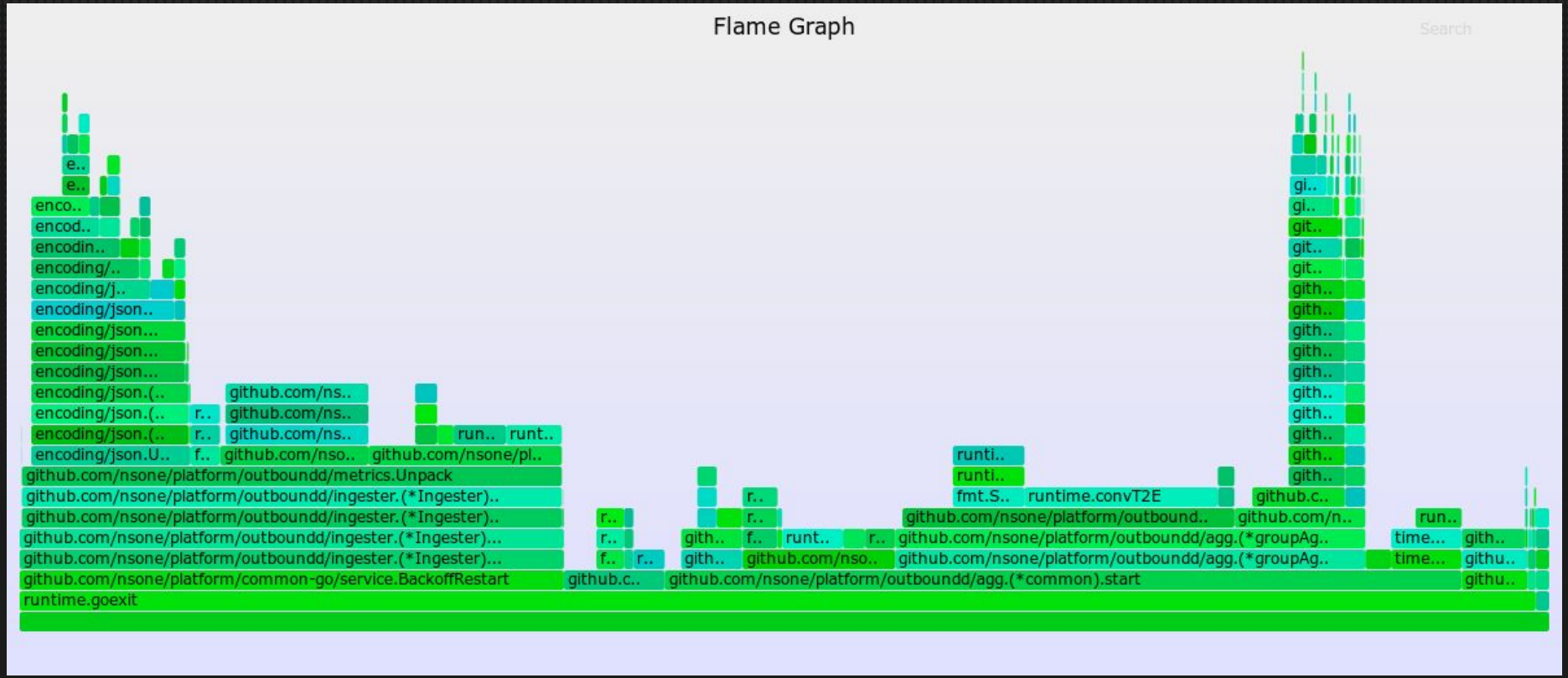
Steps taken

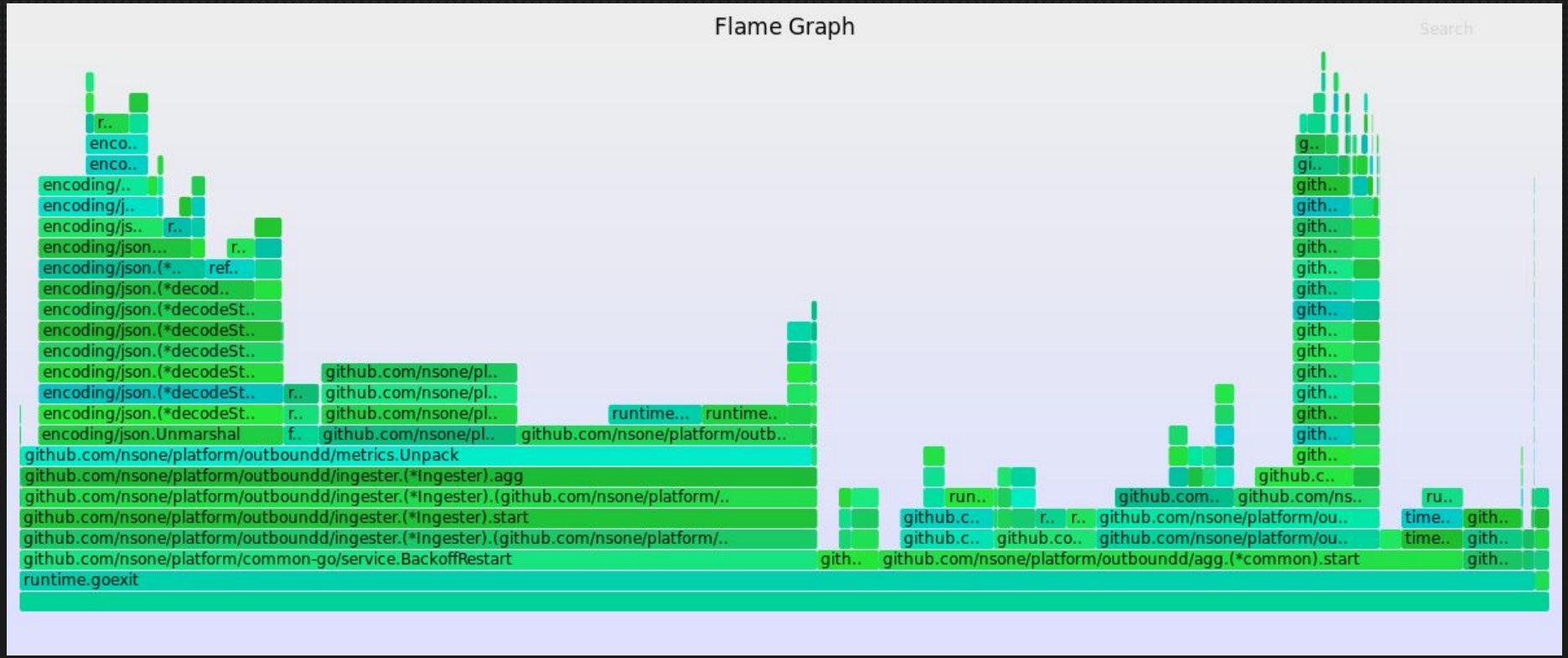
1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment
3. Profile the heap allocations of a single of the aggregation process
   - First with Go's built-in profiler and pprof visualizations

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment
3. Profile the heap allocations of a single of the aggregation process
   - First with Go's built-in profiler and pprof visualizations
   - Switched to Go-Torch (https://github.com/uber/go-torch)


Flame Graph
Search
e..
e..
enco..
encod..
encodin..
encoding/..
encoding/j..
encoding/json..
encoding/json...
encoding/json...
encoding/json...
encoding/json.(..
github.com/ns..
encoding/json.(..
r..
github.com/ns..
encoding/json.(..
r..
github.com/ns..
run..
runt..
encoding/json.U..
f..
github.com/nso..
github.com/nsone/pl..
runti..
github.com/nsone/platform/outboundd/metrics.Unpack
runti..
github.com/nsone/platform/outboundd/ingester.(*Ingester)..
r..
fmt.S..
runtime.convT2E
github.com/nsone/platform/outboundd/ingester.(*Ingester)..
r..
r..
github.com/nsone/platform/outbound..
run..
github.com/nsone/platform/outboundd/ingester.(*Ingester)...
r..
gith..
f..
runt..
r..
github.com/nsone/platform/outboundd/agg.(*groupAg..
time...
gith..
github.com/nsone/platform/outboundd/ingester.(*Ingester)...
f..
r..
gith..
github.com/nso..
github.com/nsone/platform/outboundd/agg.(*groupAg..
time...
githu..
github.com/nsone/platform/common-go/service.BackoffRestart
github.c..
github.com/nsone/platform/outboundd/agg.(*common).start
githu..
runtime.goexit
gi..
gi..
git..
git..
git..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
github.c..
github.com/n..


Flame Graph
Search
r..
enco..
enco..
encoding/..
encoding/j..
encoding/js..
r..
encoding/json...
r..
encoding/json.(*..
ref..
encoding/json.(*decod..
encoding/json.(*decodeSt..
encoding/json.(*decodeSt..
encoding/json.(*decodeSt..
encoding/json.(*decodeSt..
github.com/nsone/pl..
encoding/json.(*decodeSt..
r..
github.com/nsone/pl..
encoding/json.(*decodeSt..
r..
github.com/nsone/pl..
runtime...
runtime..
encoding/json.Unmarshal
f..
github.com/nsone/pl..
github.com/nsone/platform/outb..
github.com/nsone/platform/outboundd/metrics.Unpack
github.com/nsone/platform/outboundd/ingester.(*Ingester).agg
github.com/nsone/platform/outboundd/ingester.(*Ingester).(github.com/nsone/platform/..
github.com/nsone/platform/outboundd/ingester.(*Ingester).start
github.com/nsone/platform/outboundd/ingester.(*Ingester).(github.com/nsone/platform/..
github.com/nsone/platform/common-go/service.BackoffRestart
gith..
github.com/nsone/platform/outboundd/agg.(*common).start
gith..
runtime.goexit
run..
github.c..
r..
r..
github.c..
github.co..
github.com..
github.com/ns..
github.com/nsone/platform/ou..
github.com/nsone/platform/ou..
github.c..
g..
gi..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
gith..
ru..
time..
time..
gith..
gith..

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment
3. Profile the heap allocations of a single of the aggregation process
   - First with Go's built-in profiler and pprof visualizations
   - Switched to Go-Torch (https://github.com/uber/go-torch)
   - Used eBPF with bcc's memleak.py (slightly modified) tool for greater detail (https://github.com/iovisor/bcc)

Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment
3. Profile the heap allocations of a single of the aggregation process
   - First with Go's built-in profiler and pprof visualizations
   - Switched to Go-Torch (https://github.com/uber/go-torch)
   - Used eBPF with bcc's memleak.py (slightly modified) tool for greater detail (https://github.com/iovisor/bcc)
4. Used Go's Benchmark testing to be able to compare changes

```
BenchmarkBTree_Insert1-8                2000000              753 ns/op
BenchmarkBTree_Insert10-8                300000             5460 ns/op
BenchmarkBTree_Insert100-8                20000            67481 ns/op
BenchmarkBTree_Insert1000-8                2000           831932 ns/op
BenchmarkBTree_Insert10000-8                200          9863054 ns/op
BenchmarkBTree_Insert100000-8                10        127574333 ns/op
BenchmarkBTree_Insert1000000-8                1       1899309427 ns/op
BenchmarkBTree_Remove1-8                300000000              5.65 ns/op
BenchmarkBTree_Remove10-8               30000000              53.9 ns/op
BenchmarkBTree_Remove100-8               3000000               543 ns/op
BenchmarkBTree_Remove1000-8               200000              5481 ns/op
BenchmarkBTree_Remove10000-8               30000             55950 ns/op
BenchmarkBTree_Remove100000-8               2000            691415 ns/op
BenchmarkBTree_Remove1000000-8                1       4412983369 ns/op
BenchmarkLLRB_Insert1-8                 3000000               483 ns/op
BenchmarkLLRB_Insert10-8                 300000              4486 ns/op
BenchmarkLLRB_Insert100-8                 30000             52976 ns/op
BenchmarkLLRB_Insert1000-8                 2000            672204 ns/op
BenchmarkLLRB_Insert10000-8                 200           7843585 ns/op
BenchmarkLLRB_Insert100000-8                 10         104467790 ns/op
BenchmarkLLRB_Insert1000000-8                 1       1597383476 ns/op
BenchmarkLLRB_Remove1-8                 300000000              5.39 ns/op
BenchmarkLLRB_Remove10-8                30000000              51.6 ns/op
BenchmarkLLRB_Remove100-8                3000000               519 ns/op
BenchmarkLLRB_Remove1000-8                300000              5163 ns/op
BenchmarkLLRB_Remove10000-8                30000             54560 ns/op
BenchmarkLLRB_Remove100000-8                2000            673192 ns/op
BenchmarkLLRB_Remove1000000-8                 1       4016146227 ns/op
```

```
BenchmarkMapAggPut1-4                 1000000          52489 ns/op          1656 B/op        29
allocs/op
BenchmarkMapAggPut10-4                  50000         101305 ns/op          8763 B/op       140
allocs/op
BenchmarkMapAggPut100-4                 10000         522084 ns/op         78651 B/op      1220
allocs/op
BenchmarkMapAggPut1000-4                 1000        1343539 ns/op        777050 B/op     12020
allocs/op
BenchmarkMapAggPut10000-4                 100       13487897 ns/op       7761038 B/op    120020
allocs/op
BenchmarkMapAggPut100000-4                 10      133272840 ns/op      77601172 B/op   1200021
allocs/op
BenchmarkMapAggFlush1-4                500000          14246 ns/op          2272 B/op         37
allocs/op
BenchmarkMapAggFlush10-4                20000          86255 ns/op         11789 B/op        175
allocs/op
BenchmarkMapAggFlush100-4               10000         947855 ns/op         83198 B/op       1264
allocs/op
BenchmarkMapAggFlush1000-4               1000        3805475 ns/op        788817 B/op      12063
allocs/op
BenchmarkMapAggFlush10000-4               100       14738426 ns/op       7846638 B/op     120064
allocs/op
BenchmarkMapAggFlush100000-4               10      133869511 ns/op      78407281 B/op    1200061
allocs/op
```

NS1.

Case 1

The largest hurdle

Case 1

The largest hurdle

- Recreating production traffic

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
  - Second approach → have our message queue copy production messages into the test environment

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
  - Second approach → have our message queue copy production messages into the test environment
    - This required a lot of configuration

Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
  - Second approach → have our message queue copy production messages into the test environment
    - This required a lot of configuration
    - We were able to throttle the amount of traffic well

Case 1

What Worked

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

What Did Not Work

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

What Did Not Work

- Being able to reproduce a production level load in a dev or testing environment

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

What Did Not Work

- Being able to reproduce a production level load in a dev or testing environment
- Keeping a base comparison when making changes for the sake of performance

Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

What Did Not Work

- Being able to reproduce a production level load in a dev or testing environment
- Keeping a base comparison when making changes for the sake of performance
- Should've changed the initial service more iteratively, instead of a full rewrite immediately

Case 2: Public REST api slowing down

NS1.

Case 2

Initial Issue

Case 2

Initial Issue

- Response times of our public api service are increasing and some were timing out

Case 2

Initial Issue

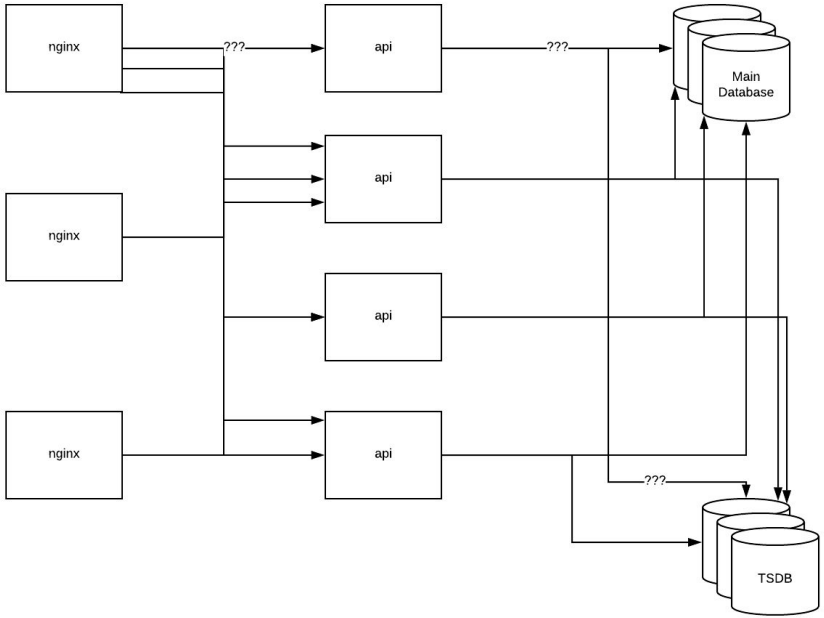
- Response times of our public api service are increasing and some were timing out
- A planned large influx of records to our database had put a much greater amount of load on our database

Case 2

Initial Issue

- Response times of our public api service are increasing and some were timing out
- A planned large influx of records to our database had put a much greater amount of load on our database
- Not all of our queries were optimized

Case 2



Case 2

Steps Taken

Case 2

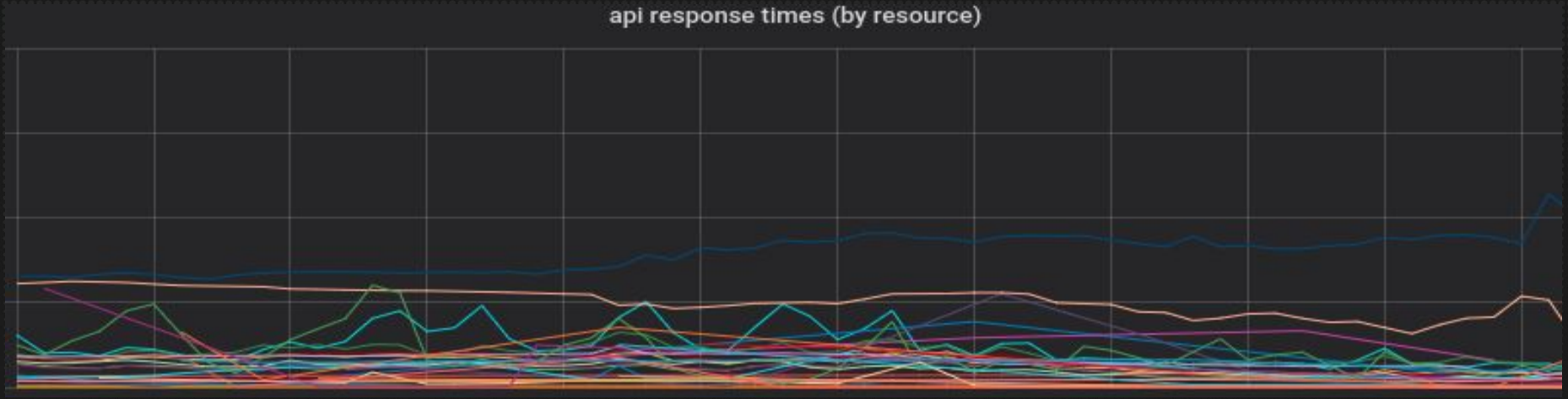
Steps Taken

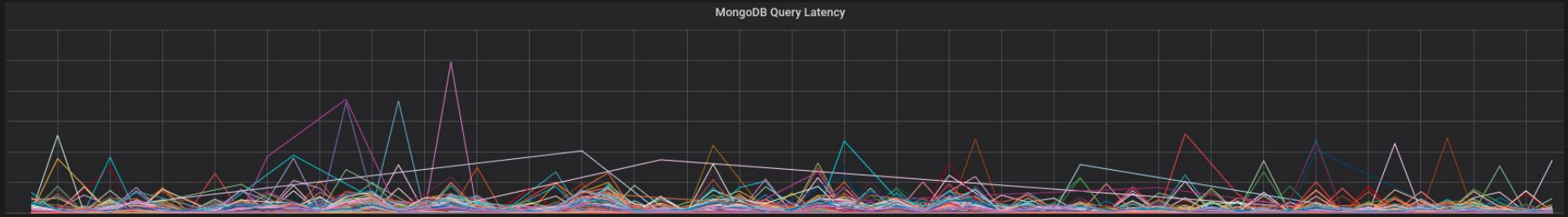
1. Instrument the IO of the api process

Case 2

Steps Taken

1. Instrument the IO of the api process
    - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?





NS1.

Case 2

Steps Taken

1. Instrument the IO of the api process
    - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
    - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking

```python
from nsone.util.tracing import ExportedTracingContext

TRACER = ExportedTracingContext(__name__)

def request(host, route, body):
    with TRACER.trace("{}.{}".format(host, route)):
        send(host, route, host)

_
```

NS1.

Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
2. Profile on-CPU for potential blocking code

Case 2

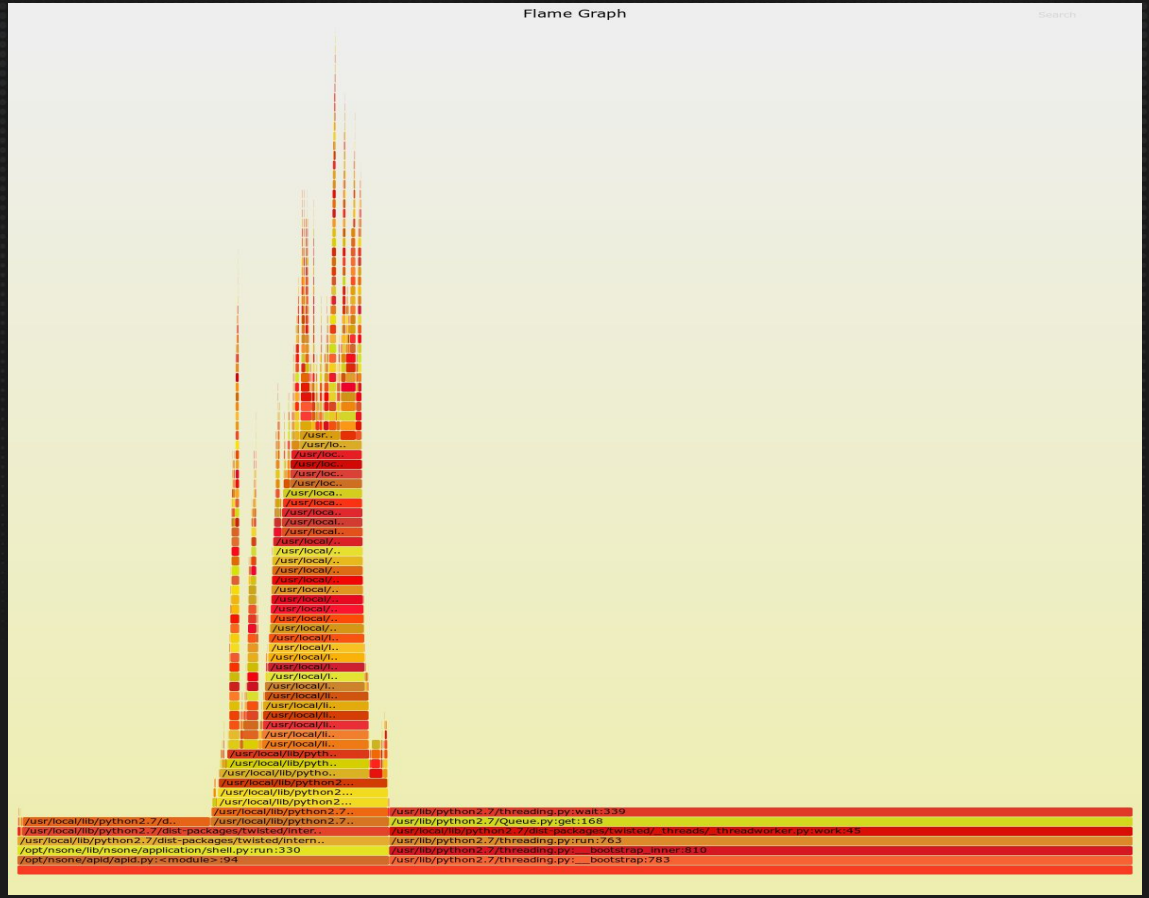
Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
2. Profile on-CPU for potential blocking code
   - Twisted uses async io

Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
2. Profile on-CPU for potential blocking code
   - Twisted uses async io
   - Large blocks would hint towards blocking code


Flame Graph
Search
/usr/local/lib/python2.7/threading.py:wait:339
/usr/local/lib/python2.7/d..
/usr/local/lib/python2.7..
/usr/lib/python2.7/Queue.py:get:168
/usr/local/lib/python2.7/dist-packages/twisted/inter..
/usr/local/lib/python2.7/dist-packages/twisted/_threads/_threadworker.py:work:45
/usr/local/lib/python2.7/dist-packages/twisted/intern..
/usr/lib/python2.7/threading.py:run:763
/opt/nsone/lib/nsone/application/shell.py:run:330
/usr/lib/python2.7/threading.py:__bootstrap_inner:810
/opt/nsone/apid/apid.py:<module>:94
/usr/lib/python2.7/threading.py:__bootstrap:783

NS1.

Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
2. Profile on-CPU for potential blocking code
   - Twisted uses async io
   - Large blocks would hint towards blocking code
3. Utilize instrumentation around our database queries

```
<TIMESTAMP> .025+0000 [conn109306794] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] zone: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:73 locks(micros) r:4090477 reslen:48 2107ms

<TIMESTAMP> .573+0000 [conn109306647] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] zone: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:104 locks(micros) r:4232800 reslen:48 2144ms

<TIMESTAMP> .214+0000 [conn109304828] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:1560 locks(micros) r:4801824 reslen:48 2558ms

<TIMESTAMP> .296+0000 [conn109304828] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:118 locks(micros) r:2828783 reslen:48 1482ms

<TIMESTAMP> .843+0000 [conn109306908] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:150 locks(micros) r:2331646 reslen:48 1194ms

<TIMESTAMP> .738+0000 [conn109307005] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] zone: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:116 locks(micros) r:4191979 reslen:48 2160ms

<TIMESTAMP> .475+0000 [conn109306647] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted], zone: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:77 locks(micros) r:4167760 reslen:48 2127ms

<TIMESTAMP> 857+0000 [conn109304828] command nsone.$cmd command: count { count: "records", fields: null, query: { customer: [redacted] } } planSummary: COUNT { <INDEX> } ntoreturn:1 keyUpdates:0 numYields:257 locks(micros) r:4973431 reslen:48 2593ms
```

NS1.

Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
2. Profile on-CPU for potential blocking code
   - Twisted uses async io
   - Large blocks would hint towards blocking code
3. Utilize instrumentation around our database queries
4. Based on the instrumentation, we identified the slow routes and queries

Case 2

Results

Case 2

Results

- Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes

Case 2

Results

- Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes
- Some of those queries we were able to remove all together

Case 2

Results

- Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes
- Some of those queries we were able to remove all together
- We were able to observe these improvements the moment they went to production

Case 2

Results

- Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes
- Some of those queries we were able to remove all together
- We were able to observe these improvements the moment they went to production
- There were definitely some queries that given more time, would be ideal to move to a different, better suited database

Lessons Learned

NS1.

Lessons Learned

- Logs and Metrics are invaluable

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
- Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
- Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  - I've been working on a project to make this easier in my spare time https://github.com/cpg1111/pprof-ebpf

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
- Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  - I've been working on a project to make this easier in my spare time https://github.com/cpg1111/pprof-ebpf
- Being able to keep your changes small and more iterative will yield you greater results in the long run.

Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
- Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  - I've been working on a project to make this easier in my spare time https://github.com/cpg1111/pprof-ebpf
- Being able to keep your changes small and more iterative will yield you greater results in the long run.
- Having a baseline to compare changes for performance to is a must

Thank You

NS1.