Performance in a High-throughput SQL Database

Real-world Profiler Tips

Zach Musgrave, DoltHub
GopherCon 2022

Agenda

1. Introduction
2. Using the profiler toolchain
3. Performance case studies
4. High performance tuple store

Introduction

Who am I?

Zach Musgrave: project lead for Dolt

DoltHub: 20 person database startup

Formerly: Google (award-winning memer), Amazon (award-winning developer)

Gopher of ~4 years

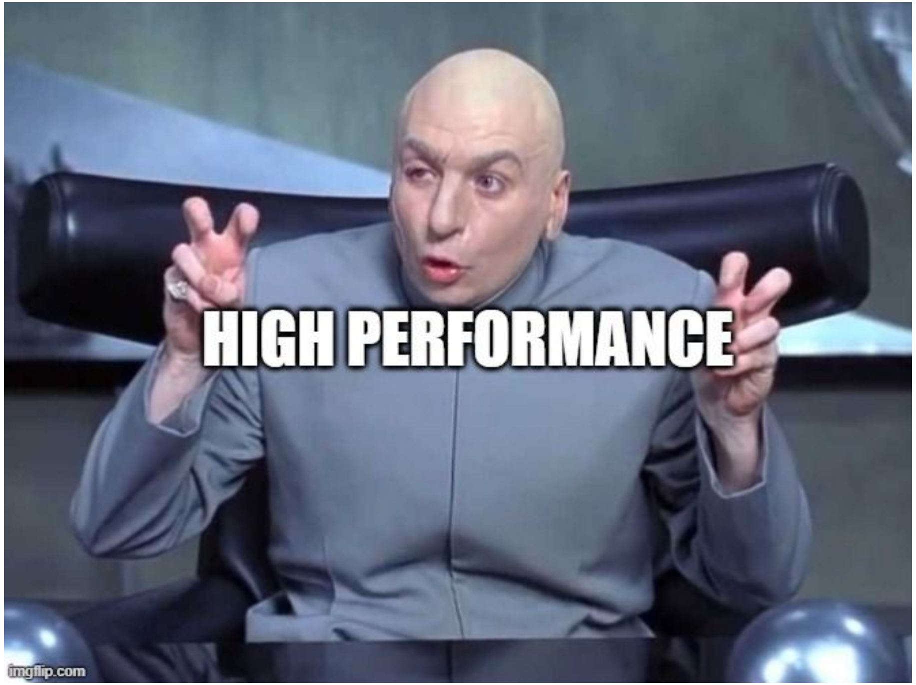
What's Dolt?

The world's first SQL database you can branch, merge, fork, clone, push, pull like a git repository

Torvalds sarcastically quipped about the name git (which means ***"unpleasant person"*** *in British English slang):*

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."

A high performance tuple store



Why is that hard?

grauenwolf 1d

I'll up vote it for a good write up. But WTF are you building your own database? Unless you have a very, very specific scenario, just don't.

7 Reply

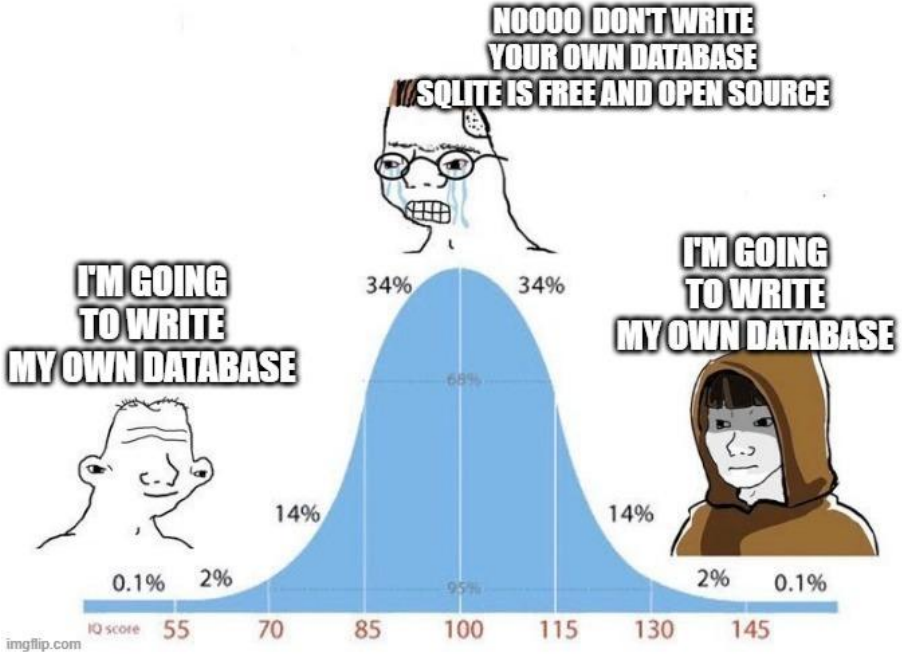
my_password_is______ 21h

why would anyone vote this down
its the correct thinking

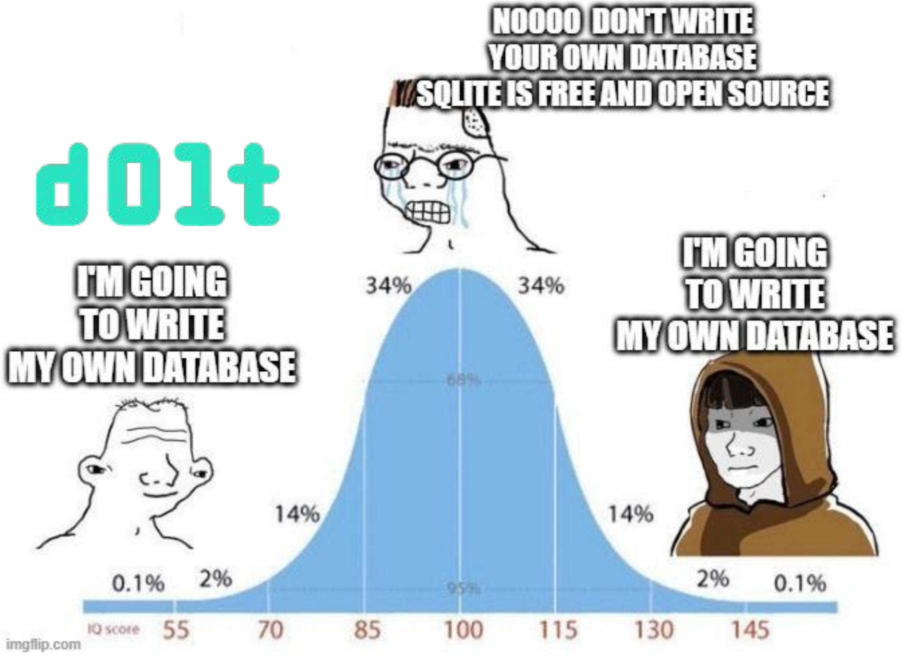
sqlite is free and open source
postgresql is free and open source

why build your own LOL

Why are we doing this?



Why are we doing this?



Using the profiler

The golang profiler

Two parts:

- `runtime/pprof`: profile your application
- `go tool pprof`: analyze profiler dumps

Instrumenting your code

```
import "runtime/pprof"
…
func main() {
    profileWr = os.Create(“/tmp/profile.prof”)
    pprof.StartCPUProfile(profileWr)
    mainLogic()
    pprof.StopCPUProfile()
}
```

Better: use a library

```
go get github.com/pkg/profile
```

```go
import (
    ...
    "github.com/pkg/profile"
)
func main() {
    defer profile.Start(profile.CPUProfile,
                        profile.NoShutdownHook).Stop()
    mainLogic()
}
```

Even better: add profiler flags

```
func main() {
    args := os.Args[1:]
    if len(args) > 0 {
        switch args[0] {
        case profFlag:
            switch args[1] {
            case cpuProf:
                defer profile.Start(profile.CPUProfile,
                  profile.NoShutdownHook).Stop()
            case memProf:
                defer profile.Start(profile.MemProfile,
                  profile.NoShutdownHook).Stop()
            }
            args = args[2:]
        }
    }
}
```

Profiler flags cont.

```
% dolt --prof cpu
% dolt --prof memory
```

Other good options: custom environment variables or cues

Bad option: changing your code to run the profiler

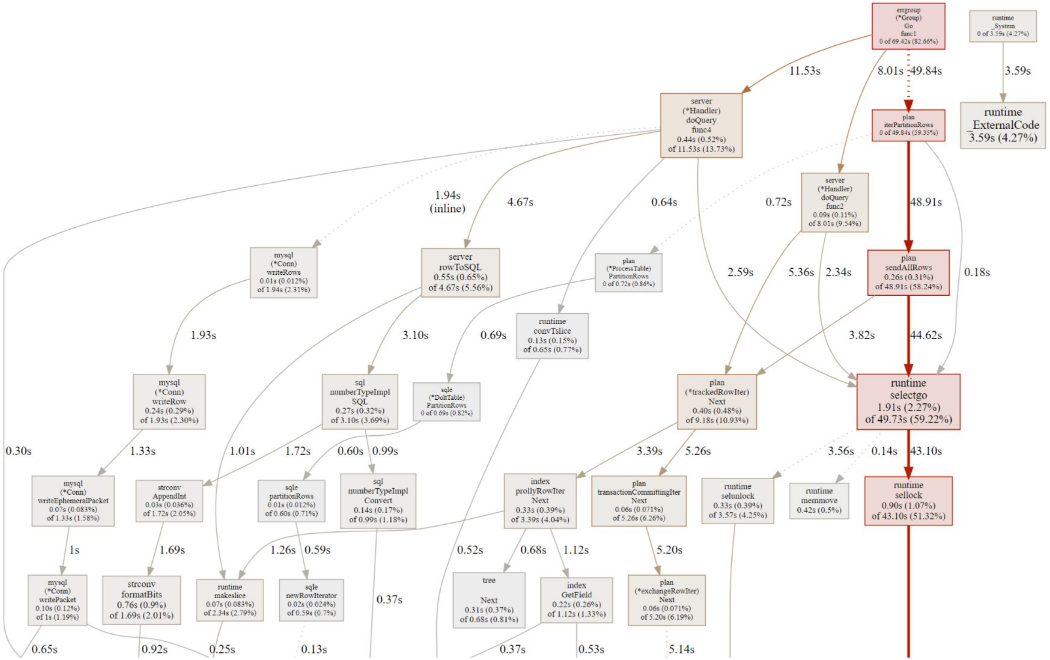
Running pprof

Basic invocation:

```
go tool pprof <options> cpu.pprof
```

Ignore all options and run `-http:8080`

Demo: using pprof http server



Performance profiling case studies

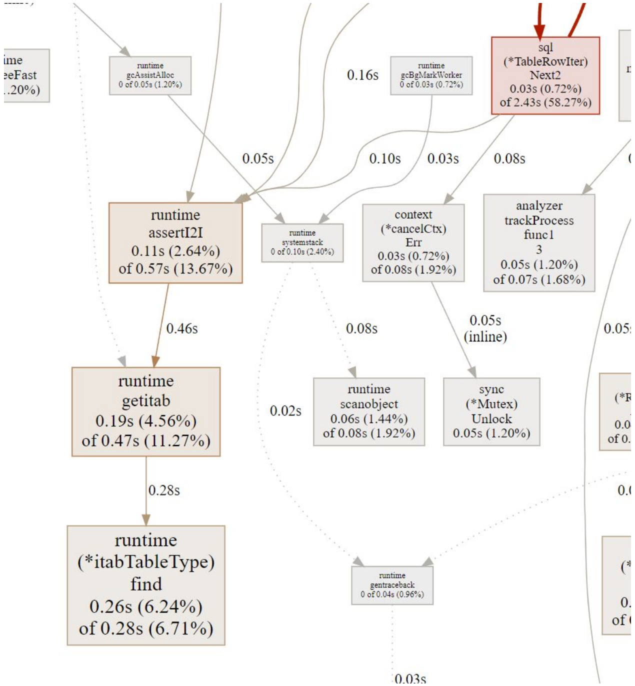
Case study: interface assertion

```go
type RowIter interface {
    Next(ctx *Context) (Row, error)
}

type RowIter2 interface {
    RowIter
    Next2(ctx *Context, frame *RowFrame) error
}
```

Case study: interface assertion

```go
type iter struct {
    childIter  sql.RowIter
}

func (t iter) Next2(
        ctx *sql.Context,
        frame *sql.RowFrame,
) error {
    return t.childIter.(sql.RowIter2).Next2(ctx, frame)
}
```

Profiler demo



Solution: remove type assertions

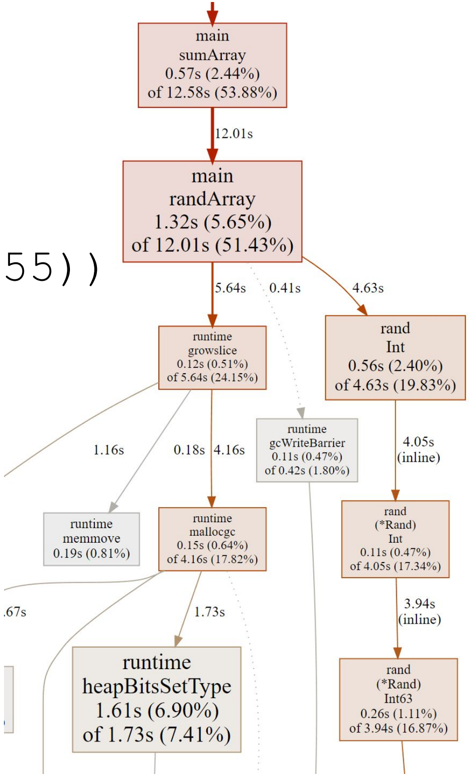
```
type iter struct {
    childIter  sql.RowIter
    childIter2 sql.RowIter2
}
func (t iter) Next(...) {
    return t.childIter.Next(ctx)
}
func (t iter) Next2(...) {
    return t.childIter2.Next2(ctx, frame)
}
```

Case study: working with slices

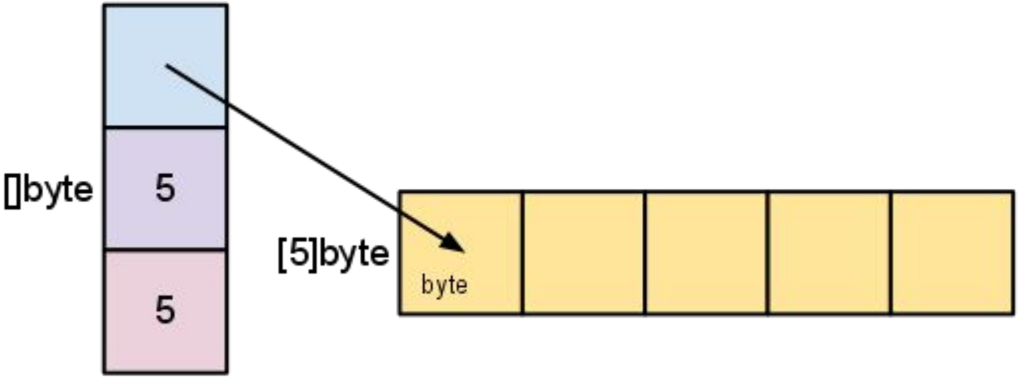
```go
func sumArray() uint64 {
    a := randArray()
    var sum uint64
    for i := range a {
        sum += uint64(a[i].(byte))
    }
    return sum
}
```

Worst: appending to a slice

```
func randArray() []interface{} {
    var a []interface{}
    for i := 0; i < 1000; i++ {
        a = append(a, byte(rand.Int() % 255))
    }
    return a
}
```

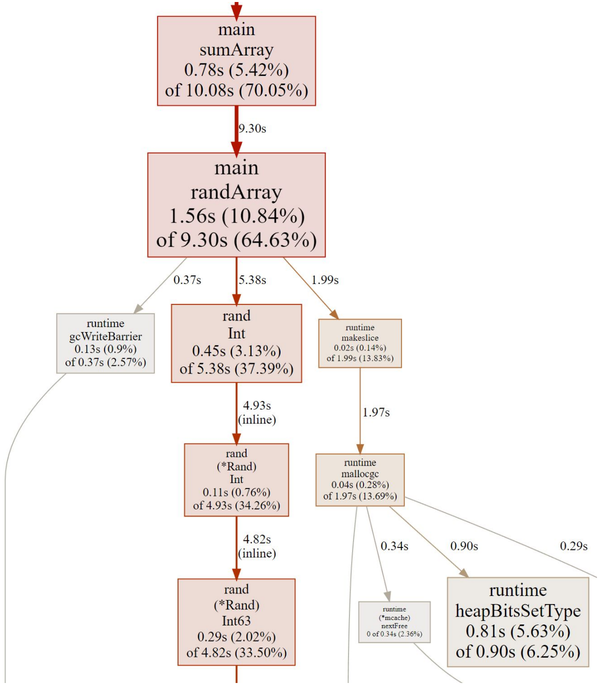


Growing a slice is slow



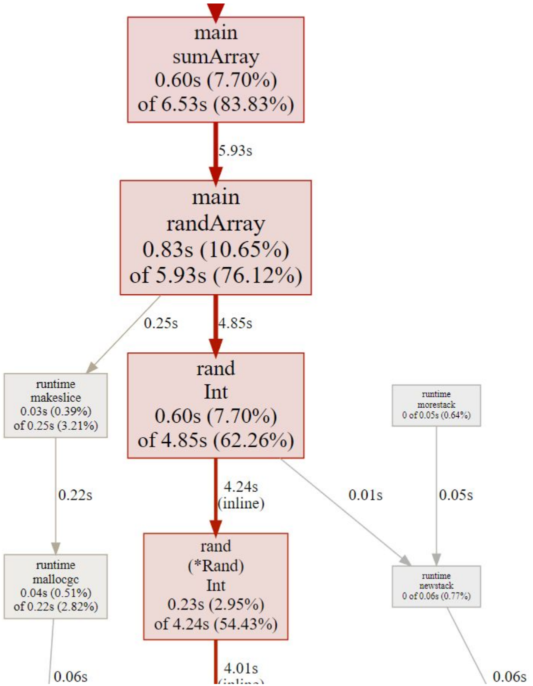
Better: static slice size

```go
func randArray() []interface{} {
	a := make([]interface{}, 1000)
	for i := 0; i < len(a); i++ {
		a[i] = byte(rand.Int() % 255)
	}
	return a
}
```



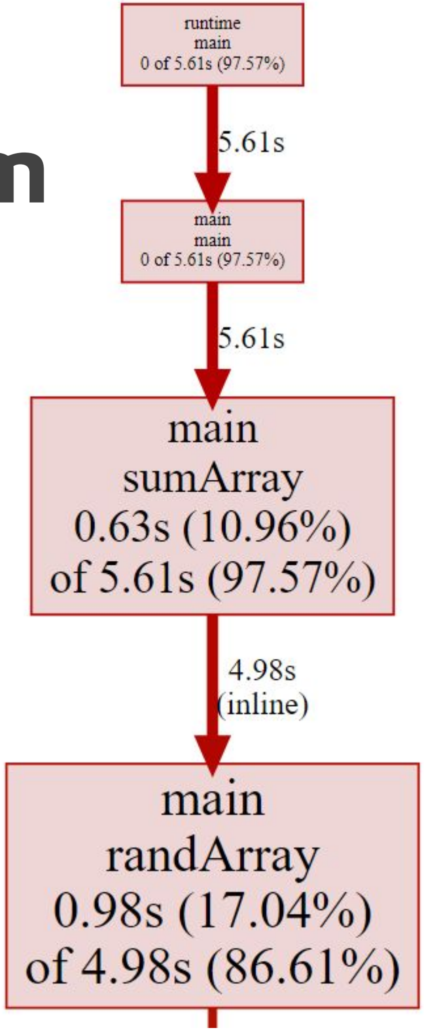
Even better: don't use interface{}

```go
func randArray() []byte {
    a := make([]byte, 1000)
    for i := 0; i < len(a); i++ {
        a[i] = byte(rand.Int() % 255)
    }
    return a
}
```



Best: pass a slice param

```go
func randArray(a []interface{}) {
	for i := 0; i < len(a); i++ {
		a[i] = byte(rand.Int() % 255)
	}
}
```

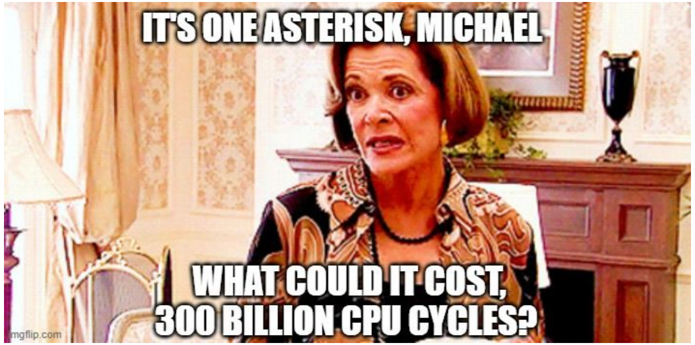


Slice operation efficiency

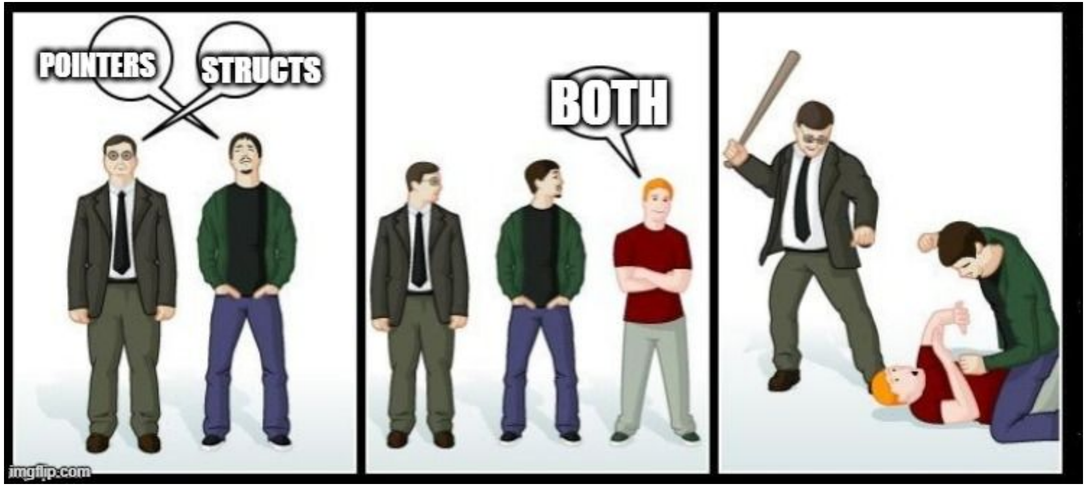
- Appending: 20%
- Static slice with interface{}: 38%
- Static slice with byte: 62%
- Slice as param: 70%

Case study: struct as receiver

```go
func (b bigStruct) randFloat() float32 {
    x := rand.Float32()
    return x
}

func (b *bigStruct) randFloat() float32 {
    x := rand.Float32()
    return x
}
```



Case study: struct as receiver



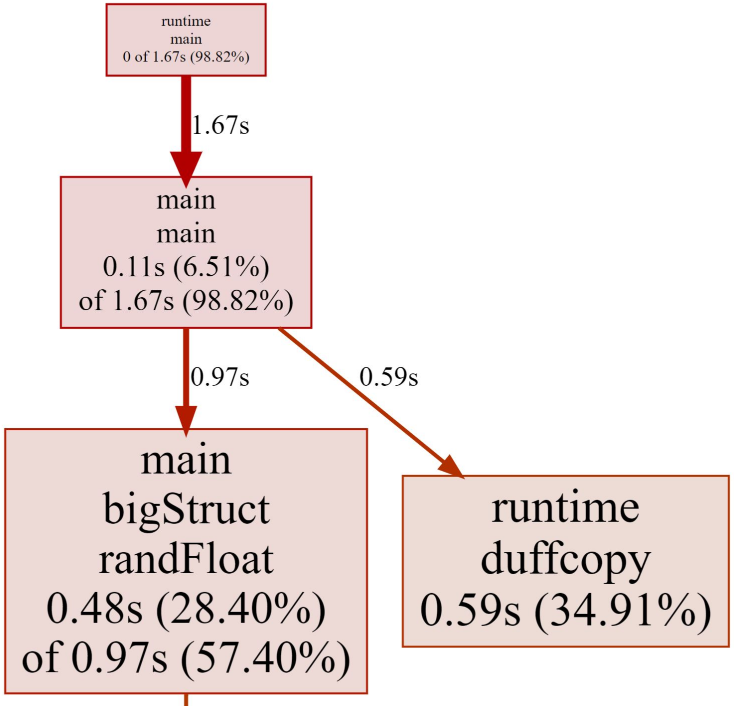
Let’s be civil

	Pros	Cons
Structs	✅Stays on the stack ✅Immutable	❌More memory to copy
Pointers	✅Mutable ✅Less memory to copy	❌Goes on the heap
Both	✅Choose your own adventure ✅Irritate pedants	❌go vet warnings ❌pedant is your tech lead

I thought this was a talk on performance

```go
type bigStruct struct {
    v1, v2, v3, v4, v5, v6, v7, v8, v9 uint64
    f1, f2, f3, f4, f5, f6, f7, f8, f9 float64
    b1, b2, b3, b4, b5, b6, b7, b8, b9 []byte
    s1, s2, s3, s4, s5, s6, s7, s8, s9 string
}

func (b bigStruct) randFloat() float64 {
    x := rand.Float32()
    switch {
    case x < .1:
        return b.f1
    …
}
```

Profiler demo



CS history aside: Duff’s Device

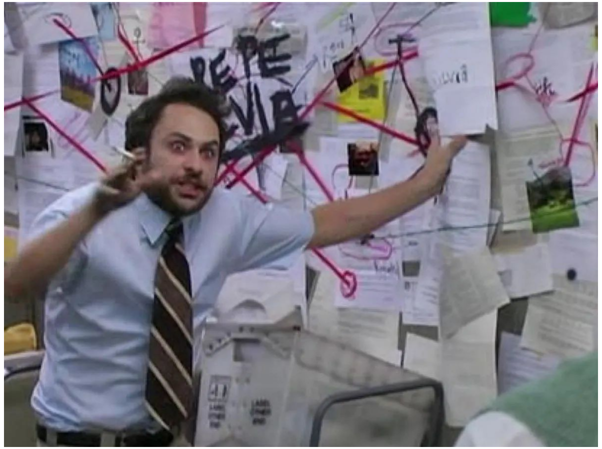
```c
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

src/runtime/mkduff.go:

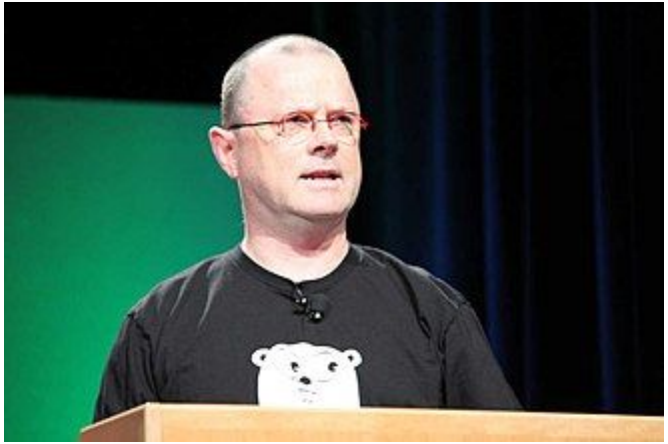
```go
// runtime.duffcopy is a Duff's device for copying memory.
// The compiler jumps to computed addresses within
// the routine to copy chunks of memory.
// Source and destination must not overlap.
// Do not change duffcopy without also
// changing the uses in cmd/compile/internal/*/*.go.
```

Tom Duff

- Canadians born in 50s
- University of Toronto
- Bell Labs
- Plan 9



Rob Pike



DigitalIlusion @Digital_ilusion · Feb 10, 2015

#go #golang Duff’s devices ow.ly/IszH0

1

Rob Pike
@rob_pike

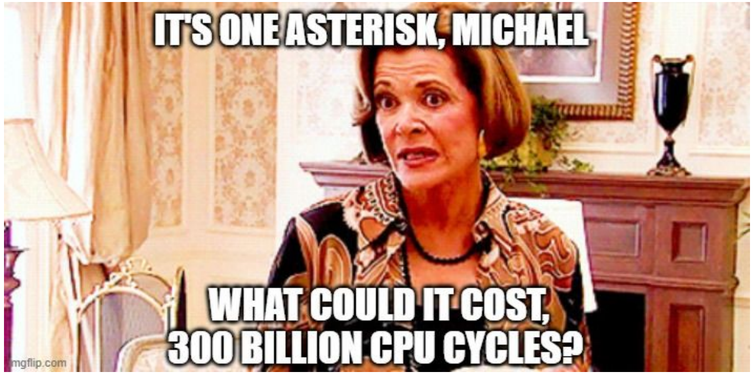
Replying to @Digital_ilusion

@Digital_ilusion That's not really a Duff's device. Duff's device is a C trick mingling a loop and a switch statement.

6:28 AM · Feb 10, 2015 · Twitter Web Client

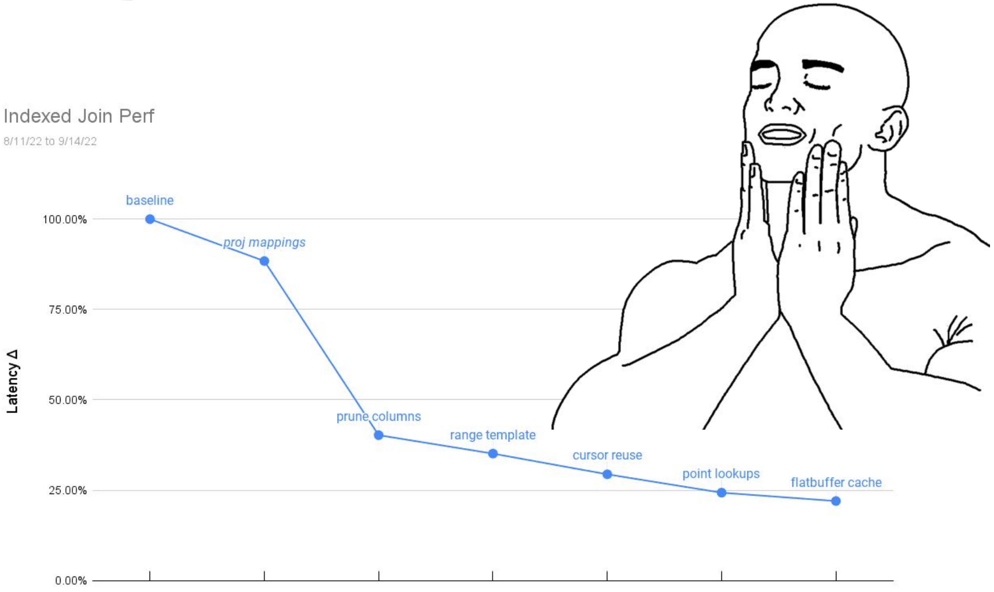
Solution: use a pointer receiver

```go
func (b *bigStruct) randFloat() float32 {
    x := rand.Float32()
    return x
}
```

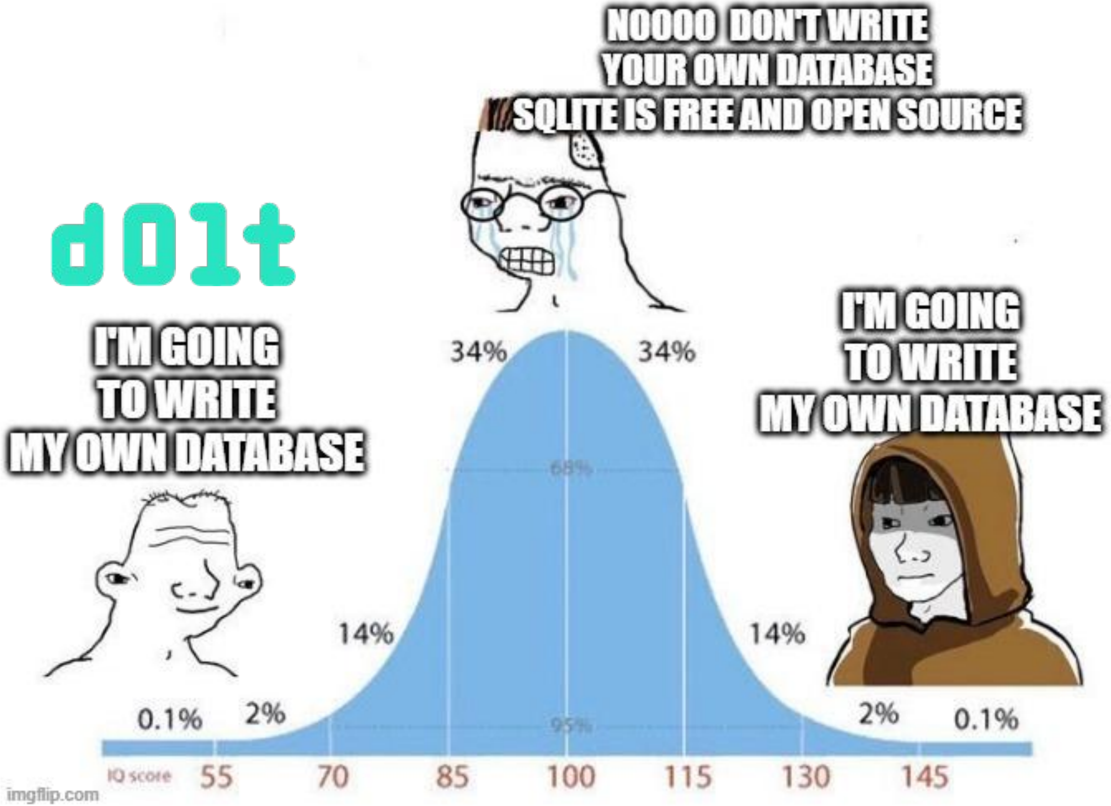


Deep dive:
Profiling a high-performance tuple store

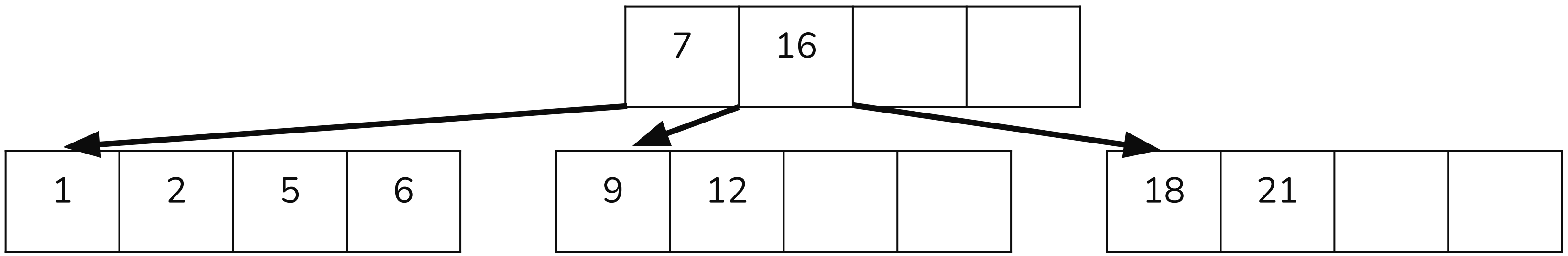
The limits of profiling



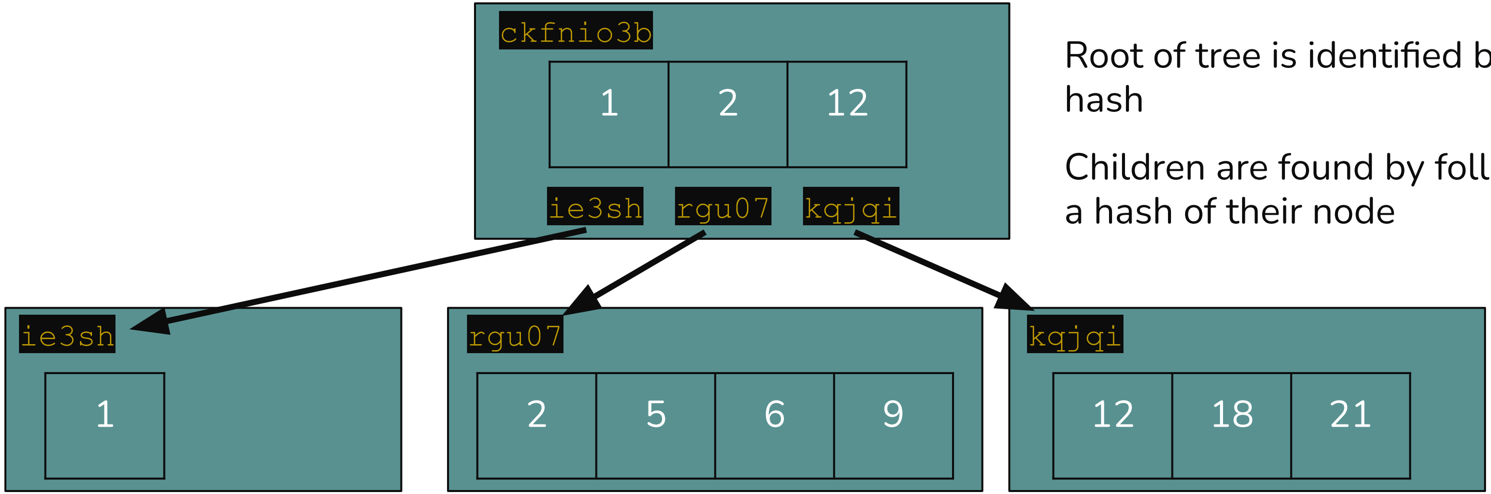
Writing your own database



B-Trees



Prolly Trees



Root of tree is identified by its hash

Children are found by following a hash of their node

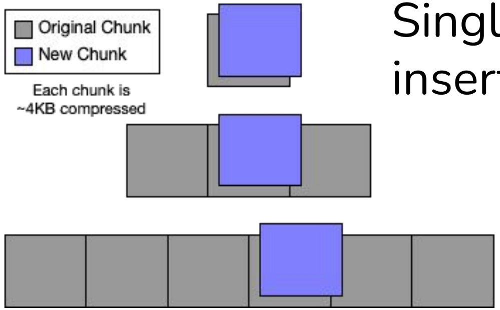
Building a prolly tree

We build a prolly tree by separating it into chunks based on a rolling hash of the contents. When the hash falls below a certain threshold, start a new chunk. Choose a threshold to get an (average) desired chunk size

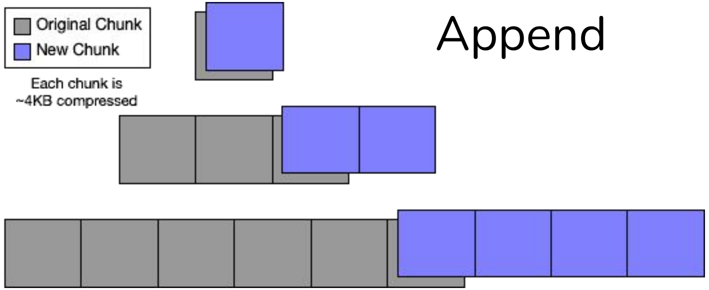
Hash	4	40	107	125	3	109	88	60
Content	1	2	5	6	9	12	18	21

Structural sharing

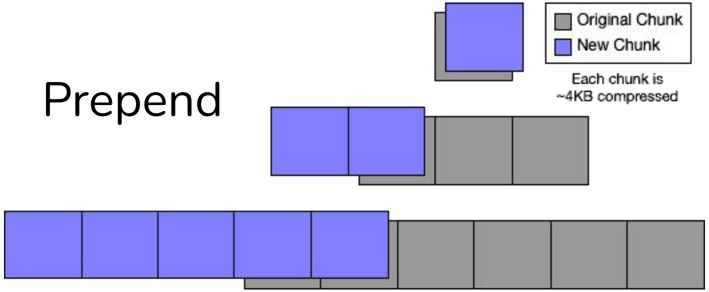
Single insert



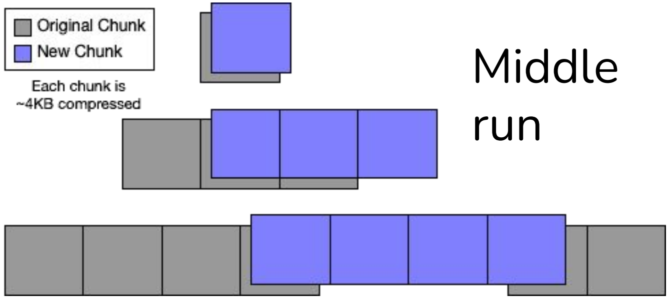
Append



Prepend



Middle run



Problems with original architecture

Too general: graph storage v. tabular

Too verbose: expensive to deserialize

Too much overhead: limits branching factor

Table data

ID (tag=3)	First_name (tag=10)	Last_name (tag=4)
1	Zach	Musgrave
2	Aaron	Son
3	Cher	

```
{
      (3,1) => (10, "Zach", 4, "Musgrave"),
      (3,2) => (10, "Aaron", 4, "Son"),
      (3,3) => (10, "Cher"),
}
```

Self-describing serialization

```
Struct Commit {
    Value: Set<String>
    Parents: Set<Ref<Cycle<Commit>> |
        Ref<Struct Commit {
            Value: Set<Number>
            Parents: Cycle<Commit>
        }>>
    }
}
```

Deserializing noms objects

$Type_0$	$Length_0$	$Value_0$	$Type_1$	$Length_1$	$Value_1$	…

1) Read type + length
2) Instantiate appropriate deserializer for type
3) Repeat for entire array

O(N) access time for any element (1st load)

New tuple storage

$Value_0$	$Value_1$	…	$Value_k$	$Offset_1$	$Offset_2$	…	$Offset_k$	Count

Types stored out of band in schema object

Each offset is uint16 (65k max row size)

O(1) random access inside a row

New object storage

Flatbuffers:

Zero-deserialization (!!) persistence

Sounds too good to be true but somehow works

Flatbuffers example

```
table ProllyTreeNode {
  key_items:[ubyte] (required);
  key_offsets:[uint16] (required);

  value_items:[ubyte];
  value_offsets:[uint16];
  value_address_offsets:[uint16];

  address_array:[ubyte];
}
```

Flatbuffer results

Format	Ops	Time	Mem	Allocs
Old	19893	60952 ns/op	21086 B/op	484 allocs/op
Flatbuffers	183279	7199 ns/op	544 B/op	5 allocs/op
Proto	153792	10043 ns/op	6912 B/op	22 allocs/op

10x faster than old format

30% faster than proto
90% less memory allocated

Performance metrics: reads

Read Tests	MySQL	Dolt	Multiple
covering_index_scan	1.96	6.55	3.3
groupby_scan	12.52	22.69	1.8
index_join	1.18	16.71	14.2
index_join_scan	1.12	16.12	14.4
index_scan	30.26	73.13	2.4
oltp_point_select	0.15	0.57	3.8
oltp_read_only	3.02	9.56	3.2
select_random_points	0.31	1.39	4.5
select_random_ranges	0.36	1.37	3.8
table_scan	30.81	69.29	2.2
types_table_scan	69.29	223.34	3.2
reads_mean_multiplier			5.2

Read Tests	MySQL	Dolt	Multiple
covering_index_scan	1.93	2.76	1.4
groupby_scan	12.08	17.32	1.4
index_join	1.18	4.57	3.9
index_join_scan	1.12	3.89	3.5
index_scan	30.26	53.85	1.8
oltp_point_select	0.15	0.46	3.1
oltp_read_only	2.91	8.28	2.8
select_random_points	0.3	0.74	2.5
select_random_ranges	0.35	1.12	3.2
table_scan	30.81	63.32	2.1
types_table_scan	69.29	193.38	2.8
reads_mean_multiplier			2.6

Performance metrics: writes

Write Tests	MySQL	Dolt	Multiple
bulk_insert	0.001	0.001	1.0
oltp_delete_insert	2.81	19.65	7.0
oltp_insert	1.55	7.98	5.1
oltp_read_write	5.28	36.89	7.0
oltp_update_index	1.58	9.39	5.9
oltp_update_non_index	1.47	6.55	4.5
oltp_write_only	2.39	26.2	11.0
types_delete_insert	2.91	155.8	53.5
writes_mean_multiplier			11.9

Write Tests	MySQL	Dolt	Multiple
bulk_insert	0.001	0.001	1.0
oltp_delete_insert	3.02	10.84	3.6
oltp_insert	1.58	2.81	1.8
oltp_read_write	5.18	16.71	3.2
oltp_update_index	1.61	4.91	3.0
oltp_update_non_index	1.55	5.18	3.3
oltp_write_only	2.3	8.28	3.6
types_delete_insert	3.13	12.08	3.9
writes_mean_multiplier			2.9

Overall metrics

50% reduction in read latency

75% reduction in write latency

3x overall speedup

Thanks for coming!

To learn more:
dolthub.com
github.com/dolthub/dolt