Profiling and Tracing Tips in Go

OLAP 데이터베이스를 개발하며 얻은 교훈들

박재완 abit.ly/hueypark

발표자 소개

박재완

- AB180 의 Backend Engineer
  - Query Engine Team 에서 Luft 개발
- 이전에는 MMORPG server 등 게임 개발
- Gopher
  - Open source: 2018 ~
  - Fulltime: 2022 ~

아젠다

- 왜 해야했나?
- 최적화 대상 쿼리 선정
- pprof
- 개별 사례를 통한 최적화 팁

왜 해야했나?

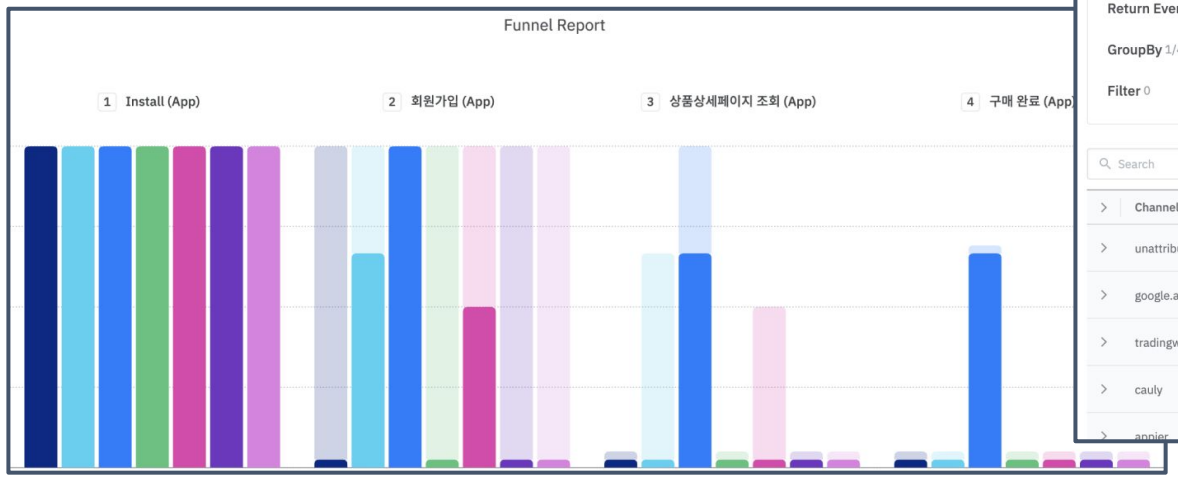
Luft: Airbridge 자체개발 OLAP 데이터베이스

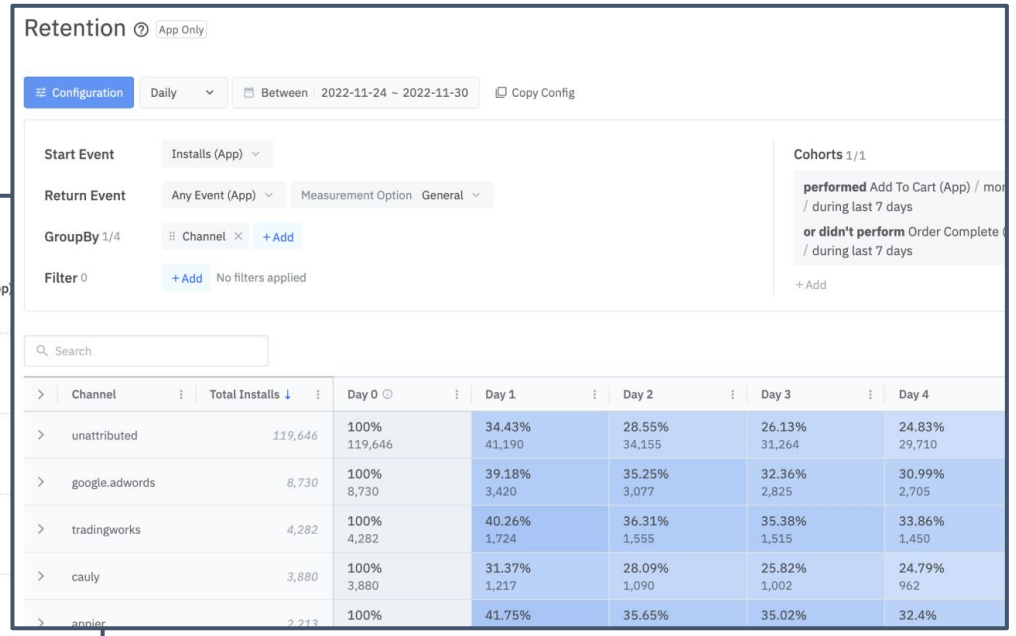
- 유저행동 분석에 최적화 되어있음
- Airbridge 워크로드에 대응하기 위해 자체개발 되었음
  - 하루 20억 건 이상의 유저 이벤트 처리
- 모든 계층 (query, processing, storage) 이 Go 로 작성되어 있음

더 자세한 내용은: abit.ly/ab180-luft

Luft 로 제공되는 데이터

- 유저행동의 시계열 분석
- **Retention, Funnel** 등의 데이터 쿼리





Luft 사용자는?

특별한 관심사를 해결하기 위한 OLAP 워크로드

- 긴 시간에 대해
  - 1개월 에서 1년까지
- 복잡한 집계 쿼리를 함
  - **Sum, Average, Median, Unique user count, Event** 순서 고려, ...
- 반응속도에 민감하지 않음
  - 밀리초 단위로 결과가 나올 것이라 기대하지 않음

Luft 사용자는?

- 예: 한 달에 3회 이상 구매한 유저의 6개월 **retention, funnel, trend**

Luft 워크로드 진화

더 많은 데이터! 더 다양하게!

- 처리 데이터 기간 증가
  - **180일 -> 1년 (2023년 2Q)**
- 처리 이벤트 수 증가 (1개월 기준)
  - 약 **36억 (2022년 3Q)**
  - 약 **50억 (2023년 1Q)**
  - 약 **67억 (2023년 3Q)**
- 프로세싱 복잡도(쿼리 다양성) 증가
  - 퍼널, 쿼리 타임 어트리뷰션, **... (~ 2023년 4Q)**
- 이런 진화는 앞으로도 계속

성능 최적화 대상 쿼리

성능 최적화 대상 쿼리

어디에서 시작해야 하나?

- ~~1분 이상의 매우 느린 쿼리~~: 납득할만큼 터무니없이 복잡한 쿼리임
- ~~1초 이하의 매우 빠른 쿼리~~: 반응속도에 민감하지 않음(OLAP)
- 적당히 빠른 쿼리(?)

적당히 빠른 쿼리는 무엇인가?

~~정답~~ 보다는 모법답안!

- P90 에서 P95 사이의 쿼리
  - 체감할 수 있는 사용자 경험 개선
  - 실행가능한 액션 아이템 도출 용이
- 예:
  - 3개월 데이터에서 퍼널을 조회하는 9초 걸리는 쿼리
  - 1개월 데이터에서 리텐션을 조회하는 13초 걸리는 쿼리

또 다른 최적화 대상 쿼리

개발환경 개선

- 사람이 직접 대응해야 하는 쿼리 😭
  - 사용량이 적은 시간에 배치 처리되는 대량의 장시간 쿼리
  - 일정 횟수 이상 실패하면 시스템이 자동으로 재시도하지 못함
  - 온콜 대상자가 수동으로 복구해야 했음

pprof

프로파일링과 pprof

- 프로파일링
  - 프로그램의 실행시간, 메모리 사용량, 함수 호출 빈도 등을 측정
  - 프로그램 최적화를 보조하기 위해 많이 사용됨
- pprof
  - 프로파일링 데이터를 분석하고 시각화하기 위한 툴
  - Go 는 이 pprof 가 언어 수준에서 잘 통합되어 있음
  - 그렇기 때문에 적은 노력으로 다양한 성능 정보 확인할 수 있음

net/http/pprof package

go standard library

- http 서버를 사용해 pprof 런타임 프로파일링 데이터를 제공
- 오늘은 아래 세 가지가 중점
  - profile: CPU profile
  - heap: 메모리 profile
  - trace
    - 시계열 프로그램 실행정보
    - goroutine 상태, 함수 실행정보, heap 사용량, block profile, ...

net/http/pprof package

profile 준비 완료!

```
package main

import (
    "fmt"
    _ "net/http/pprof"
)

func main() {
    http.ListenAndServe("localhost:6060", nil)
}
```

/debug/pprof/

Set debug=1 as a query parameter to export in legacy text format

Types of profiles available:

Count	Profile
3	allocs
0	block
0	cmdline
5	goroutine
3	heap
0	mutex
0	profile
5	threadcreate
0	trace

full goroutine stack dump

Profile Descriptions:

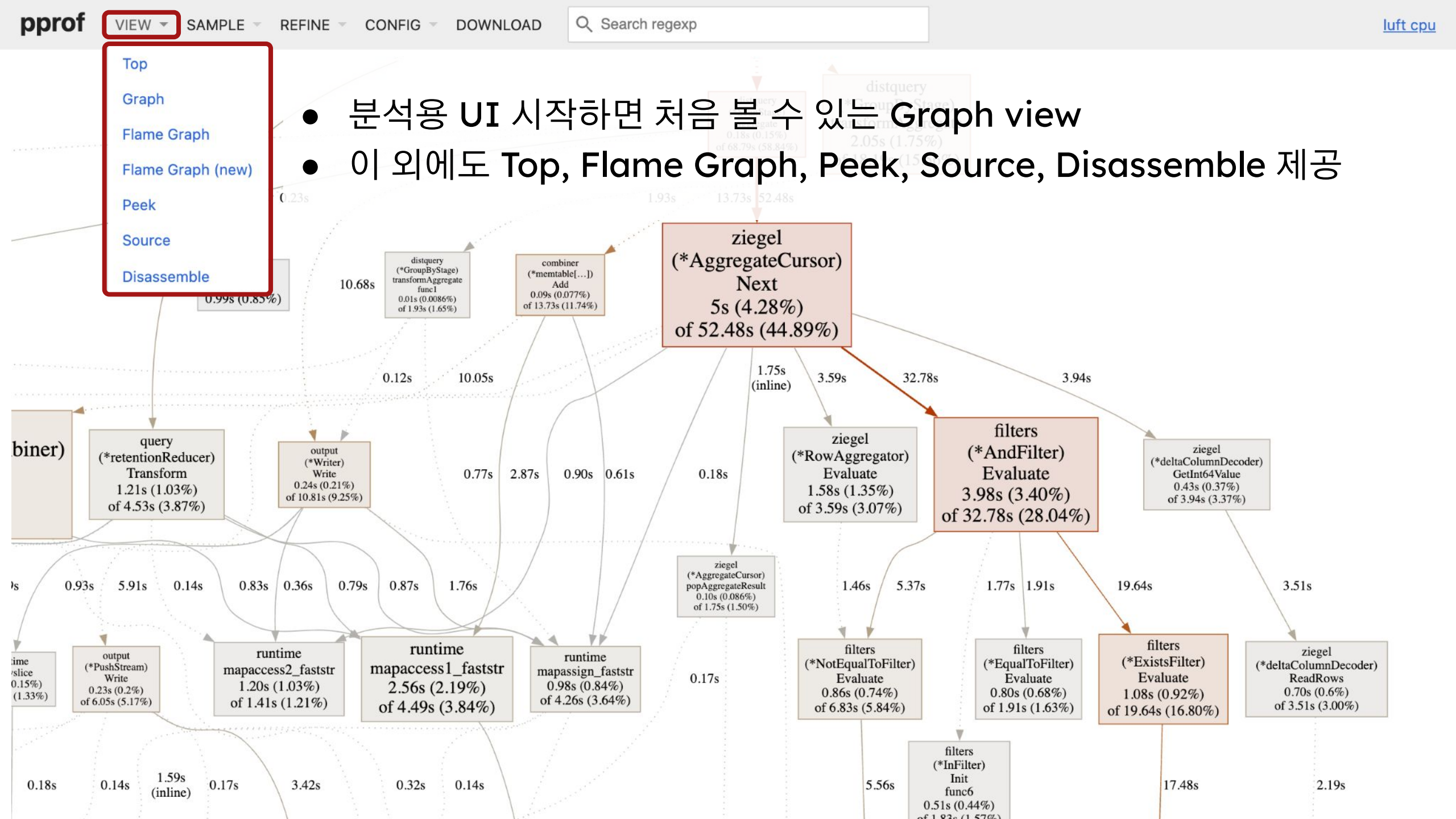
- allocs: A sampling of all past memory allocations
- block: Stack traces that led to blocking on synchronization primitives
- cmdline: The command line invocation of the current program
- goroutine: Stack traces of all current goroutines. Use debug=2 as a query parameter to export in the same format as an unrecovered panic.
- heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.
- mutex: Stack traces of holders of contended mutexes
- profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
- threadcreate: Stack traces that led to the creation of new OS threads
- trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.

- http://localhost:6060/debug/pprof/
- 사용가능한 기능과 간단한 설명을 보여줌

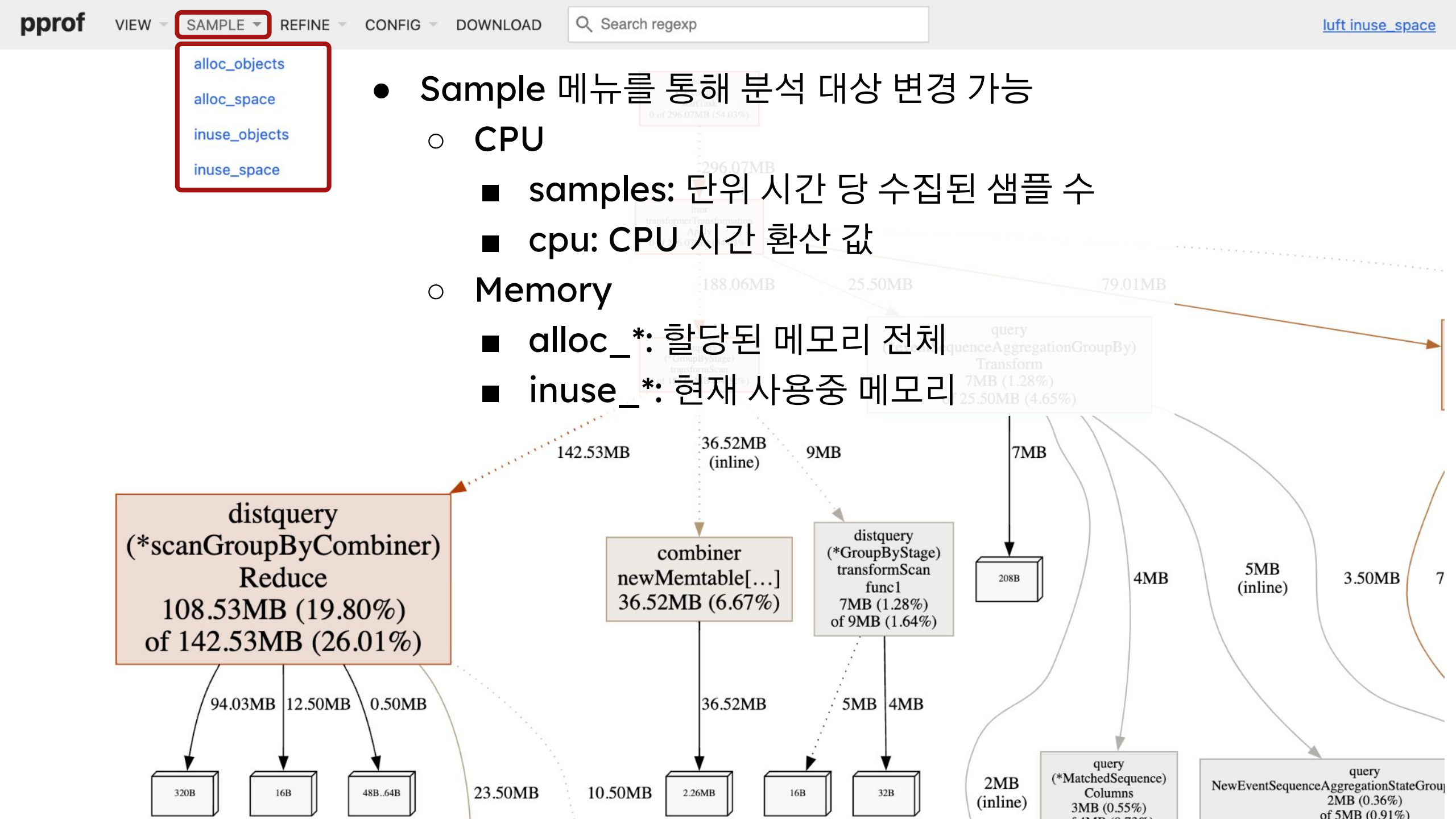
go tool pprof

- go tool pprof 명령어로 데이터 분석
  - `go tool pprof http://localhost:6060/debug/pprof/profile`

- http flag 를 추가하면 분석용 UI 가 실행됨
  - `go tool pprof -http=localhost:8080 http://localhost:6060/debug/pprof/profile`



- 분석용 UI 시작하면 처음 볼 수 있는 Graph view
- 이 외에도 Top, Flame Graph, Peek, Source, Disassemble 제공



- Sample 메뉴를 통해 분석 대상 변경 가능
  - CPU
    - samples: 단위 시간 당 수집된 샘플 수
    - cpu: CPU 시간 환산 값
  - Memory
    - alloc_*: 할당된 메모리 전체
    - inuse_*: 현재 사용중 메모리

go tool trace

- trace 의 경우 먼저 다운로드 받고, go tool trace 로 분석
  - `curl -o trace.log http://localhost:7400/debug/pprof/trace?seconds=10`
  - `go tool trace trace.log`

cmd/trace: the Go trace event viewer

This web server provides various visualizations of an event log gathered during the execution of a Go program that uses the runtime/trace package.

Trace 툴 메인 페이지

Event timelines for running goroutines

Large traces are split into multiple sections of equal data size (not duration) to avoid overwhelming the visualizer.

- View trace (0s-4.855400349s)
- View trace (4.855401661s-6.566572009s)
- View trace (6.566573001s-20.000942992s)

- **View trace:** 시각화된 시계열 데이터 확인

This view displays a timeline for each of the GOMAXPROCS logical processors, showing which goroutine (if any) was running on that logical processor at each moment. Each goroutine has an identifying number (e.g. G123), main function, and color. A colored bar represents an uninterrupted span of execution. Execution of a goroutine may migrate from one logical processor to another, causing a single colored bar to be horizontally continuous but vertically displaced.

Clicking on a span reveals information about it, such as its duration, its causal predecessors and successors, and the stack trace at the final moment when it yielded the logical processor, for example because it made a system call or tried to acquire a mutex. Directly underneath each bar, a smaller bar or more commonly a fine vertical line indicates an event occurring during its execution. Some of these are related to garbage collection; most indicate that a goroutine yielded its logical processor but then immediately resumed execution on the same logical processor. Clicking on the event displays the stack trace at the moment it occurred.

The causal relationships between spans of goroutine execution can be displayed by clicking the Flow Events button at the top.

At the top ("STATS"), there are three additional timelines that display statistical information. "Goroutines" is a time series of the count of existing goroutines; clicking on it displays their breakdown by state at that moment: running, runnable, or waiting. "Heap" is a time series of the amount of heap memory allocated (in orange) and (in green) the allocation limit at which the next GC cycle will begin. "Threads" shows the number

Above the event trace for the first logical processor are traces for various runtime-internal events. The "GC" bar shows when the garbage collector is running, and in which stage. Garbage collection may temporarily affect all the logical processors and the other metrics. The "Network", "Timers", and "Syscalls" traces indicate events in the runtime that cause goroutines to wake up.

The visualization allows you to navigate events at scales ranging from several seconds to a handful of nanoseconds. Consult the documentation for the Chromium Trace Event Profiling Tool for help navigating the view.

- Goroutine analysis

This view displays information about each set of goroutines that shares the same main function. Clicking on a main function shows links to the four types of blocking profile (see below) applied to that subset of goroutines. It also shows a table of specific goroutine instances, with various execution statistics and a link to the event timeline for each one. The timeline displays only the selected goroutine and any other it interacts with via block/unblock events. (The timeline is goroutine-oriented rather than logical processor-oriented.)

Profiles

Each link below displays a global profile in zoomable graph form as produced by pprof's "web" command. In addition there is a link to download the profile for offline analysis with pprof. All four profiles represent causes of delay that prevent a goroutine from running on a logical processor: because it was waiting for the network, for a synchronization operation on a mutex or channel, for a system call, or for a logical processor to become available.

- Network blocking profile (⬇️)
- Synchronization blocking profile (⬇️)
- Syscall blocking profile (⬇️)
- Scheduler latency profile (⬇️)

User-defined tasks and regions

- Goroutine 분석
- 다양한 profile 정보 제공

개별 사례를 통한 최적화 팁

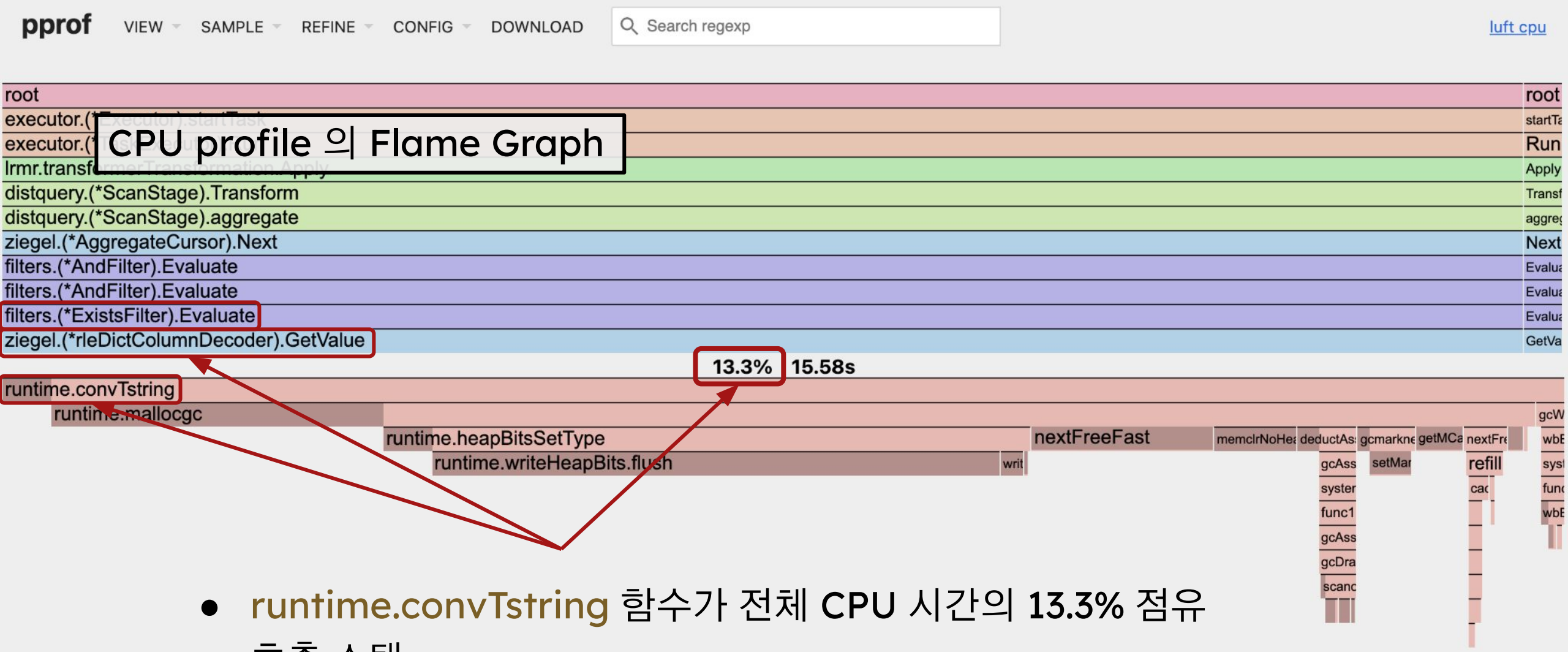
개별 사례를 통한 최적화 팁

- Exists filter 개선
- 메모리 사용량 개선
- gRPC stream 줄이기
- Global lock 제거

Exists filter 개선

개선점 식별

1. CPU profile 의 Flame Graph 에서 트렌드 확인
2. Source view 에서 개선대상 특정



- runtime.convTstring 함수가 전체 CPU 시간의 13.3% 점유
- 호출 스택
  - ExistsFilter.Evalute
  - rleDictColumnDecoder.GetValue

github.com/ab180/cohort-engine/pkg/ziegel.(*rleDictColumnDecoder).GetValue
/app/pkg/ziegel/rle_dict_column.go

```
Total:       440ms     17.48s (flat, cum) 14.95%
   108            .          .
   109            .          .           CPU profile 의 source view
   110            .          .
   111            .          .           }
   112            .          .
   113       250ms      250ms           func (d *rleDictColumnDecoder) GetValue(eid EventID) (interface{}, Repeat, error) {
   114       190ms     17.23s                return d.GetStringValue(eid)
   115            .          .           }
   116            .          .
   117            .          .           func (d *rleDictColumnDecoder) GetStringValue(eid EventID) (string, Repeat, error) {
   118            .          .                v := d.ReadRows(eid)
   119            .          .
```

- GetStringValue 함수 반환: string
- GetValue 함수 반환: interface{}
- 암시적 string to interface{} 변환 발생

runtime.convTstring
/usr/local/go/src/runtime/iface.go

```
Total:       490ms     15.58s (flat, cum) 13.34%
   383            .          .                *(*uint64)(x) = val
   384            .          .            }
   385            .          .            return
   386            .          .        }
   387            .          .
   388       120ms      120ms        func convTstring(val string) (x unsafe.Pointer) {
   389        40ms       40ms            if val == "" {
   390            .          .                x = unsafe.Pointer(&zeroVal[0])
   391            .          .            } else {
   392       170ms     14.97s                x = mallocgc(unsafe.Sizeof(val), stringType, true)
   393       110ms      400ms                *(*string)(x) = val
   394            .          .            }
   395        50ms       50ms            return
   396            .          .        }
   397            .          .
   398            .          .        func convTslice(val []byte) (x unsafe.Pointer) {
   399            .          .            // Note: this must work for any element type, not just byte.
   400            .          .            if (*slice)(unsafe.Pointer(&val)).array == nil {
```

github.com/ab180/cohort-engine/pkg/ziegel.(*rleDictColumnDecoder).GetValue
/app/pkg/ziegel/rle_dict_column.go

```
Total:       440ms     17.48s (flat, cum) 14.95%
   108            .          .
   109            .          .          mnDecoder) Type() Type {
   110            .          .          a.Type
   111            .          .          }
   112            .          .
   113       250ms      250ms          func (d *rleDictColumnDecoder) GetValue(eid EventID) (interface{}, Repeat, error) {
   114       190ms     17.23s                return d.GetStringValue(eid)
   115            .          .          }
   116            .          .
   117            .          .          func (d *rleDictColumnDecoder) GetStringValue(eid EventID) (string, Repeat, error) {
   118            .          .
   119            .          .
```

CPU profile 의 source view

- runtime.convTstring 를 통해 암시적 형변환
- 불필요한 메모리 할당
- 이 과정에서 많은 **CPU** 시간 사용

runtime.convTstring
/usr/local/go/src/runtime/iface.go

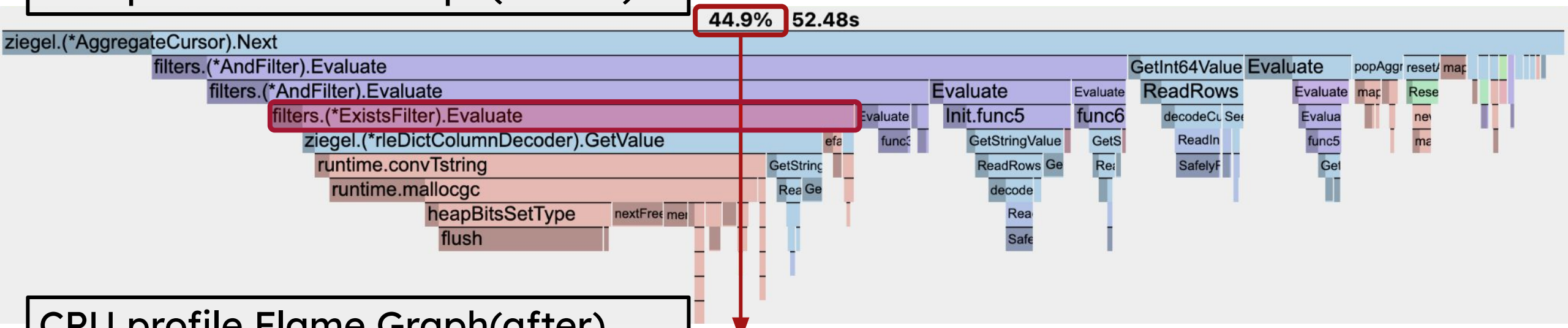
```
Total:       490ms     15.58s (flat, cum) 13.33%
   383            .          .                      *(*uint64)(x) = val
   384            .          .                }
   385            .          .                return
   386            .          .          }
   387            .          .
   388       120ms      120ms          func convTstring(val string) (x unsafe.Pointer) {
   389        40ms       40ms                if val == "" {
   390            .          .                      x = unsafe.Pointer(&zeroVal[0])
   391            .          .                } else {
   392       170ms     14.97s                       x = mallocgc(unsafe.Sizeof(val), stringType, true)
   393       110ms      400ms                       *(*string)(x) = val
   394            .          .                }
   395        50ms       50ms                return
   396            .          .          }
   397            .          .
   398            .          .          func convTslice(val []byte) (x unsafe.Pointer) {
   399            .          .                // Note: this must work for any element type, not just byte.
   400            .          .                if (*slice)(unsafe.Pointer(&val)).array == nil {
```

Exists filter 개선

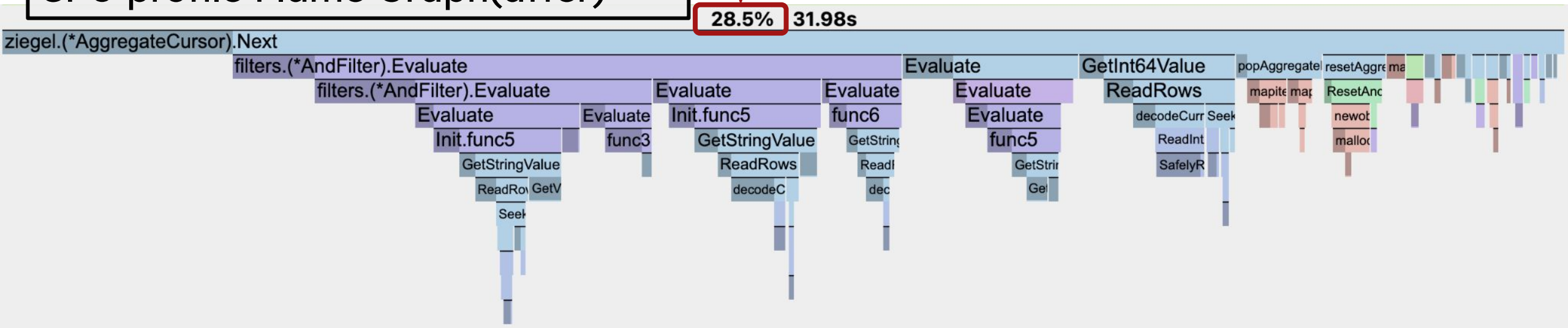
개선

- 쿼리 계획 시점에 **filter** 최적화
- 가능한 경우 **Exists filter** 를 **NotEqualTo filter** 로 변경
  - **NotEqualTo filter** 는 타입 최적화가 되어 있음
  - 그 결과로 **string to interface{}** 변환이 발생하지 않음

CPU profile Flame Graph(before)



CPU profile Flame Graph(after)

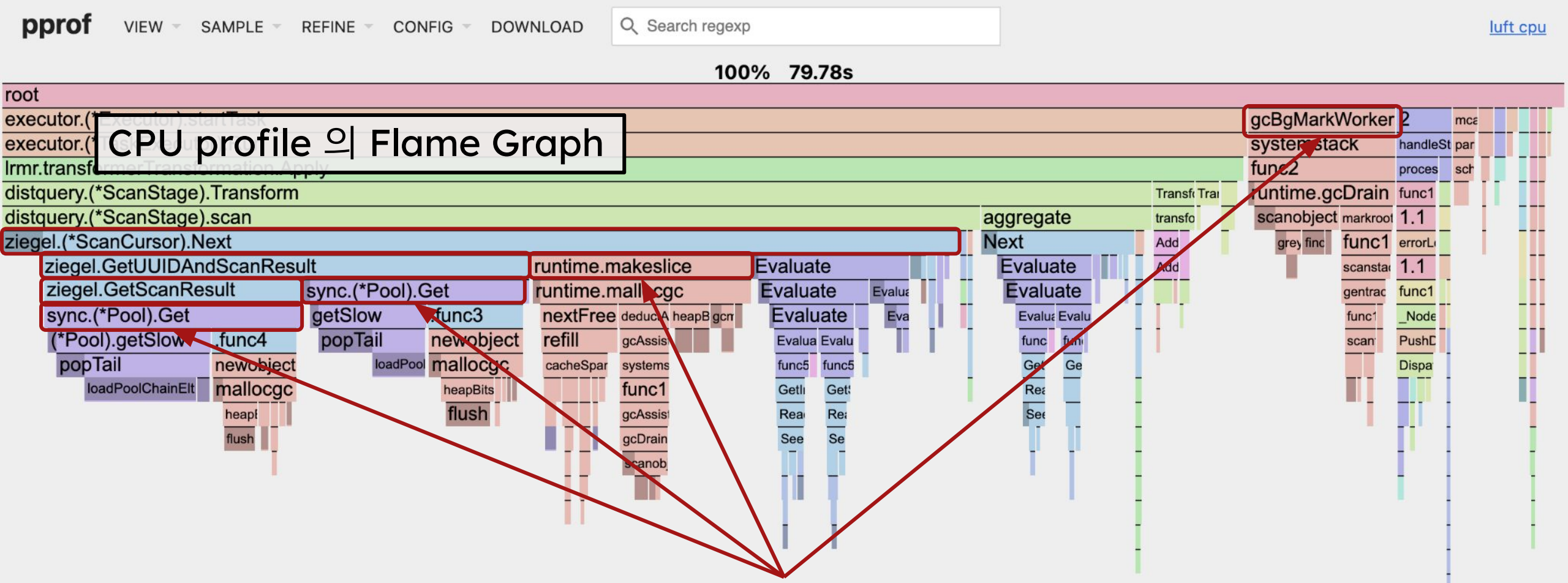


- 결과: 쿼리 응답 속도 **22%** 개선**(13.5s -> 10.5s)**

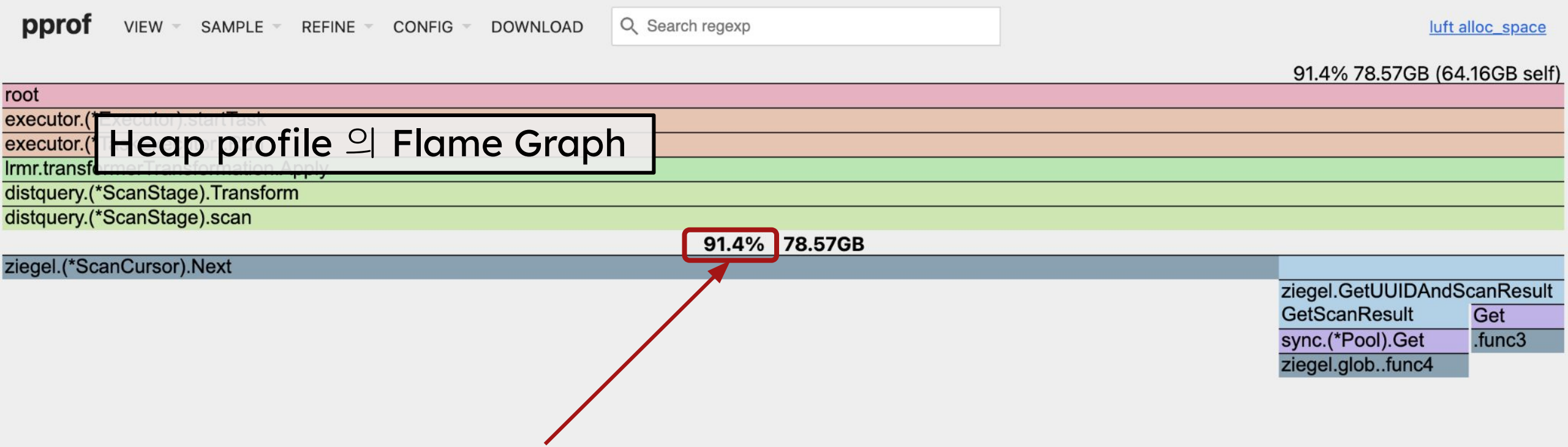
메모리 사용량 개선

개선점 식별

1. **CPU** 와 메모리 **profile** 의 **Flame Graph** 에서 트렌드 확인
2. 메모리 **profile** 의 **source view** 에서 개선대상 특정



- **CPU profile** 에서 일부 함수가 납득하기 어려운 지분 점유
  - GetUUIDAndScanResult, sync.(*Pool).Get, runtime.makeslice 등
  - ziegel.(*ScanCursor).Next 에서 호출
  - 메모리 할당과 반납에 관한 함수
- 메모리가 pool 에 잘 반납되지 않는 것으로 추정



Heap profile 의 Flame Graph

- 메모리 **profile** 에서도 같은 결과를 확인 할 수 있었음
  - 메모리의 **91.4%** 가 **CPU** 프로파일의 병목인 ziegel.(*ScanCursor).Next 에서 할당

ziegel 에 대한 더 자세한 내용은 abit.ly/ziegel

pprof VIEW SAMPLE REFINE CONFIG DOWNLOAD Search regexp luft alloc_space

github.com/ab180/cohort-engine/pkg/ziegel.(*ScanCursor).Next

/app/pkg/ziegel/scan_cursor.go

Heap profile 의 source view

```
Total:
    86          .          .                if c.opt.endTrailID <= c.currentEID.TrailID {
    87          .          .                        return false
    88          .          .                }
    89          .          .                numEvents := c.base.numEventsColumn.GetNumEvents(c.currentEID.TrailID)
    90          .          .
    91          .   14.41GB                c.current = GetUUIDAndScanResult()
    92          .          .                uuid, mapped := c.base.uuidColumn.GetUUID(c.currentEID.TrailID)
    93          .          .                if !c.opt.uuidRange.Contains(uuid.RangeKey()) {
    94          .          .                        continue
    95          .          .                }
    96          .          .                // TODO: We can use this after
    97          .          .                //if uuid.RangeKey().Compare(
    98          .          .                //      continue
    99          .          .                //}
   100          .          .                //if 0 < uuid.RangeKey().Compare(c.opt.uuidRange.End) {
   101          .          .                //      // We can end here because
   102          .          .                //      return false
   103          .          .                //}
   104          .          .                c.current.UUID = mapped
   105          .          .
   106          .          .                if cap(c.current.ScanResult.Columnss) < len(c.columnsToRead) {
   107    64.10GB    64.10GB                        c.current.ScanResult.Columnss = make([]TypeSlice, len(c.columnsToRead))
   108          .          .                } else {
   109          .          .                        c.current.ScanResult.Columnss = c.current.ScanResult.Columnss[:len(c.columnsToRead)]
   110          .          .                }
   111          .          .                for i, col := range c.columnsToRead {
   112          .          .                        if col == nil {
```

- 메모리의 대부분이 특정 부분에서 할당
  - Pool 에서 오브젝트 획득: **14.41GB**
  - slice 확장: **64.10GB**

github.com/ab180/cohort-engine/pkg/ziegel.(*ScanCursor).Next

/app/pkg/ziegel/scan_cursor.go

메모리 사용량 개선

ziegel.(*ScanCursor).Next 함수 개선

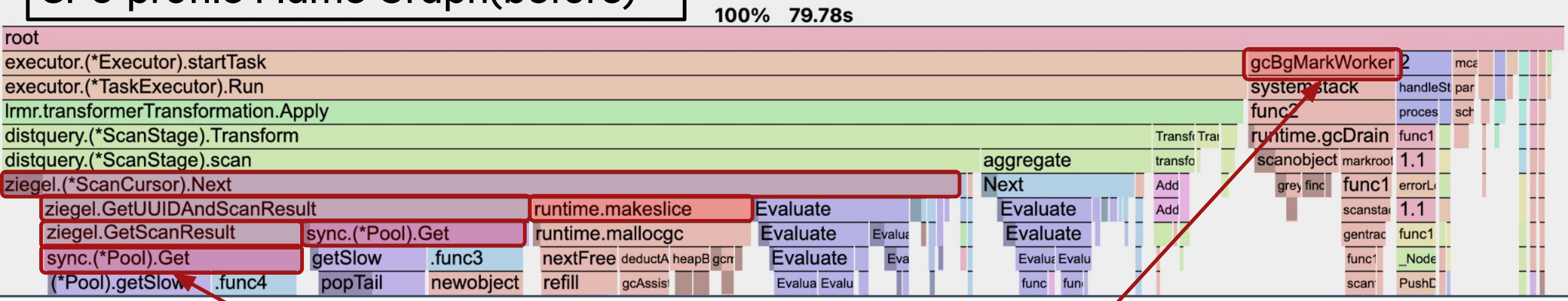
- 풀에서 메모리를 중복으로 할당받는 버그 수정
- Next 결과를 다른 goroutine 에서 사용하기 위해 복사하는 부분 제거
  - Cursor 구현체 내부에서 재사용하던 buffer 제거
  - 메모리 반환을 사용자가 책임지는 방향으로 디자인 개선
  - Luft 의 경우 다른 goroutine 이나 물리장비로 데이터를 보내는 경우가 대부분이기 때문에 이 변경이 유의미하게 동작했음

메모리 사용량 개선

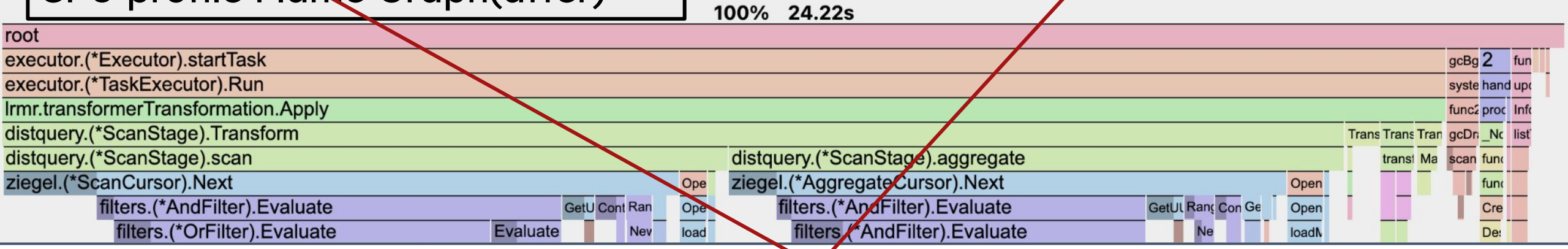
ziegel.(*ScanCursor).Next 함수 개선

```
1   for cur.Next() {
2 -     temp := cur.Current()
3 +     data := cur.Current()          // 이제 내부에서 버퍼를 사용하지 않음
4
5 -     data := getDataFromPool()      // 메모리 중복 할당 제거
6 -     copy(data, temp)               // 불필요한 메모리 복사 제거
7
8       sendToNext(data)
9   }
```

CPU profile Flame Graph(before)



CPU profile Flame Graph(after)



- 메모리 할당관련 부분이 모두 사라짐
- 결과: 쿼리 응답 속도 **51%** 개선**(8.5s -> 4.2s)**

gRPC stream 줄이기

개선점 식별

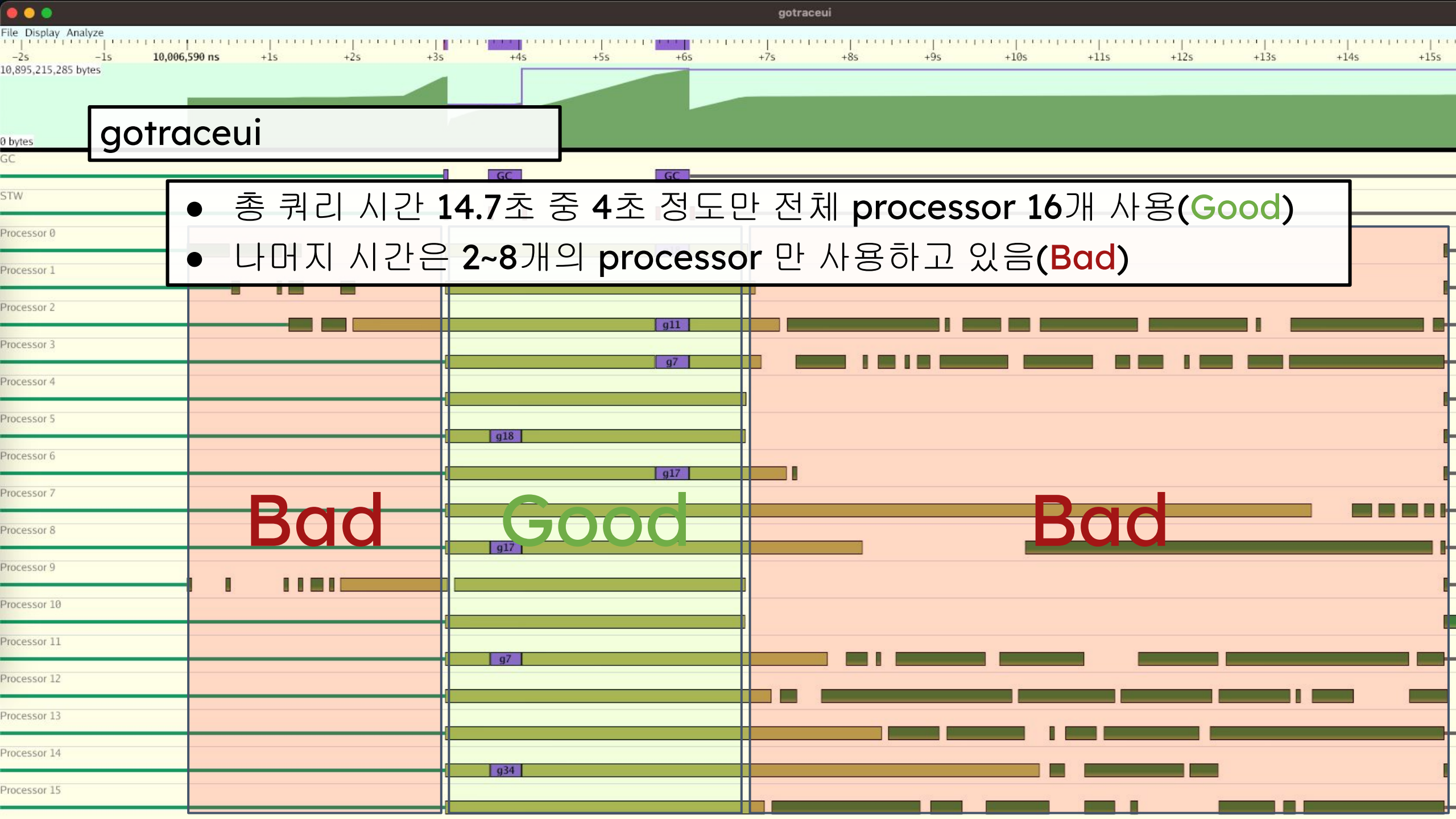
1. CPU, 메모리 프로파일을 시도했으나 개선대상 특정할 수 없었음
   a. 특별히 오래 걸리는 함수 없음
   b. 과도하게 할당되는 메모리 없음
2. Trace 분석을 통해 트렌드 확인
3. Synchronization blocking profile 의 Flame Graph 에서 개선대상 특정

gRPC stream 줄이기

잠시 감사하는 시간

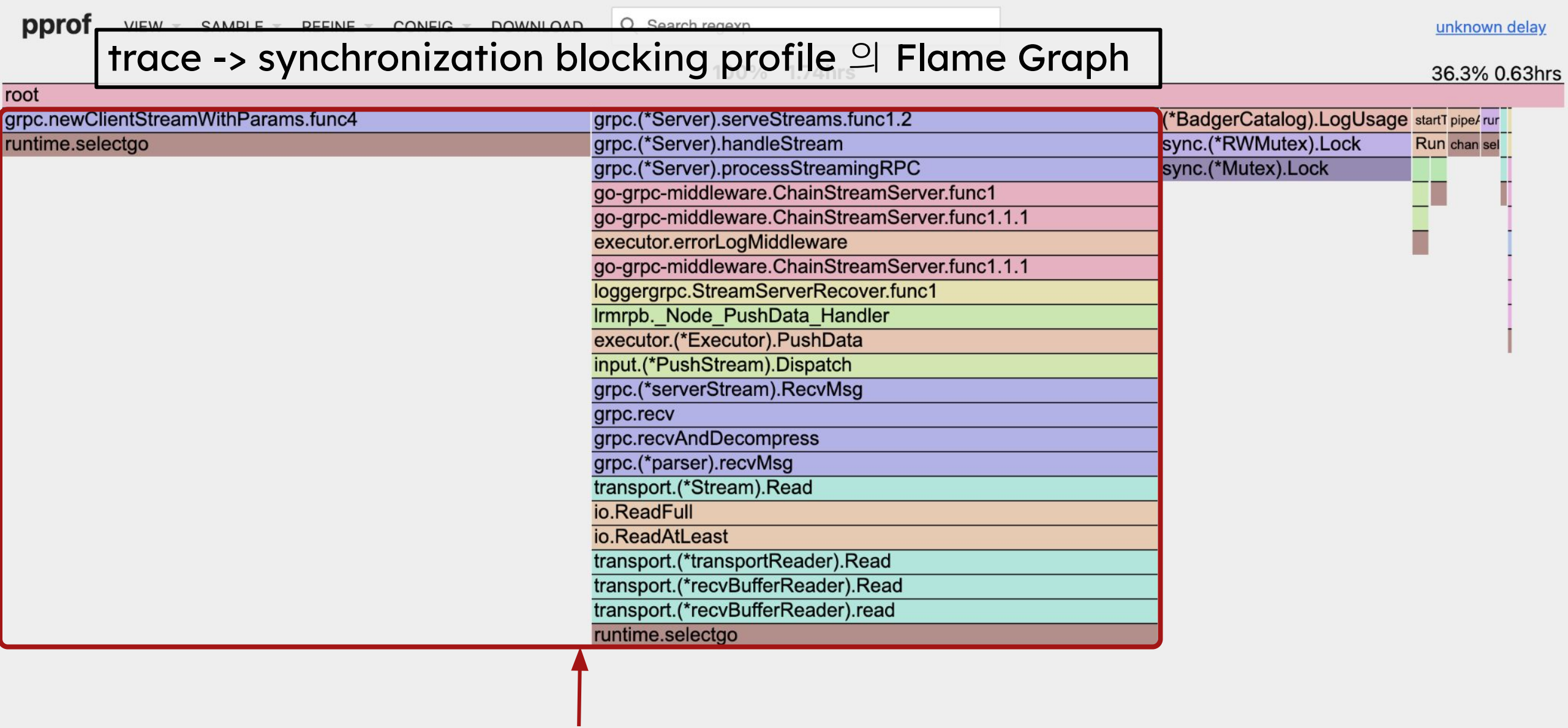
gotraceui

- 오픈소스 go trace frontend
- go tool trace 에 비해
  - 압도적으로 빠름
  - 한 번에 분석할 수 있는 기간이 김
- https://github.com/dominikh/gotraceui



gotraceui

- 총 쿼리 시간 **14.7**초 중 **4**초 정도만 전체 **processor 16**개 사용(Good)
- 나머지 시간은 **2~8**개의 **processor** 만 사용하고 있음(Bad)



- gRPC stream 관련 함수에서 과도한 blocking 발생

gRPC stream 줄이기

개선

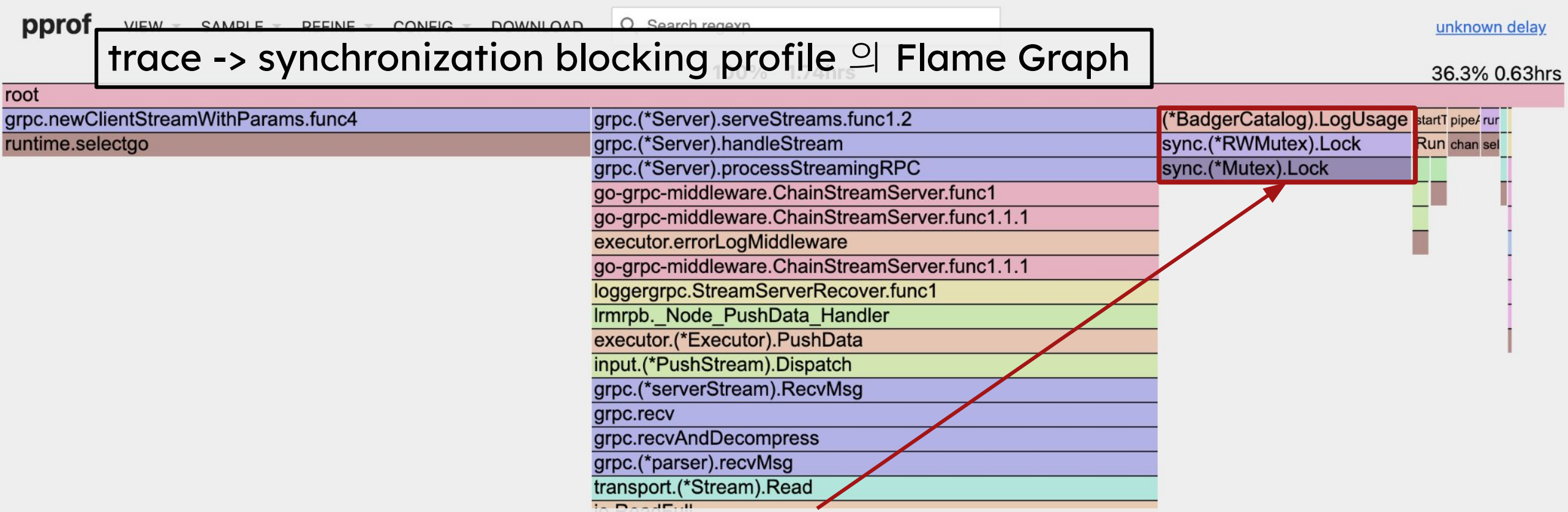
- **gRPC stream** 을 과도하게 생성하지 않게 수정
  - 논리적으로 필요한 만큼 **->** 적절히 생성 후 공유
- 결과: 쿼리 응답 속도 **56%** 개선**(14.7s -> 6.4s)**

global lock 제거

gRPC stream 줄이기의 역습

1. **gRPC stream** 을 적게 만듬
2. 여러 쿼리가 동시에 실행될 가능성이 높아짐
3. 잠재된 위험이었던 메타데이터 접근 시 **global lock** 경합이 심화됨

trace -> synchronization blocking profile 의 Flame Graph



- **production** 에서 성능문제를 겪은 후 **global lock** 을 제거함
- 지금 돌아보니 **profile** 은 힌트를 주고 있었음
- 결과(P90)
  - 작업 전: **13.2s**
  - **gRPC stream** 줄이기: **19.9s**
  - **global lock** 제거: **7.6s**

정리

성급히 유용한 아이디어를 만들어 보면

- **string to interface{}** 형변환을 피하자
- 메모리를 할당하지 말고 재사용하자
- **gRPC** 연결을 적게 만들자
- **mutex** 를 이용한 **lock** 을 사용하지 말자

저희는 실제 적용한 작업이고 유용하게 동작했지만 일반화하기엔 무리

정리

하지만! 프로파일과 함께 한다면 필요한 특정 부분에 올바르게 적용할 수 있음

- **Exists filter** 개선: **22% (13.5s -> 10.5s)**
- 메모리 사용량 개선: **51% (8.5s -> 4.2s)**
- **gRPC stream** 줄이기: **56% (14.7s -> 6.4s)**
- **Global lock** 제거**(P90): 62% (19.9s -> 7.6s)**

Thank you

박재완 abit.ly/hueypark