cache me if you can

how Grafana Labs scaled up their memcached $42x$
and improved reliability

danny kopping / sr. software engineer
65% reduction in object storage reqs
vastly improved reliability
~2% overall TCO reduction
no change in performance
but first...

loki internals
how does Loki work?

timestamp
with nanosecond precision

labels/selectors
key-value pairs

content
log line

indexed
unindexed

object storage

index
chunks
<table>
<thead>
<tr>
<th>Requests</th>
<th>Time</th>
<th>Size</th>
<th>User-Agent</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /</td>
<td>06/16/2023</td>
<td>612</td>
<td>curl/7.74.0</td>
<td></td>
</tr>
<tr>
<td>GET /index.html</td>
<td>06/16/2023</td>
<td>612</td>
<td>wget/1.21.1</td>
<td></td>
</tr>
<tr>
<td>GET /about.html</td>
<td>06/16/2023</td>
<td>396</td>
<td>Safari/15.4</td>
<td></td>
</tr>
<tr>
<td>GET /contact.html</td>
<td>06/16/2023</td>
<td>299</td>
<td>Firefox/92.0</td>
<td></td>
</tr>
<tr>
<td>GET /blog/index.html</td>
<td>06/16/2023</td>
<td>824</td>
<td>Chrome/92.0.4515.131</td>
<td></td>
</tr>
<tr>
<td>GET /blog/post-1.html</td>
<td>06/16/2023</td>
<td>543</td>
<td>Opera/12.15</td>
<td></td>
</tr>
<tr>
<td>GET /blog/post-2.html</td>
<td>06/16/2023</td>
<td>456</td>
<td>Edge/82.0.1245.153</td>
<td></td>
</tr>
<tr>
<td>GET /blog/post-3.html</td>
<td>06/16/2023</td>
<td>369</td>
<td>IE/11.0</td>
<td></td>
</tr>
</tbody>
</table>

{app="nginx", pod="nginx-a", az="us-east-1a"}

{app="nginx", pod="nginx-b", az="us-east-1b"}

{app="nginx", pod="nginx-c", az="us-east-1b"}

{app="nginx", pod="nginx-d", az="us-east-1b"}
querying

{app="nginx"}
querying

{app="nginx"}

{az="us-east-1a"}

00ba6fb0

72447153
fast queries

100TB ➔ 100GB ➔ 10GB ➔ 1TB/s

raw logs ➔ label selector ➔ timeframe ➔ brute force search, heavily parallelized

index

{app="nginx", az="us-east-1a"} | = "12.34.56.78"
slow queries

100TB → 100GB → 10GB → 1TB/s

raw logs

label selector

timeframe

brute force search, heavily parallelized

{app=~".+"} | = "12.34.56.78"
rate-limiting
rate-limiting
⇒ decreased throughput
⇒ slower queries
⇒ frustrated users
⇒ SLO budget burn alerts
⇒ operator toil
⇒ "the incident"
two choices:
query less or cache more
choice A:
query less

improve labels
accelerated "needle-in-haystack"
smarter query engine
chunk compaction
the problem:
this shit takes time
choice B: cache more

buy time to do the *right thing*™
easier lever to pull
shorter delivery timeline
caching
### Ingestion Logs

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Time</th>
<th>Request Headers</th>
<th>Status Code</th>
<th>Size</th>
<th>User-Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.1.1</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET / HTTP/1.1 200 612 &quot;curl/7.74.0&quot;</td>
<td>200</td>
<td>612</td>
<td></td>
</tr>
<tr>
<td>192.168.1.2</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /index.html HTTP/1.1 200 612 &quot;wget/1.21.1&quot;</td>
<td>200</td>
<td>612</td>
<td></td>
</tr>
<tr>
<td>192.168.1.3</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /about.html HTTP/1.1 200 396 &quot;Safari/15.4&quot;</td>
<td>200</td>
<td>396</td>
<td></td>
</tr>
<tr>
<td>192.168.1.4</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /contact.html HTTP/1.1 200 299 &quot;Firefox/03.8&quot;</td>
<td>200</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>192.168.1.5</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /blog/index.html HTTP/1.1 200 824 &quot;Chrome/82.0.4151.131&quot;</td>
<td>200</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td>192.168.1.6</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /blog/post-1.html HTTP/1.1 200 543 &quot;Opera/12.15&quot;</td>
<td>200</td>
<td>543</td>
<td></td>
</tr>
<tr>
<td>192.168.1.7</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /blog/post-2.html HTTP/1.1 200 456 &quot;Edge/82.0.41245.153&quot;</td>
<td>200</td>
<td>456</td>
<td></td>
</tr>
<tr>
<td>192.168.1.8</td>
<td>2023-06-16 11:02.123456789</td>
<td>GET /blog/post-3.html HTTP/1.1 200 369 &quot;IE/11.0&quot;</td>
<td>200</td>
<td>369</td>
<td></td>
</tr>
</tbody>
</table>

### Ingestion Diagram

[Diagram showing ingestion process and object storage]

### Ingestion Summary

- **App**: nginx
- **Pod**: nginx-a, nginx-b, nginx-c, nginx-d
- **AZ**: us-east-1a, us-east-1b

---

**Note:** The diagram illustrates the ingestion process and object storage within the system, showcasing the flow of data from the Ingestion step to the Object Storage. The logs displayed are sample entries from the ingestion process, highlighting the request methods, headers, and status codes for various requests made to the server.
querying

{app="nginx"}

{az="us-east.*"}

00ba6fb0

72447153

query engine

object storage
recency bias

global
recency bias

loki-prod3
success metric: hit rate

looks kinda... *good*, right?
problem: churn

items being evicted before being fetched even once!
problem: churn

full cache replacements per hour
how much cache do we need?
200 instances @ 1 vCPU, 6GB RAM

total capacity: ~1.2TB RAM
running on shared n2-standard-32 nodes
how much cache do we need?
how much cache do we need?
~50TB cache / ~500Gbps throughput
the challenge:
can we do it **cost-effectively**, whilst **maintaining performance**, in an **operationally-familiar** way?
200 instances @ 1 vCPU, 6GB RAM

total capacity: ~1.2TB RAM
running on shared n2-standard-32 nodes

CPU: \(\frac{\text{list}}{2} / \text{cpus}\) = \$17.72/vCPU/month
RAM: \(\frac{\text{list}}{2} / \text{gb}\) = \$4.43/GB/month

200 vCPUs \(\rightarrow\) \$3544
1200 GB RAM \(\rightarrow\) \$5316
\(\Rightarrow\) \$8860 per month
200 replicas @ 1 vCPU, 6GB RAM

Total capacity: ~1.2TB RAM
running on shared n2-standard-32 nodes

CPU: \( \frac{\text{list}}{2} / \text{cpus} = \sim \$17.72/\text{vCPU/month} \)

RAM: \( \sim \$4.43/\text{GB/month} \)

200 vCPUs \( \Rightarrow \$3544 \)
1200 GB RAM \( \Rightarrow \$5316 \)
\( \Rightarrow \$8860 \text{ per month} \)
the goal:

**drive down cost per GB of cache**
use memcached
...with SSDs!
Caching beyond RAM: the case for NVMe - Dormando (June 12th, 2018)

Caching architectures at every layer of the stack embody an implicit tradeoff between performance and cost. These tradeoffs however are constantly shifting: new inflection points can emerge alongside advances in storage technology, changes in workload patterns, or fluctuations in hardware supply and demand.

In this post we explore the design ramifications of the increasing cost of RAM on caching systems. While RAM has always been expensive, DRAM prices have risen by over 50% in 2017, and high densities of RAM involve multi-socket NUMA machines, bioating power and overall costs. Concurrently, alternative storage technologies such as Flash and Optane continue to improve. They have specialized hardware interfaces, consistent performance, high density, and relatively low costs. While there is increasing economic incentive to explore offloading caching from RAM onto NVMe or NVM devices, the implications for performance are still not widely understood.

We will explore these design implications in the context of Memcached, a distributed, simple, cache-focused key/value store. For a quick overview, see the about page or the story tutorial.
DRAM

<table>
<thead>
<tr>
<th>keys</th>
<th>objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU head</td>
</tr>
</tbody>
</table>

evict
keys

LRU head → LRU tail

objects

DRAM

SSD

objects

extstore
using extstore

memcached <other flags>
   --extended
   ext_path=/mnt/disks/ssd0/datafile:345G
using extstore

memcached <other flags>
  --extended
  ext_path=/mnt/disks/ssd0/datafile:345G,
            /mnt/disks/ssd1/datafile:345G,
            /mnt/disks/ssd2/datafile:345G...

SSDs in cloud?
no "disk over network"
no multi-tenancy
physically attached to hypervisor
“Local SSDs”

- 375GB each @ $30/month
- add up to 24 SSDs to most machine types
- 660MB/s reads, 350MB/s writes
“Instance Storage”

- varies by instance type
- included in cost
ttv: time-to-value
2 weeks!
33 instances @ 6 vCPU, 5GB RAM, 4 SSDs

total capacity: ~50TB SSD, 528Gbps
running on dedicated
n2-highcpu-8 nodes (16Gbps each)

CPU: <list> / 2 / <cpus> = ~$13.08/vCPU/month
RAM: <list> / 2 / <gb> = ~$13.08/GB/month
SSD: $30 per disk (375GB) = $0.08/GB/month

198 vCPUs ⇒ $2590
165 GB RAM ⇒ $2158
132 SSDs ⇒ $3960

⇒ $8708 per month
33 instances @ 6 vCPU, 5GB RAM, 4 SSDs

Total capacity: ~50TB SSD, 528Gbps running on dedicated n2-highcpu-8 nodes (16Gbps each)

CPU: <list> / 2 / <cpus> = ~$13.08/vCPU/month
RAM: <list> / 2 / <gb> = ~$13.08/GB/month
SSD: = $0.08/GB/month
200 instances @ 1 vCPU, 6GB RAM

total capacity: \( \sim 1.2 \) TB RAM
running on shared n2-standard-32 nodes

CPU: \( \frac{\text{<list>}}{2} \div \text{<cpus>} = \sim \$17.72/\text{vCPU/month} \)
RAM: \( \frac{\text{<list>}}{2} \div \text{<gb>} = \sim \$4.43/\text{GB/month} \)

200 vCPUs \( \Rightarrow \$3544 \)
1200 GB RAM \( \Rightarrow \$5316 \)
\( \Rightarrow \$8860 \text{ per month} \)
✅ 65% reduction in object storage reqs

✅ vastly improved reliability

✅ ~2% overall TCO reduction

😢 no change in performance
**Google Cloud Storage**

- $5 per million writes
- $0.005 per 1000

**Amazon S3**

- $0.4 per million reads
- $0.0004 per 1000

*no charge for bandwidth within the same region*
rate-limits
SLO (query throughput)
hit rate

Cache Hit Rate

rollout
churn rate

rollout
cache effectiveness

Chunk Volume Fetched Per Day

rollout
query performance

P50

P99

rollout
and then everything was perfect... the end
trade-offs
Latency Numbers Every Programmer Should Know

- 1ns
- L1 cache reference: 6.5 ns
- Branch instruction: 5 ns
- L2 cache reference: 7 ns
- Mutex lock/unlock: 20 ns
- Mutex lock/unlock: 10 ns

Random memory reference: 100 ns
- Compass 1KB with Zlib: 3 ns
- 300 random read 1GB/s SSD: 155 ns

Send 1KB over 1Gbps network: 15 µs
- Read LMB sequentially from memory: 250 µs
- Round trip to same datacenter: 500 µs

Read LMB sequentially from disk: 36 ns
- Packet roundtrip from Oe to Netherlands: 130 ns
- Read LMB sequentially from disk: 50 ns

Source: https://gitlab.com/2601/latency
Latency Numbers Every Programmer Should Know

- Main memory reference: 100 ns
- L1 cache reference: 1 ns
- L2 cache reference: 7 ns
- Mutex lock/unlock: 25 ns
- Branch hazard: 1 ns
- Compute 1 CPU with 2 threads: 3 μs
- Send 1 KB over 1 Gbps network: 19 μs
- Read L1D sequentially from 200 KB: 1 ns

SSD random read (1Gb/s SSD): 150 μs

Round trip in same datacenter: 500 μs

Packet roundtrip:
- On site: 15 μs
- Netherlands: 150 μs

Source: https://alist.github.io/2013/07/
latency
latency

rollout
durability vs volatility

Warning: The performance gains from local SSDs require certain trade-offs in availability, durability, and flexibility. Because of these trade-offs, Local SSD storage is not automatically replicated and all data on the local SSD may be lost if the VM stops for any reason. See Local SSD data persistence for details.

<table>
<thead>
<tr>
<th>Initiated instance lifecycle events</th>
<th>What happens to your data?</th>
</tr>
</thead>
<tbody>
<tr>
<td>The instance is rebooted</td>
<td>The data persists</td>
</tr>
<tr>
<td>The instance is stopped</td>
<td>The data does not persist</td>
</tr>
<tr>
<td>The instance is hibernated</td>
<td>The data does not persist</td>
</tr>
<tr>
<td>The instance is terminated</td>
<td>The data does not persist</td>
</tr>
<tr>
<td>The instance type is changed</td>
<td>The data does not persist *</td>
</tr>
<tr>
<td>A Windows AMI is created from the instance</td>
<td>The data does not persist in the created AMI **</td>
</tr>
<tr>
<td>An EBS-backed AMI is created from the instance</td>
<td>The data does not persist in the created AMI **</td>
</tr>
<tr>
<td>An instance store-backed AMI is created from the instance</td>
<td>The data persists in the AMI bundle uploaded to Amazon S3 ***</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User-initiated OS events</th>
<th>What happens to your data?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A shutdown is initiated</td>
<td>The data does not persist †</td>
</tr>
<tr>
<td>A restart is initiated</td>
<td>The data persists</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AWS scheduled events</th>
<th>What happens to your data?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance stop</td>
<td>The data does not persist</td>
</tr>
<tr>
<td>Instance reboot</td>
<td>The data persists</td>
</tr>
<tr>
<td>System reboot</td>
<td>The data persists</td>
</tr>
<tr>
<td>Instance retirement</td>
<td>The data persists</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unplanned events</th>
<th>What happens to your data?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The data does not persist</td>
</tr>
</tbody>
</table>
disk-related issues

- disk fills up
- disk has bad sectors
- disk ages, performs poorly
solution:

kill the disk!
observability

● **prometheus**
  ○ memcached-exporter
  ○ node-exporter

● **alertmanager**
  ○ disk full alert
  ○ disk latency alert
  ○ memcached latency alert

● **grafana & loki** (shocker!)
implications for the future of cloud databases?

- cost is becoming compelling
- capacious disks at near-DRAM speeds
- higher network throughput
  - getting higher as network is offloaded from CPU
thanks!

questions?

@dannykopping

shoutout to Ed & Alan!

check out the blog post for more details
why not Memorystore / Elasticache?

Memorystore for Memcached

37 x Cache

vCPU usage: 135,050 vCPU hours
RAM usage: 40,515,000 GiB hours
Location: us-central1
USD 367,336.00

Total Estimated Cost: USD 367,336.00 per 1 month

Elasticache:

Selected Instance:
cache.m6g.8xlarge
vCPU: 32
Memory: 103.68 GiB
Network Performance: 12 Gigabit

Utilization (On-Demand only)
Elasticache Instances will be always at 100% utilization, to stop incurring charges, you can either cancel your instance or change the capacity:

100

Total Upfront cost: 0.00 USD
Total Monthly cost: 865,780.00 USD