AB180

Cgo to Go:

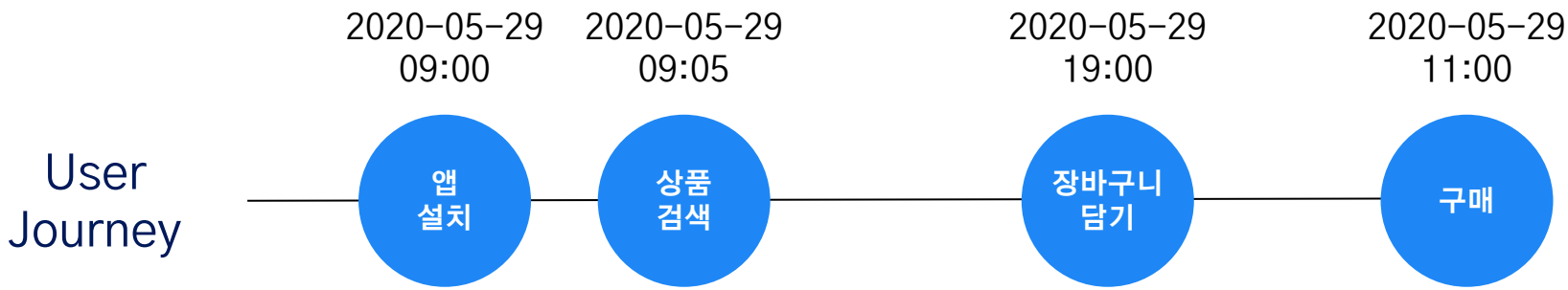
Go로 columnar storage engine 개발하기

Query Engine Team - Geon Kim
@DevFest 2022 Golang Korea (2022-12-15)

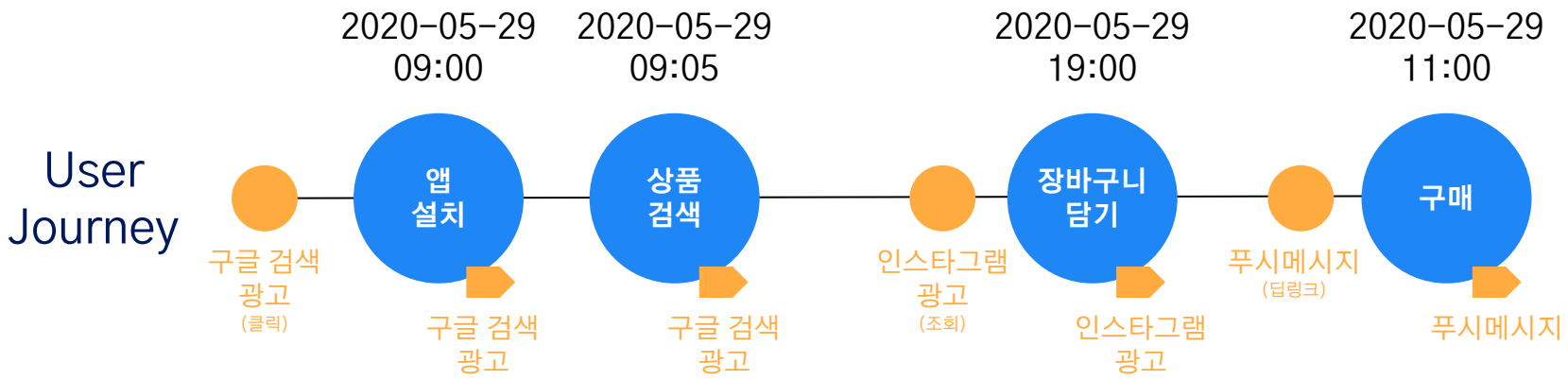
Airbridge:

People-Based Attribution and Incrementality Measurement for Web and Mobile.

Attribution Tool



Attribution Tool

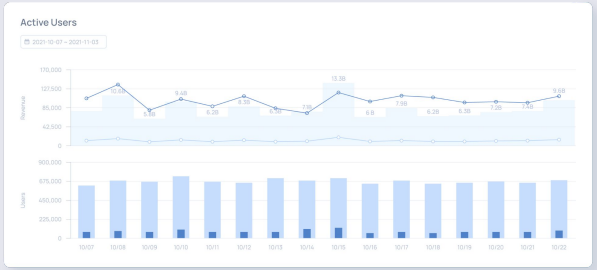


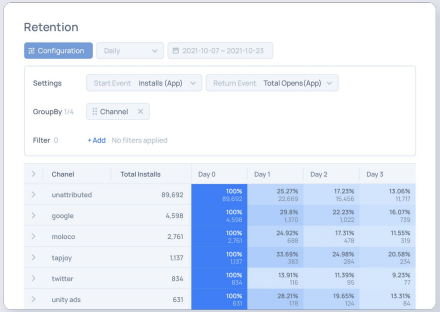
귀인 (심리학)

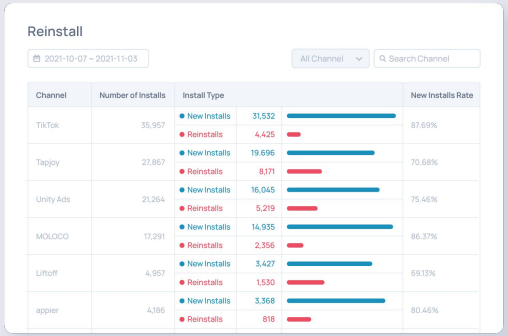
위키백과, 우리 모두의 백과사전.

귀인(歸因, attribution)은 '원인의 귀착'의 줄임말로서, 한 개인이 타인의 행동이나 사건의 원인을 어떻게 설명하느냐와 관련이 있는 말이다. 예를 들어 컵을 실수로 떨어뜨려 깨뜨렸을 때, 옆에 있는 사람과 부딪혔기 때문에 떨어뜨렸다고 생각할 수 있고, 자신이 너무 덜렁대서 깨뜨렸다고 생각할 수도 있다. 이처럼 하나의 결과를 갖고도 원인으로 생각하는 것은 개인에 따라 다를 수 있으며, 다양한 귀인이 나타난다.

Performance Reports







... and more

Luft:

Airbridge’s in-house OLAP Database

Luft

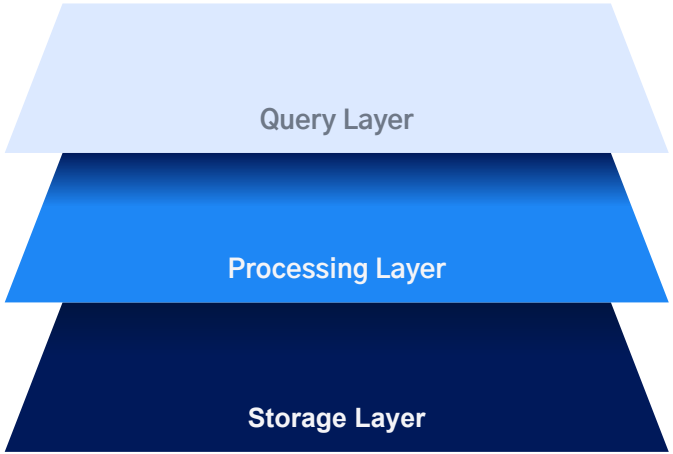
"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- **Fast:** 5초 내로 수억개의 이벤트를 분석 가능함.
- **Highly Available:** 모든 데이터는 S3에 백업되며, 여러 노드에 Replication되어 저장됨.
- **Scalable:** 복잡한 설정 없이 몇번의 클릭만으로 노드의 수를 늘리고, 줄일 수 있음.
- **Easy to Operate:** 외부 의존성 없이 단일 바이너리로 배포 가능하며, 클러스터 관리를 위한 다양한 REST API를 제공함.
- **Easy to Use:** SQL로 분석하기 어려운 다양한 복잡한 쿼리를 사용하기 쉽게 추상화한 API를 제공함.

Full version: abit.ly/ab180-luft

Luft

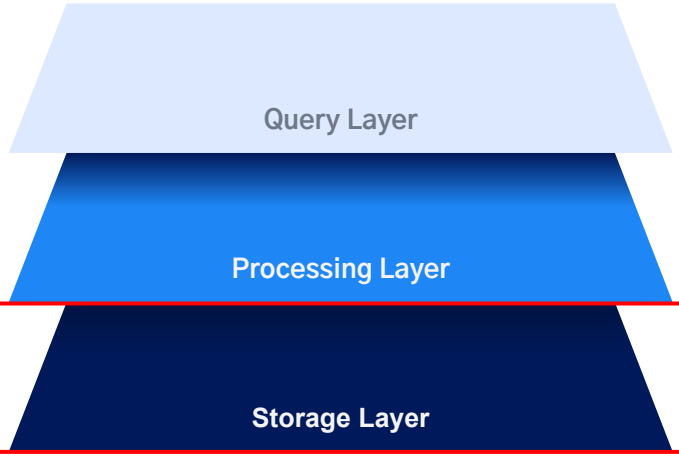
3 Layers



- **Query Layer:**
  쿼리 API 제공, 쿼리 플래닝, 쿼리 실행 등을 함.
- **Processing Layer: (feat. ab180/lrmr)**
  쿼리를 분산 처리함.
- **Storage Layer: (feat. traildb/traildb)**
  디스크에 데이터를 인덱싱, 디스크로부터 데이터를 스캔함.

Luft

3 Layers



- **Query Layer:**
  쿼리 API 제공, 쿼리 플래닝, 쿼리 실행 등을 함.
- **Processing Layer: (feat. ab180/lrmr)**
  쿼리를 분산 처리함.
- **Storage Layer: (feat. traildb/traildb)**
  디스크에 데이터를 인덱싱, 디스크로부터 데이터를 스캔함.

Why TrailDB?

Why TrailDB?

“TrailDB is an efficient tool for storing and querying series of events. ”

- **Partitioned By User:** 이벤트를 유저 단위로 묶어 시간 순으로 정렬하여 저장하여 분산처리시 발생할 수 있는 JOIN에 대한 오버헤드를 줄임.
- **High Compression Ratio:** 유저 단위로 묶어 이벤트를 저장함으로 얻을 수 있는 data correlation 특성을 활용하여 더욱 효율적인 압축이 가능함.
- **Language Bindings:** Go언어 바인딩을 자체적으로 제공하여 바인딩 작업을 직접 할 필요 없이 바로 사용 가능함.

Full version: traildb.io

Why Not TrailDB?

Why Not TrailDB?

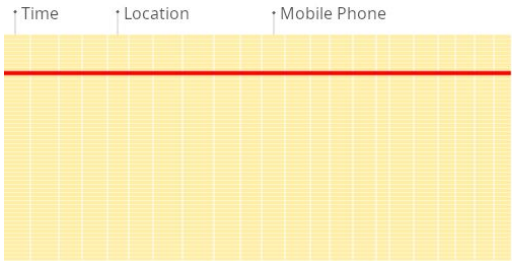
“Cgo is not Go. — Rob Pike”

- **Cgo Overhead:** C와 Go의 ABI 호환성 문제, 레이어간 값을 이동할 때 발생하는 불필요한 복사/할당 등의 이유로 성능이 좋지 않음.
- **Row-Oriented:** OLAP 워크로드의 대부분의 쿼리는 모든 컬럼을 읽을 필요가 없지만, Row-Oriented 포맷 특성상 항상 모든 컬럼의 데이터를 스캔해야함.
- **Poor Maintainability:** 프로젝트는 더이상 관리가 되지 않고 있으며, C는 AB180의 주력 언어가 아님.
- **Lack of Type Support:** 라이브러리 레벨에서 다양한 타입에 대한 지원이 없고, 인코딩도 dictionary run-length encoding만 지원함.
- **Lack of Multi-Core Support:** 멀티코어를 효율적으로 활용하지 못해 데이터 인덱싱에 너무 오랜 시간이 걸림.

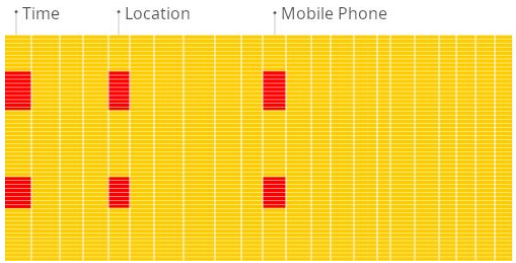
Why Ziegel?

Why Ziegel?

- **Homogeneous Implementation:** Storage Layer부터 Query Layer까지 모든 코드를 Go로 작성하면서 코드의 재사용성이 향상되고, Cgo로 인한 오버헤드도 없음.
- **Column-Oriented:** 데이터를 컬럼별로 저장하여 쿼리를 실행하는데 있어 필요 없는 컬럼의 데이터는 읽지 않음.
- **Maintainability:** Go..!
- **Type Support:** bool, int, float, string 등 다양한 타입뿐만 아니라 dictionary run-length encoding, delta encoding, raw encoding 등 컬럼의 특성에 맞춰 사용할 수 있는 다양한 인코딩 방식을 지원함.
- **Multi-Core Support:** Go..!



Row-Oriented



Column-Oriented

Images: https://clickhouse.com/docs/en/faq/general/columnar-database

Cgo to Go:

TrailDB to Ziegel

Migration Process

1. 마이그레이션 후 스펙 변경이 없었음을 보장할 수 있도록,다양한 케이스를 커버할 수 있는데 인덱싱, 쿼리에 대한 e2e 테스트를 추가 작성함.
2. Storage Layer에서 기존 TrailDB 구현체가 아닌 다른 구현체를 사용할 수 있도록 스토리지 엔진을 추상화함.
3. Ziegel로 추상화된 스토리지 엔진의 구현체를 개발. 이 단계에서는 TrailDB와 Ziegel이 동시에 함께 사용될 수 있도록 코드를 작성함.
4. 기능 동작을 위한 최소한의 스펙을 잡고, 기능 구현을 목표로 퍼포먼스는 크게 고려하지 않으며 코드를 작성함.
5. 프로덕션에서 쿼리시 사용하는 뷰가 Ziegel로 인덱싱된 테이블을 바라보도록 수정함. 언제나 롤백이 가능하도록 1~2주 동안은 TrailDB와 Ziegel로 동일한 데이터를 두 번 인덱싱함.
6. 레거시 청산 후 퍼포먼스 최적화를 진행함.
7. 새로운 기능을 개발함
8. …

Lessons Learned:

Do’s and Don’ts

Don’t

Use Empty Interface

“interface{} says nothing. — Rob Pike”

- 일반적인 상황에서 empty interface를 사용하는 것은 성능에 큰 영향을 미치진 않지만, 처리할 데이터 양이 매우 많은 경우엔 문제가 될 수 있음.
- interface type ↔ concrete type는 컴파일러가 최적화하기 어렵게 만들고, 형 변환 과정에서 할당과 복사를 자주 하게 됨.
- 성능이 아니더라도 empty interface는 유지보수성에도 영향을 미침.

Don’t

Use Dark Arts

“With the unsafe package there are no guarantees. — Rob Pike”

- 초기에 퍼포먼스 최적화를 위해 굉장히 많은 곳에서 unsafe 패키지를 사용했고, 많은 버그를 겪었음.
- Go에서 segfault와 buffer overflow를 겪는다는건… 정말 슬픈일임.
- 거의 모든 상황에서 unsafe와 같은 것을 사용한 최적화보다 유지보수하기 쉬운 코드가 훨씬 가치가 있음.
  - **대부분의 성능 저하는 Go가 아니라 내가 짠 비효율적인 로직에서 발생함.**
- 쉽게 테스트할 수 있고, 코드 수정이 잦지 않은 정말 일부 영역에서만 사용하는 것이 좋음.
  - **현재는 unsafe를 사용하는 함수에는 ‘Unsafe’라는 prefix를 강제하는 방식으로 관리하고 있음.**

```
unexpected fault address 0x722023f4b
fatal error: fault
[signal SIGSEGV: segmentation violation code=0x1 addr=0x722023f4b pc=0x10712f0]

goroutine 11787 [running]:...
```

Do

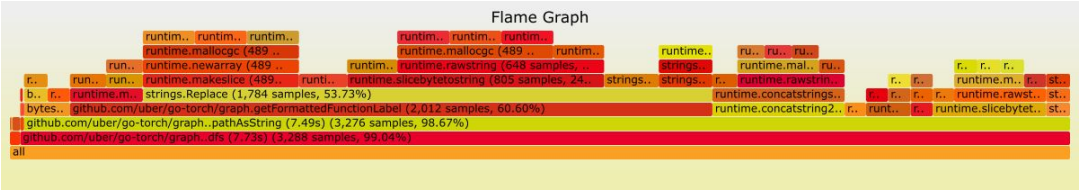
Use pprof

- 새로운 기능을 개발할 때 마다 pprof를 통해서 cpu, memory profiling을 주기적으로 함.
- pprof 자체가 성능을 개선시켜 주는 것은 아니지만 주기적인 pprof를 통해 정말 많은 인사이트를 얻을 수 있었음.

runtime.concatstrings

/Users/jbd/go/src/runtime/string.go

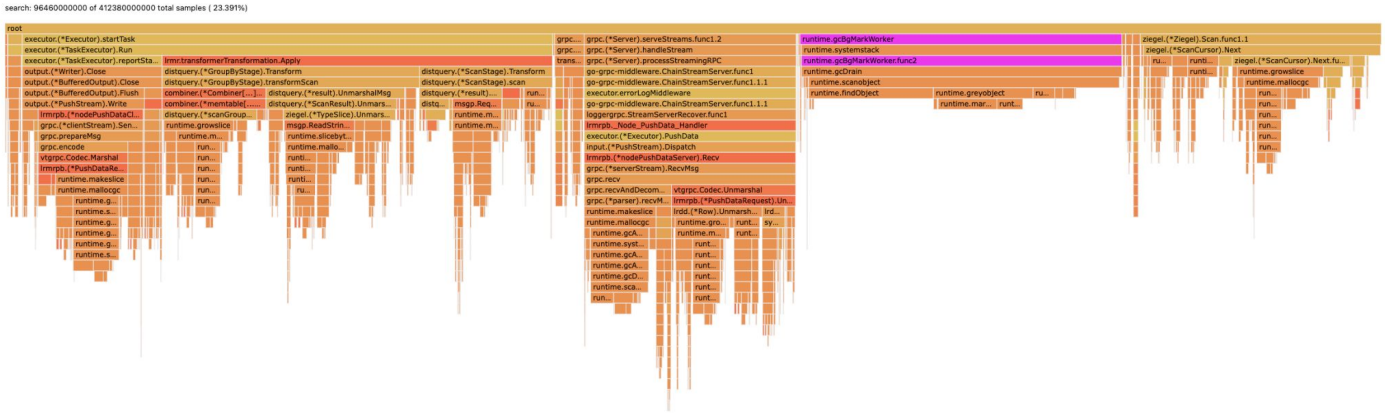
```
Total:       530ms      870ms (flat, cum) 30.63%
    15          .          . // concatstrings implements a Go string concatenation x+y+z+...
    16          .          . // The operands are passed in the slice a.
    17          .          . // If buf != nil, the compiler has determined that the result does not
    18          .          . // escape the calling function, so the string data can be stored in buf
    19          .          . // if small enough.
    20       40ms       40ms func concatstrings(buf *tmpBuf, a []string) string {
    21          .          .         idx := 0
    22          .          .         l := 0
    23          .          .         count := 0
    24       10ms       10ms         for i, x := range a {
    25          .          .                 n := len(x)
    26          .          .                 if n == 0 {
    27          .          .                         continue
    28          .          .                 }
    29       20ms       20ms                 if l+n < l {
    30          .          .                         throw("string concatenation too long")
    31          .          .                 }
    32          .          .                 l += n
    33       10ms       10ms                 count++
    34          .          .                 idx = i
    35          .          .         }
    36       10ms       10ms         if count == 0 {
    37          .          .                 return ""
    38          .          .         }
```



Do

Use sync.Pool

- 항상 문제로 제기되어온 것이지만 GC는 비쌈...
- Luft에서는 STW로 인한 latency가 문제가 아니라 background marking이 차지하는 25%의 CPU 오버헤드가 문제가 됨.
- sync.Pool을 통해 오브젝트를 재사용하고, GOGC와 GOMEMLIMIT을 잘 조절하여 GC의 발생 빈도를 줄이는 것은 큰 도움이 됨.
  - 틈새 홍보: https://github.com/KimMachineGun/automemlimit



Do

Use Latest Go Version

- 아직 공짜 점심은 있음. (6개월마다 제공중)
- 하위 호환성을 잘 지켜주는 덕분에 큰 걱정 없이 버전을 올릴 수 있는게 Go의 큰 장점 중 하나라고 생각함.

What's new?

- `sync.Pool`, a GC-aware tool for reusing memory, has a lower latency impact and recycles memory much more effectively than before. (Go 1.13)
- The Go runtime returns unneeded memory back to the operating system much more proactively, reducing excess memory consumption and the chance of out-of-memory errors. This reduces idle memory consumption by up to 20%. (Go 1.13 and 1.14)
- The Go runtime is able to preempt goroutines more readily in many cases, reducing stop-the-world latencies up to 90%. Watch the talk from Gophercon 2020 here. (Go 1.14)
- The Go runtime manages timers more efficiently than before, especially on machines with many CPU cores. (Go 1.14)
- Function calls that have been deferred with the `defer` statement now cost as little as a regular function call in most cases. Watch the talk from Gophercon 2020 here. (Go 1.14)
- The memory allocator's slow path scales better with CPU cores, increasing throughput up to 10% and decreasing tail latencies up to 30%, especially in highly-parallel programs. (Go 1.14 and 1.15)
- Go memory statistics are now accessible in a more granular, flexible, and efficient API, the runtime/metrics package. This reduces latency of obtaining runtime statistics by two orders of magnitude (milliseconds to microseconds). (Go 1.16)
- The Go scheduler spends up to 30% less CPU time spinning to find new work. (Go 1.17)
- Go code now follows a register-based calling convention on amd64, arm64, and ppc64, improving CPU efficiency by up to 15%. (Go 1.17 and Go 1.18)
- The Go GC's internal accounting and scheduling has been redesigned, resolving a variety of long-standing issues related to efficiency and robustness. This results in a significant decrease in application tail latency (up to 66%) for applications where goroutines stacks are a substantial portion of memory use. (Go 1.18)
- The Go GC now limits its own CPU use when the application is idle. This results in 75% lower CPU utilization during a GC cycle in very idle applications, reducing CPU spikes that can confuse job shapers. (Go 1.19)

Full version: https://go.dev/blog/go119runtime

What’s Next

What’s Next

- **Predicate Pushdown:** 컬럼별 min, max와 같은 메타데이터와 bitmap index 등을 활용해서 읽을 데이터의 총량을 줄임.
- **Per Column Replication:** 현재 특정 기간의 데이터는 하나의 Ziegel 파일로 묶여 분산 저장/처리됨. 이를 컬럼 단위로 쪼개서 자주 사용되는 컬럼과 자주 사용되지 않는 컬럼에 다른 replication strategy를 지정하여 관리함.
- **Vectorized Execution (feat. SIMD):** 데이터 처리 과정을 벡터화하여 데이터 처리 성능을 개선함.
  - 현재 Graviton 인스턴스를 사용하고 있어서 Neon Instruction을 지원하는 라이브러리를 간절히 기다리는중.
- **Manual Memory Management (feat. Arena?):** 다소 논란이 있는 주제지만 Go 1.20에서 시험적으로 도입될 arena 패키지와 같은 기능을 활용하여 데이터 serialization 중 발생하는 GC 오버헤드를 줄임.

QnA

감사합니다.

Geon Kim, Backend Engineer – Query Engine Team
Email : geon@ab180.co
GitHub: @KimMachineGun

AB180 INC.
서울특별시 서초구 강남대로61길 17, 3층
www.ab180.co