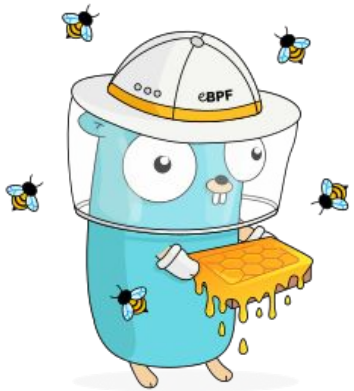
StateDB

A database for your program state

Whoami

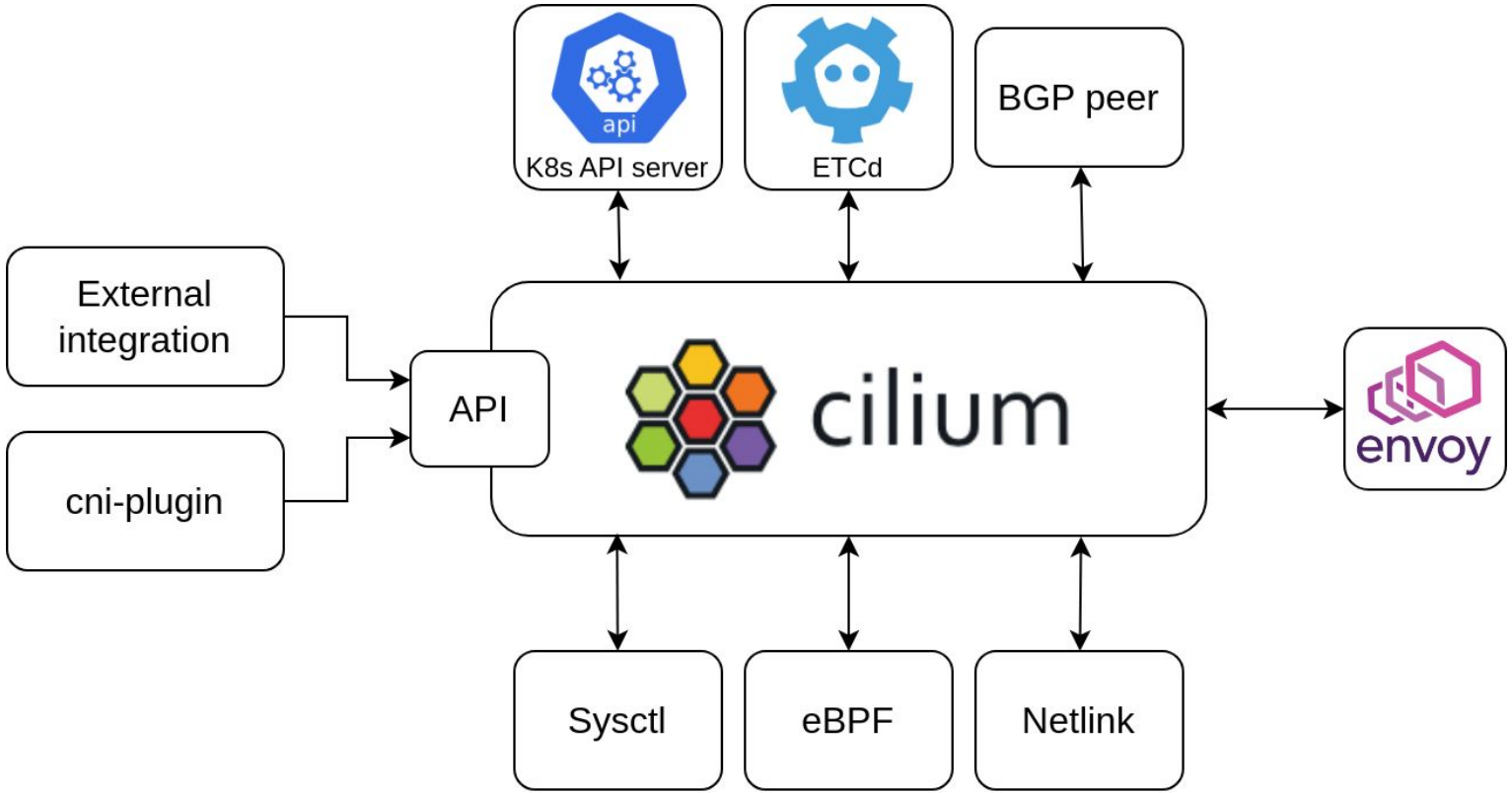
- Isovalent @ Cisco
- cilium/cilium contributor
- cilium/ebpf reviewer



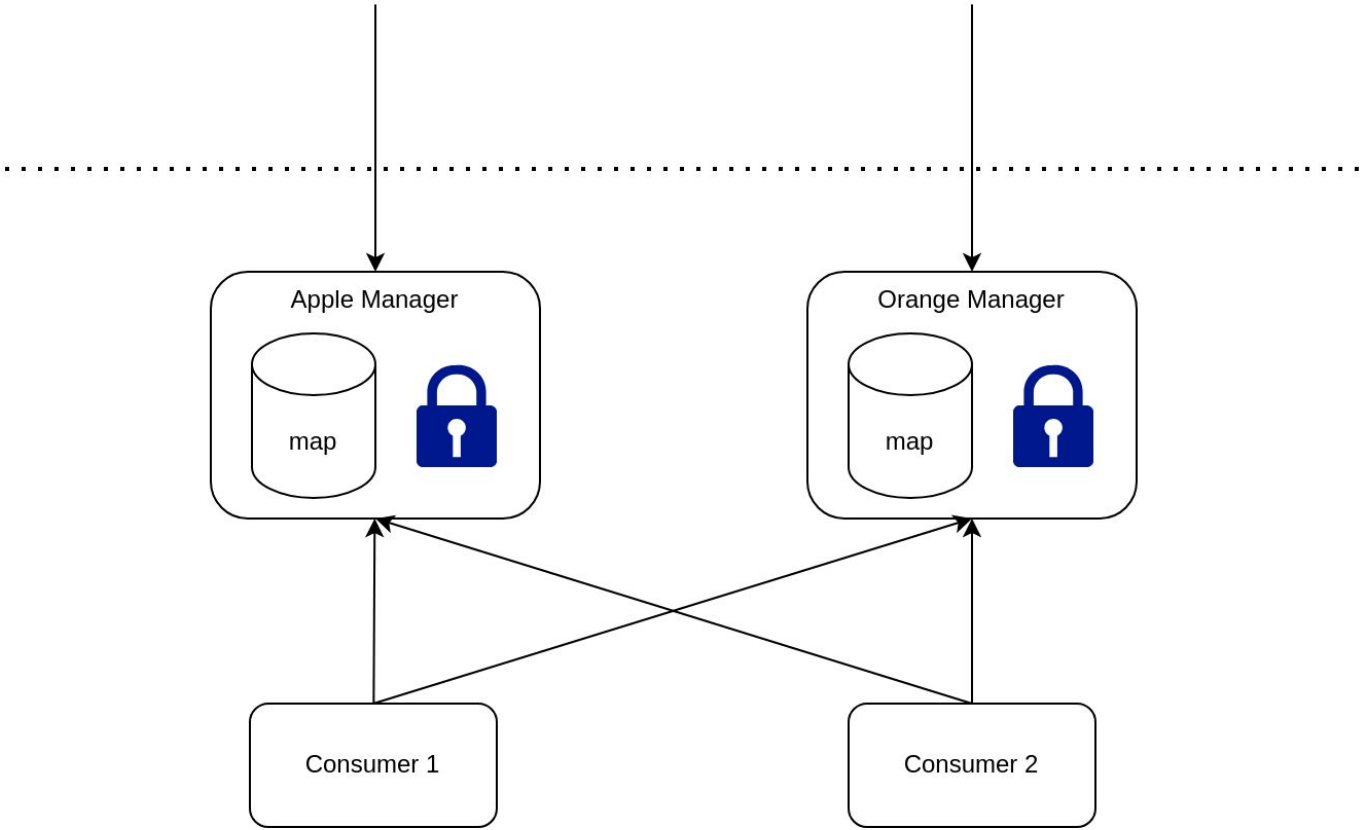
CISCO + ISOVALENT

cilium

Cilium


api
K8s API server
ETCd
BGP peer
External integration
cni-plugin
API
cilium
envoy
Sysctl
eBPF
Netlink

The naïve approach



The problem

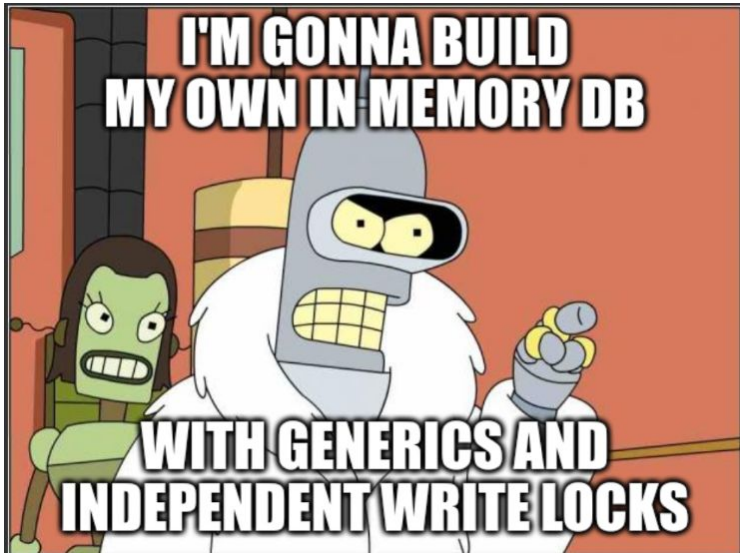
- Deadlocks (often when accessing multiple “stores”)
- Data duplication (and thus high memory utilization)
- API variation

Inspiration

- Hashicorp Nomad
- hashicorp/go-memdb
- Close, but not exactly it
  - DB wide write lock
  - Lots of interface{} usage
  - A number of missing features

So StateDB was born

- gomem-db with ~~blackjack and hookers~~ generics and independent write locks
- Also
  - Table initializers
  - Table change iteration
  - Go 1.23 iterators
  - Metrics
  - Optimistic concurrency
  - Some other features….



So StateDB is…

- In-memory
  - No persistence to disk, rebuild every time the process restarts
  - Complex data types can be stored (maps, channels, pointers)
- MVCC (Multi-Version Concurrency Control)
  - Data and indexes are versioned, different goroutines can view different versions
- Cross table write locks and transactions
  - Lock A and B at the same time. Do not lock C
  - Multiple changes are applied together or not at all
- Multiple indexes
  - Always 1 unique primary index, zero or many secondary indexes
- Watch channels
  - Watch for fine grained changes to specific parts of the db

How to use it

```
type Apple struct {
    AppleID
    Grade AppleGrade
}

type AppleID struct {
    Orchard uint32
    Tree    uint32
    Number  uint32
}

var AppleTable = func() statedb.RWTable[Apple] {
    tbl, err := statedb.NewTable("apple", AppleIDIndex, AppleGradeIndex)
    if err != nil {
        panic(err)
    }
    return tbl
}()
```

How to use it

```
var AppleIDIndex = statedb.Index[Apple, AppleID]{
    Name: "ID",
    FromObject: func(a Apple) index.KeySet {
        return index.NewKeySet(a.AppleID.toKey())
    },
    FromKey: func(key AppleID) index.Key {
        return key.toKey()
    },
    Unique: true,
}
```

How to use it

```
var AppleGradeIndex = statedb.Index[Apple, AppleGrade]{
    Name: "Grade",
    FromObject: func(a Apple) index.KeySet {
        return index.NewKeySet(index.Uint32(uint32(a.Grade)))
    },
    FromKey: func(key AppleGrade) index.Key {
        return index.Uint32(uint32(key))
    },
    Unique: false,
}
```

How to use it

```
var DB = statedb.New()

func main() {
    if err := DB.RegisterTable(AppleTable); err != nil {
        panic(err)
    }

    if err := DB.Start(); err != nil {
        panic(err)
    }
    defer DB.Stop()

    //[...]
}
```

How to use it

```
func populateAppleTable() error {
    txn := DB.WriteTxn(AppleTable)
    defer txn.Commit()
    // (oldObj Apple, hadOld bool, err error)
    _, _, err := AppleTable.Insert(txn, Apple{
        AppleID: AppleID{
            Orchard: 1,
            Tree:    1,
            Number:  9,
        },
        Grade: AppleGradeB,
    })
    if err != nil {
        txn.Abort()
        return err
    }
    return nil
}
```

How to use it

```
rx := DB.ReadTxn()
// (obj Apple, rev uint64, found bool)
apple, _, found := AppleTable.Get(rx, AppleIDIndex.Query(AppleID{
    Orchard: 1,
    Tree:    1,
    Number:  9,
}))
if found {
    fmt.Printf("Apple %v", apple)
}
```

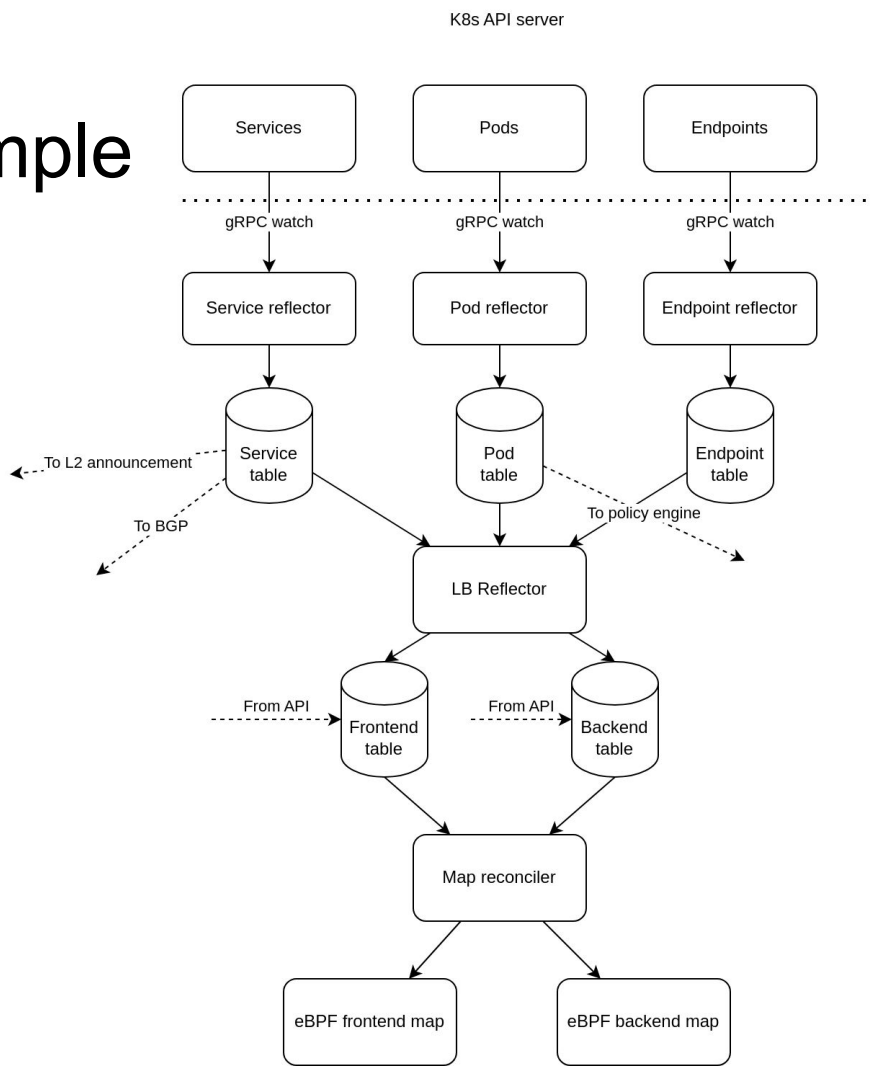
How to use it

```
for {
    rx := DB.ReadTxn()
    iter, watch := AppleTable.AllWatch(rx)

    for apple := range iter {
        fmt.Printf("Apple %v\n", apple)
    }

    <-watch
}
```

How to use it

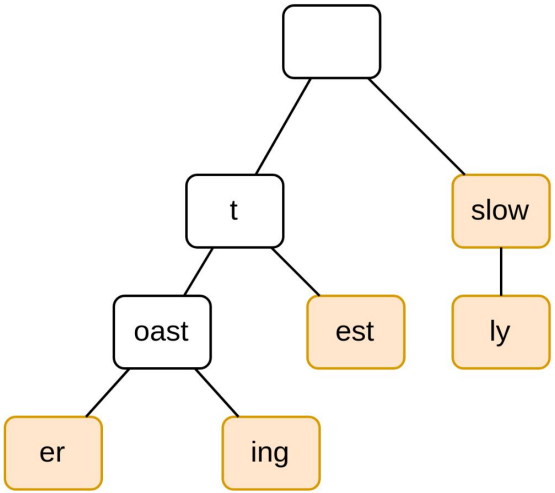
```
for {
    rx := DB.ReadTxn()
    iter, watch := AppleTable.PrefixWatch(rx, AppleIDIndex.Query(AppleID{
        Orchard: 3,
        Tree:    2,
    }))

    for apple := range iter {
        fmt.Printf("Apple %v\n", apple)
    }

    <-watch
}
```

Practical example

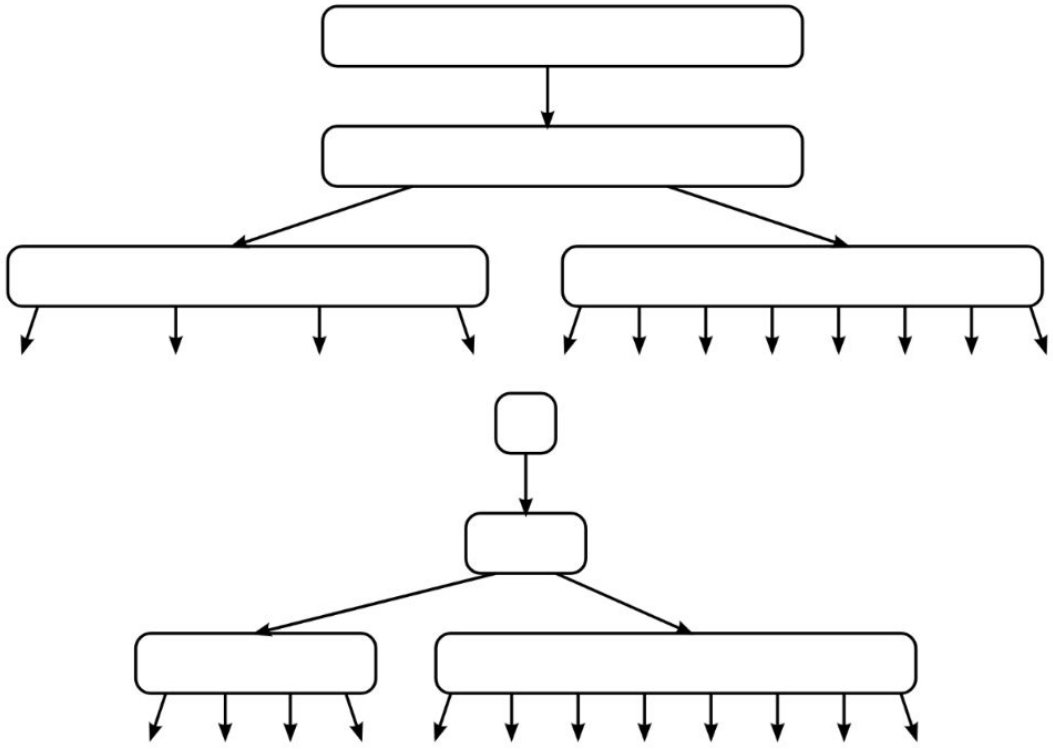


Internals - (Immutable) (Addaptive) Radix tree

- toaster
- toasting
- test
- slow
- slowly

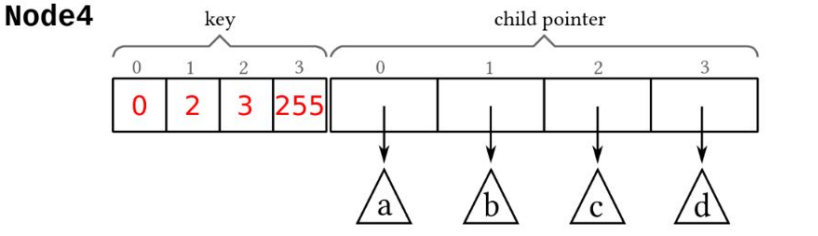


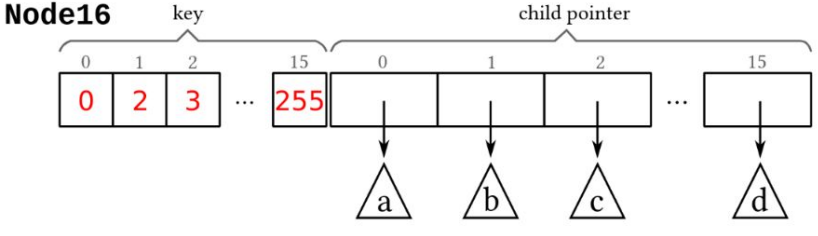
Internals - (Immutable) (Addaptive) Radix tree

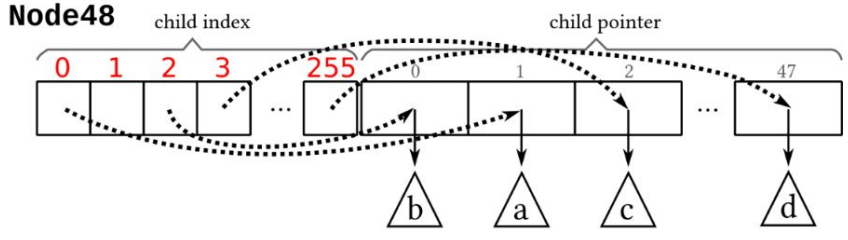


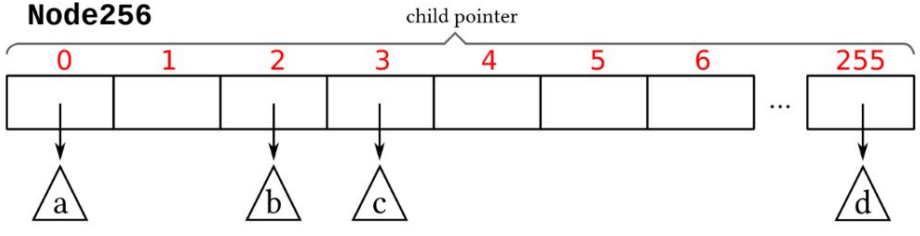
https://db.in.tum.de/~leis/papers/ART.pdf

Internals - (Immutable) (Addaptive) Radix tree





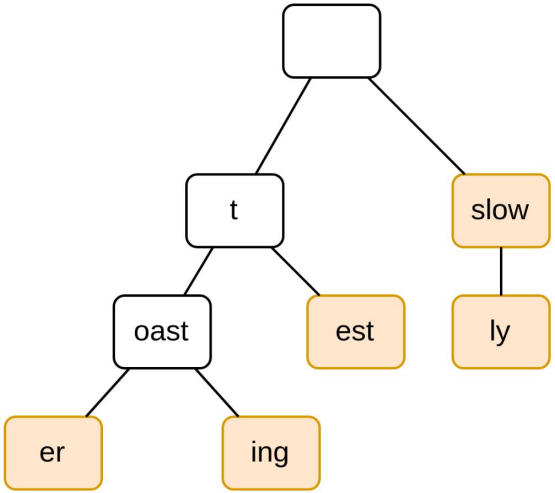




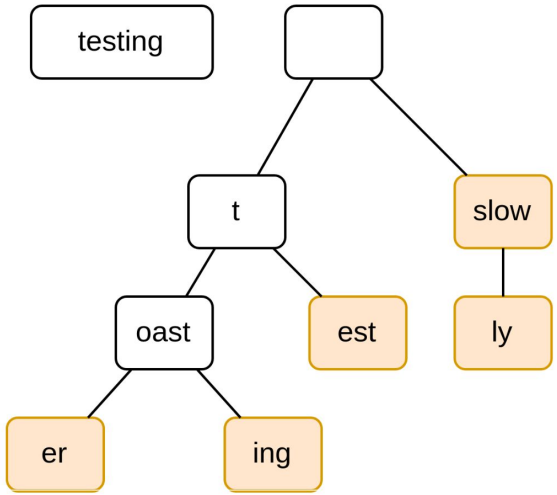
https://db.in.tum.de/~leis/papers/ART.pdf

Internals - (Immutable) (Addaptive) Radix tree

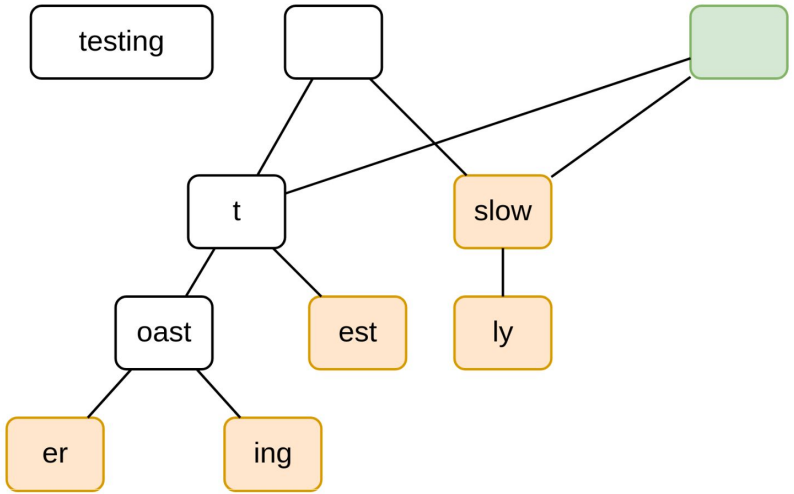
- toaster
- toasting
- test
- slow
- slowly



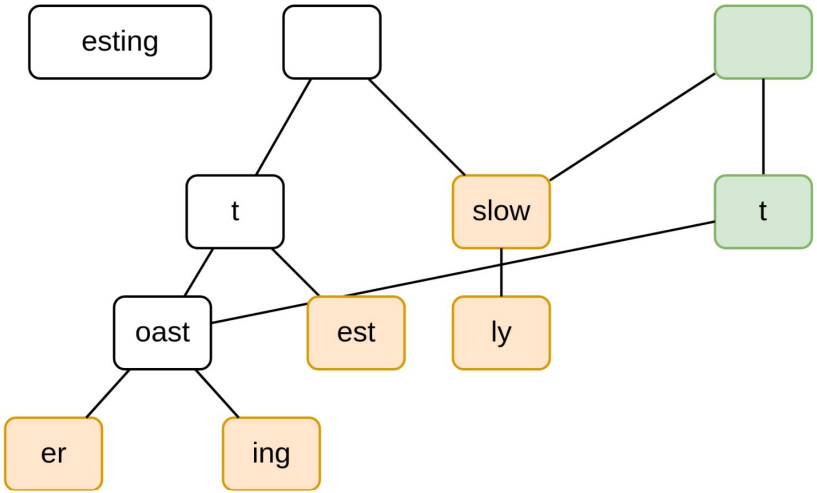
Internals - (Immutable) (Addaptive) Radix tree



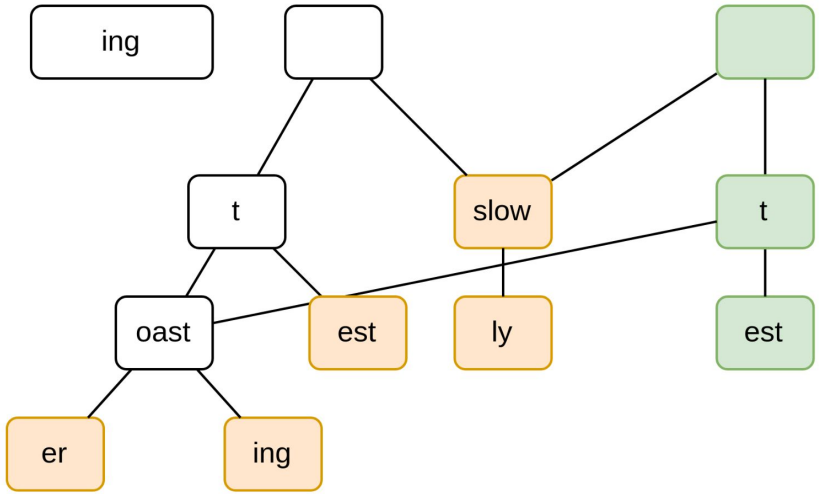
Internals - (Immutable) (Addaptive) Radix tree



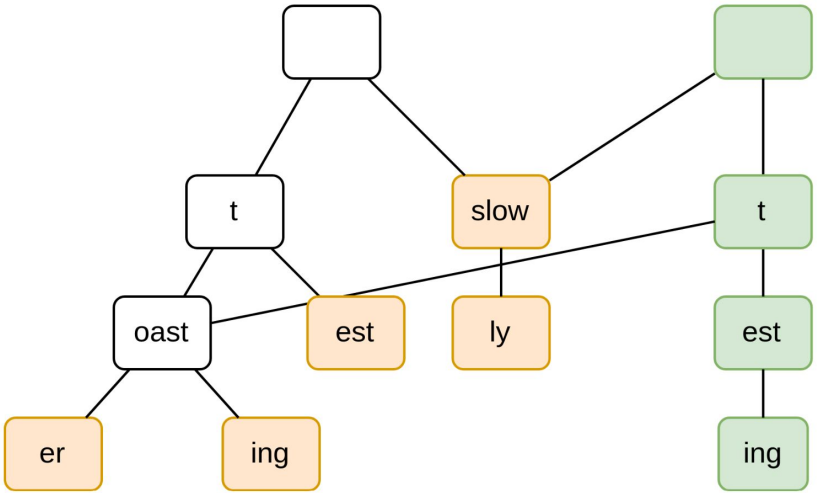
Internals - (Immutable) (Addaptive) Radix tree



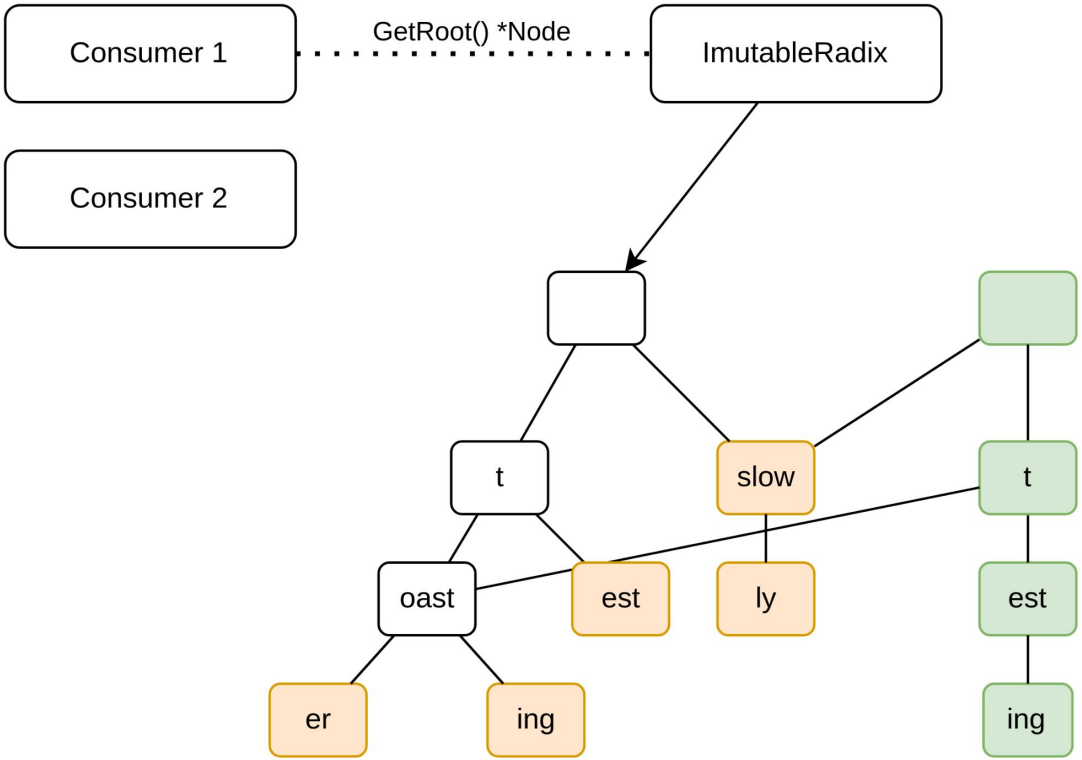
Internals - (Immutable) (Addaptive) Radix tree



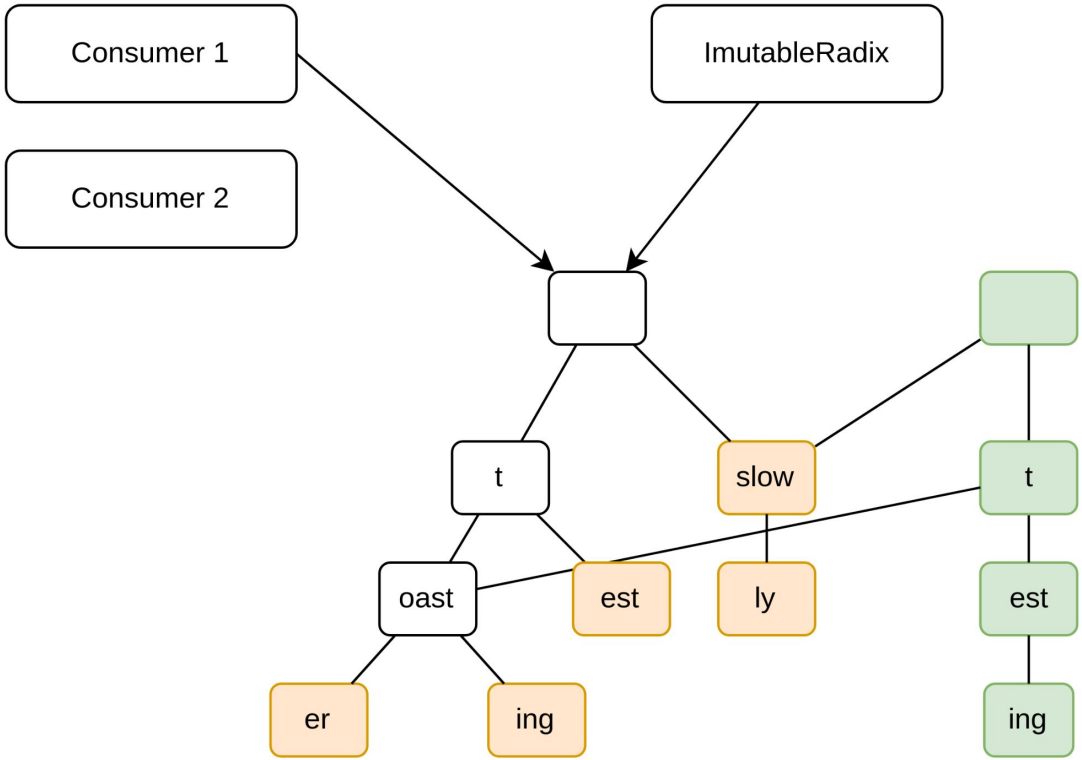
Internals - (Immutable) (Addaptive) Radix tree



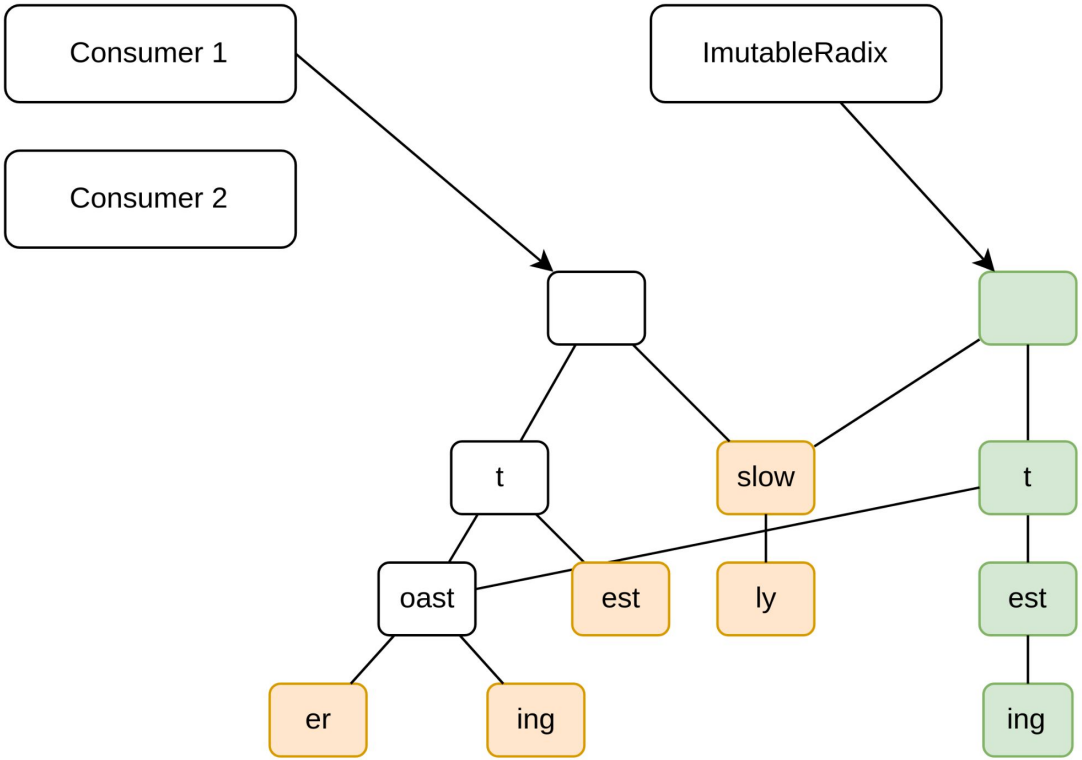
Internals - (Immutable) (Addaptive) Radix tree



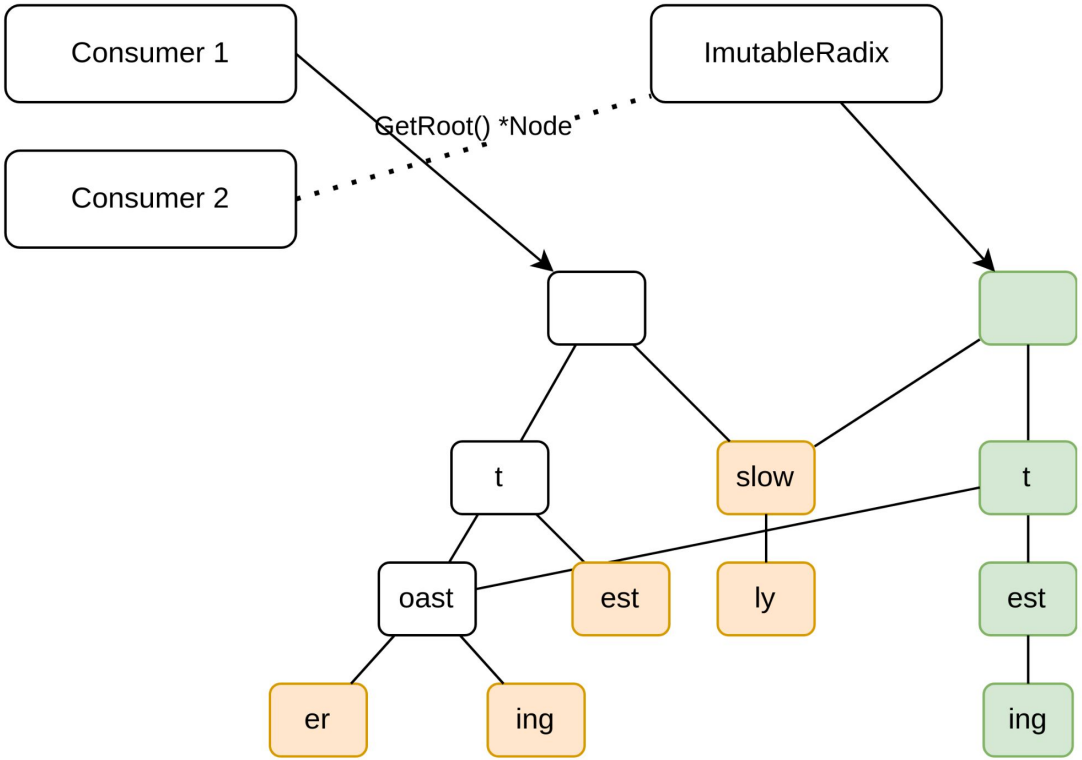
Internals - (Immutable) (Addaptive) Radix tree



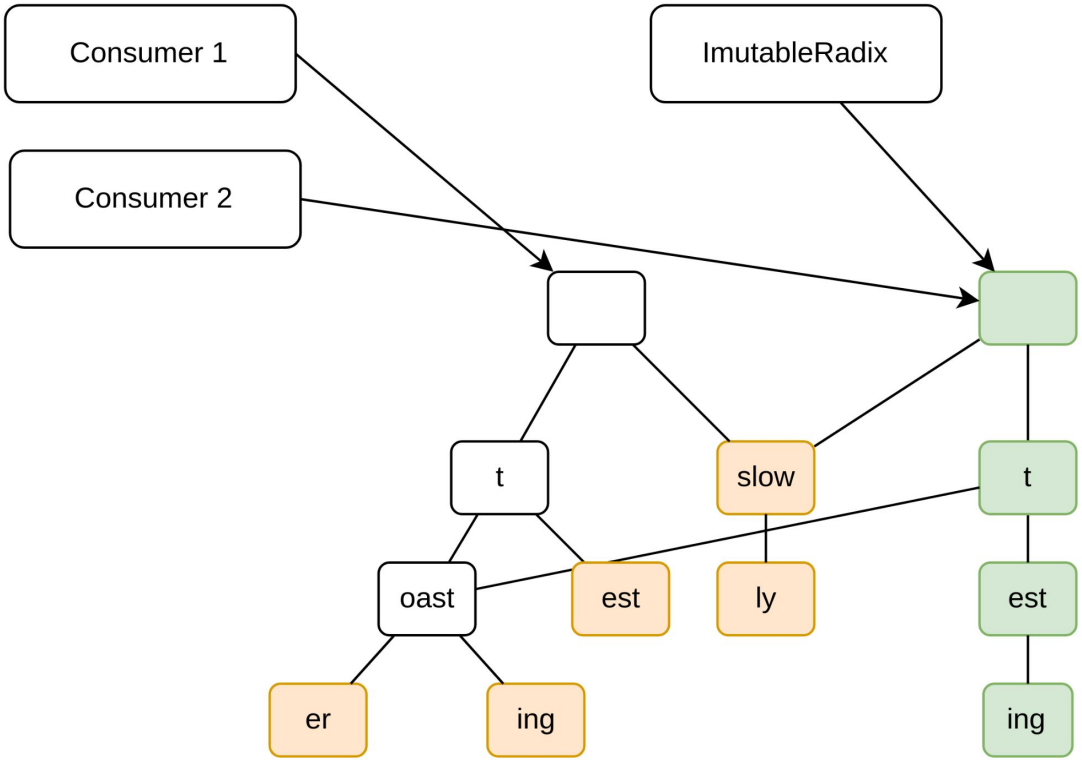
Internals - (Immutable) (Addaptive) Radix tree



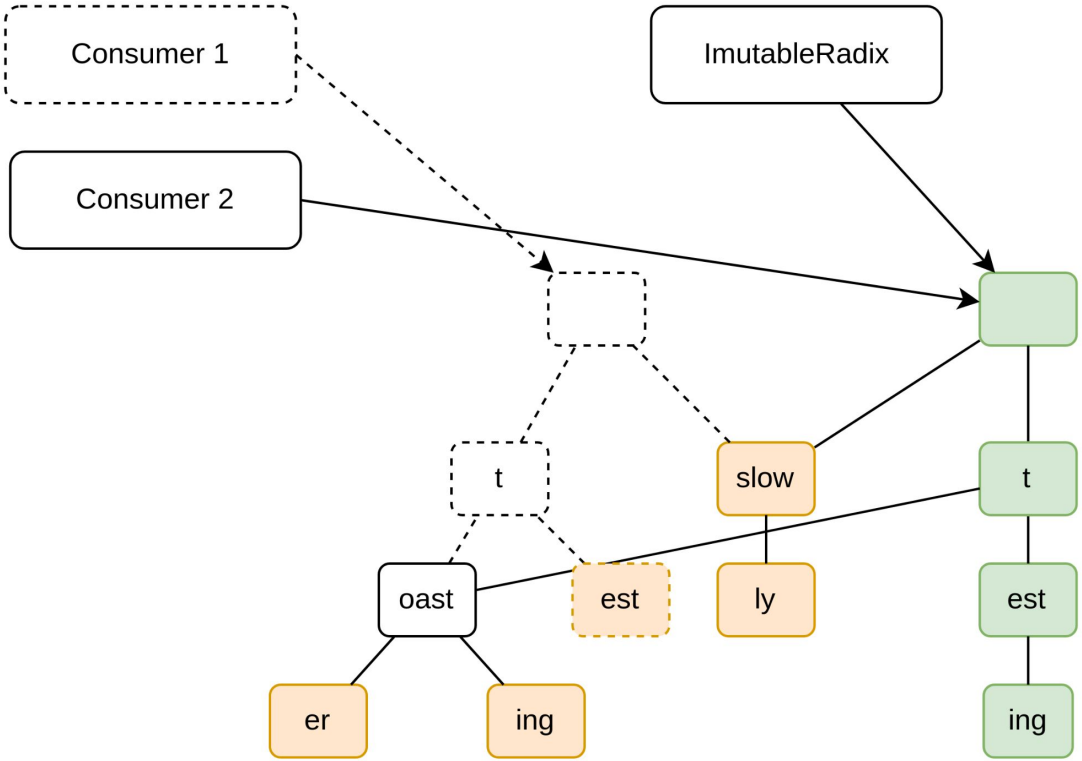
Internals - (Immutable) (Addaptive) Radix tree



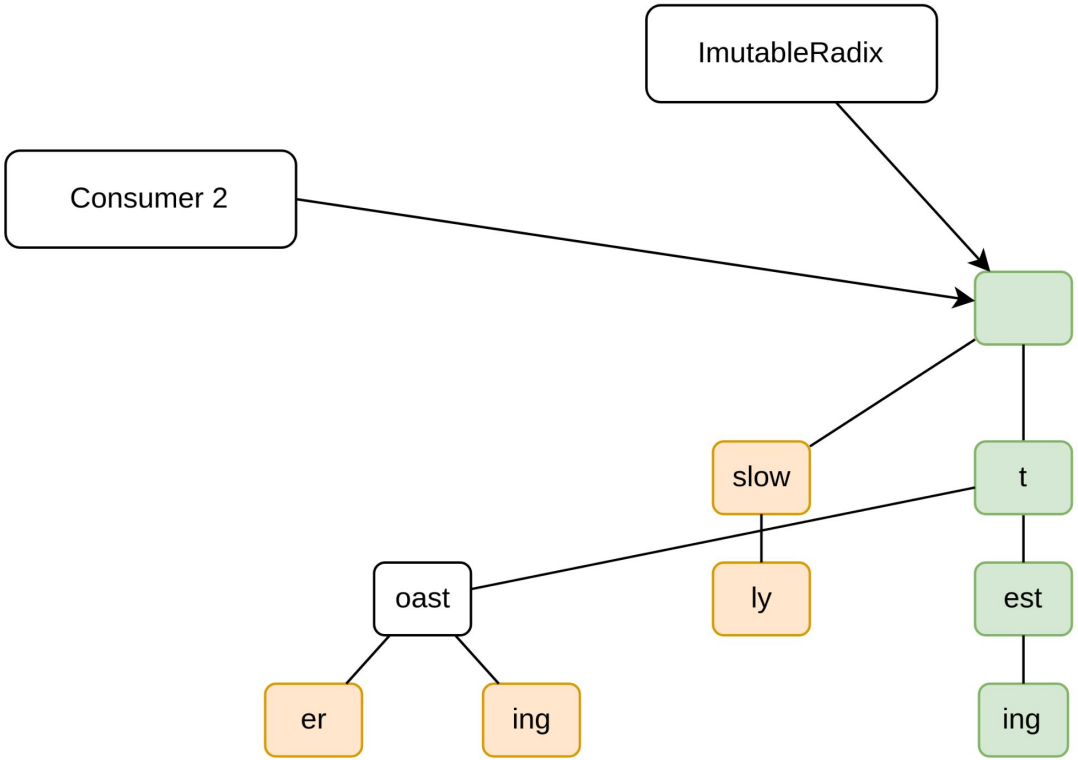
Internals - (Immutable) (Addaptive) Radix tree



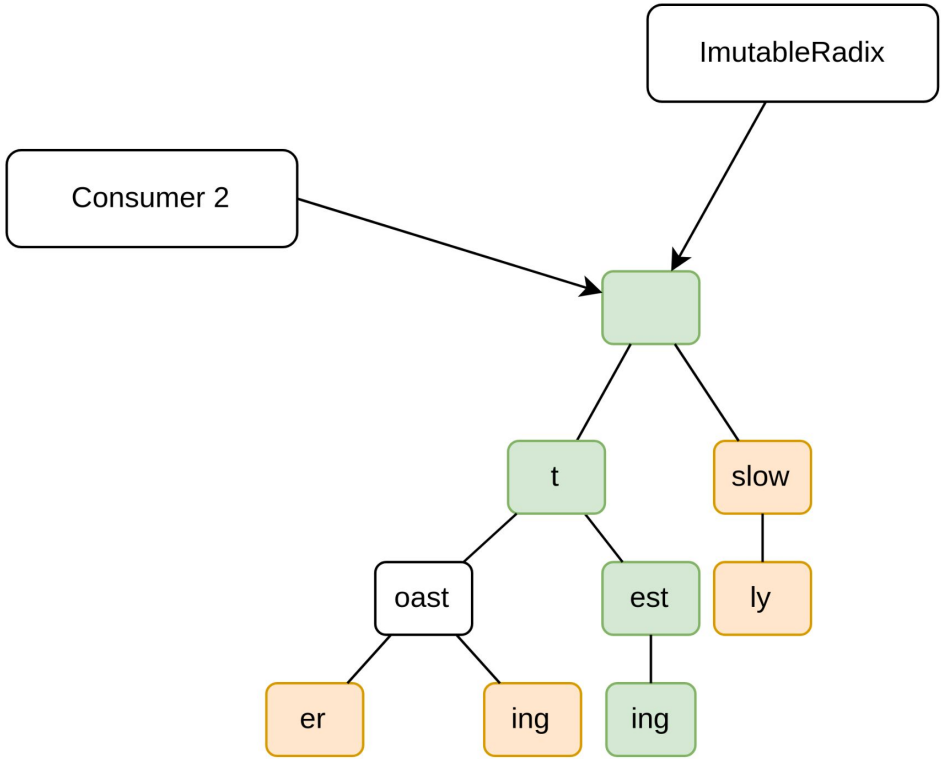
Internals - (Immutable) (Addaptive) Radix tree



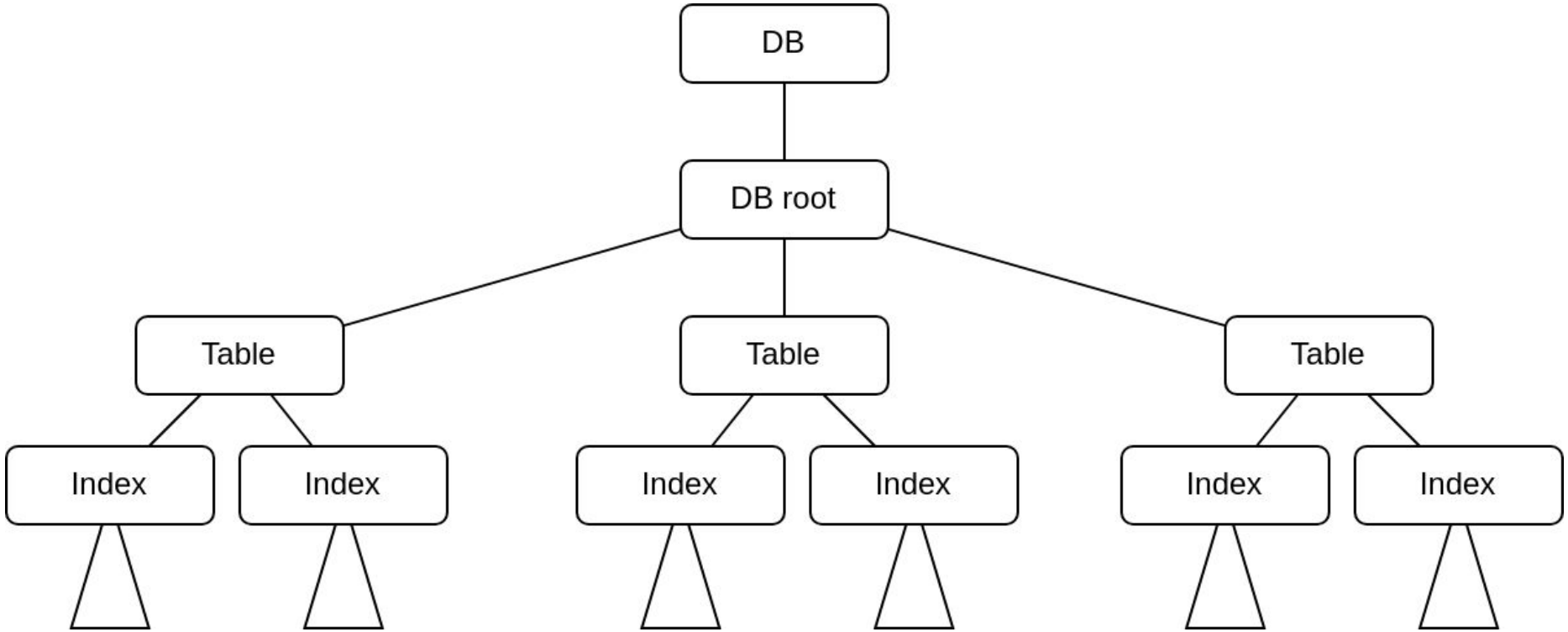
Internals - (Immutable) (Addaptive) Radix tree



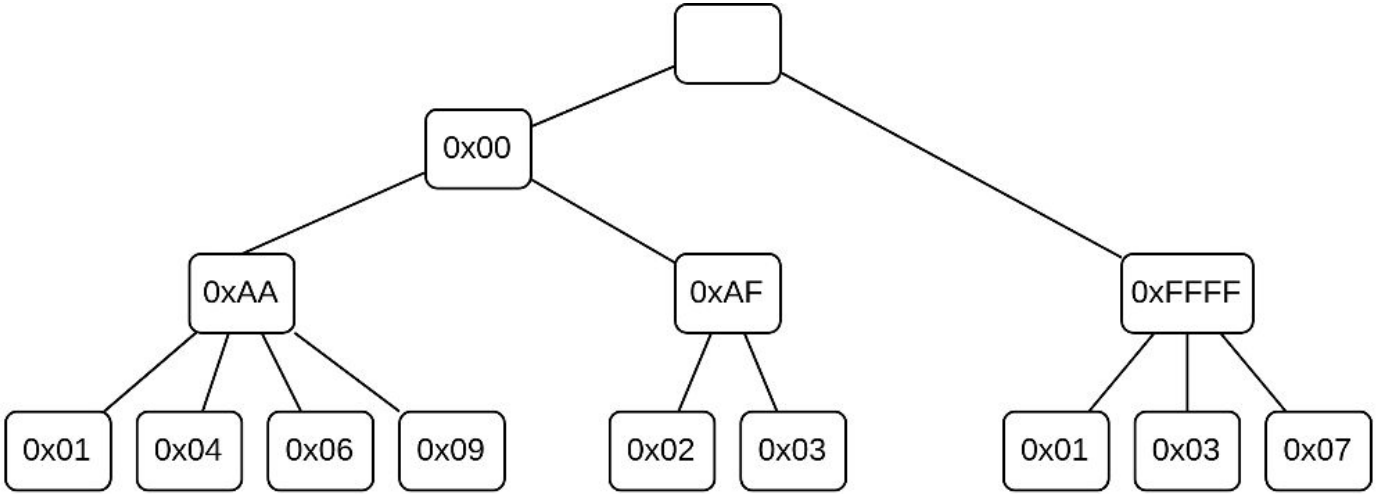
Internals - (Immutable) (Addaptive) Radix tree



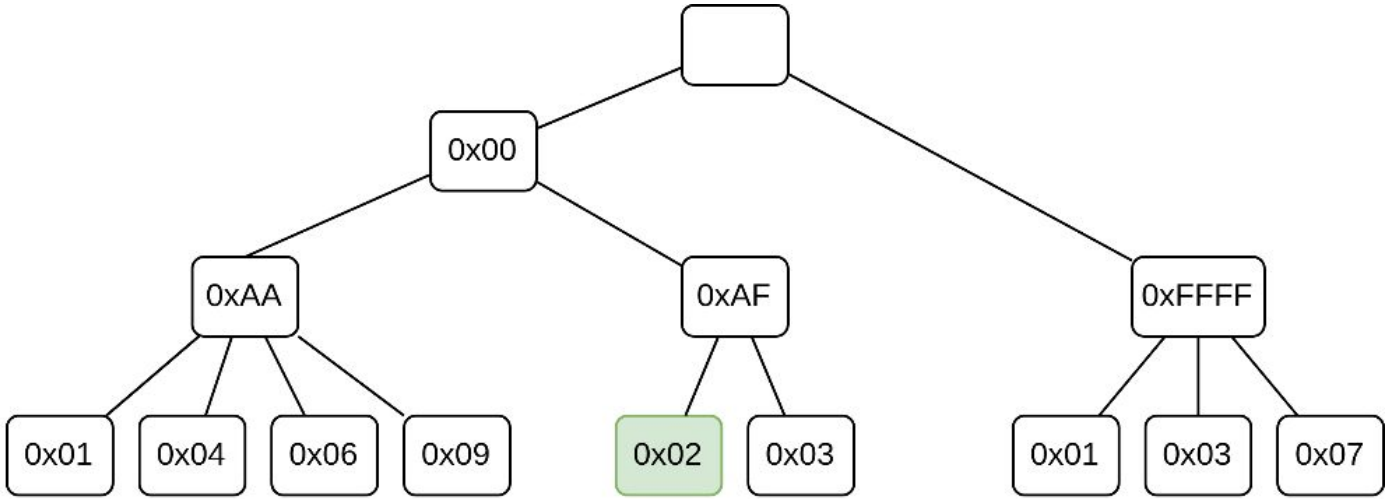
Internals - DB, Tables, Indexes



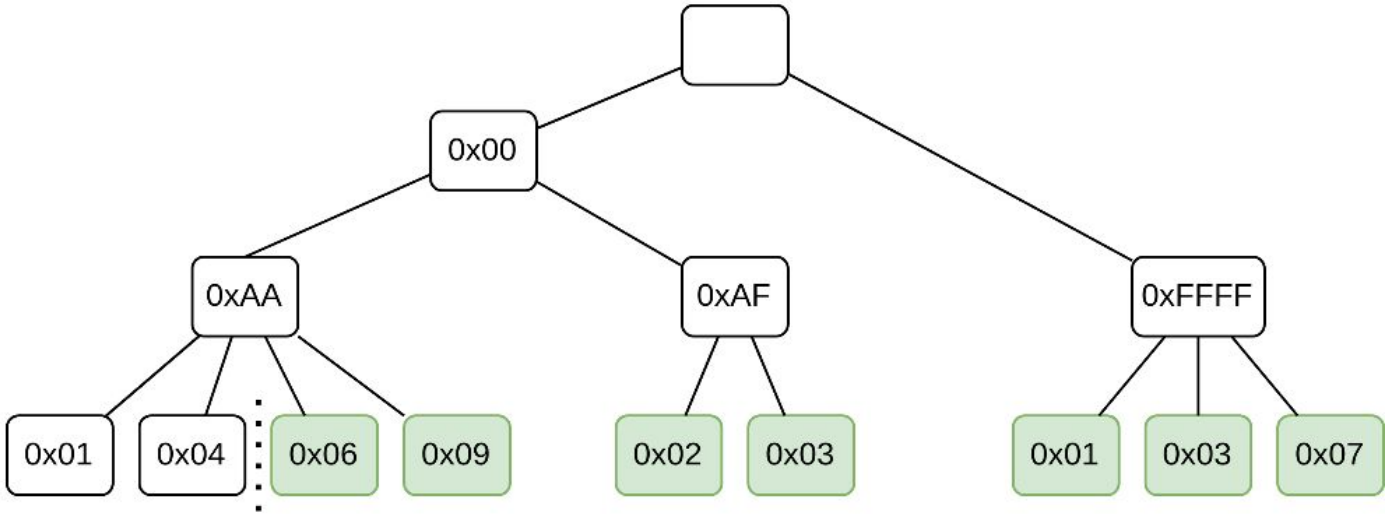
Internals - Queries and secondary indexes



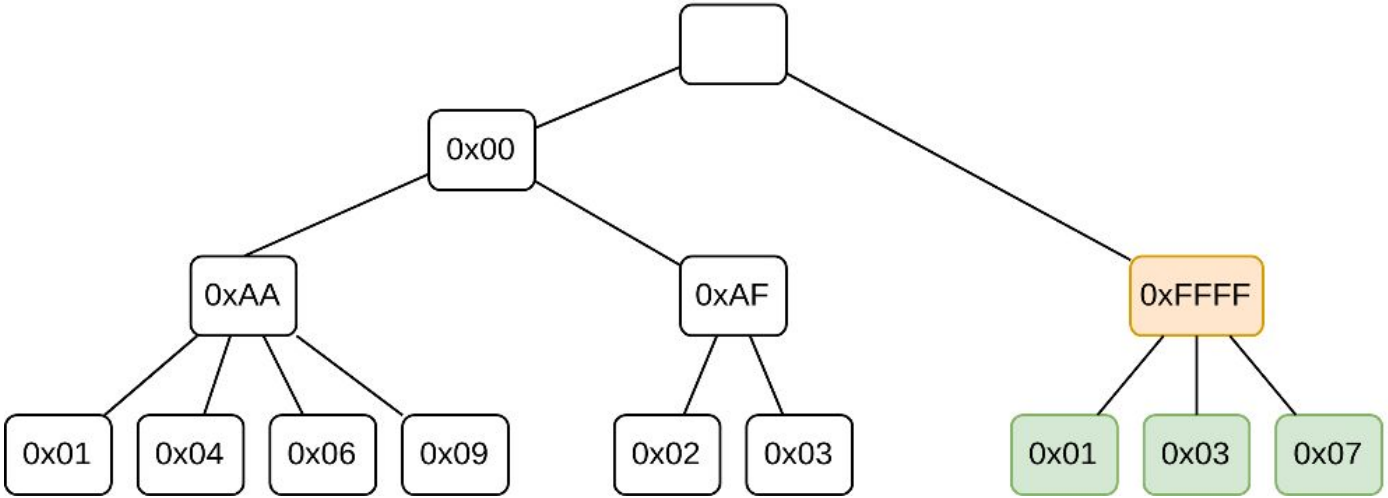
Internals - Queries and secondary indexes



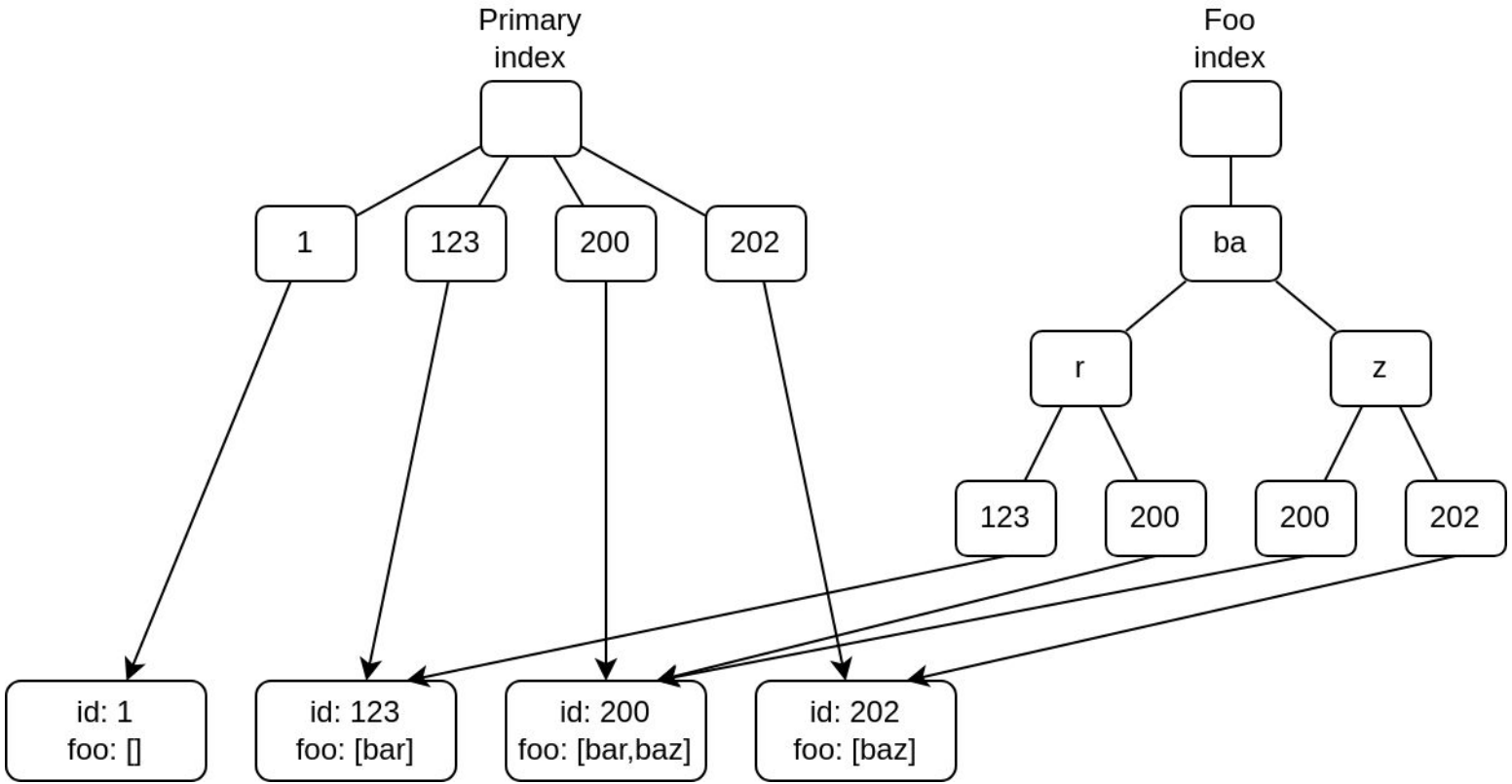
Internals - Queries and secondary indexes



Internals - Queries and secondary indexes

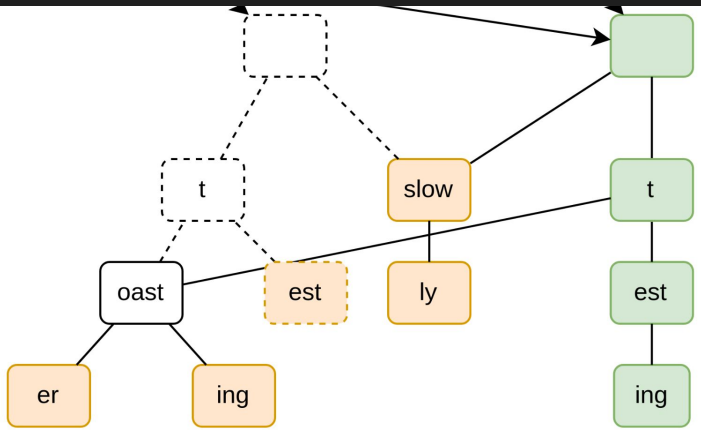


Internals - Queries and secondary indexes



Internals - Watches

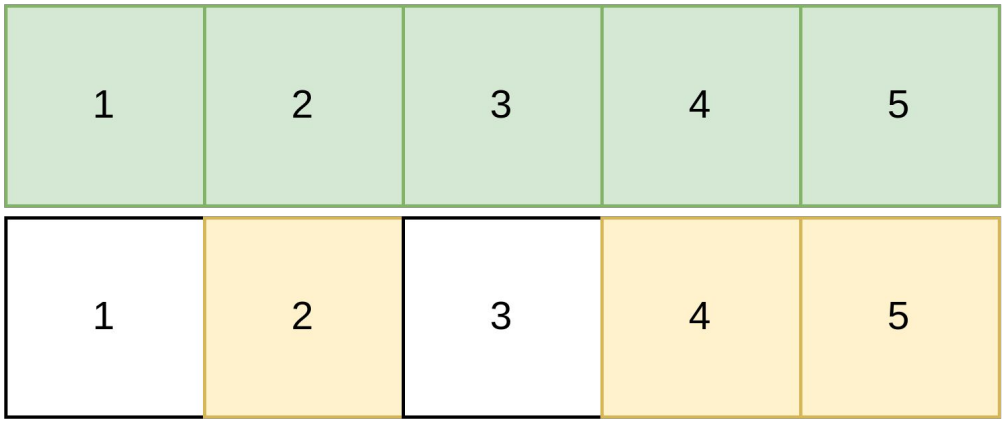
```
// header is the common header shared by all node kinds.
type header[T any] struct {
   flags  uint16         // kind(4b) | unused(3b) | size(9b)
   prefix []byte         // the compressed prefix, [0] is the key
   watch  chan struct{} // watch channel that is closed when this node mutates
}
```



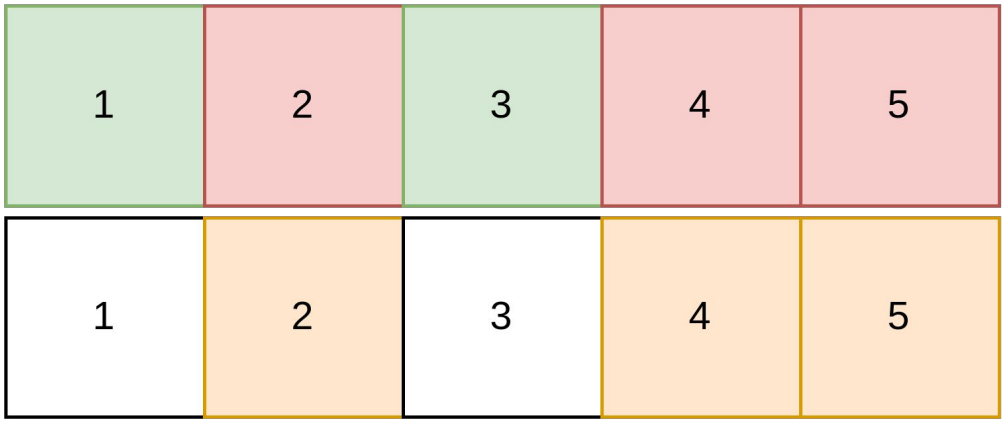
Internals - Transactions and locks

- Take locks for all tables using sortable mutexes
- Create new copies of changed bits
- On commit
  - Atomic pointer swap of root
  - closing watch channels
  - Metrics
- On abort
  - Release resources, and let the GC clean up the uncommitted copies
- Release the locks

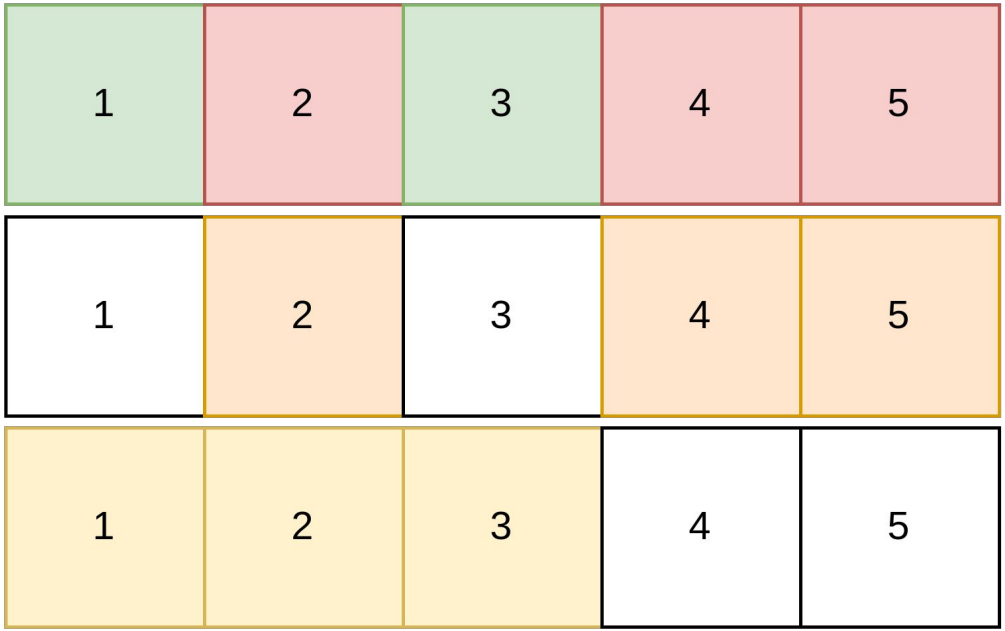
Internals - Sortable mutexes



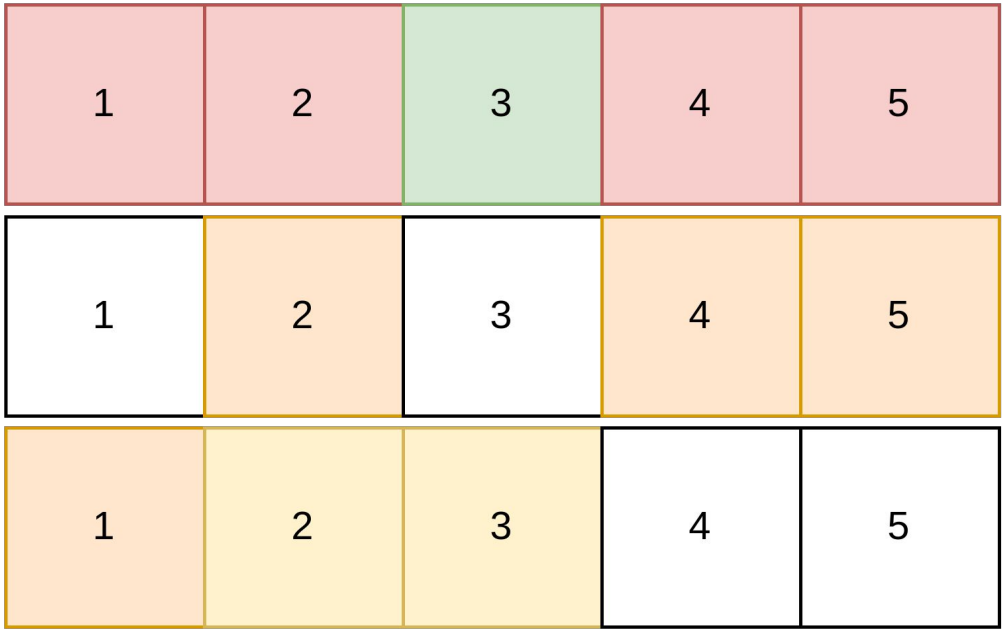
Internals - Sortable mutexes



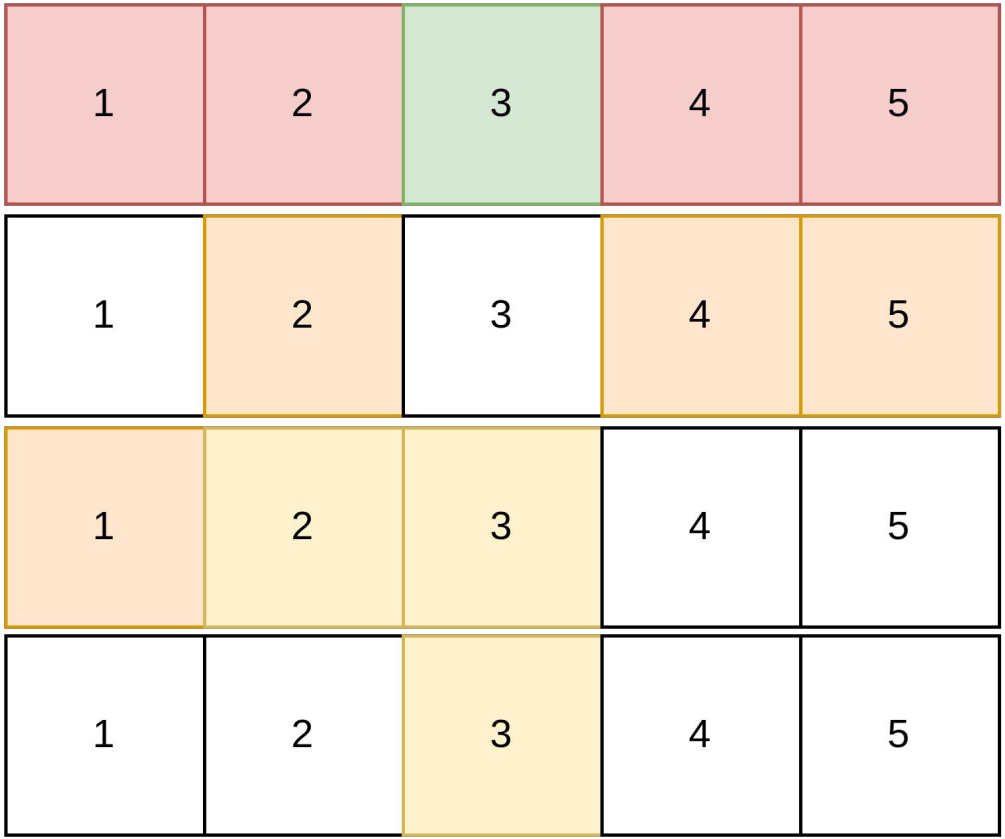
Internals - Sortable mutexes



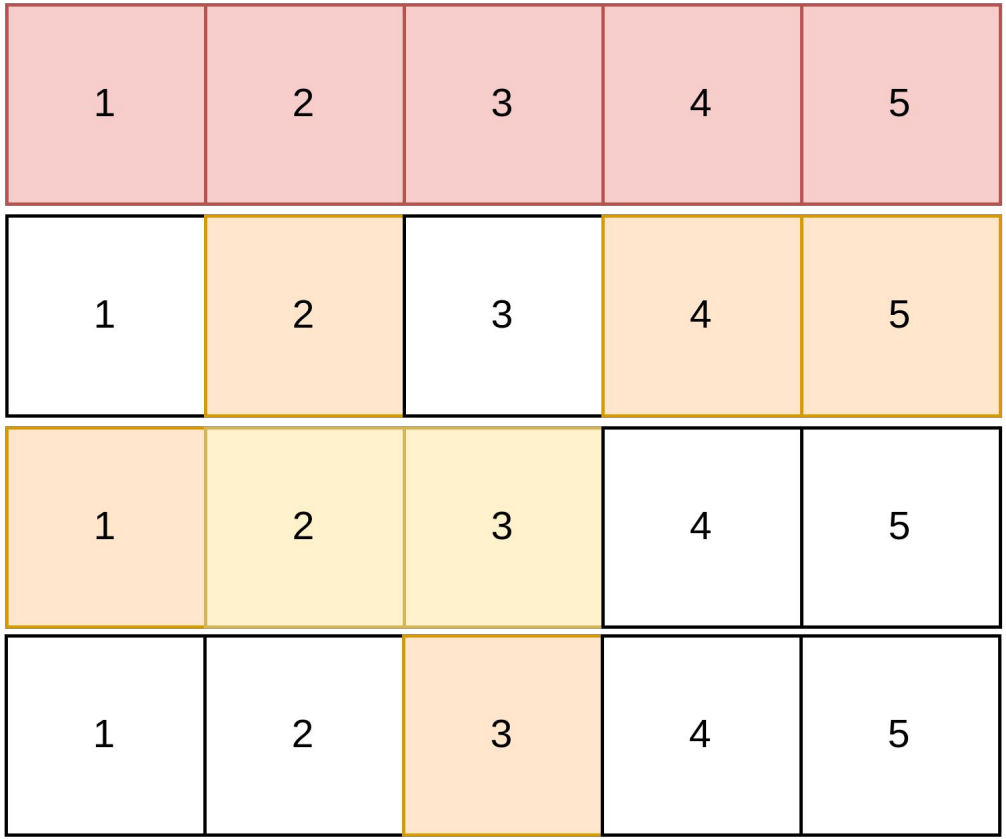
Internals - Sortable mutexes



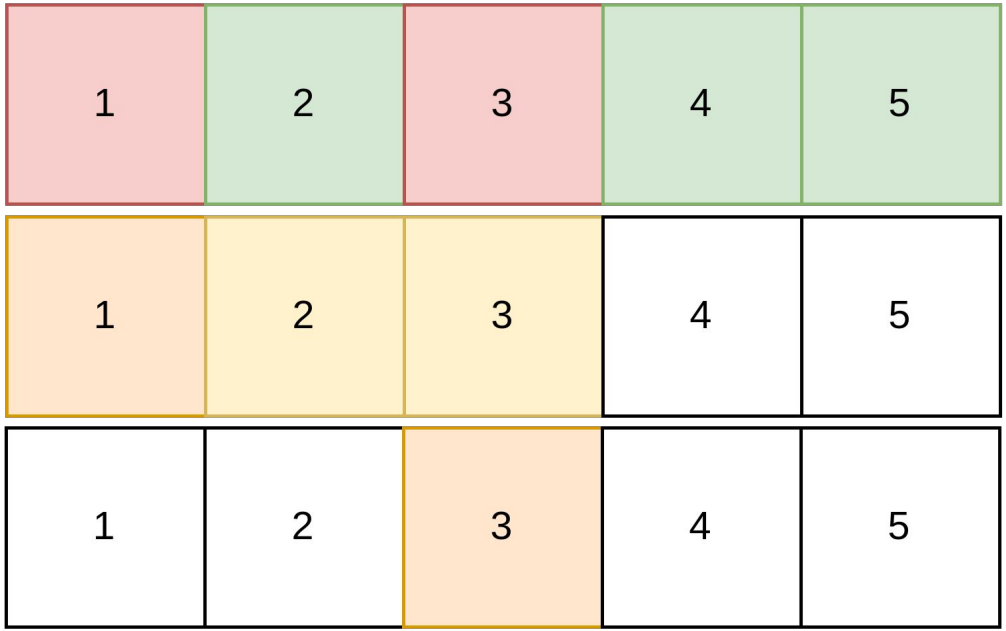
Internals - Sortable mutexes



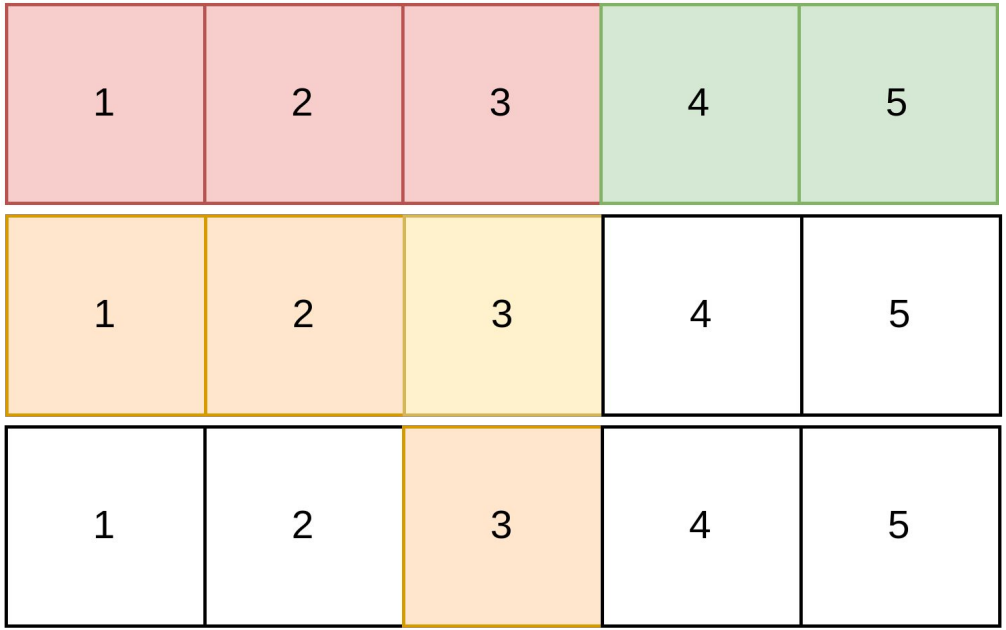
Internals - Sortable mutexes



Internals - Sortable mutexes



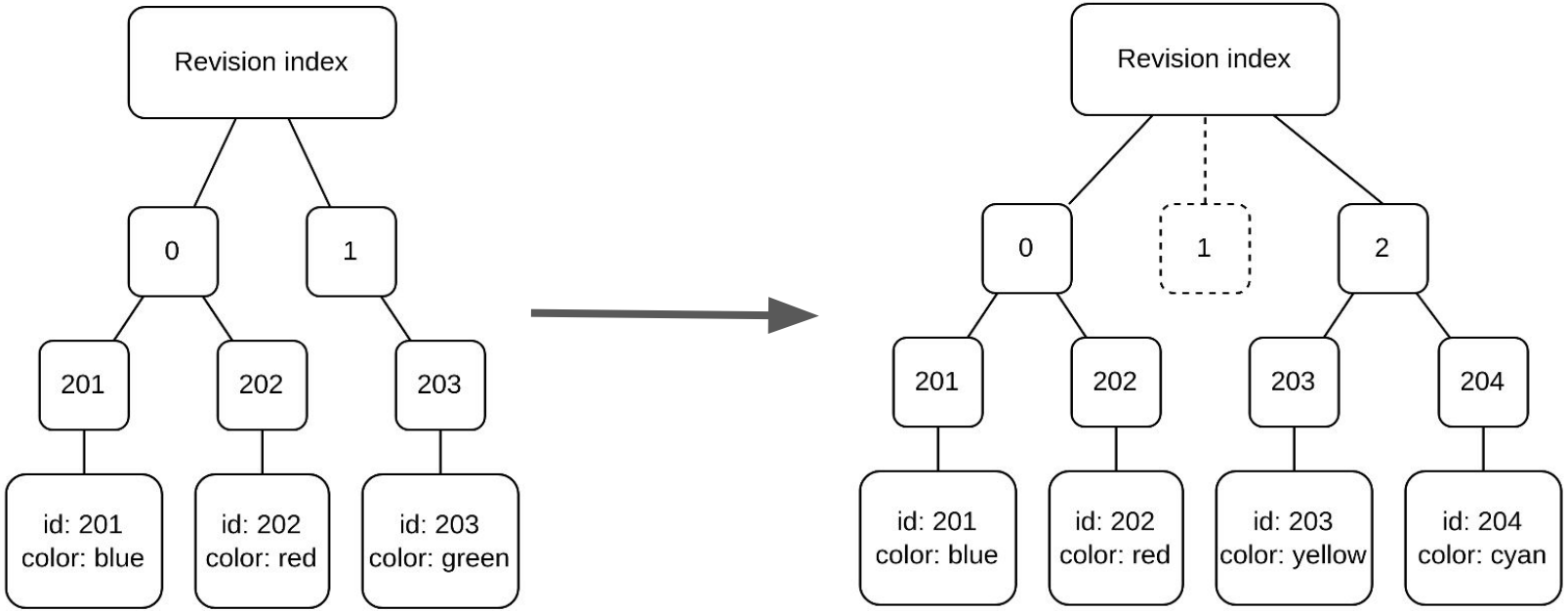
Internals - Sortable mutexes



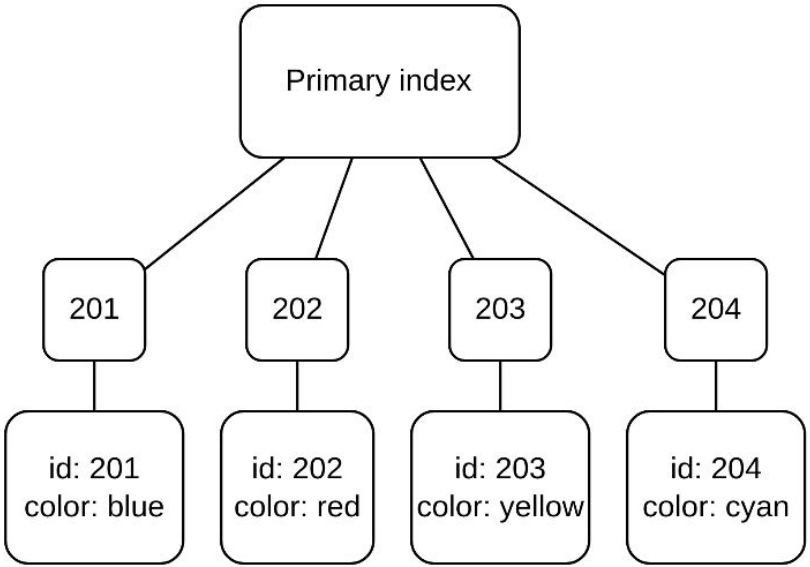
Tracking changes

```
txn := DB.WriteTxn(AppleTable)
iter, err := AppleTable.Changes(txn)
if err != nil {
    txn.Abort()
    return
}
txn.Commit()

for {
    rx := DB.ReadTxn()
    changes, watch := iter.Next(rx)
    for change := range changes {
        fmt.Printf("Deleted: %v, Obj: %v\n", change.Deleted, change.Object)
    }
    <-watch
}
```

Internals - Tracking changes



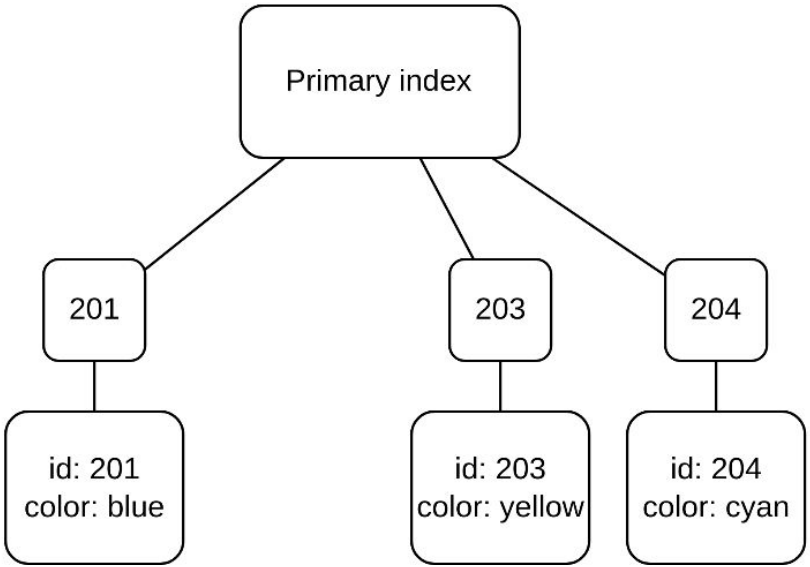
Internals - Tracking changes

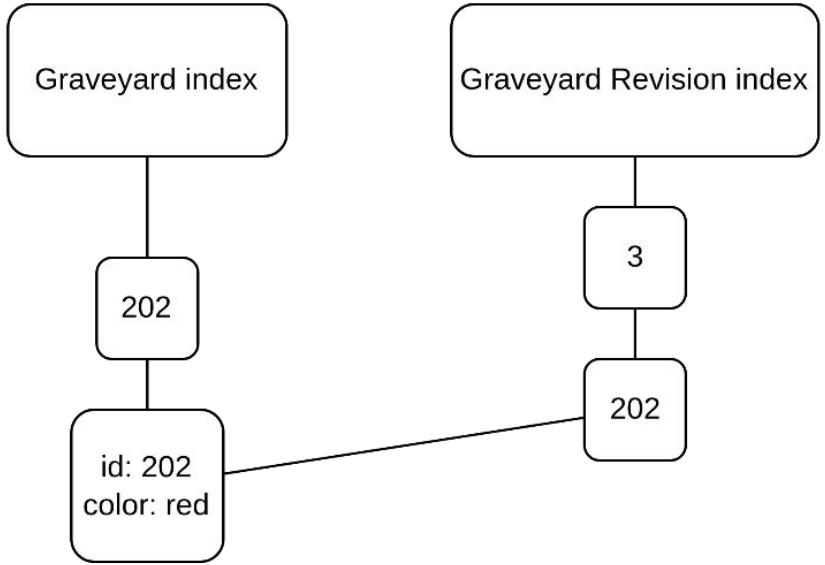


Graveyard index

Graveyard Revision index

Internals - Tracking changes





Final notes

- Sources at github.com/cilium/statedb
- Much more to see
- Credits to Jussi Mäki