Stefano Scafiti

Lead Software Engineer at Codenotary

Exploring Go's Prowess in Immersive Database Development: immudb as a Case Study

AGENDA

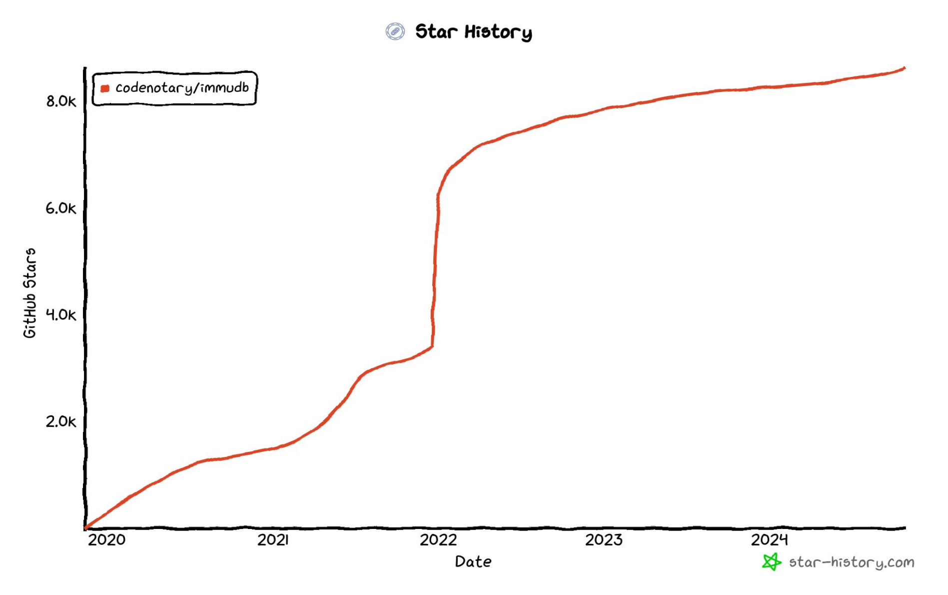
1. immudb: The Immutable Database
2. Why Golang?
3. The Garbage Collector
4. Conclusion

immudb:
The Immutable Database

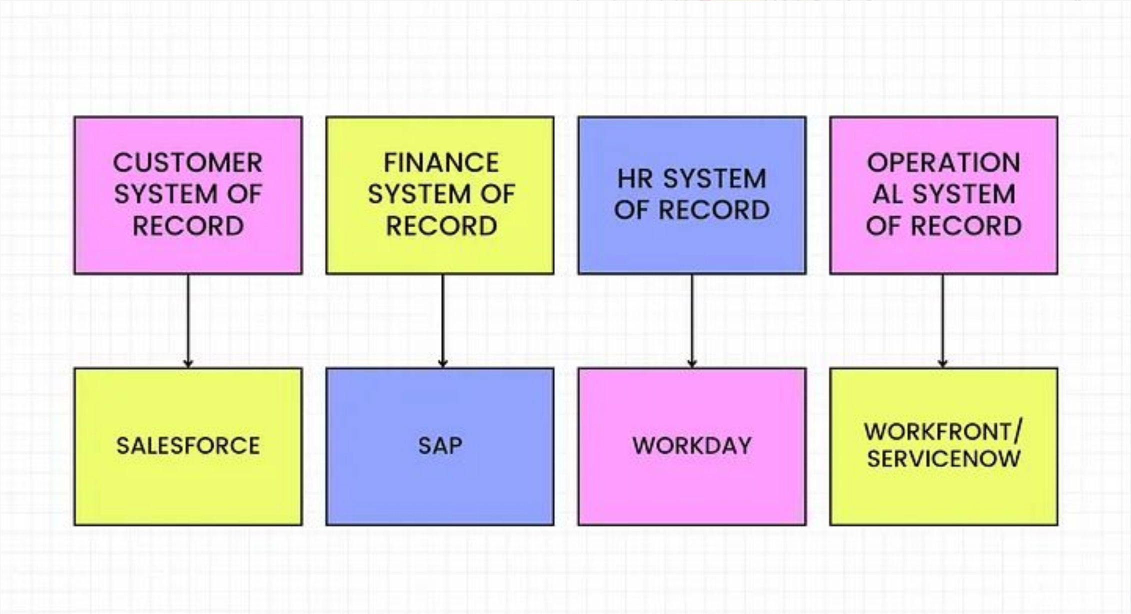
- **immudb** is a database with built-in cryptographic proof and verification
- It operates as a key-value store, SQL or document oriented database.
- Engineered for scalability, immudb can handle millions of transactions per second.

Project History

- Started in **2020**
- **4785** commits, **~70** contributors
- **47** releases (latest version is **v1.9.5**)
- **~146k** lines of code (Golang only)

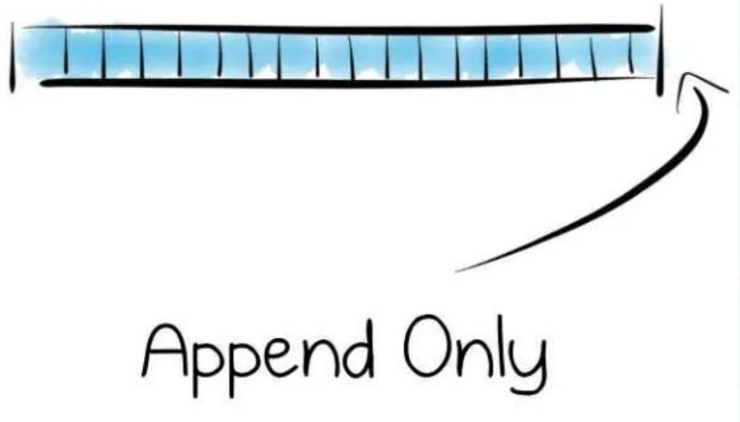


Applications of an immutable database

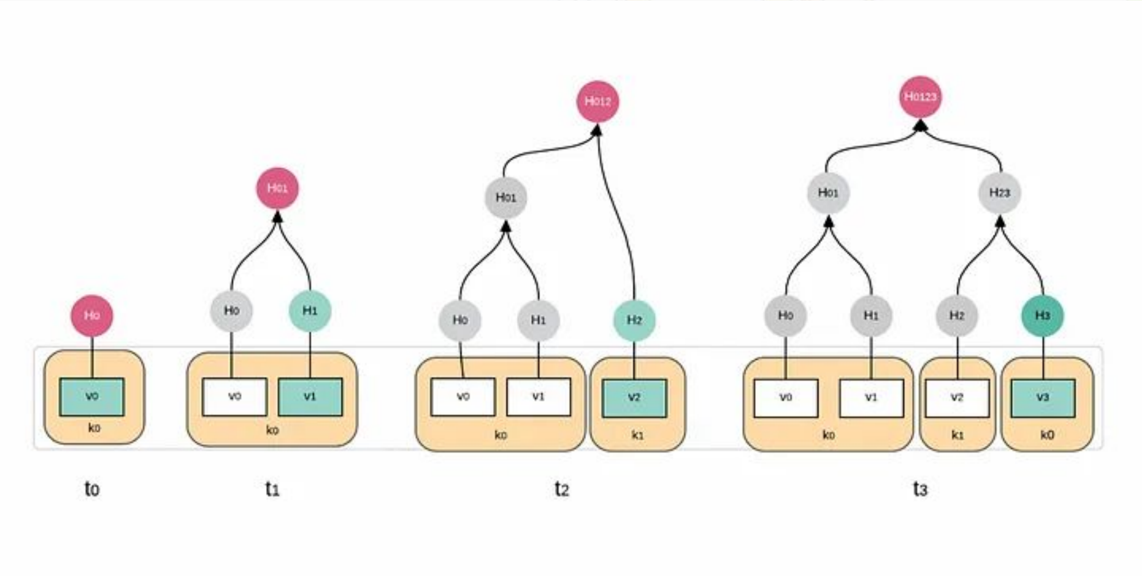


Immutable append only logs

- An append-only log file is a data structure where new entries are added sequentially at the end, and existing entries cannot be modified or deleted.

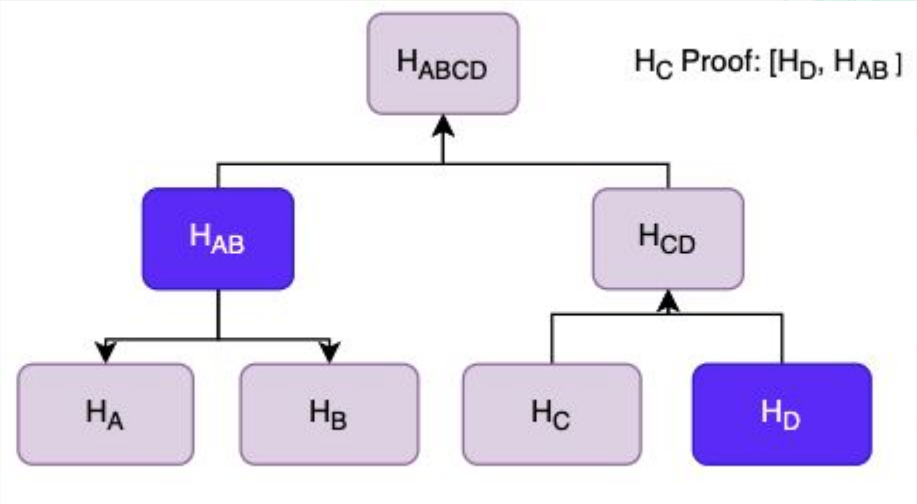


immudb: the Merkle Tree



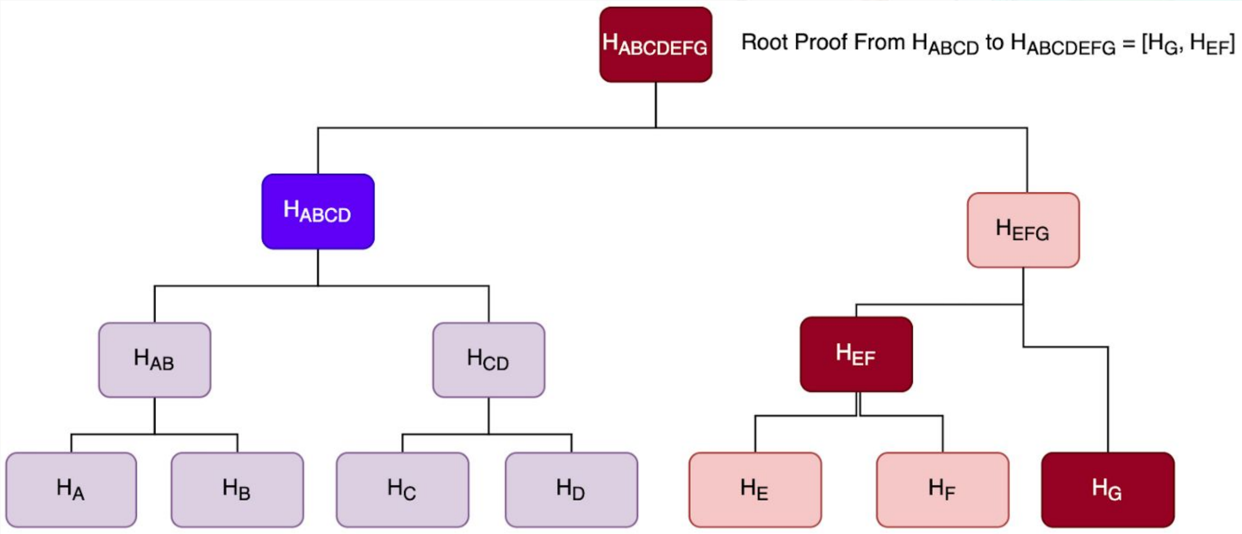
The inclusion proof

- It proofs a given leaf's membership in the tree



The consistency proof

- It proofs that a merkle tree (root hash) is a superset of a previous merkle tree



How immutability is proved to clients?

1. **Root Hash Tracking**: clients retain the most recent known Merkle tree root hash *H*;

2. **Proofs for Integrity**: when fetching or setting a new key-value entry *E*, clients can optionally request:

    a. a *consistency proof* between *H* and the latest database state *H'*;

    b. an *inclusion proof* confirming that *E* belongs to the Merkle tree represented by the current root hash *H'*.

Why Golang?

- Extensive standard library, including cryptographic support
- Native concurrency features (goroutines, channels and more)
- Successful examples of adoption in large database and storage projects (CockroachDB, BadgerDB)
- Tools for reducing GC load: sync.Pool, unsafe package, etc.

Rich Standard Library

```
func (t *HTree) BuildWith(digests [][sha256.Size]byte) error {
        ...
        b := [1 + 2*sha256.Size]byte{NodePrefix}
        for w > 1 {
                wn := 0
                for i := 0; i+1 < w; i += 2 {
                        copy(b[1:], t.levels[l][i][:])
                        copy(b[1+sha256.Size:], t.levels[l][i+1][:])
                        t.levels[l+1][wn] = sha256.Sum256(b[:])
                        wn++
                }
                if w%2 == 1 {
                        t.levels[l+1][wn] = t.levels[l][w-1]
                        wn++
                }
                l++
                w = wn
        }
        ...
}
```

Goroutines

```
resCh := make(chan appendableResult)
go func() {
      offsets, err := s.appendValuesToAnyVLog()
      if err != nil {
            resCh <- appendableResult{nil, err}
      }
      resCh <- appendableResult{offsets, nil}
}()

err = tx.BuildHashTree()
...
valuesRes := <-resCh
...
```

Channels

- Channels enable safe communication between goroutines, avoiding explicit locks and simplifying concurrent code.
- For simple local data sharing, channels might add unnecessary complexity, as opposed to direct data passing or using mutexes.
- Circular waiting patterns, where goroutines are blocked waiting on each other, can lead to deadlocks.

Channels in the WatchersHub component

```
type WatchersHub struct {
      ...
      value uint64
      ...
}

func (s *WatchersHub) WaitFor(ctx context.Context, v uint64) error {
      ...
}

func (s *WatchersHub) Advance(v uint64) error {
      ...
}
```

WatchersHub usage

```go
func (s *Store) commit(ctx context.Context, …) error {
    hdr, err := s.precommit(ctx, …)
    ...

    // note: durability is only ensured when sync mode is enabled
    err := s.commitWHub.WaitFor(ctx, hdr.TxID)
    ...
}
```

WatchersHub: implementation

```go
type WatchersHub struct {
	wPoints map[uint64]waitPoint
	value uint64
	...
	mtx sync.Mutex
}
```

```go
type waitPoint struct {
	v uint64
	ch chan struct{}
	count int
}
```

WatchersHub: the WaitFor method

```go
func (w *WatchersHub) WaitFor(ctx context.Context, v uint64) error {
	if w.value >= v {
		return nil
	}
	...
	wp, waiting := w.wPoints[v]
	if !waiting {
		wp = waitingPoint{v: v, ch: make(chan struct{})}
	}
	...
	cancelled := false
	select {
		case <-wp.ch:
			break
		case <-ctx.Done():
			cancelled = true
	}
	...
}
```

WatchersHub: the Advance method

```
func (w *WatchersHub) Advance(v uint64) error {
      ...
      for i := s.value+1; i <= v; i++ {
            wp, waiting := w.wPoints[i]
            if waiting {
                  close(wp.ch)
                  ...
                  delete(w.wPoints, i)
            }
      }
      w.value = v
      ...
}
```

The Garbage Collector

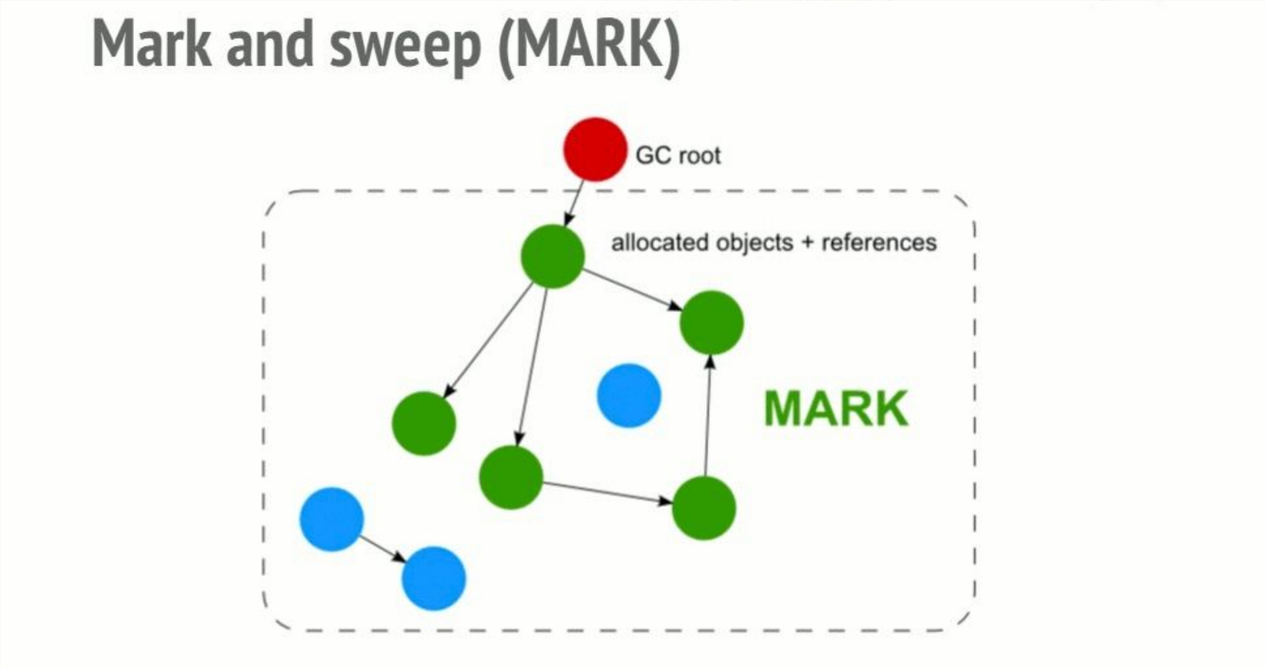
- Use sync.Pool to reduce the number of total allocations
- Prefer value types over pointers when feasible
- Eliminate unnecessary copying

The mark and sweep algorithm

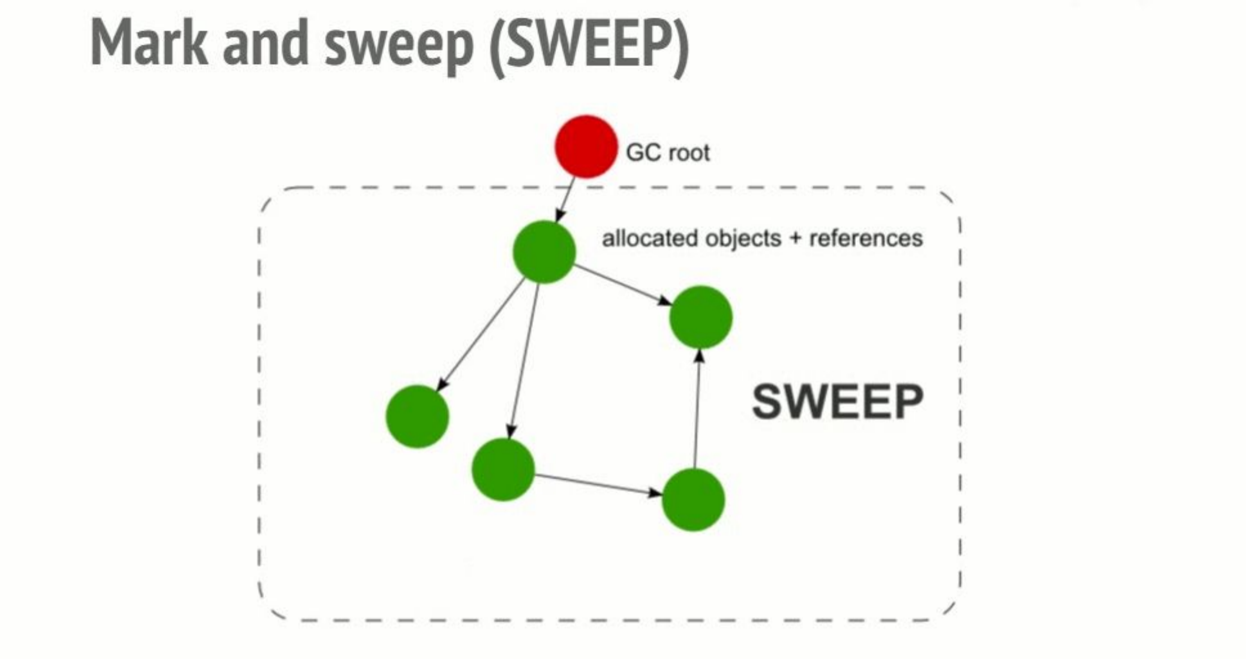
- It walks the program’s **object graph** to identify objects that are in use by the program

- It uses a two phase technique, called **mark**-and-**sweep**.

The GC Cycle: mark phase



The GC Cycle: sweep phase



The GC Cycle

- GC cycles pause time increases as we allocate more and more objects

- It is not affected by the total amount of bytes allocated

Reducing the number of allocations with **sync.Pool**

```
type Row struct {
    ...
}

var rowsPool = sync.Pool {
     New: func() any {
          return &sql.Row{...}
     }
}

s := rowsPool.Get().(*sql.Row)
s.reset()

... // do something with r

rowsPool.Put(s) // release r back to the pool
```

Prefer values to pointers

```go
type MyStruct struct {
    Field1 *uint64
    Field2 *Object
    ...
}
```

```go
type MyStruct struct {
    Field1 uint64
    Field2 Object
    ...
}
```

Cache

```go
type cacheShard struct {
	offsets map[string]uint32
	buf []byte
	...
}

func (c *cacheShard) Put(key, value []byte) error {
	...
}

func (c *cacheShard) Get(key []byte) ([]byte, error) {
	...
}
```

unsafe operations

```go
func BytesToString(buf []byte) error {
	return unsafe.String(unsafe.SliceData(buf), len(buf))
}
```

Conclusions

- Golang simplifies complex tasks, from cryptography to concurrency, with minimal dependencies.
- While GC can introduce delays, there are several options to minimize its impact.
- Projects like immudb showcase its effectiveness in real-world, performance critical applications.

STEFANO SCAFITI

stefano@codenotary.com

https://codenotary.com

https://immudb.io

https://github.com/codenotary/immudb

Thank You
For Your Time!