

애드리,
줄여야 한다

@GDG Golang Korea May Meetup

발표자 소개

김효준

Lead of Data R&D Team, AB180

다양한 기술을 사용자 경험에 녹여내,
뛰어난 제품을 만드는데 관심이 많습니다.

백엔드 안드로이드
딥러닝 블록체인
데이터 엔지니어링



Go 개발을 하면서
우리는 반드시 메모리 문제를 겪는다

우리의 문제 컴포넌트



Luft

Luft: Realtime OLAP Database by AB180

에어브릿지 (airbridge.io) 에서 실시간으로 유저 행동을 분석하기 위해 만든 데이터베이스

- **Fast:** 5초 내로 수억개 Row를 스캔해 쿼리 제공
- **Real-Time:** 람다 아키텍쳐로 실시간 데이터 처리
- **High Availability:** 데이터는 색인되고 S3에 저장됨
- **Cloud Native:** 클라우드에 직접 연동되어 유연한 스케일링 및 확장

Luft: Realtime OLAP Database by AB180



메모리, 줄여야한다 — 인트로

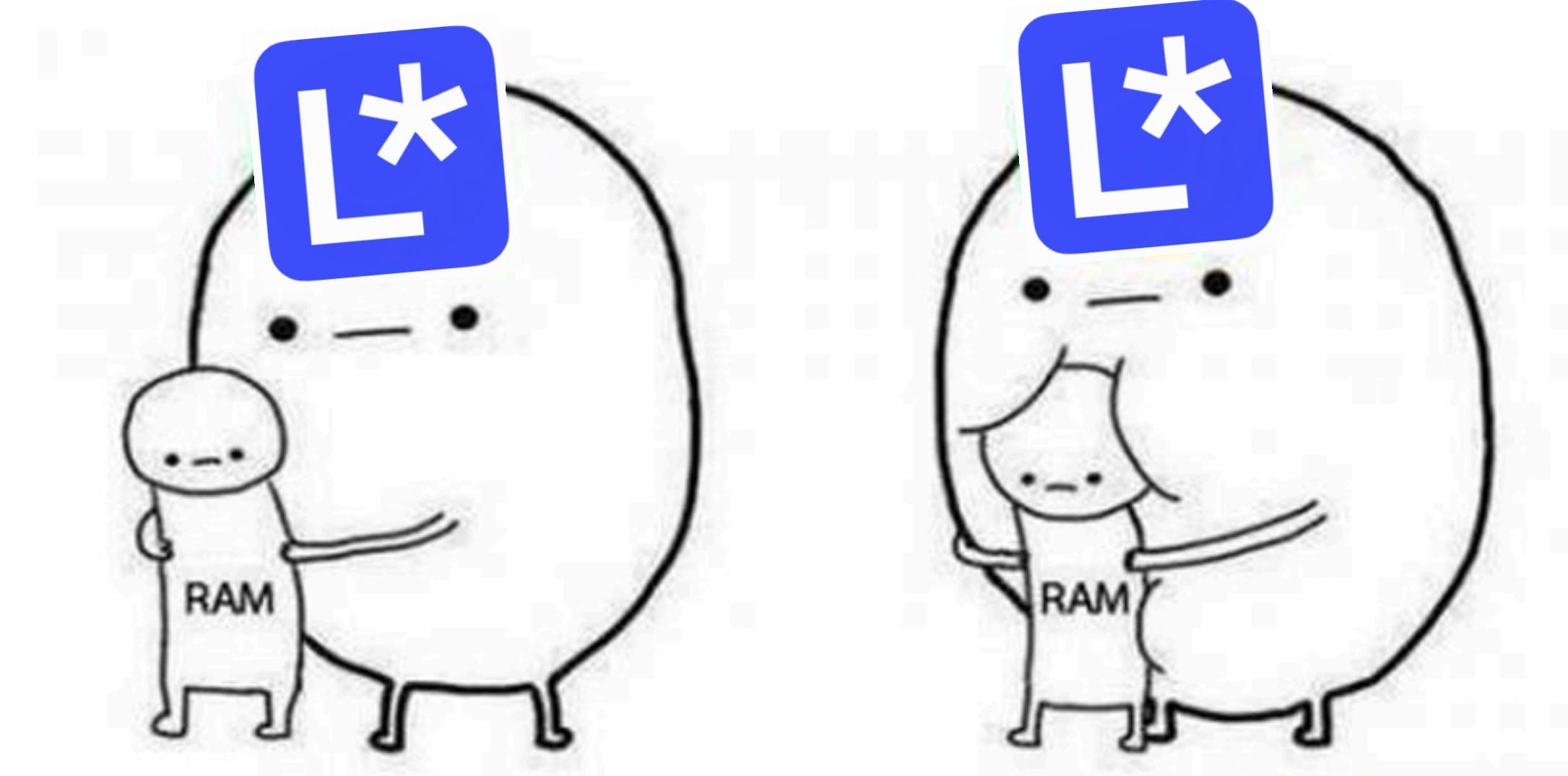
자세한 소개 → abit.ly/luft

Luft에서 생긴 메모리 문제

문제: 데이터를 쿼리할 때 OOM이 남

- MAU 500만 고객사의 수억건 데이터가 Luft에서는 단 3GB밖에 되지 않음
- 문제는 3GB를 쿼리하는데 노드당 메모리를 18GB 이상 점유하면서 OOM 발생
- **프로덕션까지 기간은 3주, 메모리를 줄여야 한다.**

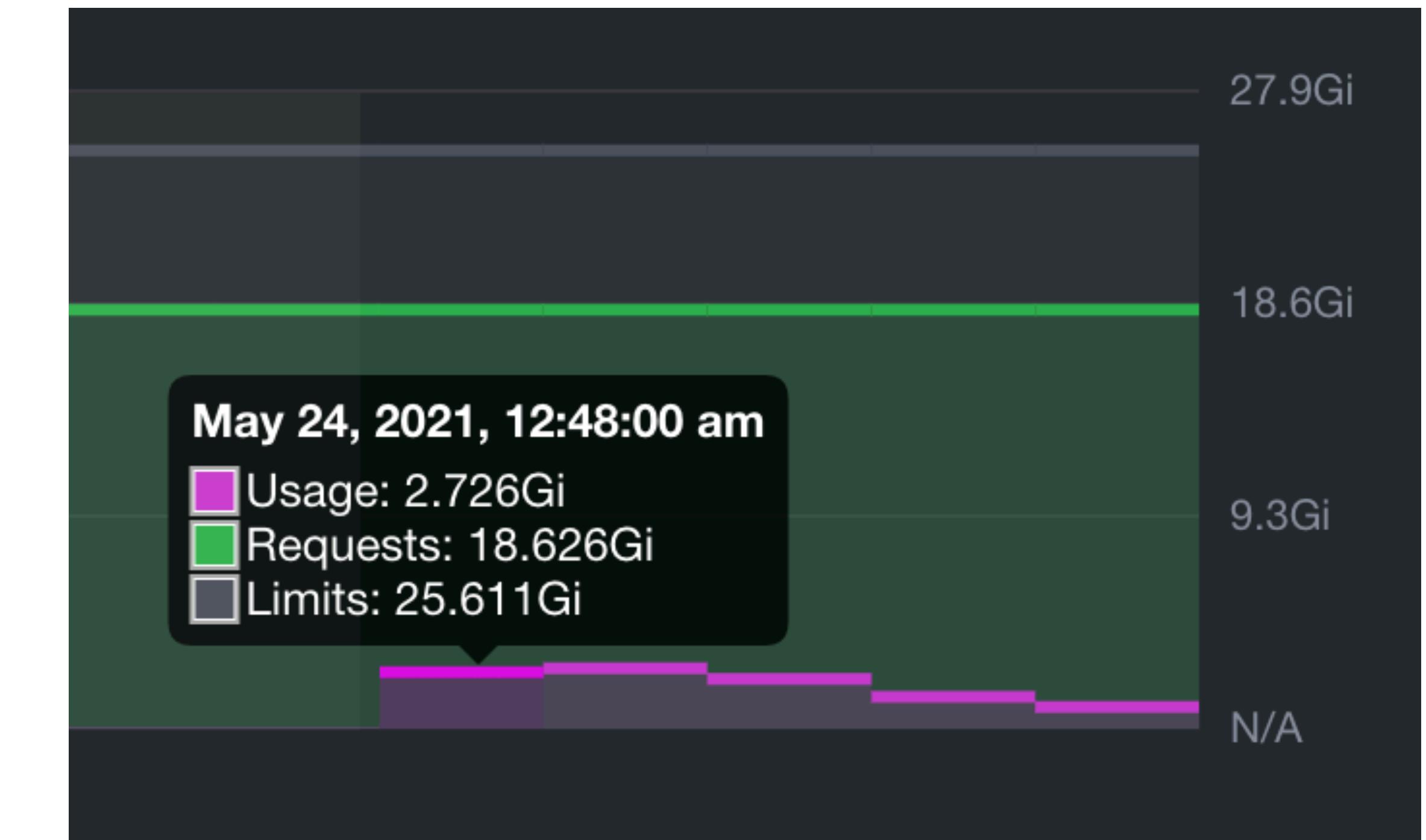
메모리, 줄여야한다 — 인트로



결말 스포: 그래서 결국 얼마나 최적화했나?

18GB → 2.76GB!

기존엔 한두번의 쿼리로도 OOM이 났지만,
이제는 피크 2.7GB 찍은 후 안정적으로 램 유지



그래서 어떻게 이렇게 줄일 수 있었을까요...?

메모리, 줄여야한다 – 인트로

1. 메모리를 줄이는 습관

메모리, 줄여야한다 – 메모리를 줄이는 습관

1. 1. 재사용 할 수 있으면 재사용하기

I/O 처리 (e.g. 파일, 스트림) 할 땐 대부분의 경우 **중간 버퍼 재사용이 가능하다.**
보통 프레임워크에서 해주지만, 그렇지 않은 경우 재사용하는 습관 들이기

```
for {  
    row := new(Buffer)  ↗  
    read(row)  
    ...  
    ...  
    row := new(Buffer)  
    for {  
        read(row)  
        ...  
    }  
}
```

메모리, 줄여야한다 – 메모리를 줄이는 습관

[실제 32GB짜리 OOM을 막은 감동적인 커밋 내용]

The screenshot shows a side-by-side code diff tool comparing two versions of a Go file named `stages.go`. The left pane is labeled `f767a0696a53f36fb044be0b6ce3fd968ffd4b22` and the right pane is labeled `97f85f6d3b34846541756c2e939b90d9c30ff19e`. The interface includes standard diff controls like arrows, a pencil icon, and a search bar. The code itself is in Go, showing a function that processes input rows and partitions them by key. The right version contains several changes, notably at lines 309, 310, 314, and 315, which appear to be optimizations or bug fixes related to memory management.

```
stages.go (/Users/vista/Dropbox/Projects/cohort-engine-poc/pkg/indexer)

Side-by-side viewer | Do not ignore | Highlight words | ? | 4 differences

f767a0696a53f36fb044be0b6ce3fd968ffd4b22
✓ if err != nil {
    return errors.Wrapf(err, "locate w")
}
chunksByPartitionKey := make(map[string]chan<parse.ResultRow>)
// takes parsed rows from previous stage
for in := range inputs {
    row := new(parse.ResultRow)
    in.UnmarshalValue(row)
    partitionKey := in.Key
    if partitionKey == "" && s.Table.SecondaryKey != nil {
        partitionKey = ctx.PartitionID()
    }
}

97f85f6d3b34846541756c2e939b90d9c30ff19e
if err != nil {
    return errors.Wrapf(err, "locate wor
}
chunksByPartitionKey := make(map[string]chan<parse.ResultRow>)
// takes parsed rows from previous stage
row := new(parse.ResultRow)
for in := range inputs {
    in.UnmarshalValue(row)
    partitionKey := in.Key
    if partitionKey == "" && s.Table.SecondaryKey != nil {
        partitionKey = ctx.PartitionID()
    }
}
```

메모리, 줄여야한다 – 메모리를 줄이는 습관

재사용 예시: csv.Reader

csv.Reader의 경우 각 row를 담는 버퍼를
재사용할 수 있는 옵션이 있음

```
reader := csv.NewReader(file)
reader.ReuseRecord = true

for {
    row, err := reader.Read()
    if err == io.EOF {
        break
    }
}
```

재사용 주의: row를 다른 곳에서 참조하면
내용물이 바뀌어버리는 버그가 발생!

메모리, 줄여야한다 – 메모리를 줄이는 습관

직접 재사용이 힘들다면: sync.Pool

여러 고루틴에서 동시다발적 I/O가 일어난다면
공통 버퍼를 만들고 재사용하기 어렵다.

객체를 알아서 쓰고 반납할 수 있게 해주는
sync.Pool을 활용해보자.

```
var pool = sync.Pool{
    New: func() interface{} {
        return &Object{}
    },
}

func myHandler() {
    obj := pool.Get().(*Object)
    // ...
    pool.Put(obj)
}
```

More Control: Buffered Channel

sync.Pool의 단점: 오브젝트가 견잡을 수 없이
늘어나거나, 멋대로 GC되는 걸 제어할 수 없다.

Buffered Channel을 응용하면 메모리 풀의 크기와
재사용 동작을 직접 커스터마이징할 수 있음

참조: [Recycling Memory Buffers in Go - Cloudflare](#)

메모리, 줄여야한다 – 메모리를 줄이는 습관

```
pool := make(chan *BigObj, 100)
for i := 0; i < cap(pool); i++ {
    pool <- new(BigObj)
}

...
select {
    // try to get one from pool
    case obj = <-pool:
        // don't forget to return!
        defer func() { pool <- obj }()
}

default:
    // all objects are in use
    obj = new(BigObj)
}
```

1. 2. Map/Slice 할당 시 Capacity 정해서 쓰기

우리가 만드는 대부분의 Map, Slice는 사실 크기를 미리 알고 있을 가능성이 높다.
Capacity를 정해놓고 할당하는 습관을 들이자.

```
arr := make([]byte, 0, SIZE)  
arr = append(arr, elem)
```

메모리, 줄여야한다 – 메모리를 줄이는 습관

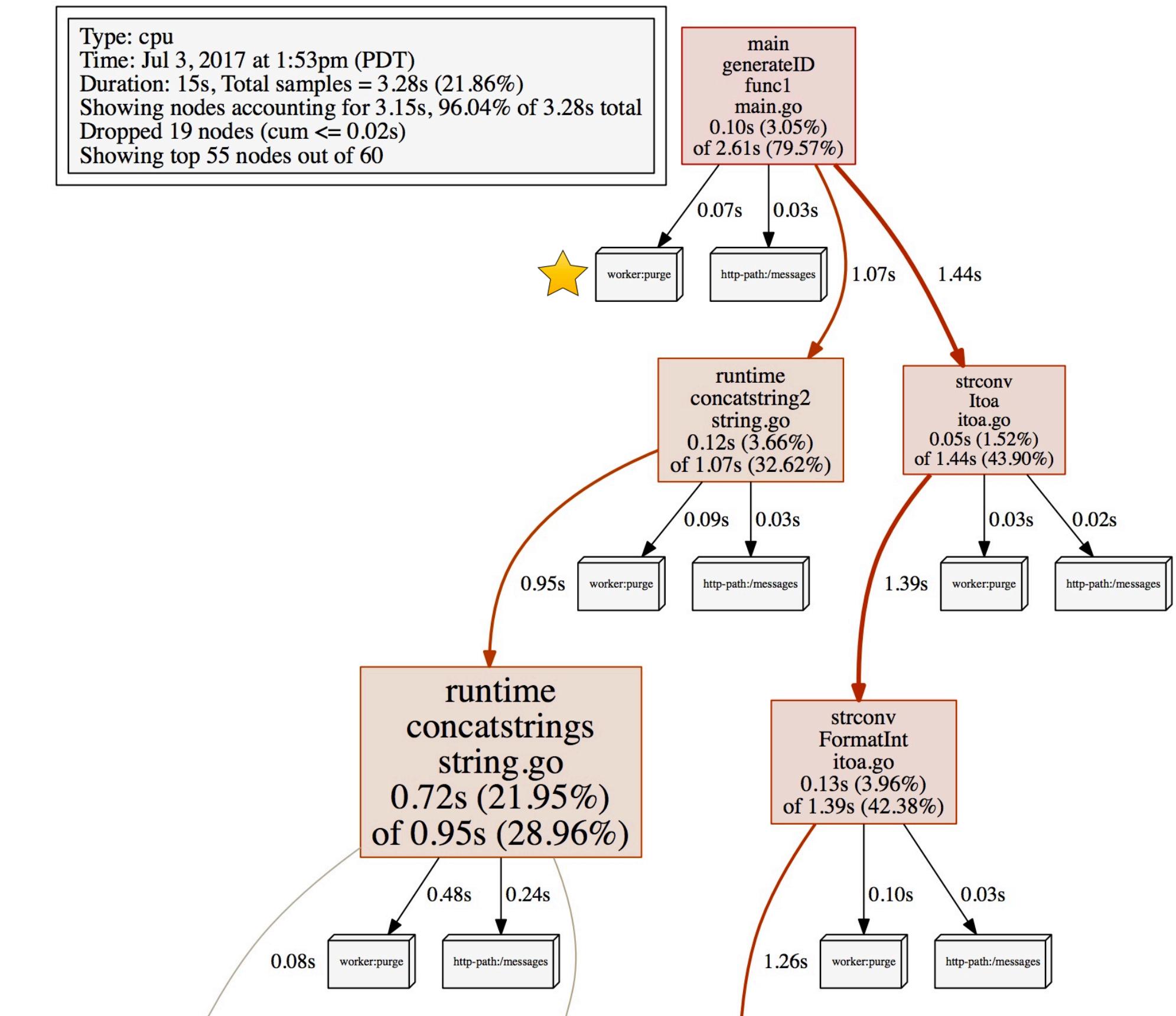
2. 그래도 메모리를 많이 먹는다면 어떻게 진단할 수 있을까?

2.1. pprof

문제가 생겼을 땐 무조건 프로파일링부터 시작하자.

Go가 공식적으로 채택한 성능 프로파일링 도구

- **CPU Profile:** 메서드별 CPU 시간 측정
- **Heap Allocation Profile:** 할당한 메모리 프로파일
- **Heap Usage Profile:** 사용 중인 메모리 프로파일
- **Goroutine Profile:** 현재 실행중인 고루틴 보기
- **Many more...**



메모리, 줄여야한다 – 메모리 진단하기

pprof 분석 사례: Luft의 쿼리 OOM

3GB의 데이터를 쿼리하는데 16GB의 메모리를 전부 탕진함.

어디가 문제인지 알아보기 위해 pprof으로 inuse_space 프로파일 캡쳐

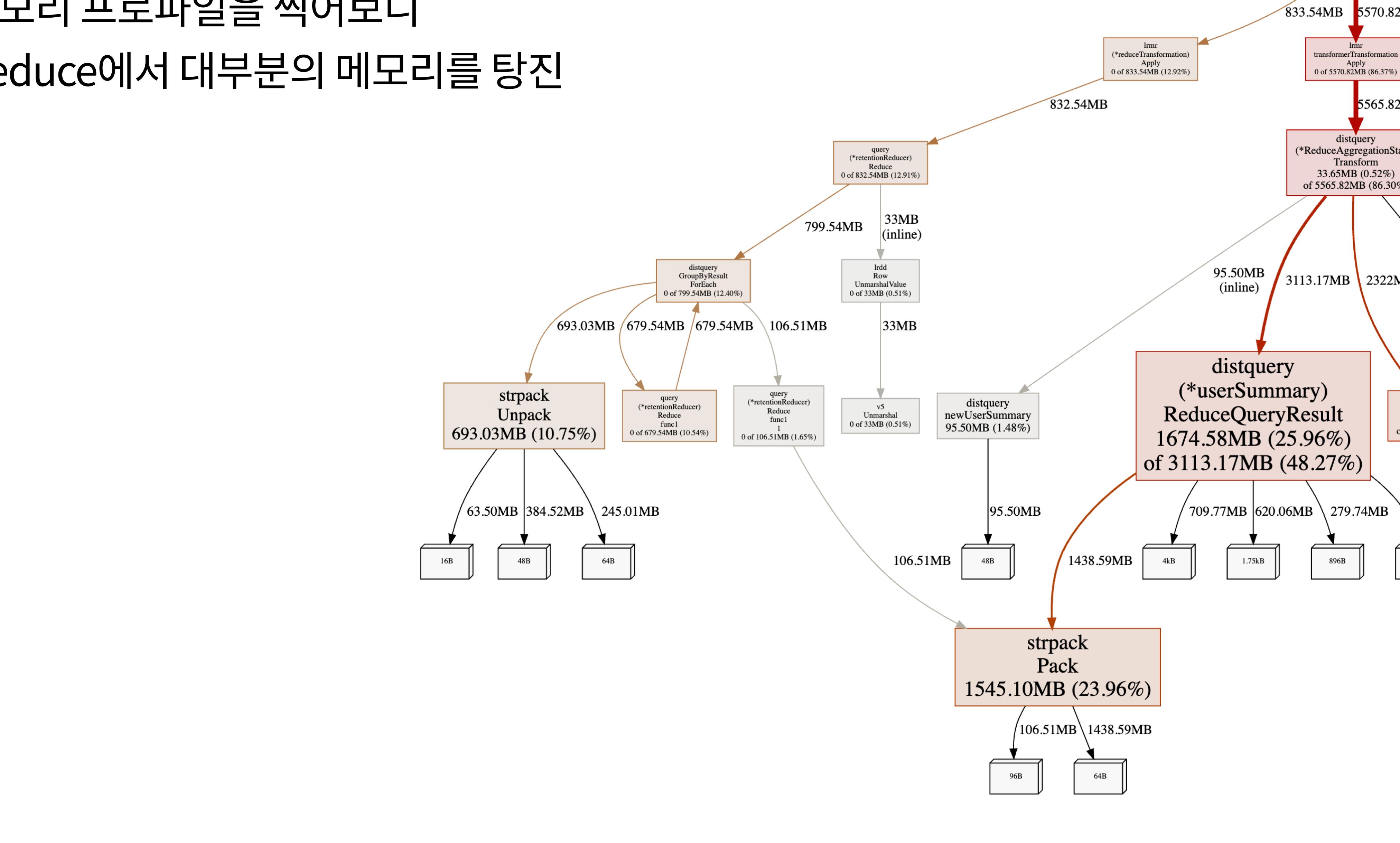
```
go tool pprof http://luft-master/debug/pprof
```

메모리, 줄여야한다 – 메모리 진단하기

분석 사례: Luft

메모리 프로파일을 찍어보니
Reduce에서 대부분의 메모리를 탕진

File: luft
Type: inuse_space
Time: May 11, 2021 at 12:40pm (KST)
Showing nodes accounting for 6350.86MB, 98.47% of 6449.76MB total
Dropped 103 nodes (cum <= 32.25MB)

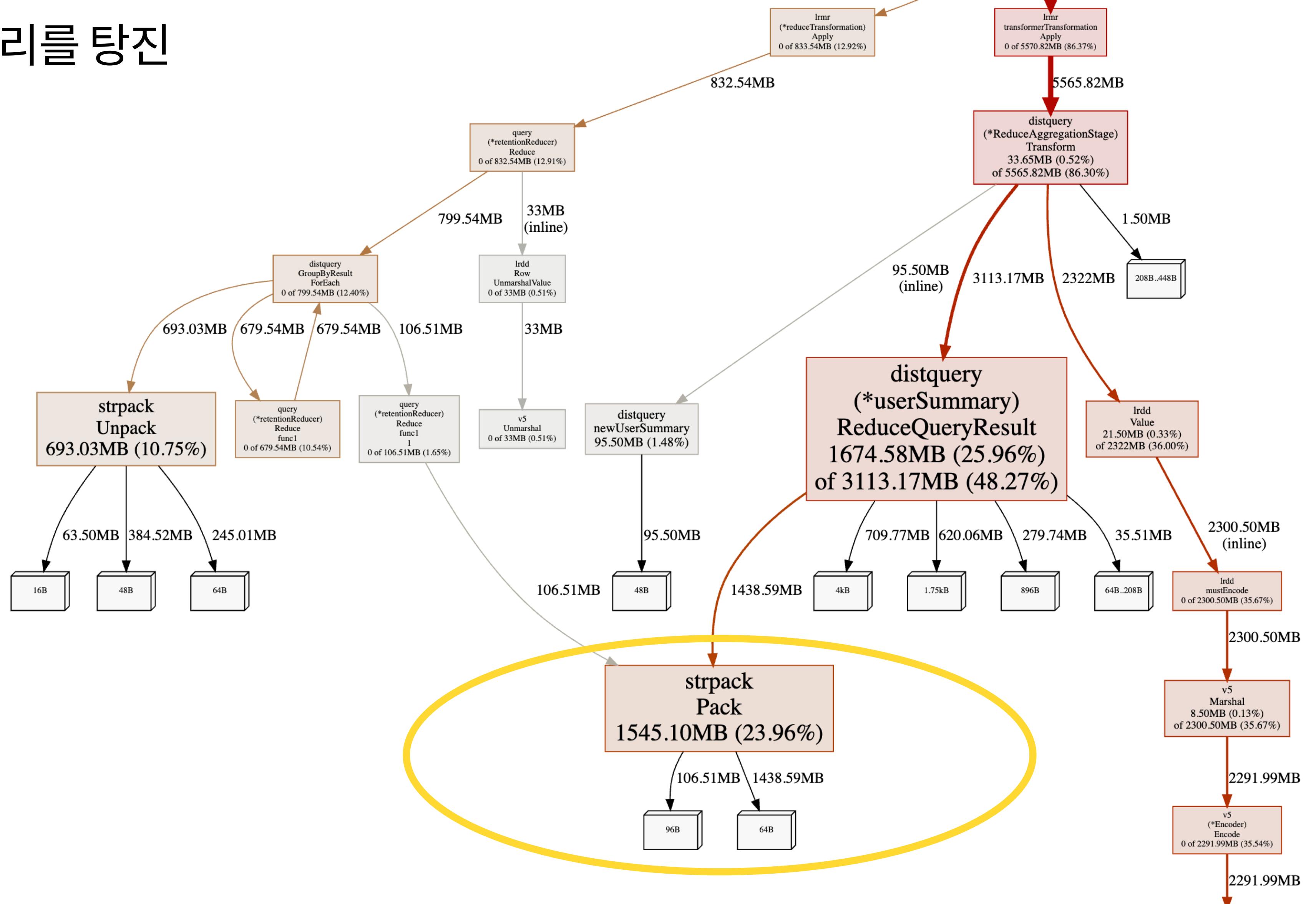


분석 사례: Luft

메모리 프로파일을 찍어보니

Reduce에서 대부분의 메모리를 탕진

File: luft
Type: inuse_space
Time: May 11, 2021 at 12:40pm (KST)
Showing nodes accounting for 6350.86MB, 98.47% of 6449.76MB total
Dropped 103 nodes (cum <= 32.25MB)

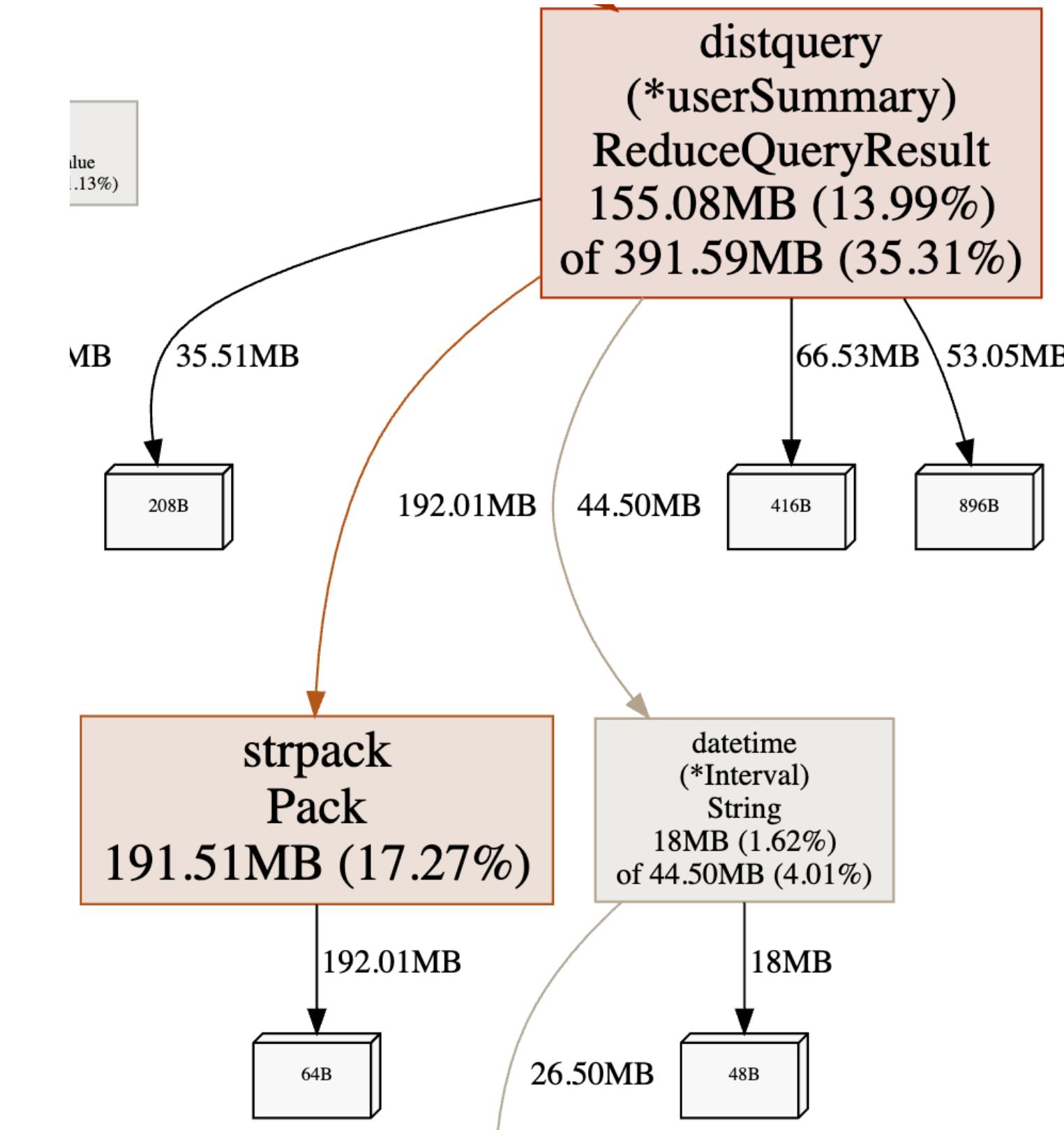
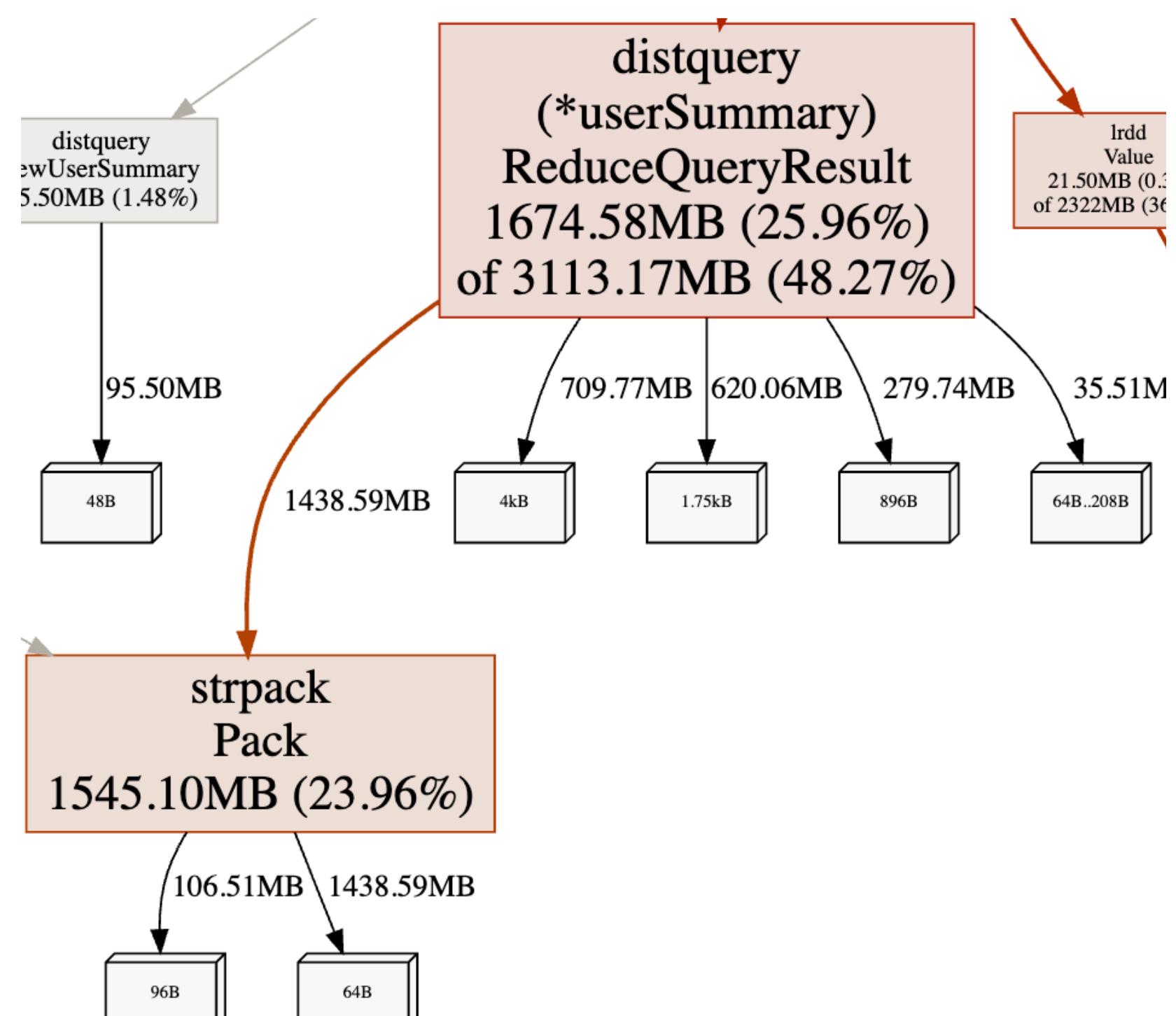


```
$ go test -bench=github.com/ab180/luft/pkg/strpack
```

BenchmarkPack-12	616 ns/op	392 B/op	17 allocs/op
BenchmarkUnpack-12	627 ns/op	352 B/op	15 allocs/op

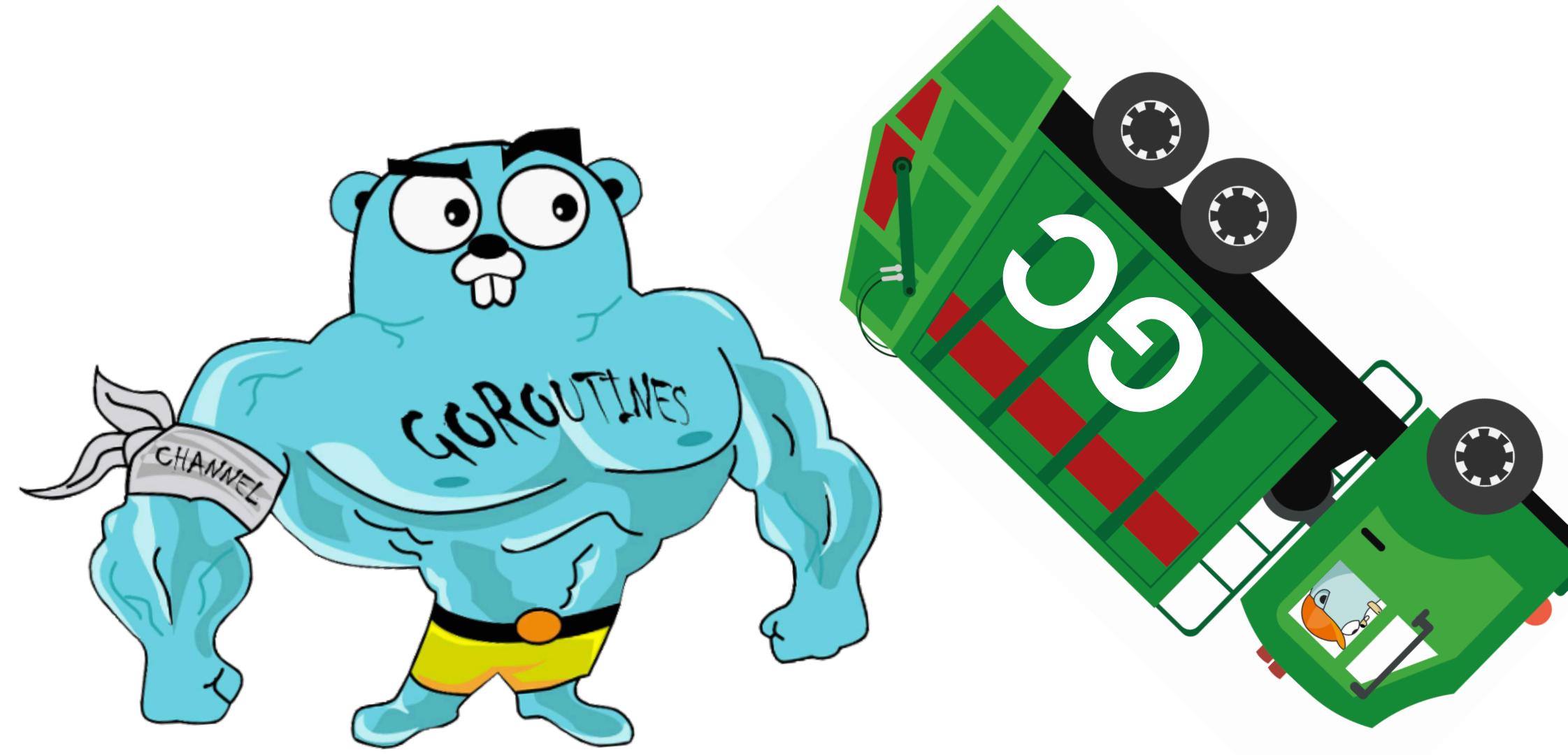


BenchmarkPack-12	112 ns/op	80 B/op	1 allocs/op
BenchmarkUnpack-12	446 ns/op	272 B/op	10 allocs/op



메모리, 줄여야한다 – 메모리 진단하기

3. 어떻게 메모리 누수를 예방할 수 있을까?



대부분 메모리 누수는
고루틴 누수로 인해 발생함

메모리, 줄여야한다 – 메모리 누수 예방하기

leaktest github.com/fortytw2/leaktest

테스트 종료 후에도 고루틴이 돌아가고 있으면 해당 테스트는 실패하게 됨.

대부분의 테스트에 달아놓으면 좋다! (`testing.Main` / `Setup` / `Teardown` 등 사용)

```
// Default "Check" will poll for 5 seconds to check that all
// goroutines are cleaned up
func TestPool(t *testing.T) {
    defer leaktest.Check(t)()

    go func() {
        for {
            time.Sleep(time.Second)
        }
    }()
}
```

4. 그래도 정 안된다면... 메모리를 줄이는 흑마법

4.1. 흑마법: Unsafe String <-> Byte

원래 String과 Byte간에 컨버젼을 하면 메모리 복사가 일어남

그러나 `unsafe.Pointer`로 변환하면, 복사 없이도 형변환 가능

(실제로 Zero Allocation를 표방하는 라이브러리에서 많이 쓰이는 기법)

```
buf := *(*[]byte)(unsafe.Pointer(&str))  
str := *(*string)(unsafe.Pointer(&buf))
```

4.1. 흑마법: Unsafe String <-> Byte

원래 String과 Byte는
그러나 unsafe

```
// StringHeader is the runtime representation of a string.  
// its representation may change in a later release.
```

```
type StringHeader struct {
```

```
    Data uintptr
```

```
    Len int
```

```
}
```

```
(상태)
```

```
(상태)
```

```
// SliceHeader is the runtime representation of a slice.
```

```
// its representation may change in a later release.
```

```
type SliceHeader struct {
```

```
    Data uintptr
```

```
    Len int
```

```
    Cap int
```

```
}
```

```
buf :=
```

```
(상태) &str )
```

```
str :=
```

```
(상태) &buf )
```

4.2. 흑마법: runtime.SetFinalizer

내가 할당한 객체가 언제 필요 없어질지 안다면
sync.Pool을 사용해서 재사용 가능.

하지만 내가 할당한 객체가 내가 통제할 수 없는
구역으로 보내진 상황에서도 재사용할 수 있을까?
(e.g. GRPC, 각종 HTTP 프레임워크)

`runtime.SetFinalizer(...)`

GC가 메모리를 해제하는 시점에 대한 콜백을 지정할 수 있는
runtime.SetFinalizer를 이용해보자.

func SetFinalizer

```
func SetFinalizer(obj interface{}, finalizer interface{})
```

`SetFinalizer` sets the finalizer associated with `obj` to the provided finalizer function. When the garbage collector finds an unreachable block with an associated finalizer, it clears the association and runs `finalizer(obj)` in a separate goroutine. This makes `obj` reachable again, but now without an associated finalizer. Assuming that `SetFinalizer` is not called again, the next time the garbage collector sees that `obj` is unreachable, it will free `obj`.

```
var responsePool = sync.Pool{
    New: func() interface{} { return &pb.SayHiOutput{} },
}

func (g *grpc) SayHi(...) (*pb.SayHiOutput, error) {
    resp := responsePool.Get().(*pb.SayHiOutput)
    resp.Message = "Hello world"

    // enter the dark side
    runtime.SetFinalizer(resp, func(obj interface{}) {
        responsePool.Put(obj)
    })
    return resp, nil
}
```

runtime.SetFinalizer, 얼마나 효과적일까?

출처: <http://www.golangdevops.com/2019/12/31/autopool/>

- **1000 Iterations:** 시간 9.9ms 단축, 메모리 433KiB 감소
- **100,000 Iterations:** 시간 484ms 단축, 메모리 4GiB 감소
- **3MiB Slices:** 시간 53초 단축 (-33%!!), 메모리 141GiB 감소

요약: 대규모 스케일에서는 시간 및 공간 최적화에 효과적이다.

4.3. 흑마법: nullify

메모리 누수 상황에서, 메모리 프로파일링을 통해서 누수 지점은 정확히 알았지만
무슨 짓을 해도 GC가 메모리를 해제하지 않을 때는 어떻게 해야 할까?

답: nil로 만들면 GC가 결국엔 해제한다.

```
// close frees occupied resources and memories.  
func (e *TaskExecutor) close() {  
    if closing := e.closed.CAS(false, true); !closing {  
        return  
    }  
    e.cancel()  
    e.Input.Close()  
    e.function = nil  
    e.Input = nil  
}
```

[수십GB짜리 00M을 막은 코드 두줄]

메모리, 줄여야한다 – 특이법

요약

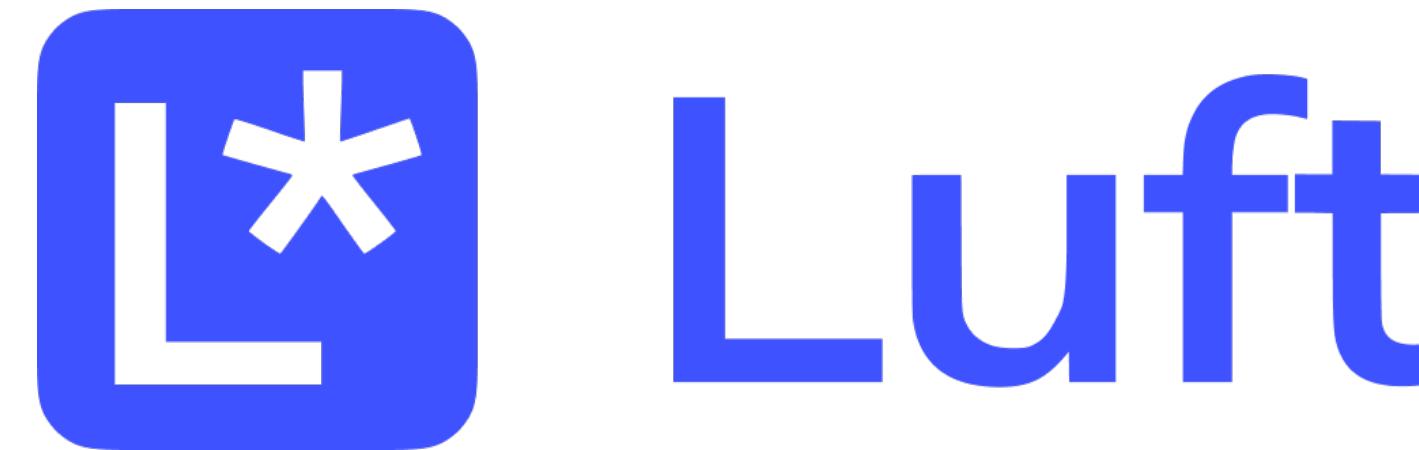
Summary

데이터베이스를 Go로 개발하면서 OOM이 남. 18GB → 2.7GB로 최적화

- **코드:** Raw I/O가 많은 만큼 메모리 재사용에 공을 들임
- **대응:** pprof으로 문제의 모듈을 찾음 → gobench를 통해 유닛별 벤치마크
- **예방:** leaktest를 통한 고루틴 누수 방지
- **그래도 안되면:** 흑마법

성능 저하는 많은 문제들의 집합이기 때문에, 시스템을 잡아 나가는게 중요하다.

One more thing



10초에 10억 데이터를 쿼리하는 데이터베이스를
함께 만들어나갈 분을 찾습니다!

dev-recruit.ab180.co/database-engineer

Q & A

김효준

hyojun@ab180.co

감사합니다.

김효준

hyojun@ab180.co