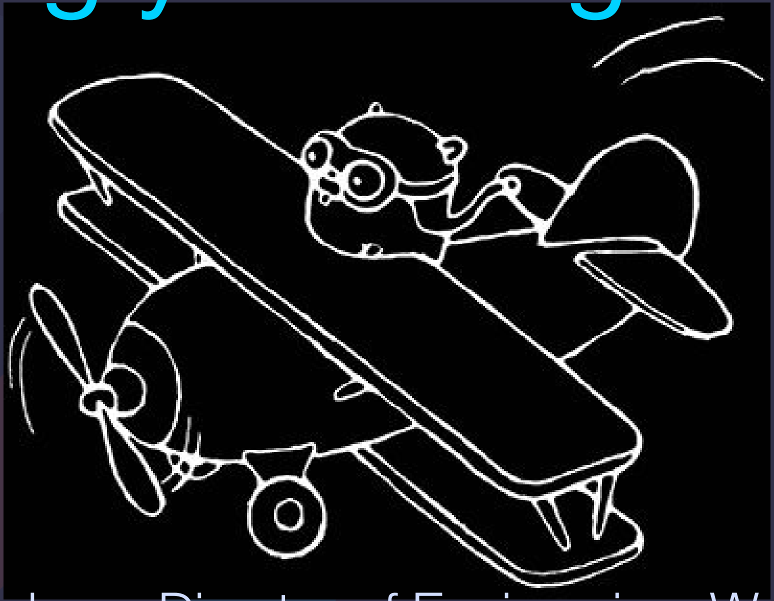
Making your Go go Faster



Bryan Boreham, Director of Engineering, Weaveworks

@bboreham

weaveworks

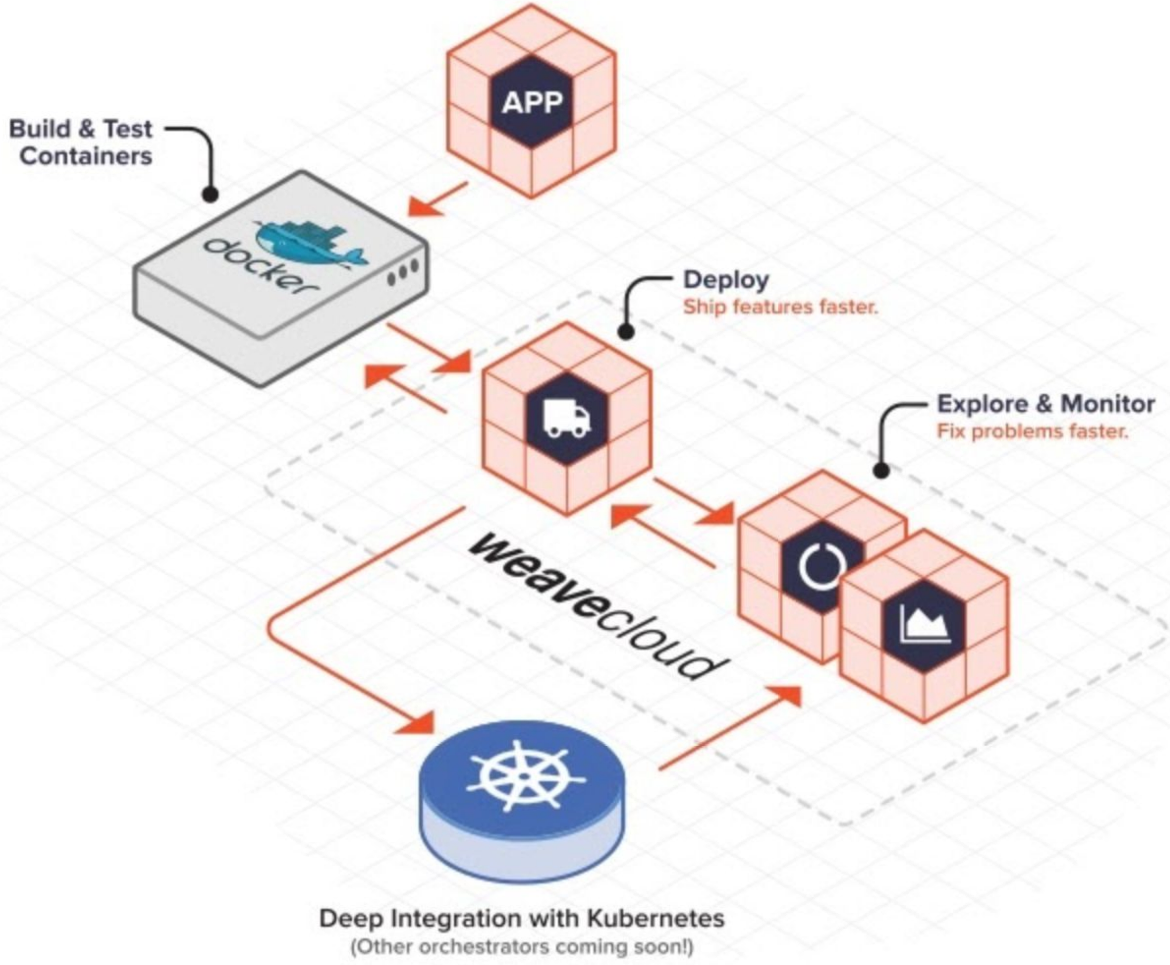
What does Weave do?

Weave lets devops iterate faster with:

- observability & monitoring
- continuous delivery
- container networks & firewalls

Kubernetes is our #1 platform

weaveworks



Hi, I’m Bryan Boreham

At Weaveworks, I work on system visualisation, observability & monitoring, CI/CD

I also contribute to Container Network Interface, Kubernetes, Prometheus

Program optimisation is my video-game.

Who is working with...

- Go
- Prometheus
- Weaveworks

What I will cover

- **How** to drill into the perf of your Go code
- **When** to look at the perf of your Go code
- Some **patterns** to look out for
- Things that matter more than you might **think**

https://media.timeout.com/images/103755908/630/472/image.jpg

The three most important things in software optimisation

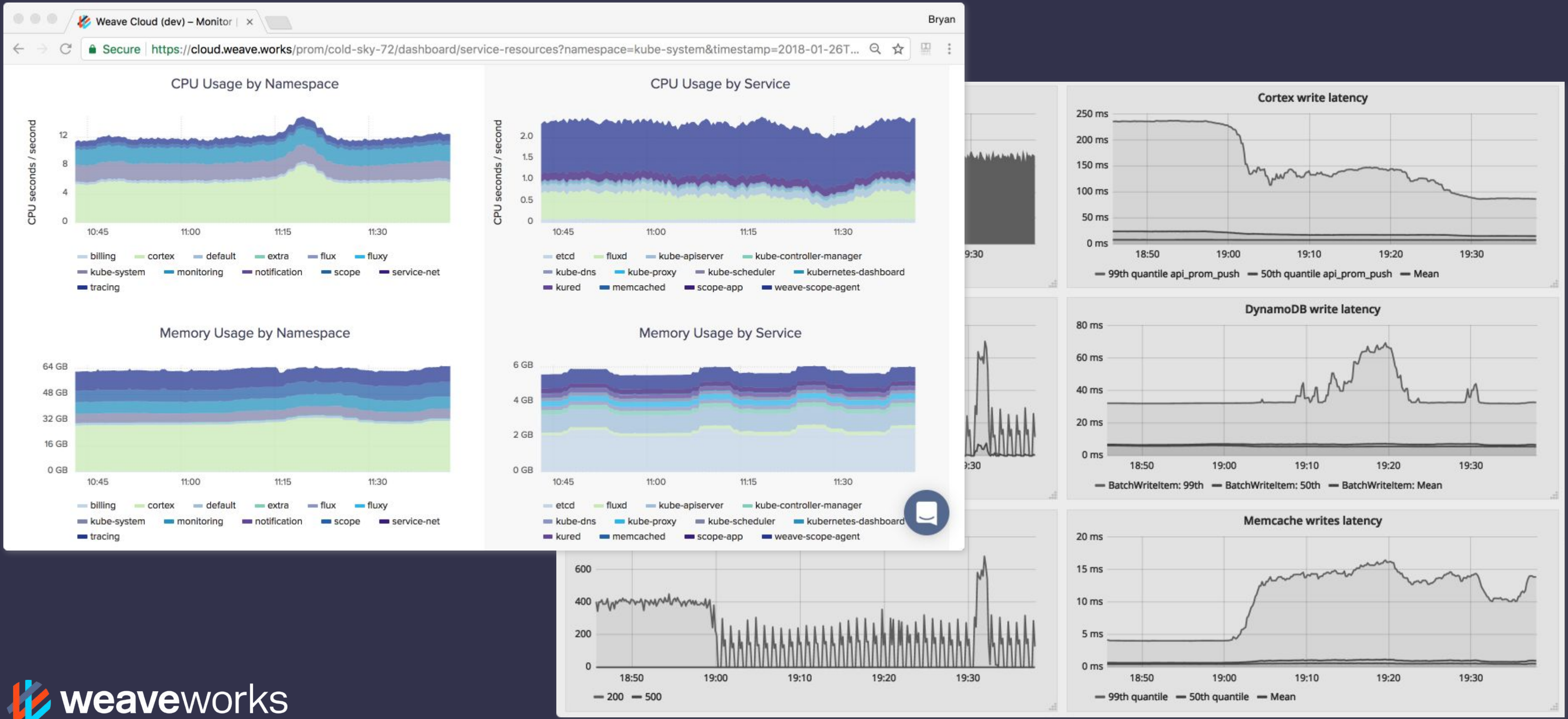
Measure,

Measure,

Measure.

Measure big things

Measure all the time

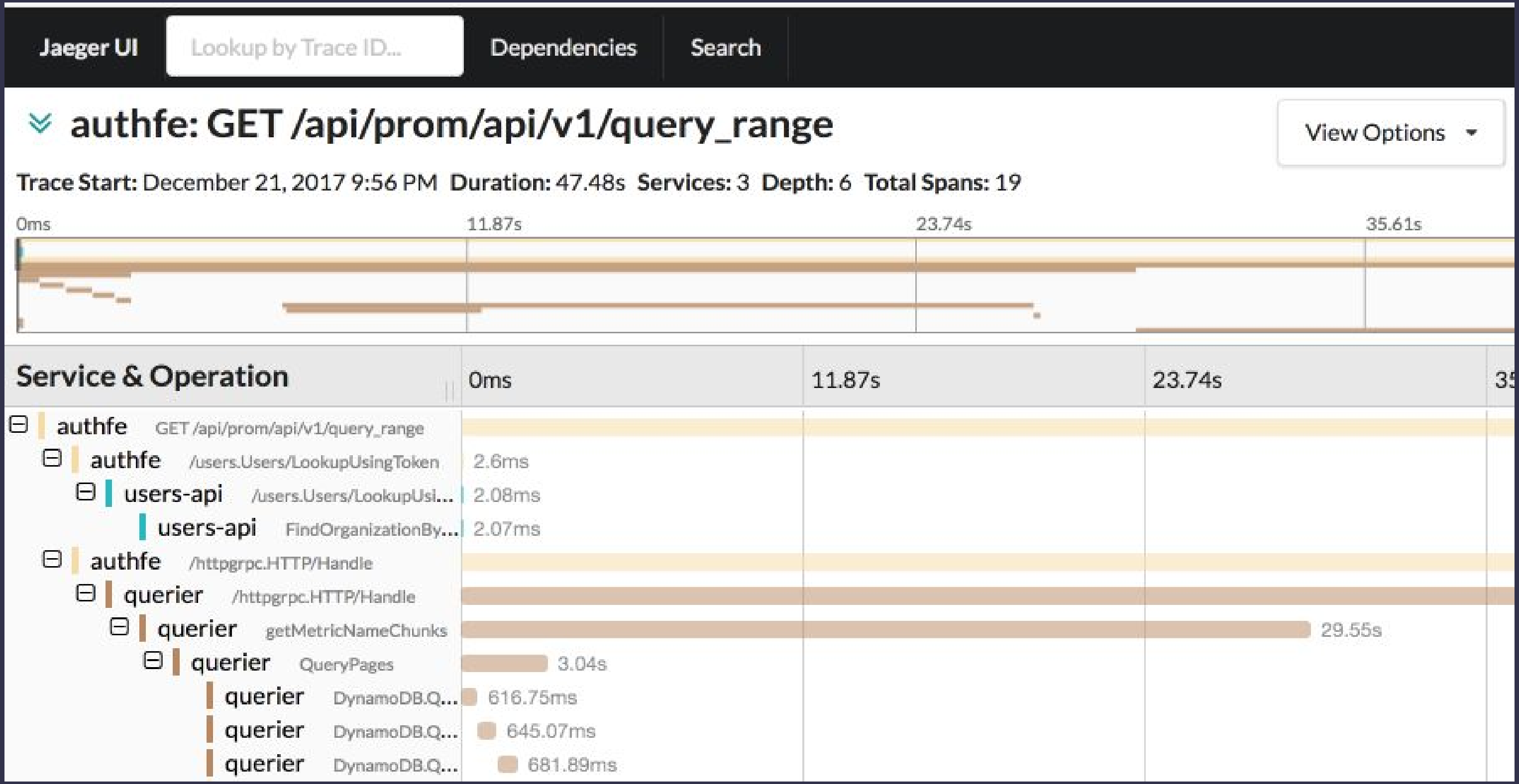


Weave Cloud (dev) – Monitor

Secure https://cloud.weave.works/prom/cold-sky-72/dashboard/service-resources?namespace=kube-system&timestamp=2018-01-26T...

weaveworks

Measure in detail



authfe: GET /api/prom/api/v1/query_range

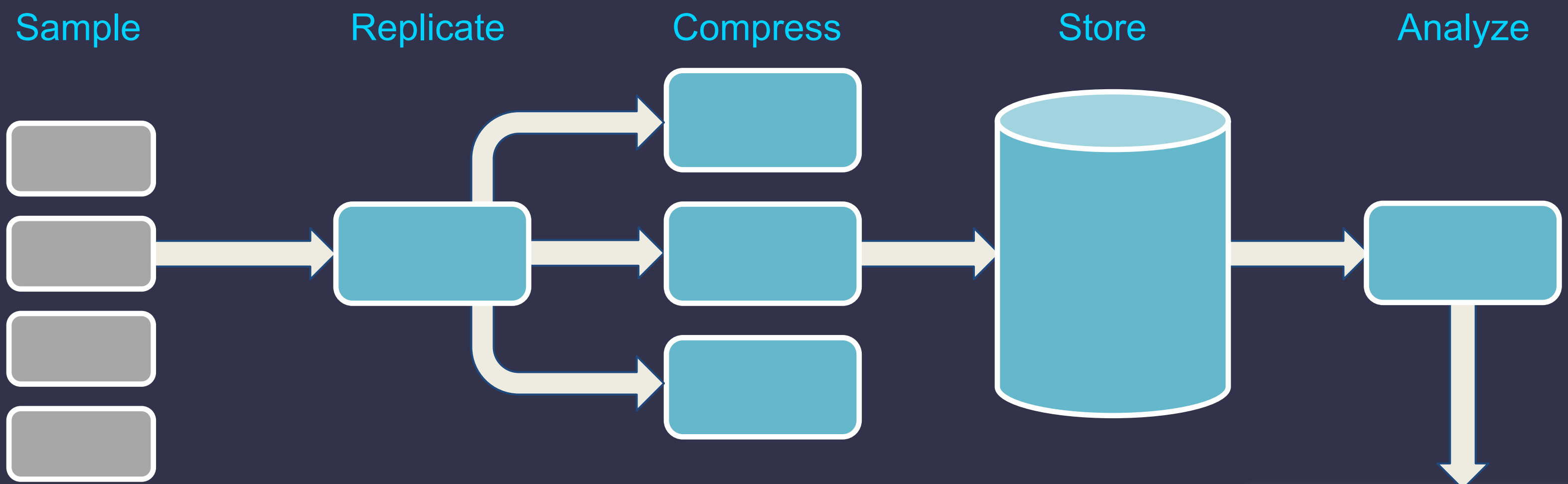
Trace Start: December 21, 2017 9:56 PM **Duration:** 47.48s **Services:** 3 **Depth:** 6 **Total Spans:** 19

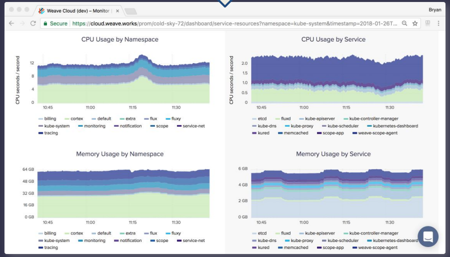
Service & Operation		0ms 11.87s 23.74s 35
authfe	GET /api/prom/api/v1/query_range	
authfe	/users.Users/LookupUsingToken	2.6ms
users-api	/users.Users/LookupUsi...	2.08ms
users-api	FindOrganizationBy...	2.07ms
authfe	/httpgrpc.HTTP/Handle	
querier	/httpgrpc.HTTP/Handle	
querier	getMetricNameChunks	29.55s
querier	QueryPages	3.04s
querier	DynamoDB.Q...	616.75ms
querier	DynamoDB.Q...	645.07ms
querier	DynamoDB.Q...	681.89ms

weaveworks

Jaeger - Distributed tracing https://jaegertracing.io

Drawing the charts




Weave Cloud (dev) – Monitor
Bryan
Secure
https://cloud.weave.works/prom/cold-sky-72/dashboard/service-resources?namespace=kube-system&timestamp=2018-01-26T...

CPU Usage by Namespace

CPU Usage by Service

Memory Usage by Namespace

Memory Usage by Service

Weave Cortex- multi-tenant Prometheus
https://github.com/weaveworks/cortex

OK, now Profiling

Basic instructions: http://blog.golang.org/profiling-go-programs

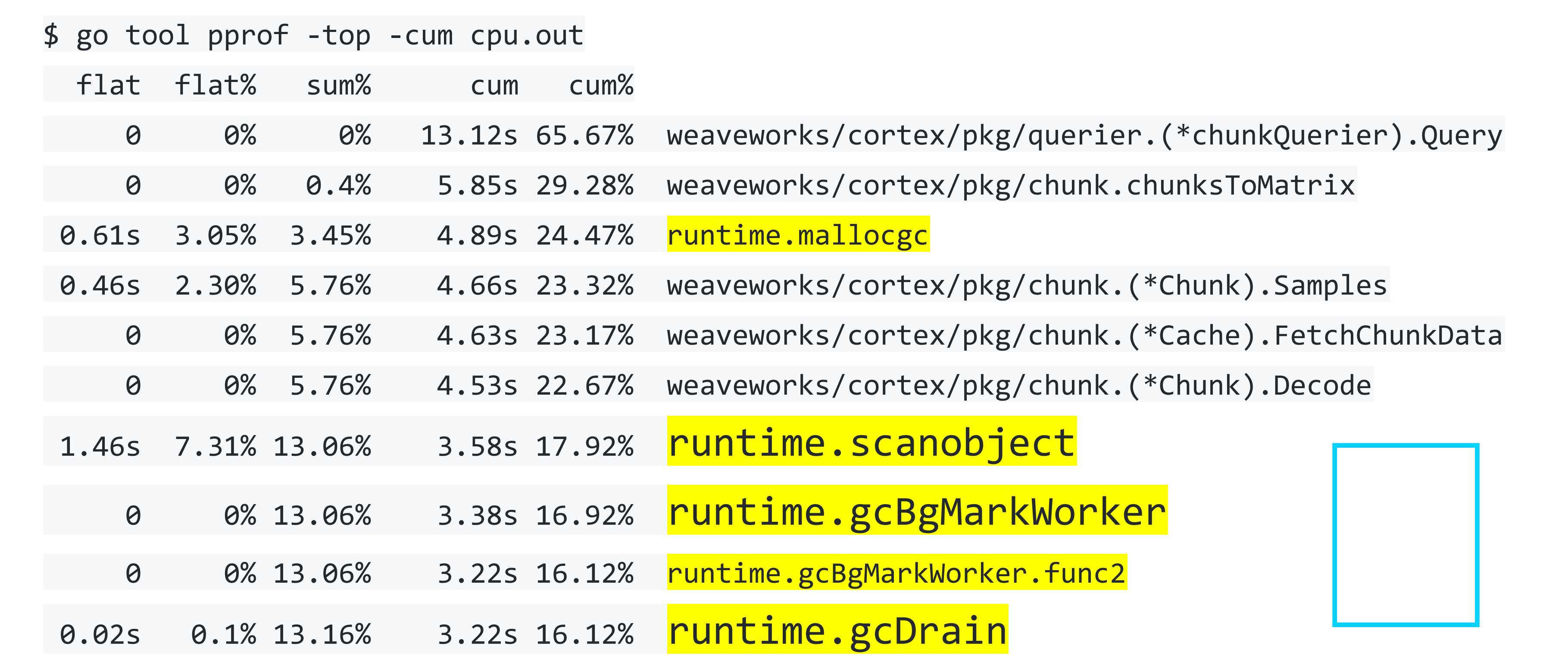
```
$ go test -cpuprofile=cpu.out
$ go tool pprof cpu.out
```

```
import _ "net/http/pprof"
```

What’s going on here?

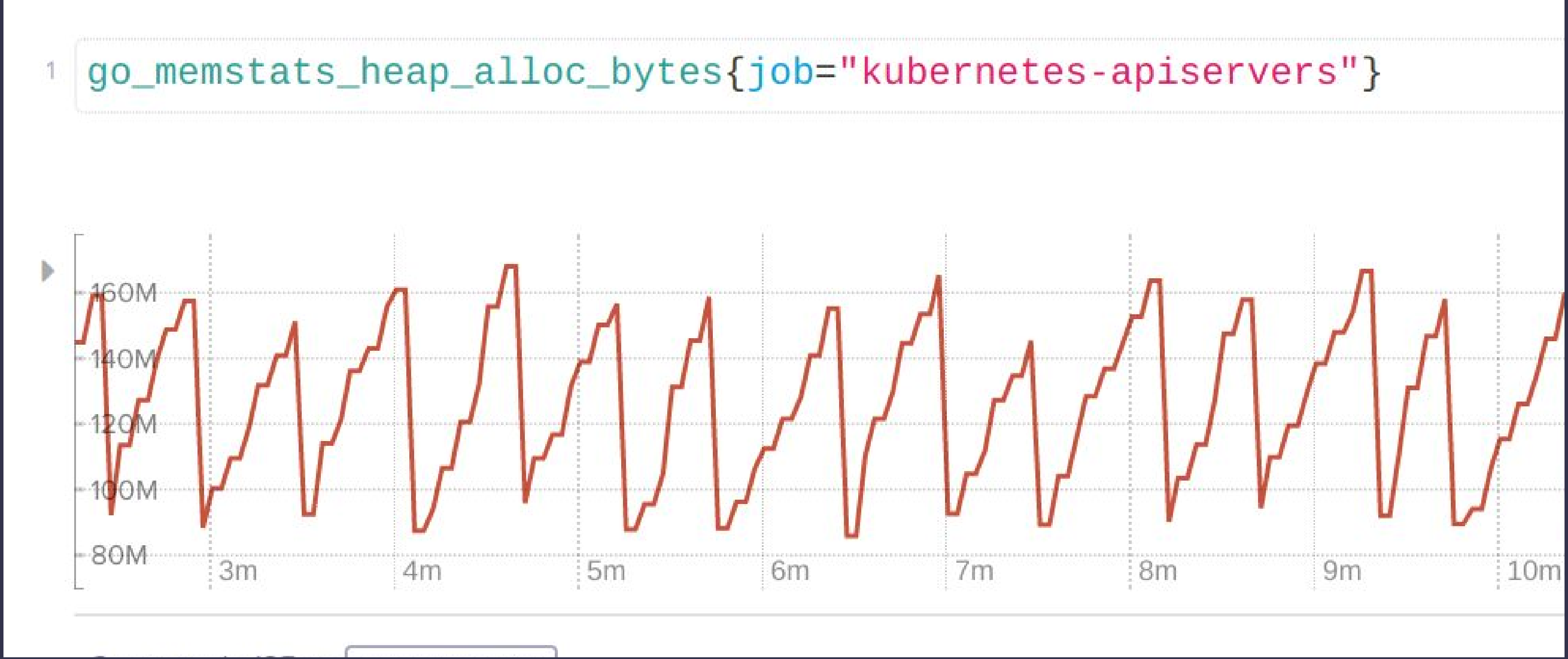
```
$ go tool pprof -top -cum cpu.out
  flat  flat%   sum%        cum   cum%
     0     0%     0%     13.12s 65.67%  weaveworks/cortex/pkg/querier.(*chunkQuerier).Query
     0     0%   0.4%      5.85s 29.28%  weaveworks/cortex/pkg/chunk.chunksToMatrix
 0.61s  3.05%  3.45%      4.89s 24.47%  runtime.mallocgc
 0.46s  2.30%  5.76%      4.66s 23.32%  weaveworks/cortex/pkg/chunk.(*Chunk).Samples
     0     0%  5.76%      4.63s 23.17%  weaveworks/cortex/pkg/chunk.(*Cache).FetchChunkData
     0     0%  5.76%      4.53s 22.67%  weaveworks/cortex/pkg/chunk.(*Chunk).Decode
 1.46s  7.31% 13.06%      3.58s 17.92%  runtime.scanobject
     0     0% 13.06%      3.38s 16.92%  runtime.gcBgMarkWorker
     0     0% 13.06%      3.22s 16.12%  runtime.gcBgMarkWorker.func2
 0.02s   0.1% 13.16%      3.22s 16.12%  runtime.gcDrain
```

Garbage Collection!



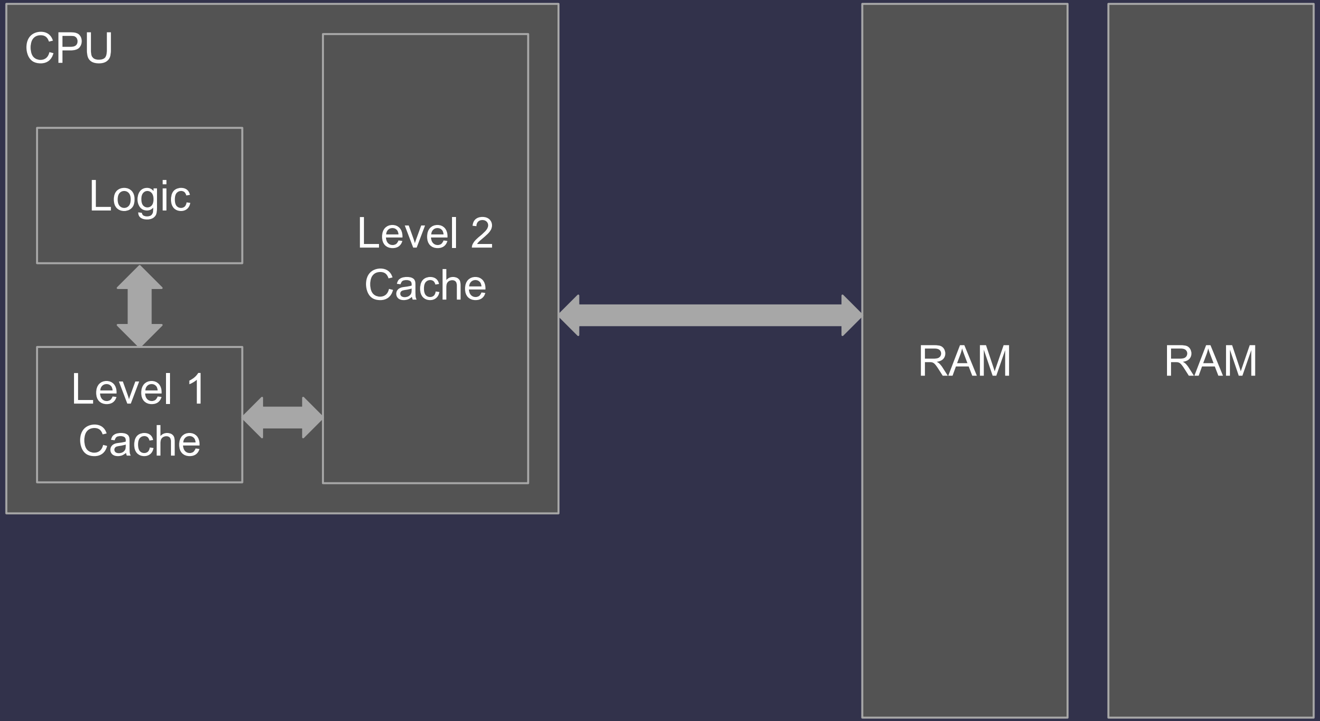
```
$ go tool pprof -top -cum cpu.out
  flat  flat%   sum%        cum   cum%
     0     0%     0%     13.12s 65.67%  weaveworks/cortex/pkg/querier.(*chunkQuerier).Query
     0     0%   0.4%      5.85s 29.28%  weaveworks/cortex/pkg/chunk.chunksToMatrix
 0.61s  3.05%  3.45%      4.89s 24.47%  runtime.mallocgc
 0.46s  2.30%  5.76%      4.66s 23.32%  weaveworks/cortex/pkg/chunk.(*Chunk).Samples
     0     0%  5.76%      4.63s 23.17%  weaveworks/cortex/pkg/chunk.(*Cache).FetchChunkData
     0     0%  5.76%      4.53s 22.67%  weaveworks/cortex/pkg/chunk.(*Chunk).Decode
 1.46s  7.31% 13.06%      3.58s 17.92%  runtime.scanobject
     0     0% 13.06%      3.38s 16.92%  runtime.gcBgMarkWorker
     0     0% 13.06%      3.22s 16.12%  runtime.gcBgMarkWorker.func2
 0.02s   0.1% 13.16%      3.22s 16.12%  runtime.gcDrain
```

Garbage Collection, visualised



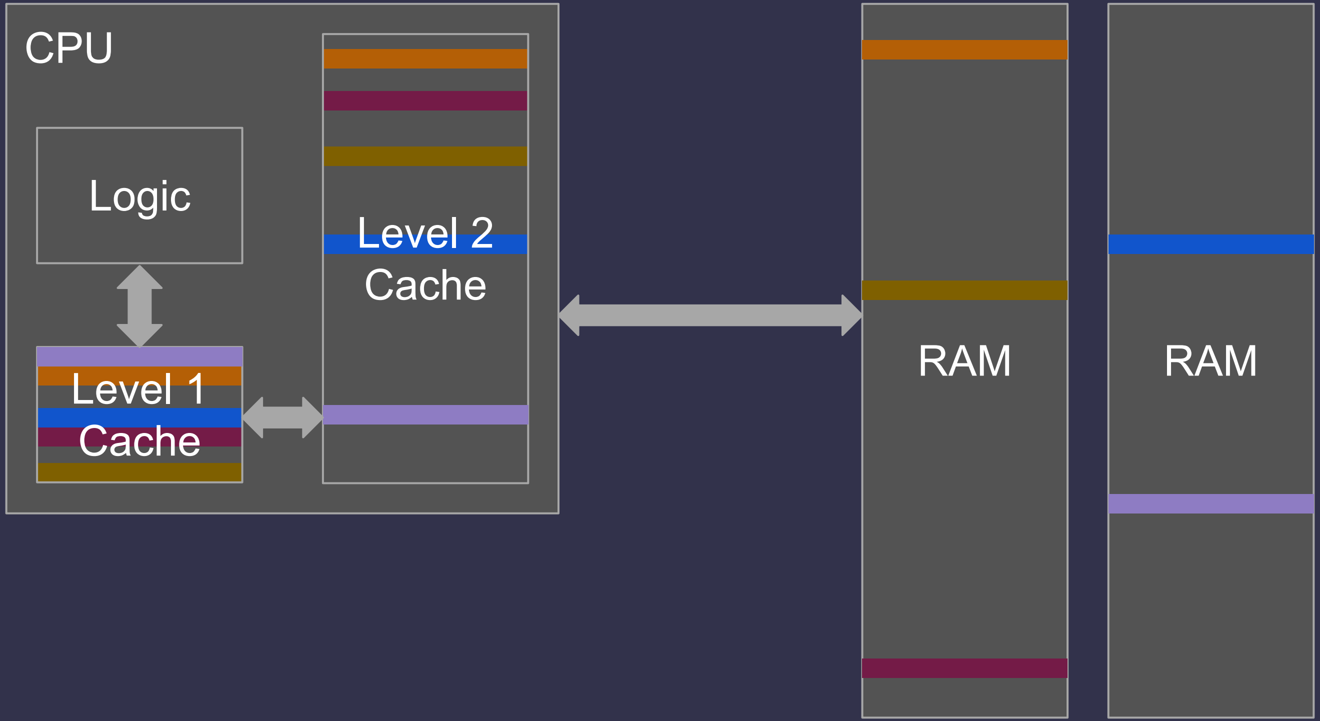
```
go_memstats_heap_alloc_bytes{job="kubernetes-apiservers"}
```

CPU memory architecture

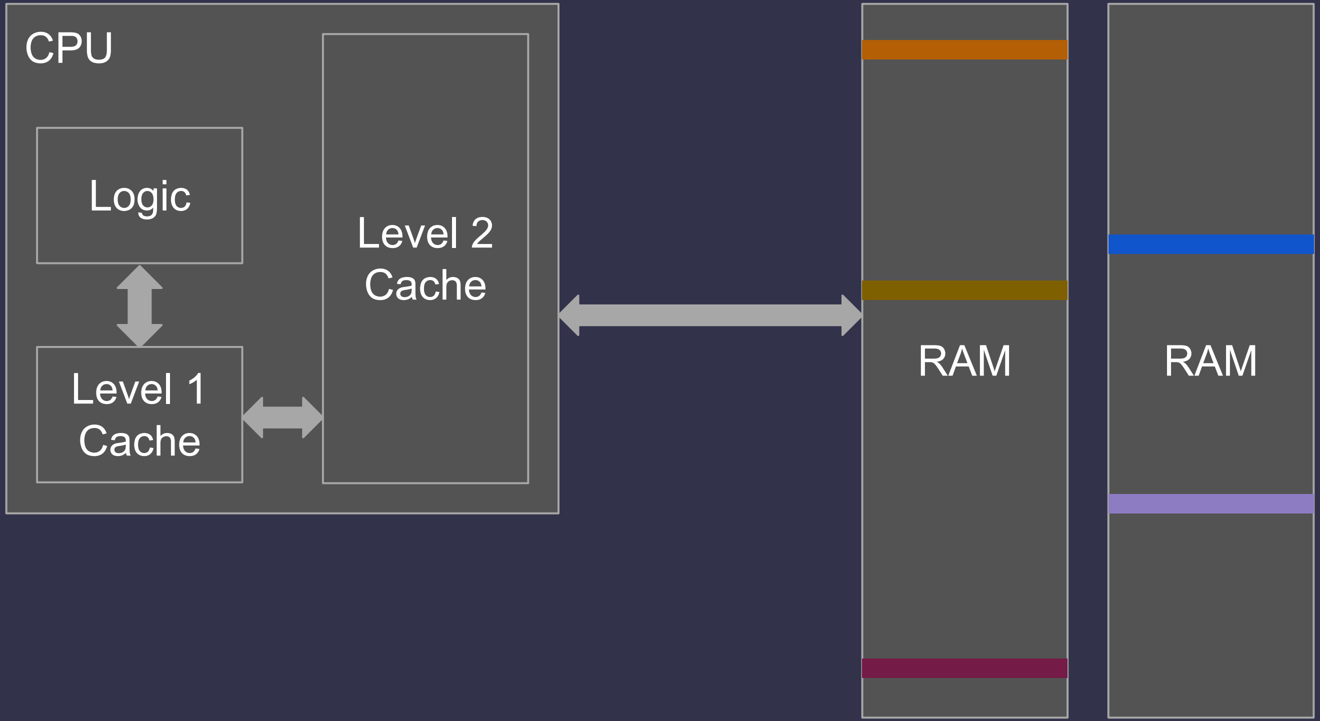


(Not to scale)

Caches in use



After GC has run



Memory Profile

```
$ go tool pprof -alloc_objects -top -cum mem.profile
    flat  flat%        cum   cum%
       0     0%   85063816 80.84%  .../cortex/pkg/chunk.(*Store).Get
45679033    43%   61078016 58.05%  .../cortex/chunk.(*Chunk).ExternalKey
       0     0%   56523148 53.72%  .../cortex/chunk.ByKey.Less
       0     0%   56523148 53.72%  sort.Sort
 1818786   1.7%   22562147 21.44%  .../cortex/chunk.(*Chunk).Decode
       0     0%   19784133 18.80%  encoding/json.(*decodeState).unmarshal
 3227746   3.1%   19456448 18.49%  encoding/json.(*decodeState).object
15398983 14.6%   15401714 14.64%  fmt.Sprintf
...
```

Memory profile options

`-inuse_space` - bytes allocated but not freed

`-inuse_objects` - count of objects allocated but not freed

`-alloc_space` - bytes allocated, including those freed

`-alloc_objects` - count of objects allocated

`-memprofilerate` - how often samples are taken

Avoidance strategies

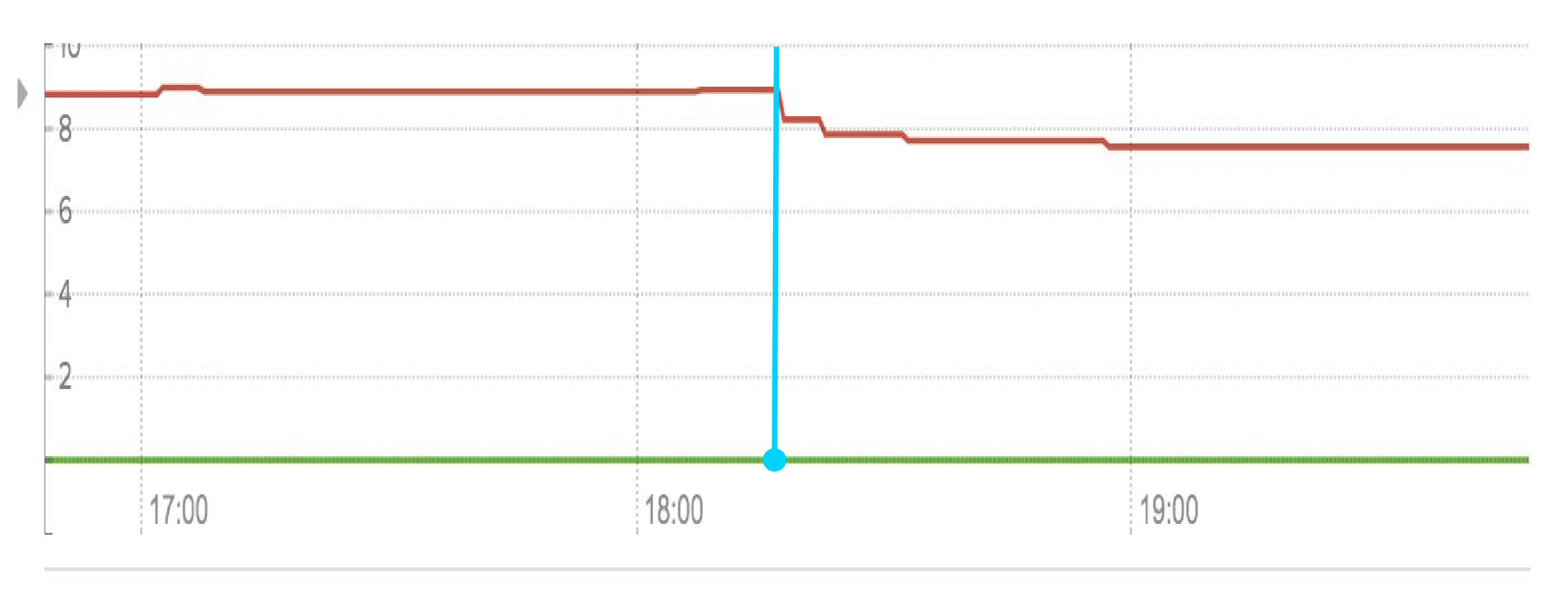
- Reuse
- Reduce
- Recycle

Anecdote: Decompressor

```
$ go tool pprof -alloc_space -top -cum mem.out
      flat  flat%   sum%          cum   cum%
         0     0%     0%   1529.93MB   100% chunk.BenchmarkDecode
                                weaveworks/cortex/chunk/chunk_test.go
1442.37MB 94.25% 94.25%   1442.37MB 94.25% chunk.Decode
                                .../vendor/github.com/golang/snappy/decode.go
```

`snappy.NewReader(r)` → `sync.Pool`

“Reuse” impact



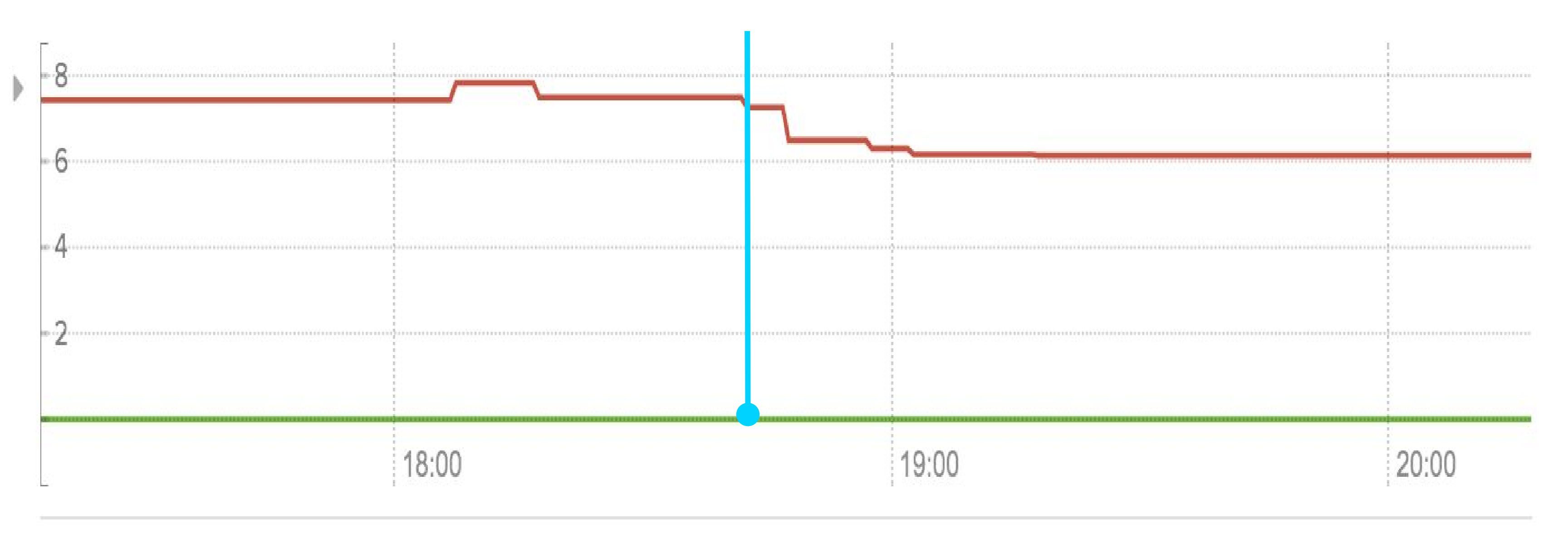
Anecdote: Sort Comparison

```
func (cs ByKey) Less(i, j int) bool {
    return cs[i].ExternalKey() < cs[j].ExternalKey()
}

func (c *Chunk) ExternalKey() string {
    return fmt.Sprintf("%s/%d:%d:%d", c.UserID, c.Fnprint,
       c.From, c.Through)
}
```

→ Compare data directly

“Reduce” impact



weaveworks

Stack vs Heap

```
var x int
```

```
var y = make([]int, n)
```

Stack vs Heap

http://www.clipartpanda.com/clipart_images/stack-files-max-39545601

Photo: JohnNyberg, rgbstock.com

Which of these is on the heap?

```
func BenchmarkOne(b *testing.B) {
    var buf io.Writer = &bytes.Buffer{}

    for i := 0; i < b.N; i++ {
        var data = []byte("hello")
        buf.Write(data)
    }
}
```

Benchmark Stats

```
    var buf io.Writer = &bytes.Buffer{}
    for i := 0; i < b.N; i++ {
        var data = []byte("hello")
        buf.Write(data)
    }

$ go test -bench=. -benchmem
BenchmarkOne    30000000   47.2 ns/op    27 B/op    1 allocs/op
```

Memory Profile

```
    var buf io.Writer = &bytes.Buffer{}
    for i := 0; i < b.N; i++ {
        var data = []byte("hello")
        buf.Write(data)
    }
```

```
$ go test -bench=. -memprofile=mem.out
$ go tool pprof -alloc_objects -top -cum mem.out
      flat  flat%   sum%        cum   cum%
         0     0%     0%    9830599   100%  testing.(*B).launch
   9830550   100%   100%    9830565   100%  BenchmarkOne
```

Line-by-line profile

```
$ go tool pprof -alloc_objects -list=BenchmarkOne mem.out
9830550  9830565 (flat, cum)   100% of Total
      .        .      9: func BenchmarkOne(b *testing.B) {
      .        .     10:    var buf io.Writer = &bytes.Buffer{}
      .        .     11:
      .        .     12:    for i := 0; i < b.N; i++ {
9830550  9830550     13:        var data = []byte("hello")
      .       15     14:        buf.Write(data)
      .        .     15:    }
```

Escape Analysis

```
    for i := 0; i < b.N; i++ {
        var data = []byte("hello")
        buf.Write(data)
    }

$ go test -gcflags '-m -m'
test.go:13:27: ([]byte)("hello") escapes to heap
test.go:13:27: from data (assigned) at ./one_test.go:13:7
test.go:13:27: from buf.Write(data) (parameter to indirect call)
at test.go:14
```

Which kinds of things escape?

- Address is passed out of a function
- Parameters of indirect calls
- Passed to a chan or a goroutine, or defer
- Others...
  - Arguments of recursive calls
  - Added to a slice or map
  - Passed to `panic()`
  - Too large for stack

Action points

- Measure your system
  - If CPU is high, look at profile
  - If GC is high, look at memory allocations
- It’s always memory allocations. 😉
- Avoid via:
  - Stack instead of heap
  - Different algorithm
  - Pooled objects

Thanks! Questions?

We are hiring!

Engineers in Berlin & SF

weave.works/hiring