Optimizing Heap Allocations in Golang: A Case Study

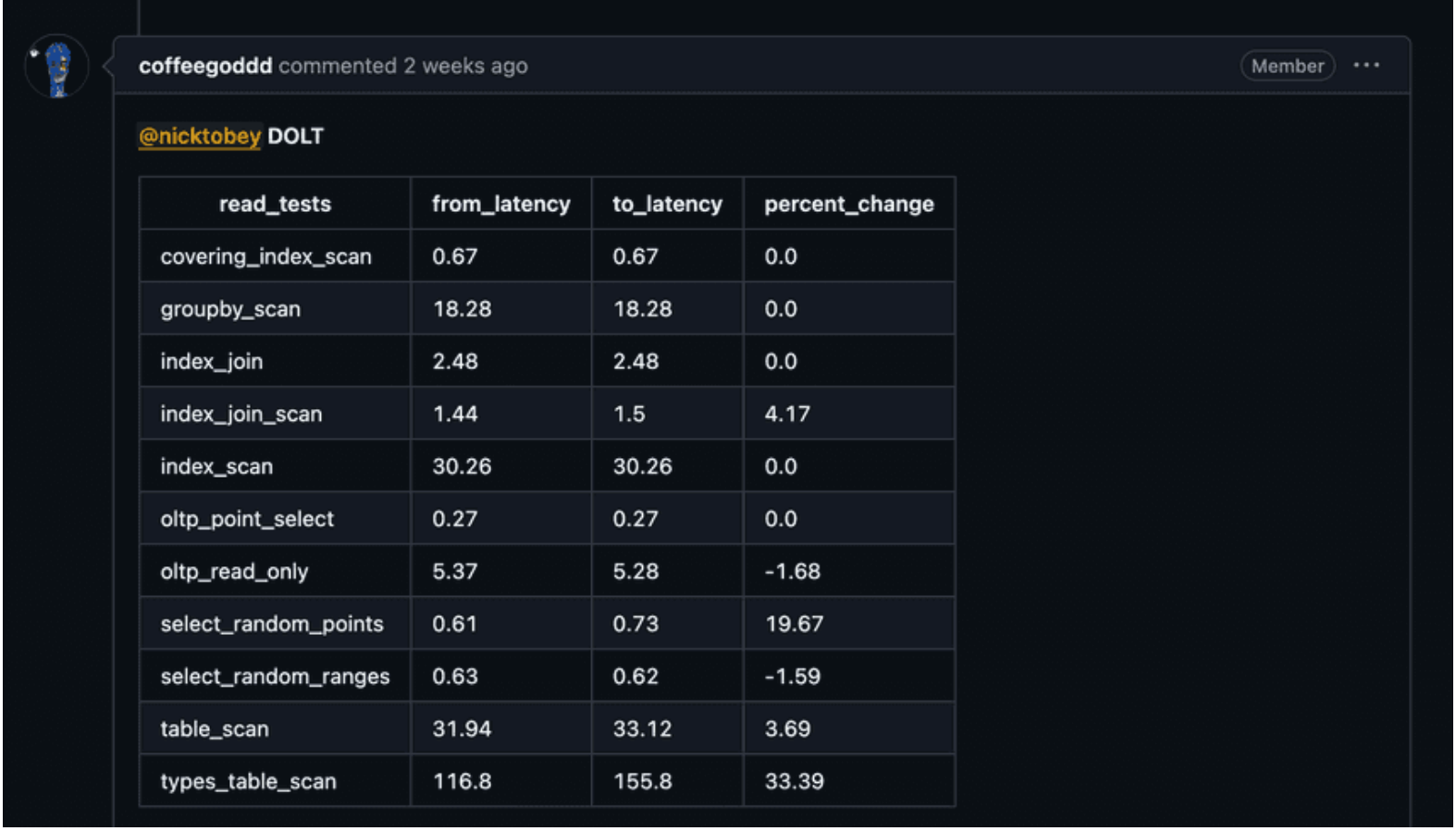
Nick Tobey

April 18, 2025

GOLANG 9 min read

I work on **Dolt, the first SQL database with git-like version control**, written entirely in Go. And as a rule, databases need to be fast. So we have lots of tests in our CI workflow to monitor performance regressions before they ever hit our main branch.

Last month, a commit that was supposed to be a no-op refactor caused a 30% regression in **sysbench**'s `types_scan` benchmark.



coffeegoddd commented 2 weeks ago Member

@nicktobey DOLT

read_tests	from_latency	to_latency	percent_change
covering_index_scan	0.67	0.67	0.0
groupby_scan	18.28	18.28	0.0
index_join	2.48	2.48	0.0
index_join_scan	1.44	1.5	4.17
index_scan	30.26	30.26	0.0
oltp_point_select	0.27	0.27	0.0
oltp_read_only	5.37	5.28	-1.68
select_random_points	0.61	0.73	19.67
select_random_ranges	0.63	0.62	-1.59
table_scan	31.94	33.12	3.69
types_table_scan	116.8	155.8	33.39

Pictured: An example of what you don't want to see on your GitHub PR.

Uncovering the root cause of this regression turned out to be quite illuminating in how Golang handles memory allocations, and I want to share it here as an exercise. The details have been simplified, but the lessons are real.

The change has to do with a type called `ImmutableValue`, which represents a content hash, an optional byte buffer containing the data with that hash, and a `ValueStore` interface that can resolve content hashes to binary blobs. `ImmutableValue` has a `GetBytes` method that checks if the blob has already been loaded, uses the `ValueStore` to load it into the buffer if it hasn't already, and then returns the buffer. Part of the goal of the refactor was to hide certain implementation details around how the `ValueStore` resolves hashes by moving those details into an interface method.

Here's a cleaned-up version of the original `GetBytes` implementation:

```
func (t *ImmutableValue) GetBytes(ctx context.Context) ([]byte, error) {
  if t.Buf == nil {
      err := t.load(ctx)
      if err != nil {
```

```
            return nil, err
        }
    }
    return t.Buf[:], nil
}

func (t *ImmutableValue) load(ctx context.Context) {
    if t.Addr.IsEmpty() {
        t.Buf = []byte{}
        return
    }
    WalkNodes(ctx, &t.valueStore, h, func(ctx context.Context, n Node) {
          if n.IsLeaf() {
              t.Buf = append(t.buf, n.GetValue(0)...)
          }
    })
}
```

And here's a cleaned-up version of the new implementation:

```
func (t *ImmutableValue) GetBytes(ctx context.Context) ([]byte, error) {
    if t.Buf == nil {
        if t.Addr.IsEmpty() {
            t.Buf = []byte{}
            return t.Buf, nil
        }
        buf, err := t.valueStore.ReadBytes(ctx, t.Addr)
        if err != nil {
            return nil, err
        }
        t.Buf = buf
    }
    return t.Buf, nil
}

// Assert that nodeStore implements ValueStore
var _ ValueStore = &nodeStore{}

type nodeStore struct {
    chunkStore interface {
      WalkNodes(ctx context.Context, h hash.Hash, cb CallbackFunc)
    }
    // Other fields removed for simplicity
}

func (ns nodeStore) ReadBytes(ctx context.Context, h hash.Hash) (result []byte) {
    ns.chunkStore.WalkNodes(ctx, h, func(ctx context.Context, n Node) {
        if n.IsLeaf() {
            result = append(result, n.GetValue(0)...)
```

```
        }
    })
    return result
}
```

Profiling showed that the new `ReadBytes` method was spending a third of its runtime calling `runtime.newobject`, a builtin function that Go uses for allocating memory on the heap. This allocation only happened in the new implementation.

Given this, there are two questions to this exercise:

1. Where is the extra allocation happening?
2. Why is this allocation happening on the heap? Why is it not happening on the stack?

Experienced Golang devs might spot the problem immediately and shout at me for making such a foolish, obvious mistake. But everyone else should take a moment to try and come up with an answer.

▶ When you're ready, click here to reveal the answer.

How did this happen?

First, we need to understand what methods in Golang actually are.

Value Receivers vs Pointer Receivers

Go is not an object oriented language. It doesn't have inheritance or abstract classes. In fact, it doesn't have classes at all. It has dynamic dispatch, but only for interfaces.

Go does have methods, which specify a receiver type and are callable on values of that type. Methods are used for implementing interfaces, but also serve as a namespace for functions. So ignoring interfaces, `func (receiver ReceiverType) Foo(param ParamType)` is equivalent to `func ReceiverType.Foo(receiver ReceiverType, param ParamType)`, with the following additional properties:

- ReceiverType must be either a value type (a type that is not a pointer or an interface), or a pointer to a value type. This value type is called the base type of the method. So if the base type is `T`, then the receiver is either `T` or `*T`.
- For a base type `T` (meaning the receiver type is either `T` or `*T`), the expression `a.Foo` is valid if `a` is either `T` or `*T`.

This means that the following are all valid ways to call a method:

```
type ReceiverType struct {
  s string
}

func (receiver ReceiverType) ValueReceiver() {
  fmt.Printf("called ValueReceiver(%s)", receiver.s)
}
func (receiver *ReceiverType) PointerReceiver() {
    fmt.Printf("called PointerReceiver(%s)", receiver.s)
}

value := ReceiverType{"value"}
pointer := &ReceiverType{"pointer"}

func main() {
  value.ValueReceiver()    // 1) This is equivalent to ReceiverType.ValueReceiver(value)
  value.PointerReceiver()  // 2) This is equivalent to ReceiverType.PointerReceiver(&value)

  pointer.PointerReceive() // 3) This is equivalent to ReceiverType.PointerReceive(pointer)
  pointer.ValueReceiver()  // 4) This is equivalent to ReceiverType.ValueReceiver(*pointer)
}
```

Note that option 4 involves dereferencing the receiver and passing the value as a function parameter. This means that the receiver value gets copied.

And that's our answer to question 1: because the `ReadBytes` method has a value receiver, every time it gets called, a new `nodeStore` is created.

But the problem isn't just that calling the method copies the parameters: in a pass-by-value language like Go, function parameters get copied all the time. But usually, these copies exist on the stack. Creating

values on the stack is relatively cheap. The issue is that these copies are being created on the heap, and even small heap allocations can be expensive when there's a lot of them.

So what's *really* going on here?

Stack Allocation vs Heap Allocation

In many languages, it's always obvious whether something is created on the stack vs the heap. In C++ for instance, function parameters and local variables are always on the stack, and the only way to do heap allocation is explicitly, with something like the `new` keyword. That level of control can make program performance more predictable, but it can also make it easy to have pointers to freed memory. Take this C++ example:

```
#include <iostream>
#include <string>

int* getPointer(int x) {
    return &x;
}

int main()
{
  int* ptr1 = getPointer(1);
  int* ptr2 = getPointer(2);
  std::cout << *ptr1;
}
```

(**Try it yourself**)

Running the above code prints "2", not "1" as you might expect.[1] This is because the function parameter `x` exists on the stack, and ceases to exist as soon at the `getPointer` function returns. That memory gets reused for the subsequent call, resulting in the pointed-to memory being overwritten with a new stack frame.

One of the most surprising parts of Go for people coming from C++ is that the equivalent Go code is perfectly safe and correct:

```
func GetPointer(x int) *int {
    return &x
}

func main() {
    ptr1 := GetPointer(1)
    ptr2 := GetPointer(2)
    fmt.Println(*ptr1)
}
```

(**Try it yourself**)

The pointer created in the first call remains valid, despite the fact that it's a pointer to a function parameter from a stack frame that no longer exists. This seems like sorcery, but it's not: the reference to `x` is valid here because `x` was allocated on the heap. The Go compiler is allowed to allocate local variables on the heap, and this includes function arguments. In this case, the compiler deduces that the pointer to `x` will outlive the function call, and so it guarantees that the value is allocated on the heap so that the pointer remains valid.

In our actual code, the `ns` receiver value is being heap-allocated. But why? How do you influence whether Go will allocate a variable on the heap or the stack?

Unfortunately, you can't. It would be nice if the language had a way to signal to the compiler that a variable *must* be stack-allocated and force a compile-time error if it can't. But Go doesn't offer such a mechanism. The notion of stack vs heap allocation isn't something that even exists in the language. Users are expected to not worry about it… until you're optimizing performance and you *need* to worry about it.

So let's look at our problem code, bearing this in mind. The newly copied receiver value is stored in the variable `ns`. Is it possible for a pointer to that memory to outlive the call to `GetBytes`?

Here's the function again, along with the `nodeStore` type definition:

```
type nodeStore struct {
  chunkStore interface {
    WalkNodes(ctx context.Context, h hash.Hash, cb CallbackFunc)
  }
  // Other fields removed for simplicity
```

```
}

func (ns nodeStore) ReadBytes(ctx context.Context, h hash.Hash) (result []byte) {
    ns.chunkStore.WalkNodes(ctx, h, func(ctx context.Context, n Node) {
        if n.IsLeaf() {
            result = append(result, n.GetValue(0)...)
        }
    })
    return result
}
```

We see that `ns` gets dereferenced when the method calls `ns.chunkstore.WalkNodes`. A callsite like this could in theory be a way for a reference to leak: if `chunkstore` is a value type but `WalkNodes` has a pointer receiver, then the call will implicitly take the address of `chunkstore`, which points within our newly copied struct. But looking closer, that's not the case here, because `chunkstore` is an interface. In Go, an interface value is essentially a smart pointer. Thus, the value that will get passed to the `WalkNodes` function will either be a pointer to the value implementing the interface, or a copy of that value. In both cases, the receiver does not point into `ns`. `ns` does not escape. It does not outlive the call to `GetBytes`. It doesn't need to be stored on the heap.

But the compiler stores it on the heap anyway. Why?

Earlier I said the compiler stores local variables on the heap when it "deduces that the pointer to `x` will outlive the function call". But it would be more accurate to say that the compiler chooses heap allocation when it "*can't* deduce that the pointer to `x` *won't* outlive the function call." This is because if the compiler can't determine whether the escape occurs, it needs to play it safe and assume that it will. Heap allocations are always safe, if slow.

The process by which the compiler attempts to prove that a variable *can't* outlive the current stack from is called **escape analysis.**

The official documentation talks about the compiler's escape analysis here. To quote:

> In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a *basic escape analysis* recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

The emphasis is mine: a basic escape analysis. So even though we know there won't be any references to the copied receiver after the method finishes, the escape analysis might not be sophisticated enough to figure this out.

The language does throw us one little bone: although we can't control where a value will be allocated, we can get the compiler to tell us where it's being allocated and why, by running `go build -gcflags "-m"`.

When I built Dolt with this additional flag, I found this in the output:

```
store/prolly/tree/node_store.go:93:7: parameter ns leaks to {heap} with derefs=1:
store/prolly/tree/node_store.go:93:7:   flow: {heap} = *ns:
store/prolly/tree/node_store.go:93:7:     from ns.chunkStore (dot of pointer) at store/prol
store/prolly/tree/node_store.go:93:7:     from ns.chunkStore.WalkNodes(ctx, ref) (call para
store/prolly/tree/node_store.go:93:7: leaking param content: ns
```

The compiler sees that `ns.chunkStore` is passed as a parameter to a function and considers that to mean that the value escapes. Thus, the compiler can't definitively conclude that the receiver is safe to store on the stack.

It's possible that this is a compiler bug. It's also possible that there's some fringe case where the reference actually *can* escape via that method call, and the compiler doesn't have enough context to rule it out. Compilers are complicated, and I can easily see it going either way. If someone reading this has more insight into why Golang thinks that value escapes here, join our **Discord** and drop me a line and I'll give you a shoutout in my next blog.

The Takeaway

The fix was simple enough: use a pointer receiver instead of a value receiver to prevent the unnecessary copy. But uncovering the *why* of this regression revealed surprisingly complex behavior.

If we learn anything from this investigation, it would be this:

- Understanding *why* compilers make decisions with regard to memory allocation is important for writing performant code. Tools like `-gcflags "-m"` that provide insight into compiler decisions are incredibly useful for understanding and optimizing performance.
- Storing a value on the heap is always safe in a garbage-collected language, but comes at a performance cost: both during the initial allocation and again during garbage collection. Since it's always safe, Golang can always choose to allocate a value on the heap, and will unless it can prove that a stack allocation is also safe.
- Prefer pointer receivers in order to avoid unnecessary copies, because those copies can easily result in extra heap allocations.

As always, if you think I got something wrong, or think I'm being too critical of Golang, or think I'm not being critical *enough*, feel free to join our **Discord** and share your thoughts! Our Golang blogs tend to generate a lot of passionate discussion.

---

1. Technically the program is undefined behavior, but will almost certainly print "2".↩

< Blog

Dolt is open source

JOIN THE DATA EVOLUTION

Get started with Dolt

Install Dolt

Create an account ↗

Or join our mailing list to get product updates.

your email address

Submit