Radix Trees
Transactions, and MemDB

Armon Dadgar

@armon

HashiCorp

MemDB

- Used in Consul, Nomad, Docker Swarm
- Built on Immutable Radix Trees
- Inspired by Radix Trees

Radix Trees

Radix Trees

- Tree Data Structure, used as a Dictionary / Map
- Directed (parent / child relationship)
- Acyclic (cannot contain a cycle)
- Keys are *strings**
- Values can be arbitrary

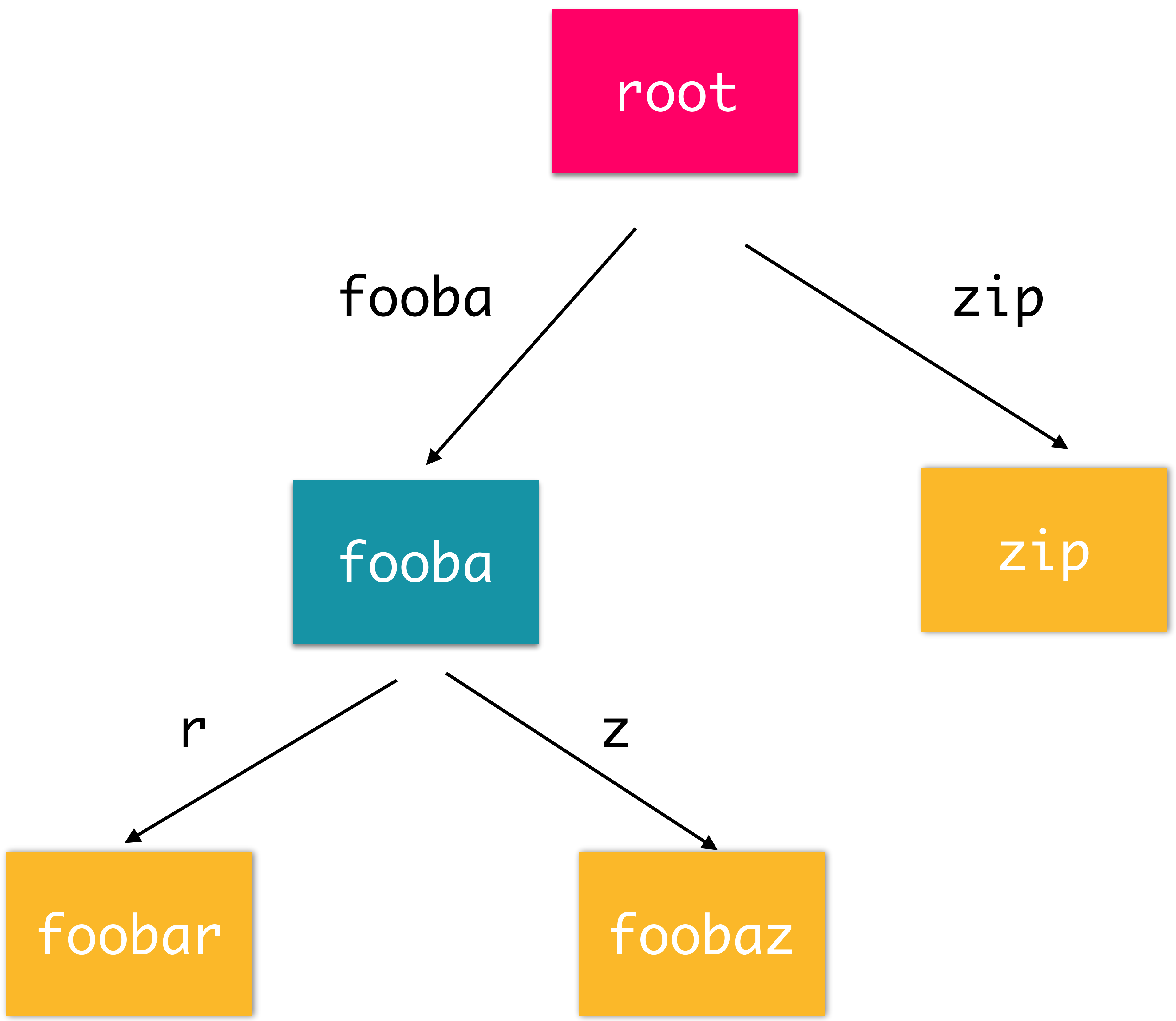
Properties

- `O(K)` operations instead of `O(log N)` for most trees
  - K is length of the input Key
  - Hash functions also `O(K)`, can be deceptive for Hash Tables
- Tunable sparsity vs depth

Operations

- CRUD (Create, Read, Update, Delete)
- Find predecessor / successor of a key
- Min / Max Value
- Find common prefix of keys
- Find longest matching prefix
- Ordered Iteration

Radix Structure



Basic Operations

- Start at the root and with the input key K
- Follow the pointers from the current node using the offset into the key
- Number of iterations linear with length of key
- May need to split nodes on Insert or merge on Delete

Uses Cases at HashiCorp

- Consul / Vault ACLs
- Vault Request Routing
- CLI Library
- etcetera

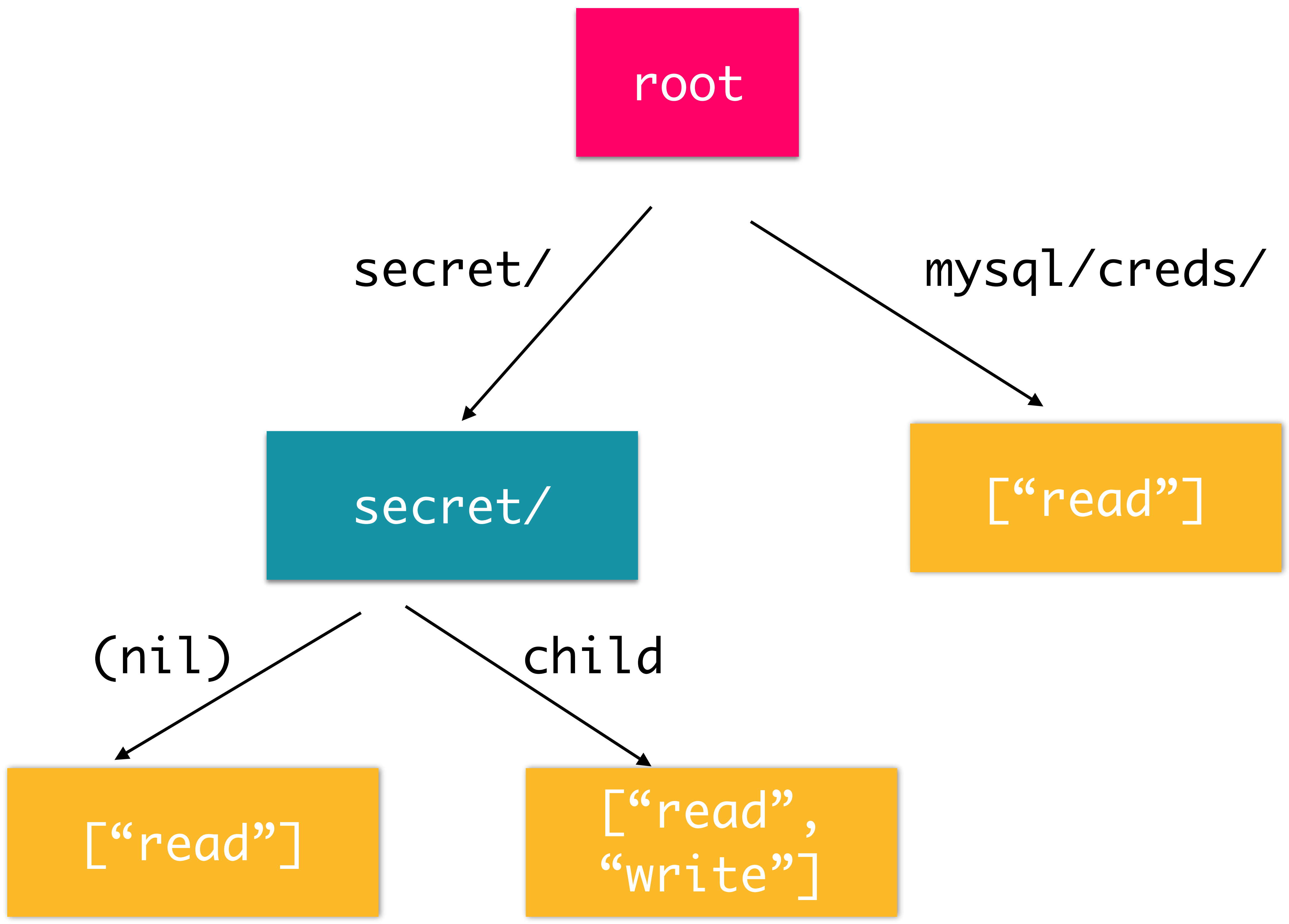
Vault ACLs

```
path “secret/*” {
    capabilities = [“read”]
}

path “secret/child” {
    capabilities = [“read”, “write”]
}

path “mysql/creds/*” {
    capabilities = [“read”]
}
```

ACL Structure

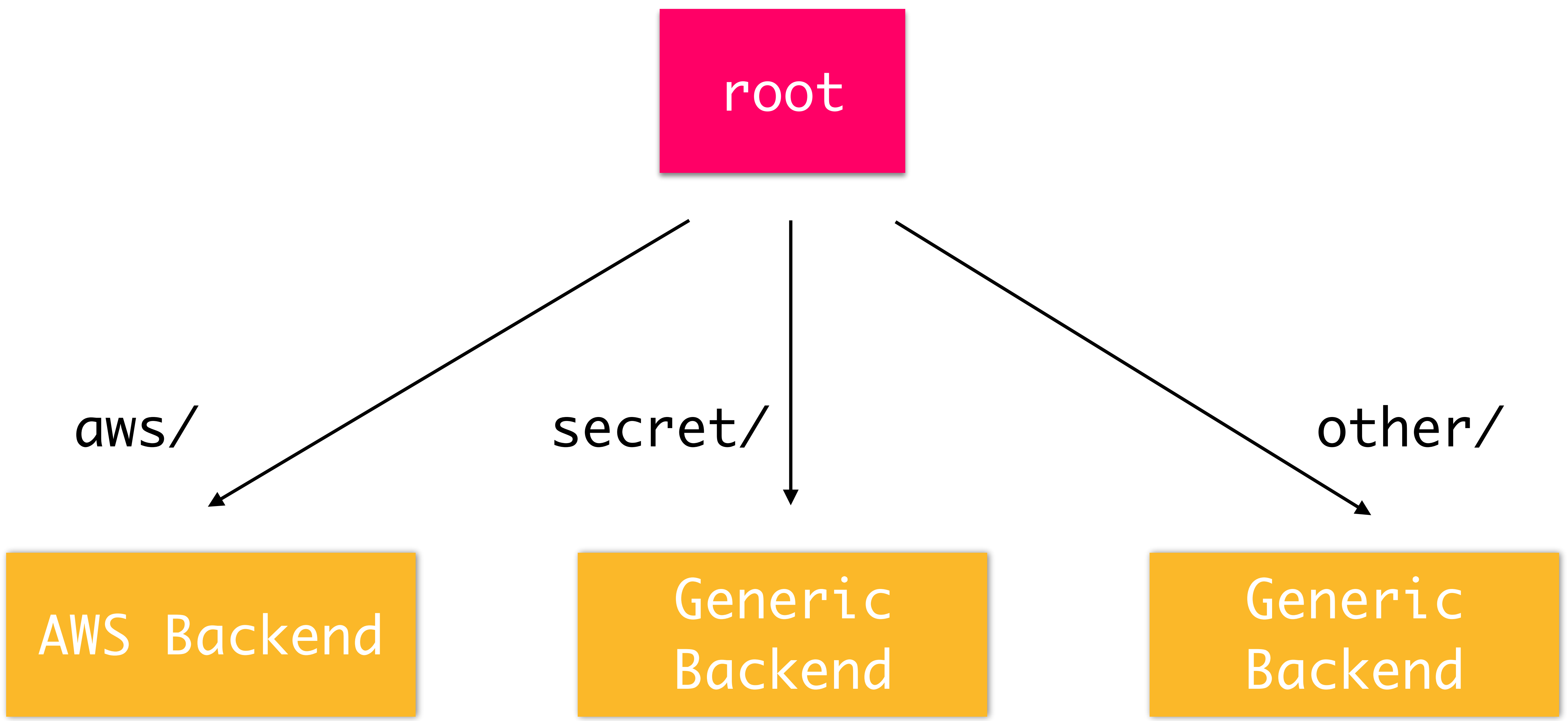


Vault Request Routing

```
$ vault mount -path=other generic
Successfully mounted 'generic' at ‘other’!

$ vault mount aws
Successfully mounted 'aws' at 'aws'!
```

Routing Structure



Request Routing

- `$ vault read secret/foobar`
- Uses the longest prefix (`secret/*`) on ACLs to determine which policy is applicable and if the operation should be allowed
- Uses the Routing tree to find longest prefix (`secret/`) to determine the backend that services the request

Immutable Radix Tree

Immutability

- The inability to be changed, e.g. not mutable
- Every modification returns a *new* tree, existing tree is unmodified
- Uses more memory, reduces need for read coordination

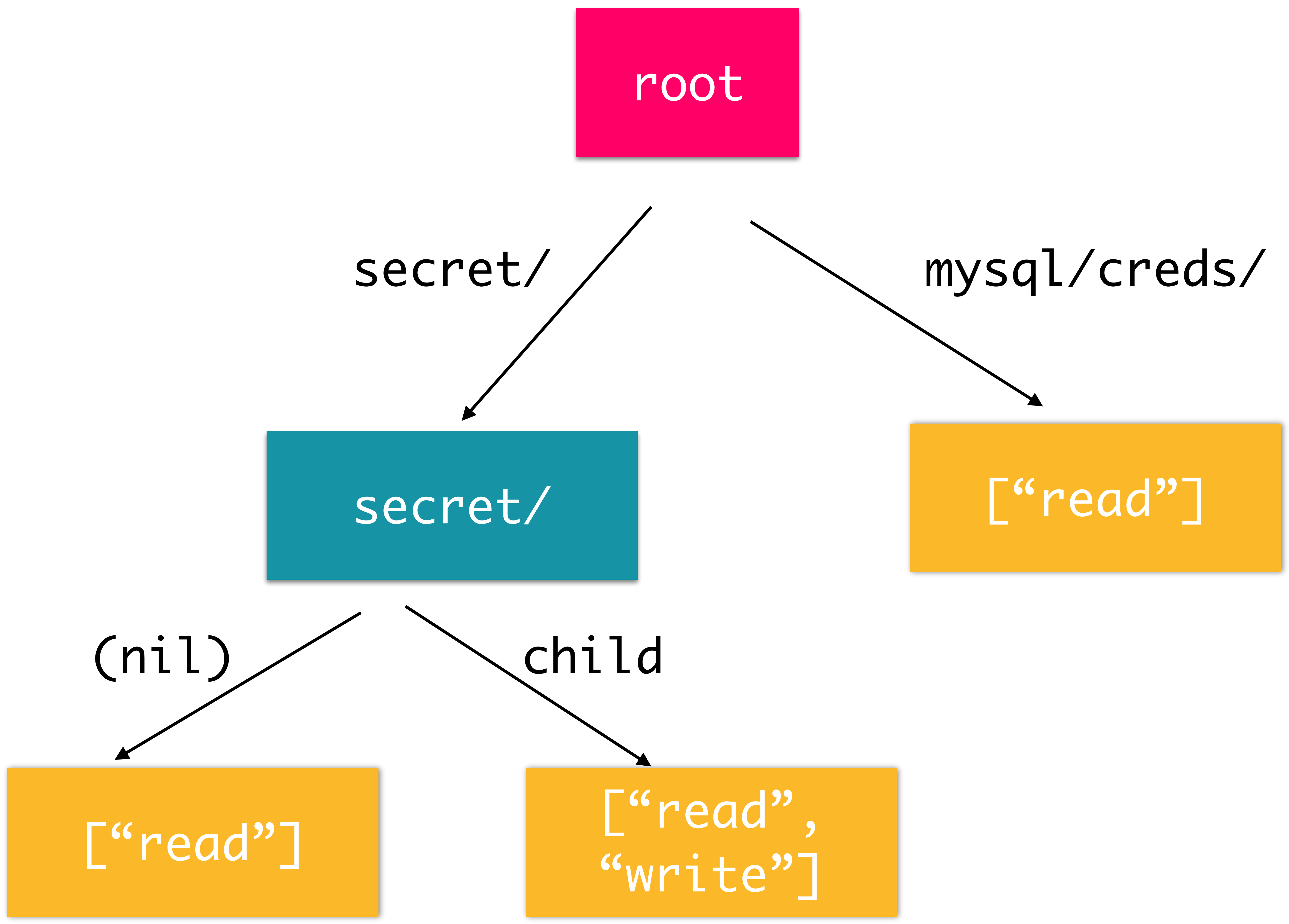
Immutable Radix

- Same operations and properties of mutable Radix
- Every modification returns a new root
- Mutable: `Insert(root, key, value) = (void)`
- Immutable: `Insert(root, key, value) = root’`

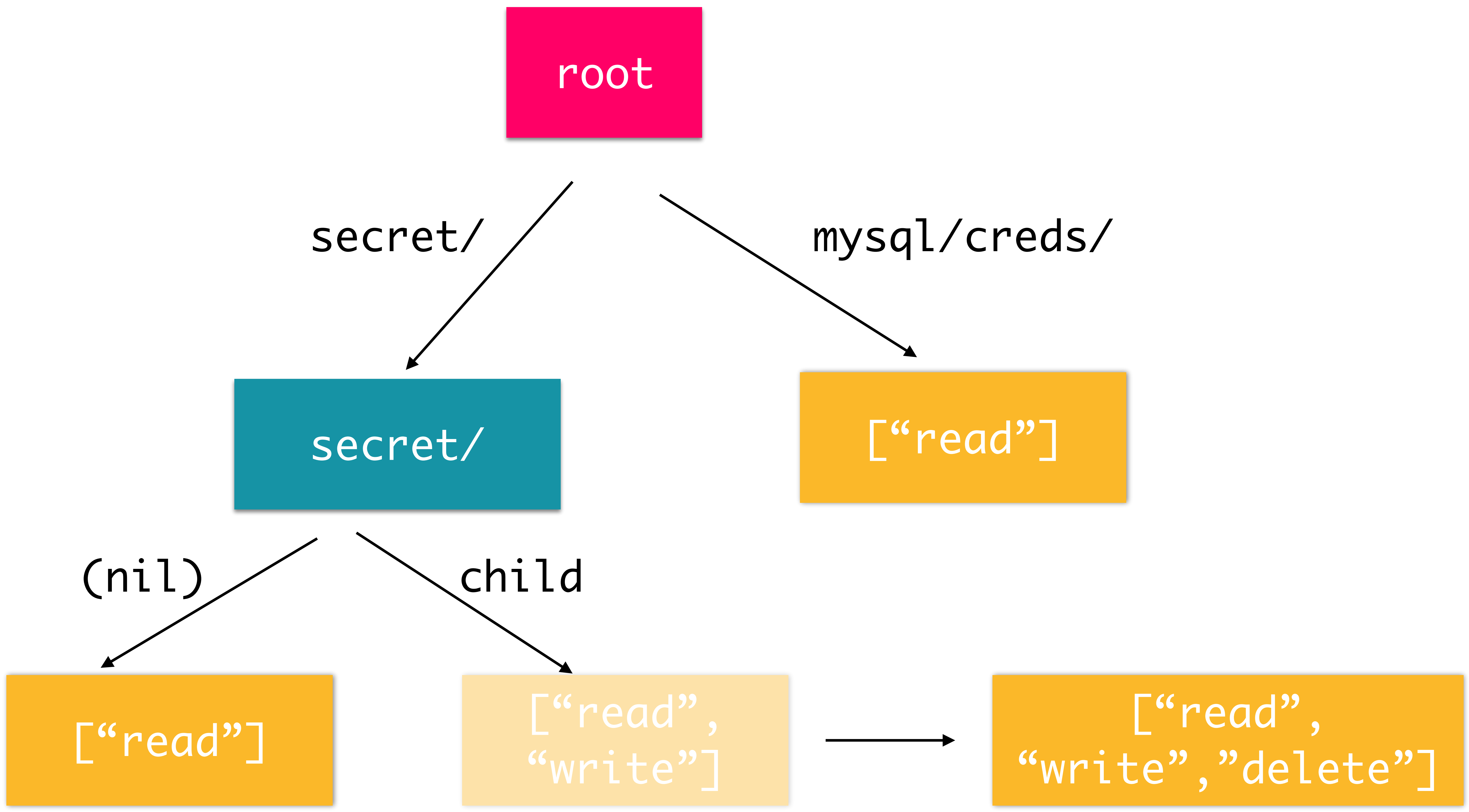
Copy On Write

- Any time a node or leaf is going to be modified, we copy the node and update the copy
- K nodes updated per modification

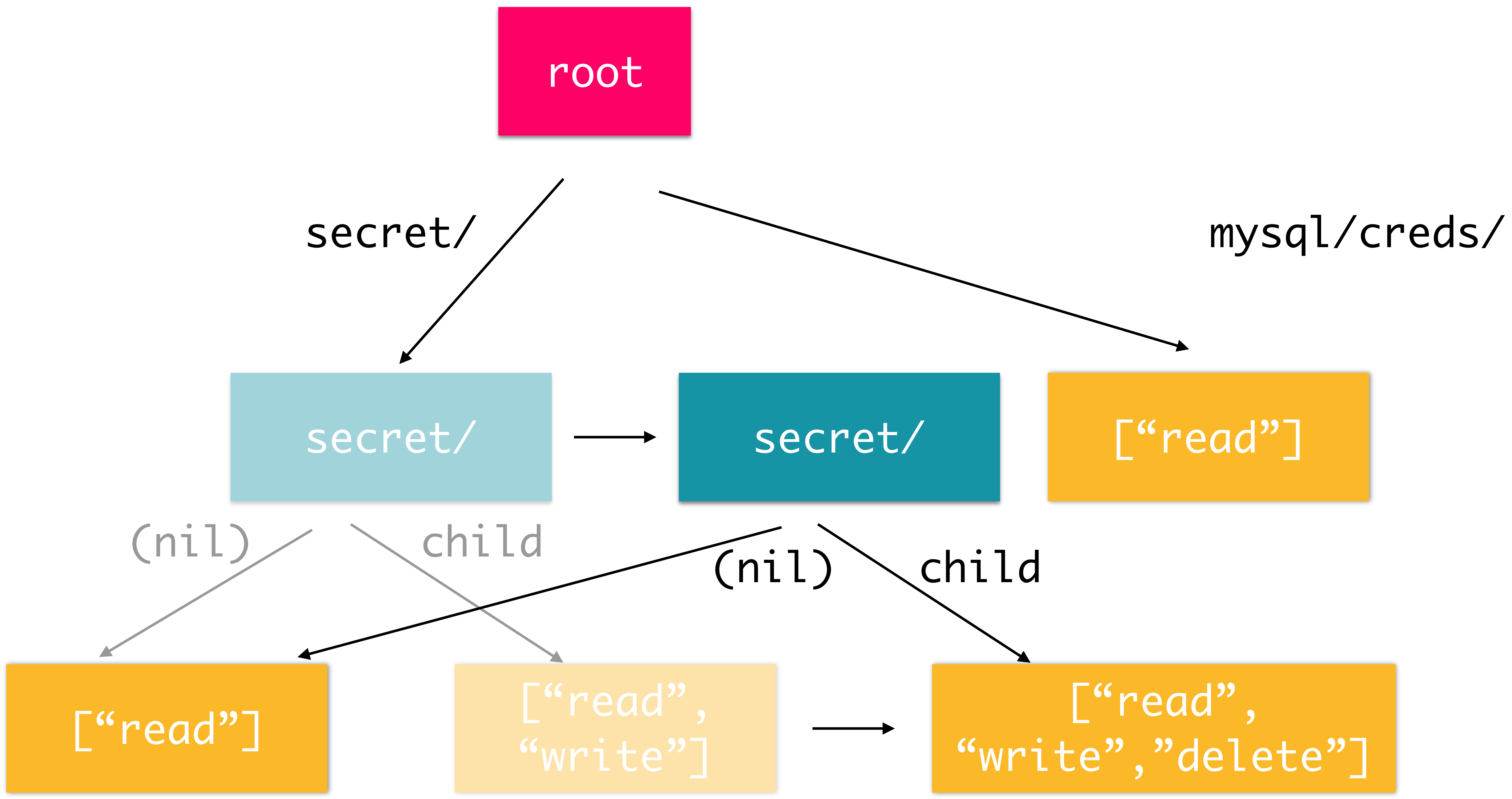
Original Tree



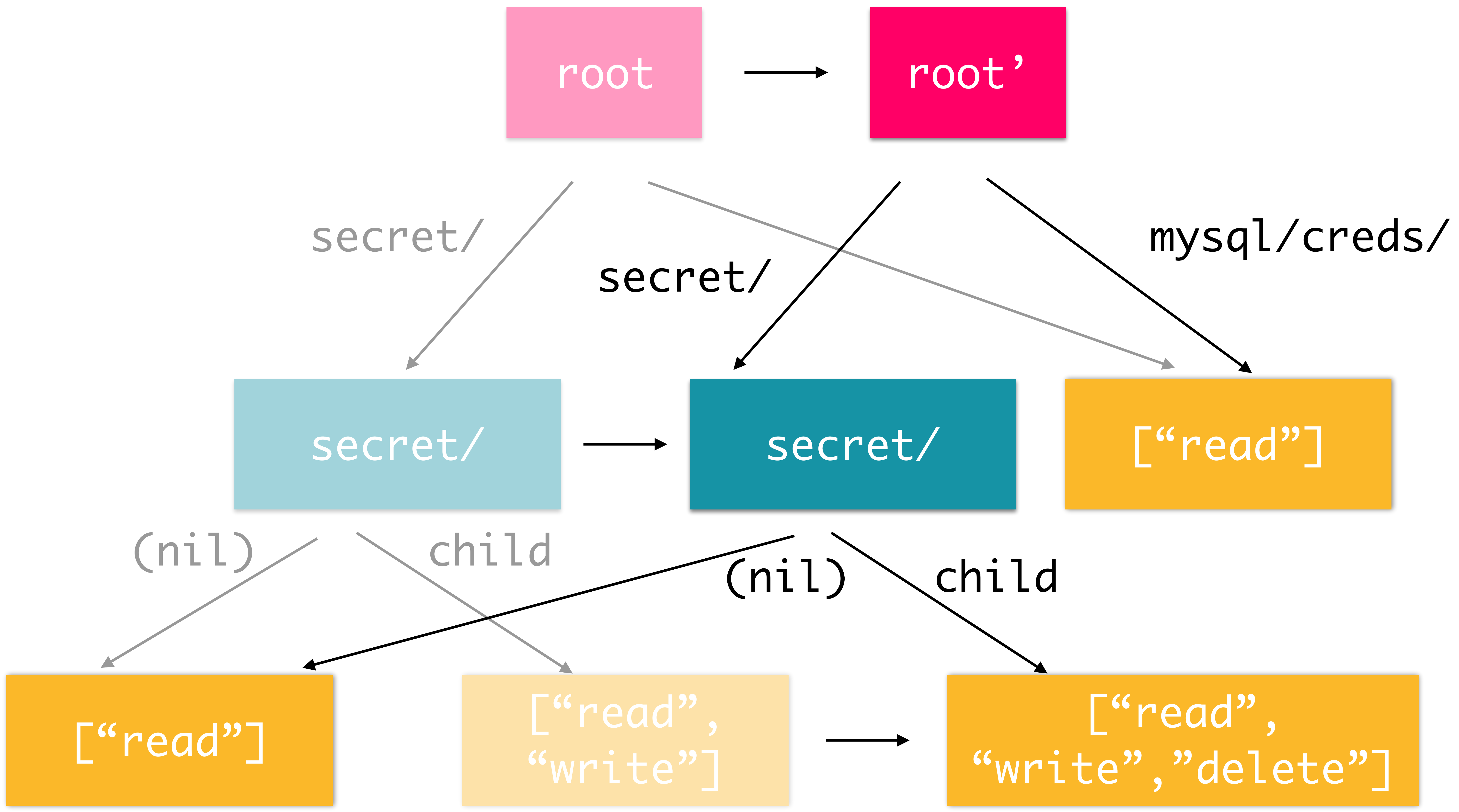
Update secret/child



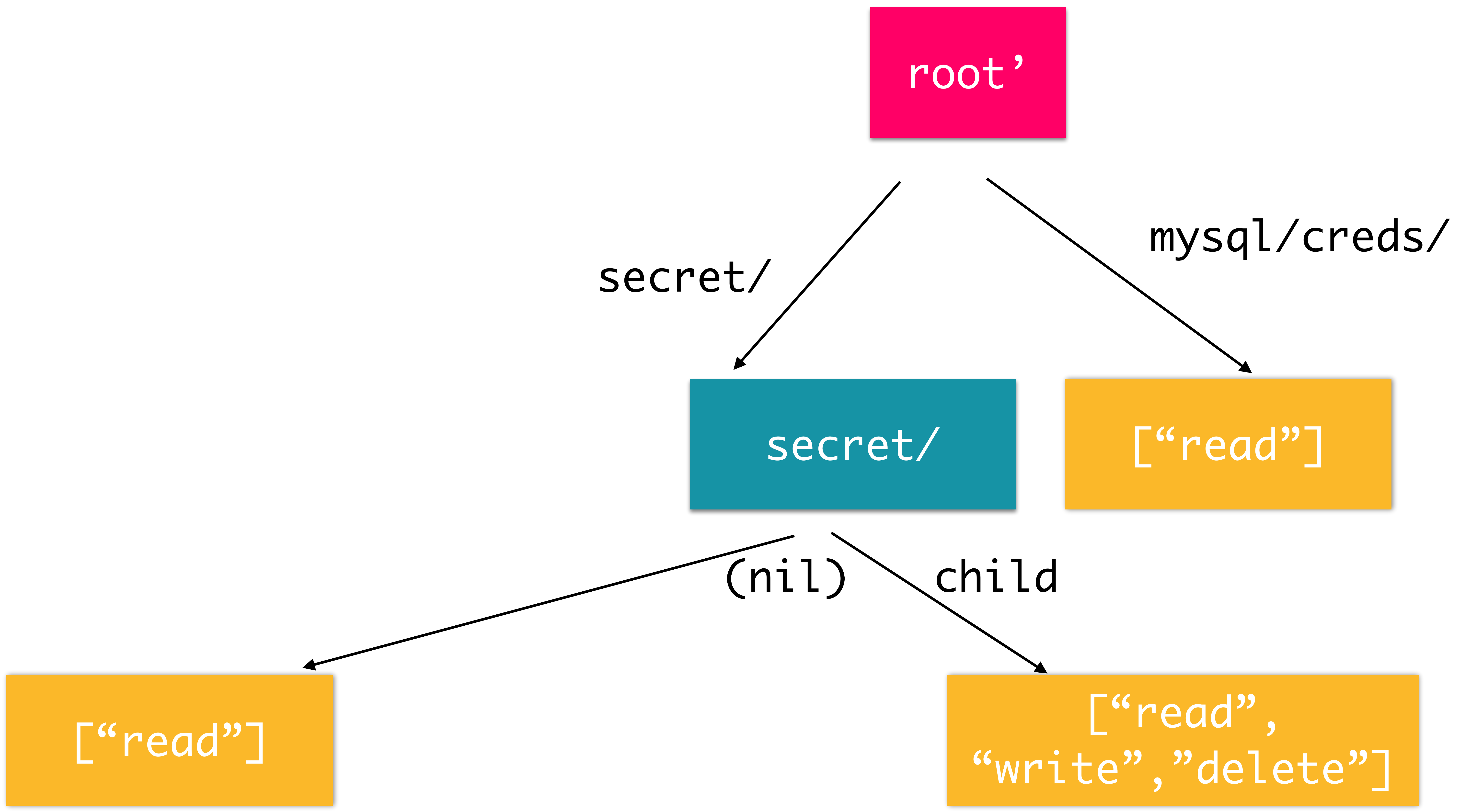
Update secret/child



Update secret/child



Update secret/child



Immutable vs Mutable

- Mutable Radix requires synchronization for reads/writes
  - Concurrent reads allowed
  - Concurrent read/writes disallowed
- Immutable Radix requires synchronization for writes only
  - Concurrent read/writes allowed
  - Each write returns a new tree, existing tree is unmodified
  - Good for heavy read, low write workloads

Uses Cases at HashiCorp

- MemDB (Consul, Nomad, Docker Swarm)
- Vault Enterprise

Transactions

Transaction

- Standard usage is RDBMS (ACID)
- **A**tomicity: Completely fails or completely succeeds
- **C**onsistency: Does not result in any integrity violations (e.g. User ID with does not map to blank e-mail)
- **I**solation: Transaction is not visible to others until completed
- **D**urability: Once completed, the changes are permanent

Immutable Radix

- We can use an immutable radix tree to implement in-memory transactions!
- Provides us with **A and I** properties
  - Consistency is domain specific
  - In-memory only, so not Durable in the ACID sense
  - Can be used to build ACID system (e.g. Consul, Nomad)

Atomicity and Isolation

- Many keys can be Created, Updated, Deleted in a single transaction
- Atomicity: transaction creates new root on commit, retains existing root on abort. Check-And-Set (CAS) operation to swap root pointers.
- Isolation: Copy-On-Write of each transaction prevents readers of the existing root from witnessing any of the changes.

MemDB

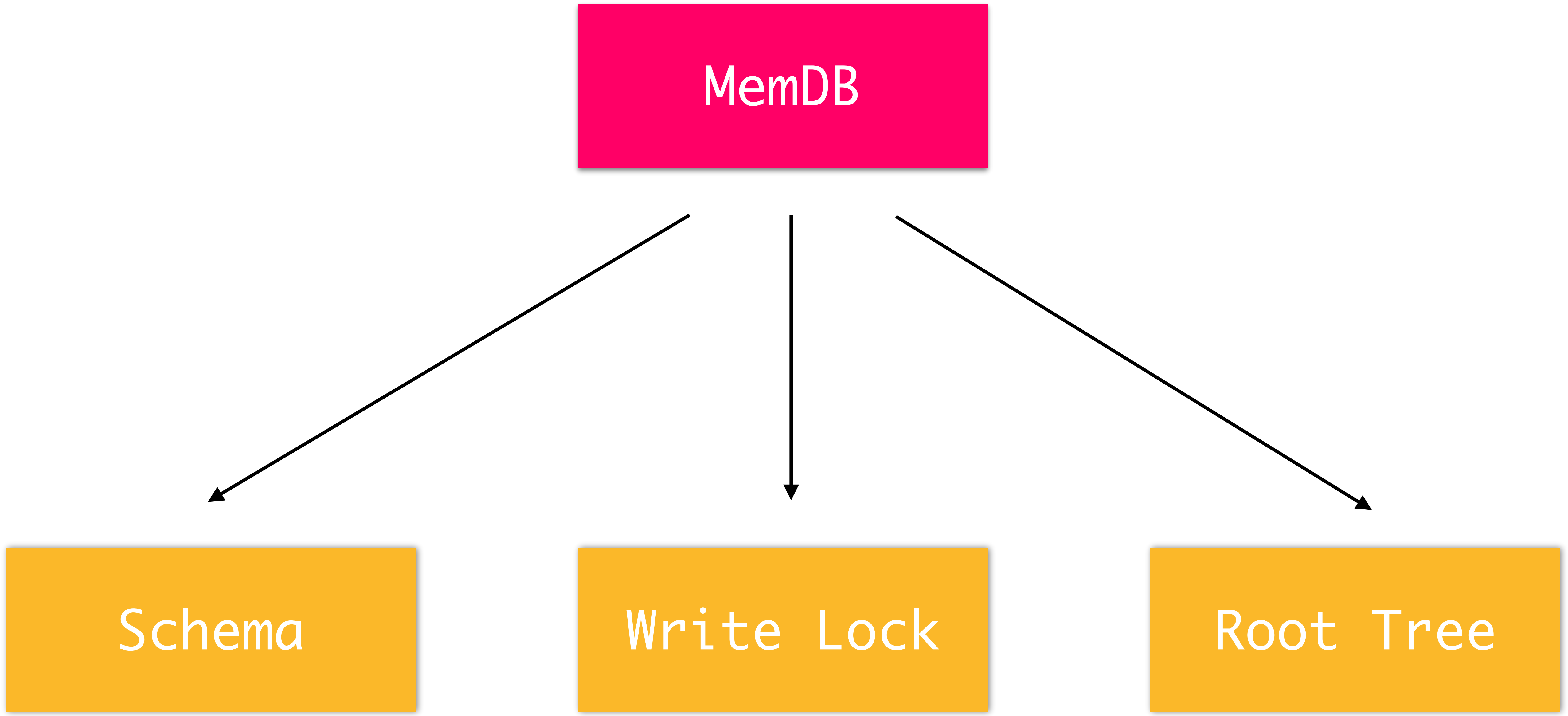
MemDB Goals

- **MVCC**: Multi-Version Concurrency Control. Support multiple versions of an object so that you can have concurrent read/writes.
- **Transaction Support**: Update many objects in a transaction to support richer high level APIs. Should be atomic and isolated.
- **Rich Indexing**: Allow a single object to be indexed in multiple ways (e.g. User ID, email, DOB, etc)

Why those requirements?

- Consul needs to be able to snapshot current state to disk while accepting new writes. Long running read cannot block writes.
- A single event such as a node failure may need to update multiple pieces of state (Health Checks, Sessions, K/V locks)
- Many different query paths. Services by node, services by name, services in a failing state, etc.

MemDB Structure



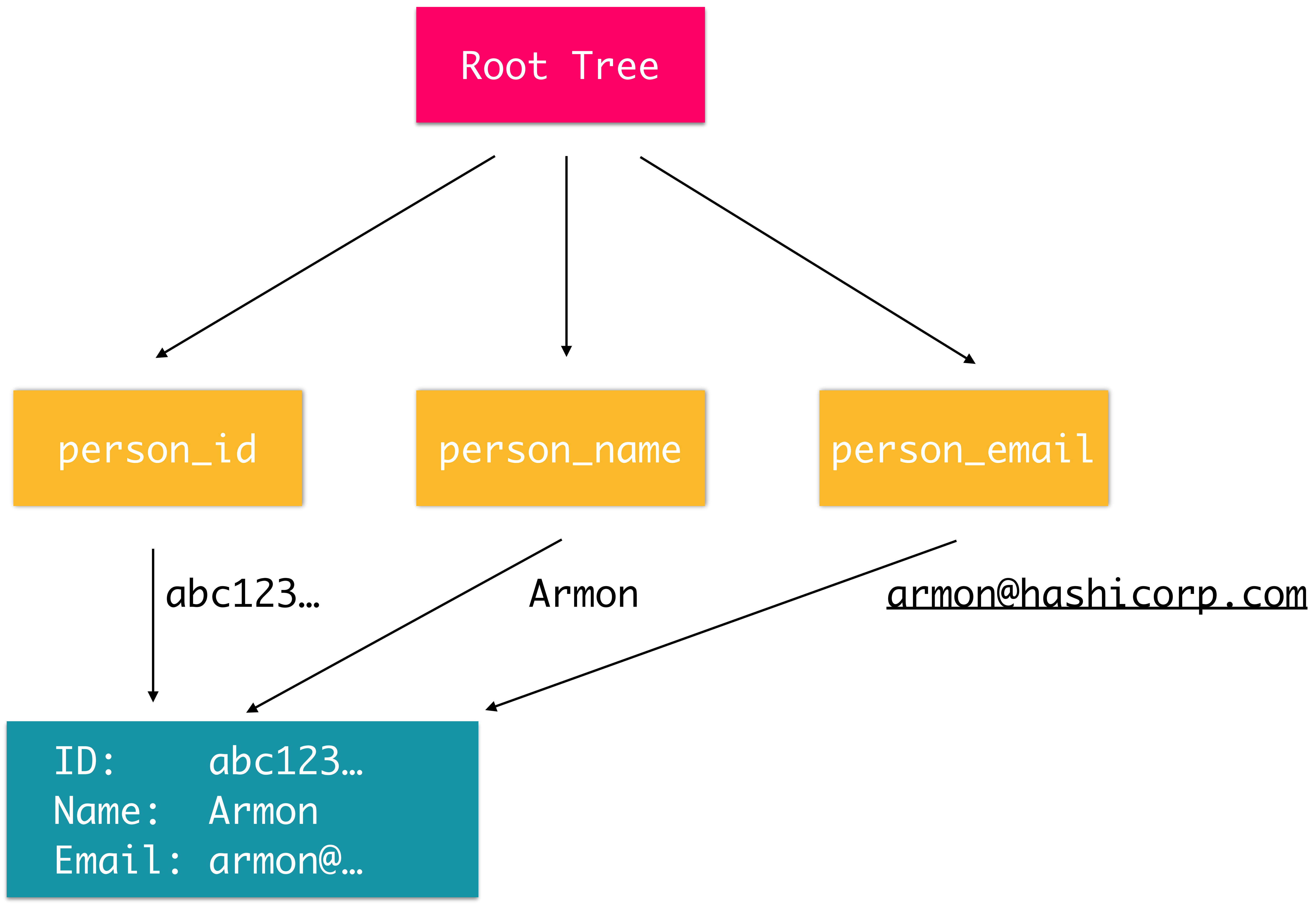
Schema

- Schema defines tables and indexes at creation time
- Allows for efficient storage and indexing of objects
- Sanity checking of objects (ensure Consistency)

Example Schema

```
&DBSchema{
    Tables: map[string]*TableSchema{
      "people": &TableSchema{
        Name: "people",
        Indexes: map[string]*IndexSchema{
          "id": &IndexSchema{
            Name:    "id",
            Unique:  true,
            Indexer: &UUIDFieldIndex{Field: "ID"},
          },
          "name": &IndexSchema{
            Name:    "name",
            Indexer: &StringFieldIndex{Field: "Name"},
          },
          “email”: &IndexSchema{
             Name: “email”,
             Indexer: &StringFieldINdex{Field: “Email”},
           },
```

MemDB Tree Structure



MemDB Tree Structure

- Each table has a primary tree, keyed by a unique ID
- Each table can have 0+ indexes, unique or non-unique
- Single copy of the object is stored in the primary tree, indexes point to the object

Indexes

- Each index has an Indexer which extracts a value from an object and turns it into an index key
  - `StringFieldIndex`: Extracts string value field
  - `UUIDFieldIndex`: Extracts string or []byte field
  - `FieldSetIndex`: Checks if a field has non-zero value (is set)
  - `ConditionalIndex`: Extracts field as boolean value
  - `CompoundIndex`: Combines multiple indexes

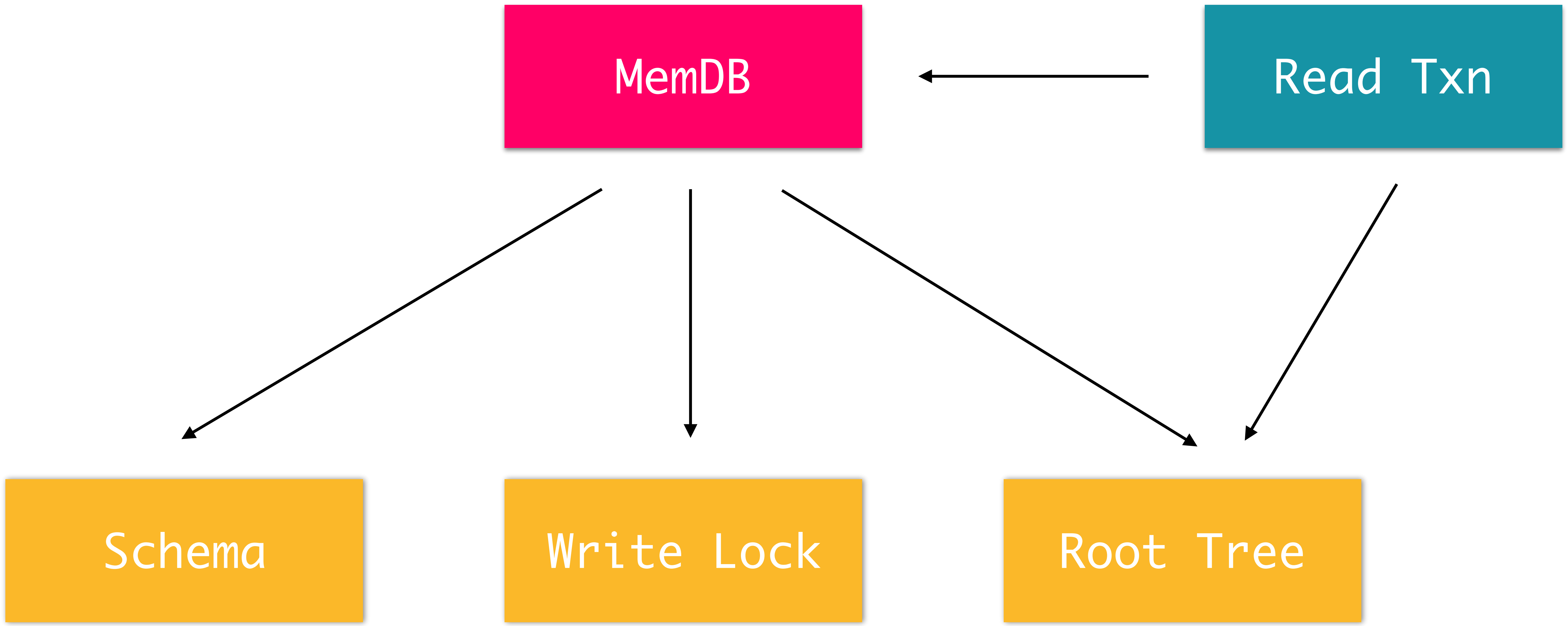
Compound Index

- CompoundIndex{StringFieldIndex{“First”}, StringFieldIndex{“Last”}}
- Extracts {“First”: “Armon”, “Last”: “Dadgar”} as “Armon\x00Dadgar\x00”
- Queries like “first = ‘Armon’ and last starts with ‘D’”

Read-only Transactions

- Snapshot MemDB, retain a copy of the root pointer
- Read against the Snapshot
- Immutable trees allow us to avoid locking across reads and isolation from other transactions

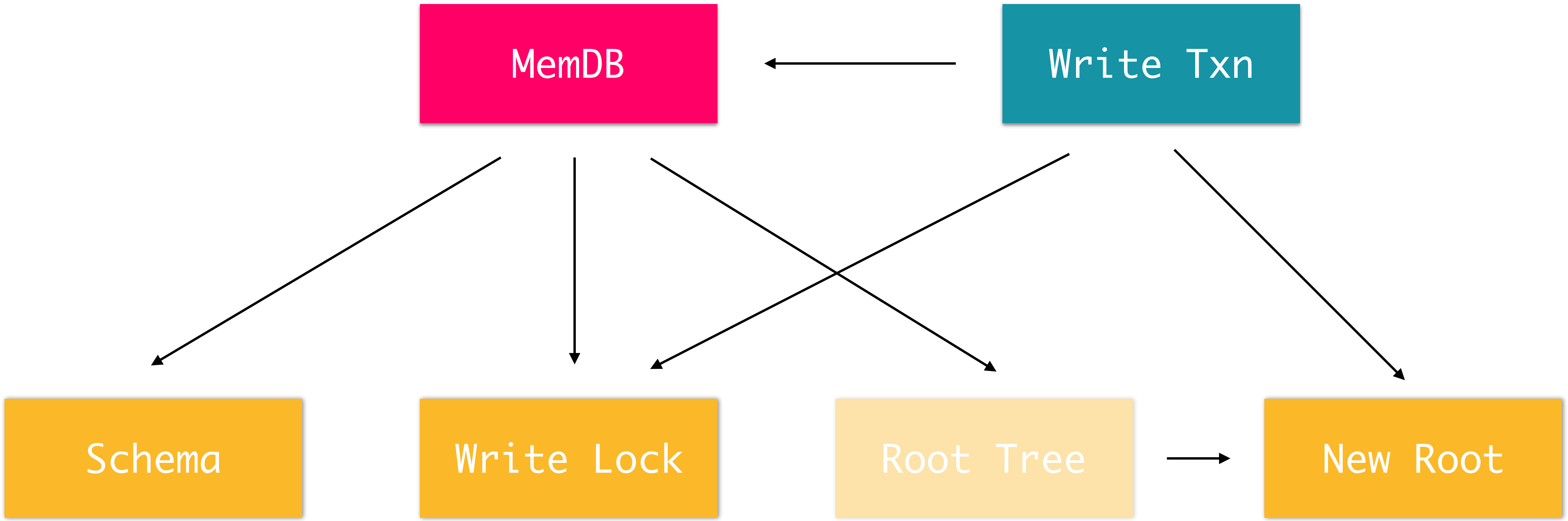
Read-only Transaction



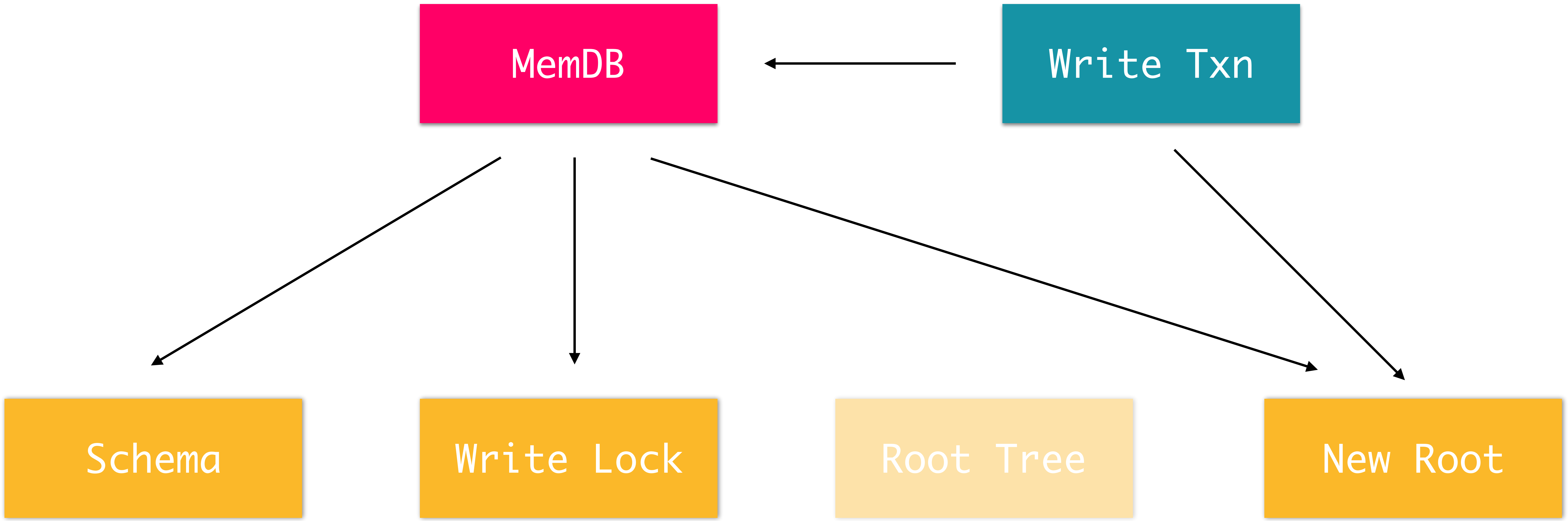
Mixed Transactions

- Acquire the write lock, serializes writes
- Write to the root, creating a new root
- Atomic swap the root pointers on commit, do nothing on abort
- Release the write lock

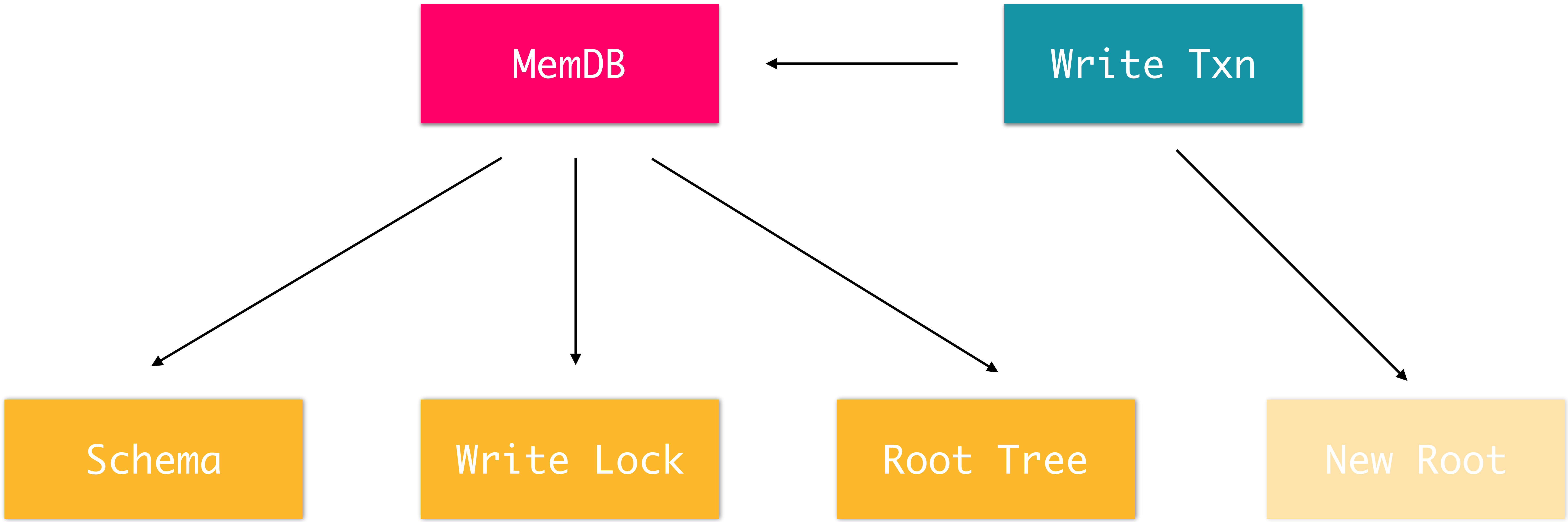
Mixed Transaction (Progress)



Mixed Transaction (Commit)



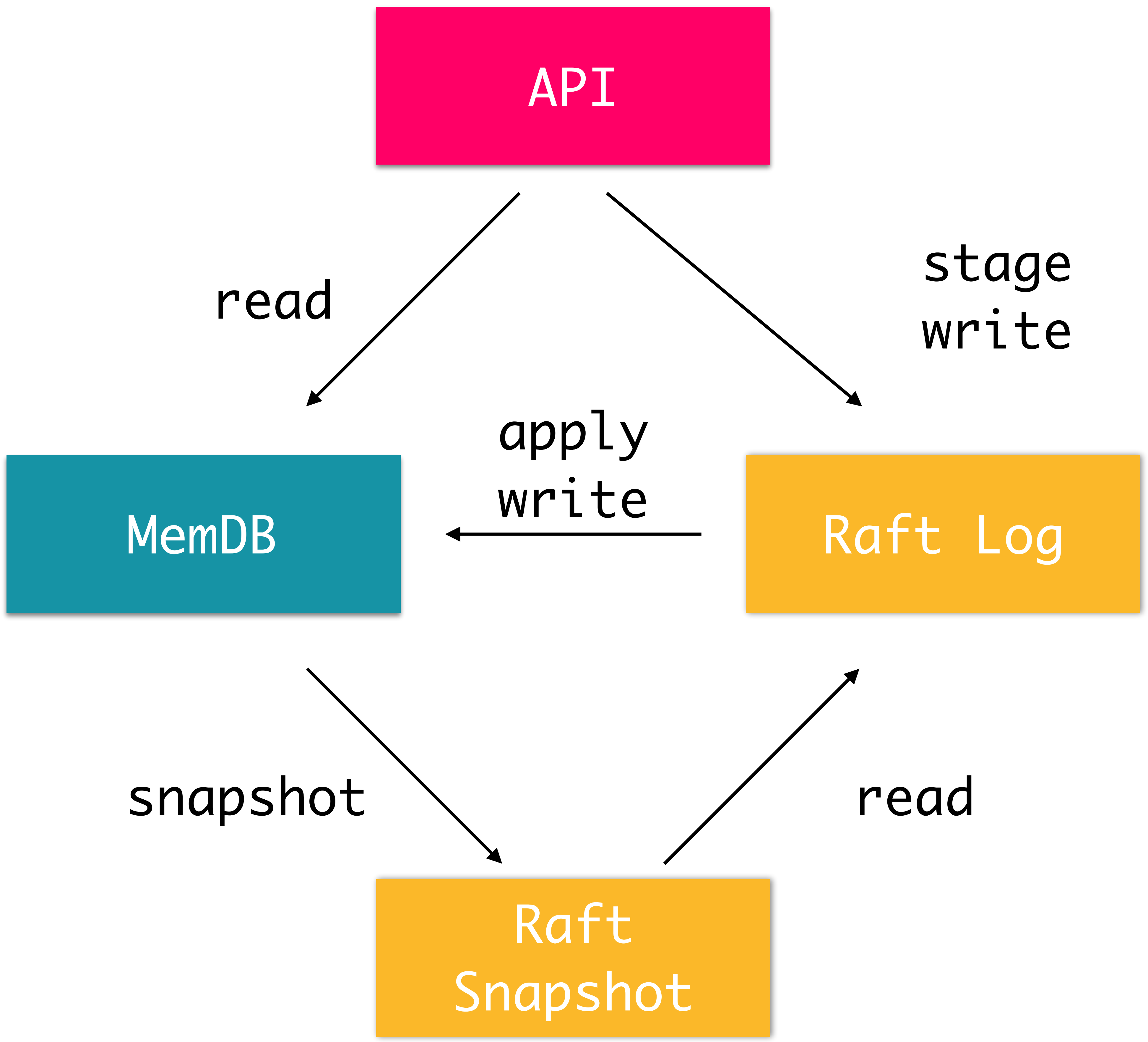
Mixed Transaction (Abort)



Uses Cases

- Consul
- Nomad
- Docker Swarm

Consensus Based Systems



MemDB

- Allows highly concurrent reads to state
- Long running reads to snapshot without blocking writes
- Single threaded writer from Raft has no write contention
- Raft ensures consistent state for all copies of MemDB

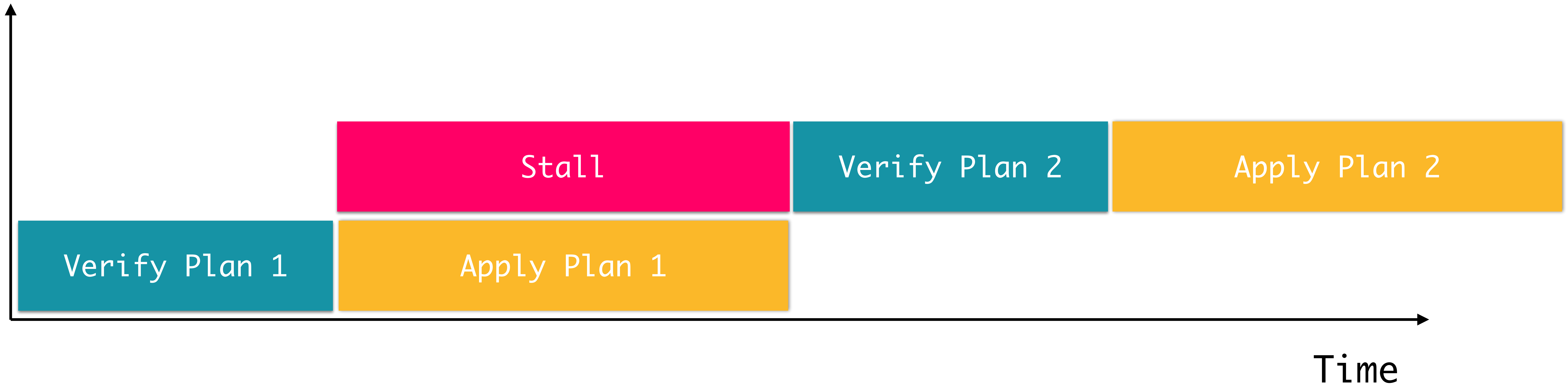
Nomad Advanced Usage

- Schedulers use snapshots of state to determine placement
- Leader provides coordination through evaluation queue and plan queue
  - Evaluation Queue: Dequeues work to schedulers, provides at-least-once semantics
  - Plan Queue: Controls placement to prevent data races and over-allocation

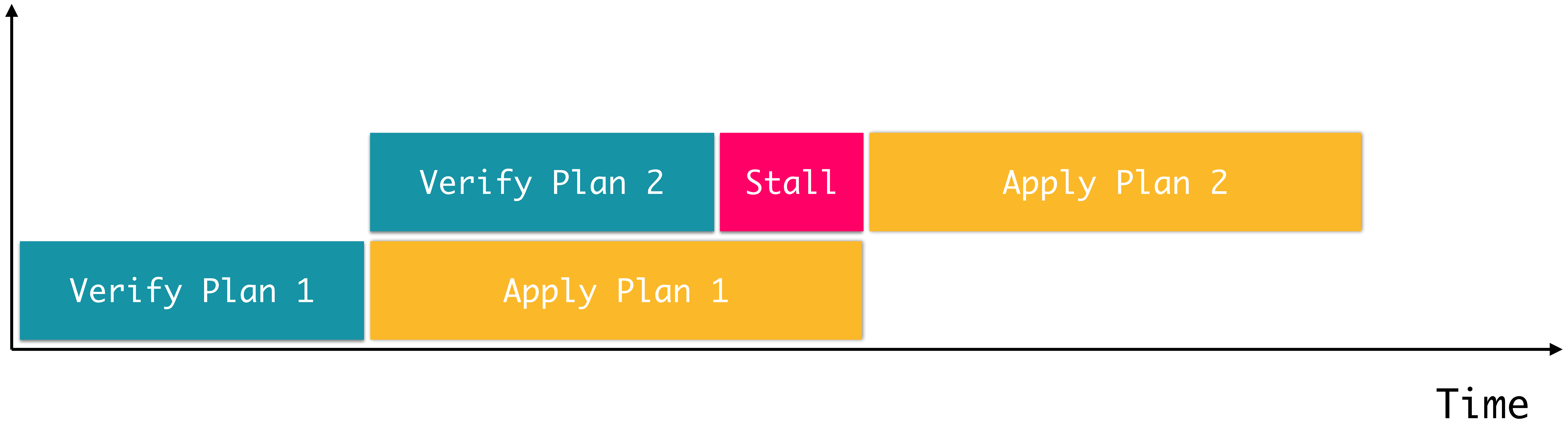
Plan Queue

- Receives placement plans from schedulers
- Verifies plan and writes to Raft to commit the plan
- Read, Verify, Write loop causes a stall while we are waiting for Raft to commit
- MemDB allows us to optimistically evaluate plans while we wait!

No Overlapping



Plan Overlapping



Plan Overlapping

- Plan 1 is applied to a snapshot of the state
- Plan 2 is verified against the optimistic state copy
- Once plan 1 commits, we can submit plan 2
- Allows CPU to verify plan while waiting on I/O to apply writes

Conclusion

Radix Trees

- High performance tree data structure
- Comparable to Hash Tables usually, richer set of operations supported
- I’ve used them in probably every project I’ve ever worked on

Immutable Radix Trees

- Similar to mutable radix tree
- Simplifies concurrency
- Allows for highly scalable reads

MemDB

- Abstracts radix trees to provide object store
- Provides MVCC, transactions, and rich indexing
- Simplifies complex state management
- Allows for highly scalable reads

Thanks!

Q/A

go-radix: https://github.com/armon/go-radix
go-immutable-radix: https://github.com/hashicorp/go-immutable-radix
MemDB: https://github.com/hashicorp/go-memdb