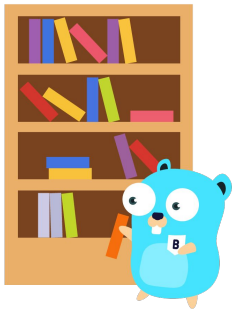
Key-Value Storage Workshop

Vangelis Katikaridis | **Sr. Backend Engineer**

github.com/drakos74
Twitter: @vangkos

BEAT

KEYS AND VALUES



KEY-VALUE STORAGE

There are several aspects / considerations tied to it.

- API
- Consistency
- Performance
- Durability
- Underlying Storage technologies
- Scalability
- Failure scenarios

API

Basic

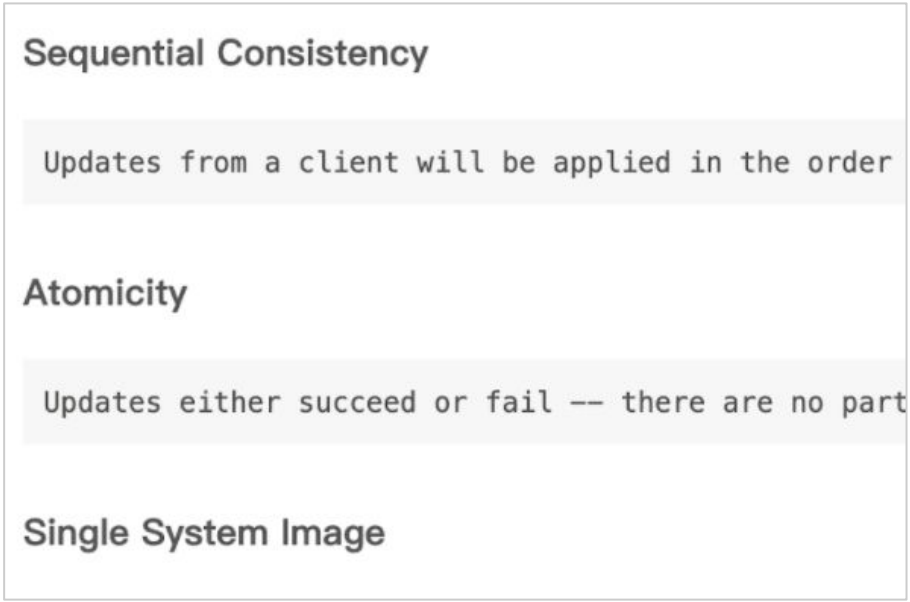
- **Put** (key,value) => error
- **Get** (key) => value

Advanced

- PutIfAbsent (key , value) => value
- ForEach (func(key,value)) => error
- Query (regex) => value[]
- ...

CONSISTENCY

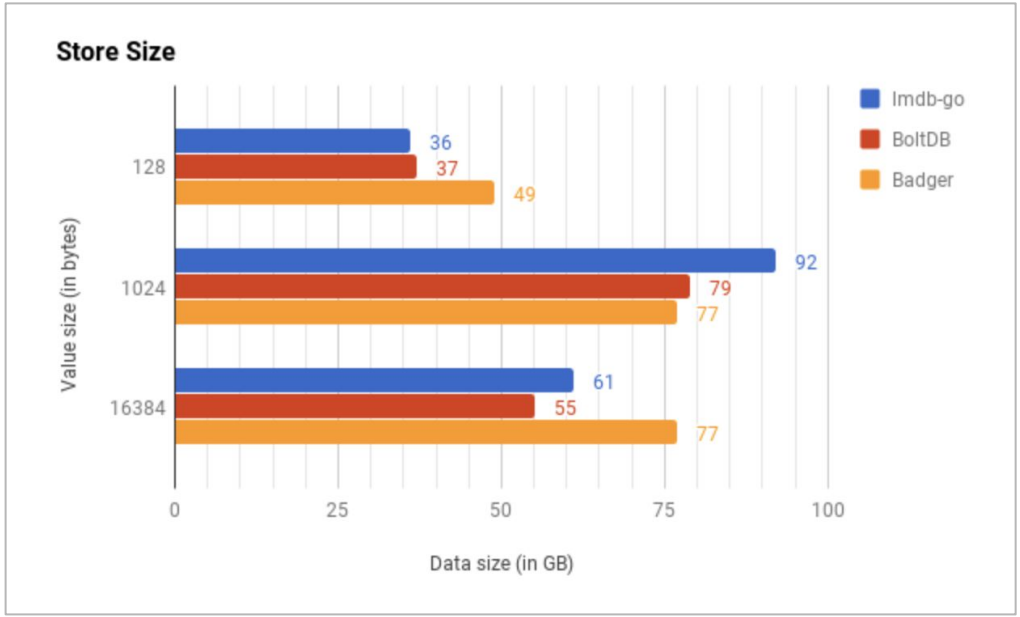
- What changes Readers see..
- While Writers are updating the store..



‘Extract’ from [zookeeper consistency guarantees]

PERFORMANCE

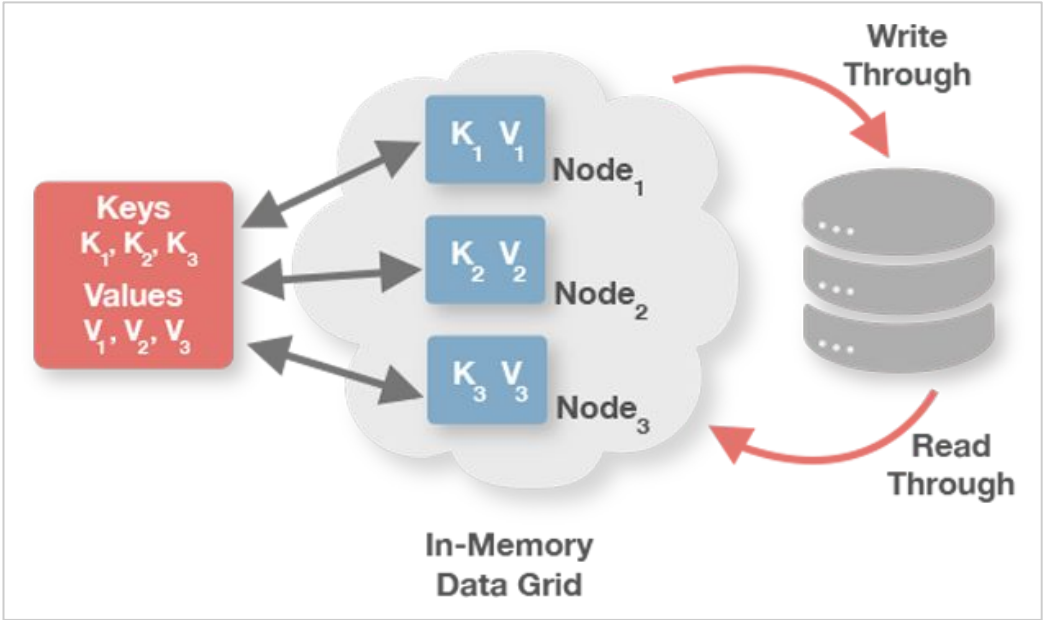
- Key-Value stores perform differently depending on ...
- Size and amount of data



Sample Benchmarking extract from Dgraph (Badger based graph database) [blog]

DURABILITY

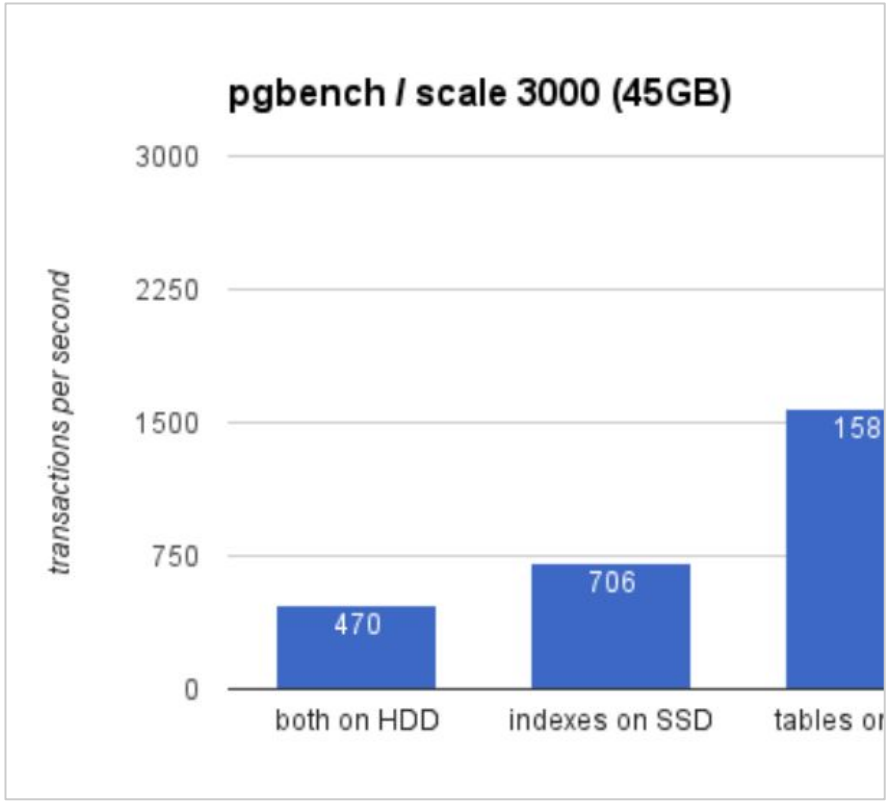
- Memory
- Persistent
- Hybrid



STORAGE TECHNOLOGIES

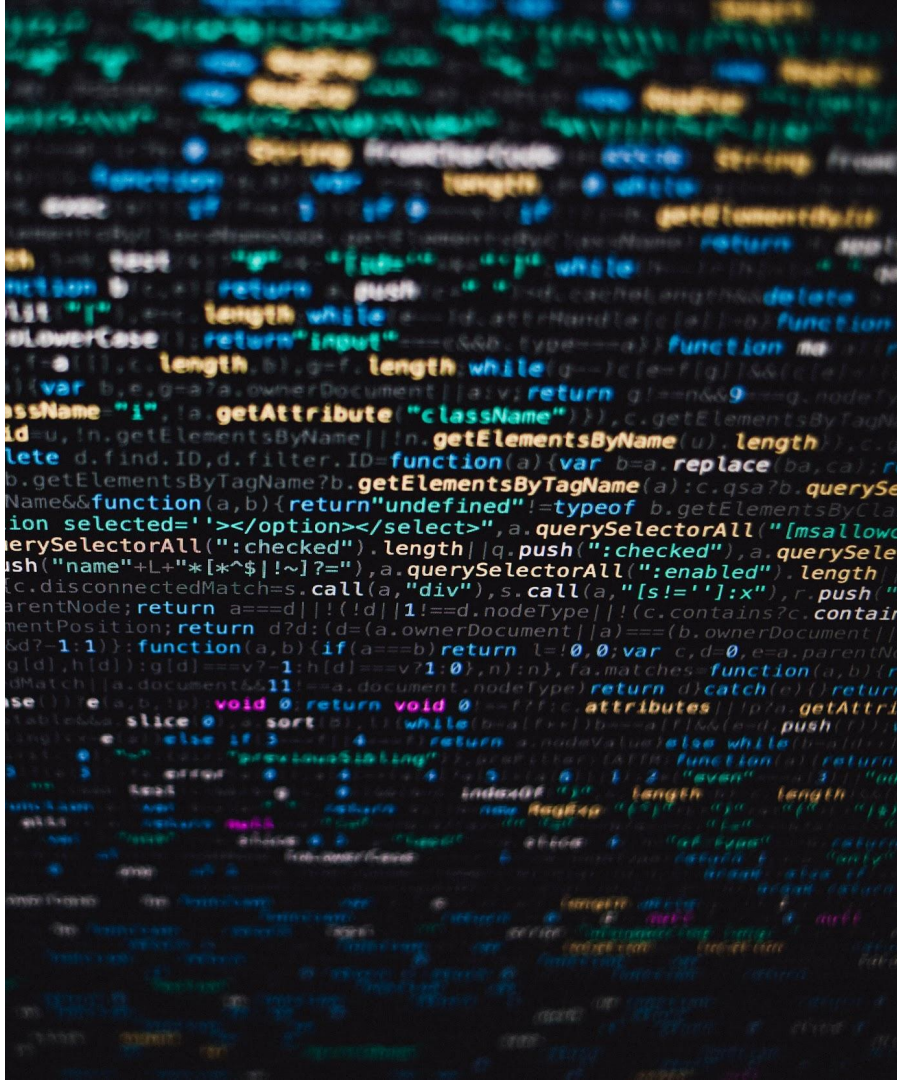
- In-Memory (**RAM**)
- Spinning Disks (**HDD**)
- Solid State Drives (**SSD**)

PostgreSQL comparison with different storage technologies



SCALABILITY 01

- Big Data
- Data Processing
- Machine Learning
- IoT



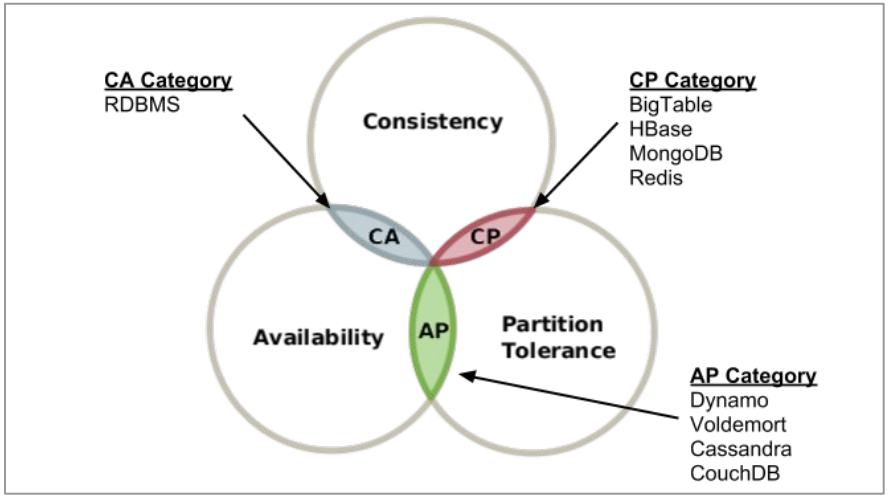
SCALABILITY 02

- BigTable
- Map/Reduce
- Horizontal scalability
- Sharding
- ...

FAILURE SCENARIOS

One failure each year for one server..

corresponds to one failure every day, for a system of 365 servers.



BASH DB

A minimal key-value store example in … bash

```
1   #!/bin/bash
2
3   db_set() {
4       echo "$1,$2" >> database
5   }
6
7   db_get() {
8       grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
9   }
10
11
```

Set

- Concatenate the key and value,
- Append the result on one line

Get

- Search for line prefix (key)
- Return the the value

EXPERIMENTING TOOLS

LACHESIS

A go repository containing key-value storage experiments
https://github.com/drakos74/lachesis

- Generic Storage interface
- Testing Suite
- Benchmarking Tools
- Visualisation tool

STORAGE INTERFACE

- Minimal
- Generic
- Storage factory

```go
// Storage is the low level interface
type Storage interface {
    Put(element Element) error
    Get(key Key) (Element, error)
}
```

```go
// Key identifies the byte arrays used a
type Key []byte

// Value identifies the byte arrays used
type Value []byte
```

```go
// StorageFactory generates a storage
type StorageFactory func() Storage
```

TESTING SUITE(S)

- Suite
- Test categories
- Operations

```go
// Suite is the testing suite base struct
type Suite struct {
    suite.Suite
    t          *testing.T
    limit      Limit
    newStorage store.StorageFactory
}
```

```go
// Consistency is the storage consistency test suite
type Consistency struct {
    Suite
}

// TestVoidReadOperation tests the given storage on a r
func (s *Consistency) TestVoidReadOperation() {
    storage := s.newStorage()
    VoidReadOperation(s.t, storage, checkMeta: false)
}
```

BENCHMARKING TOOLS

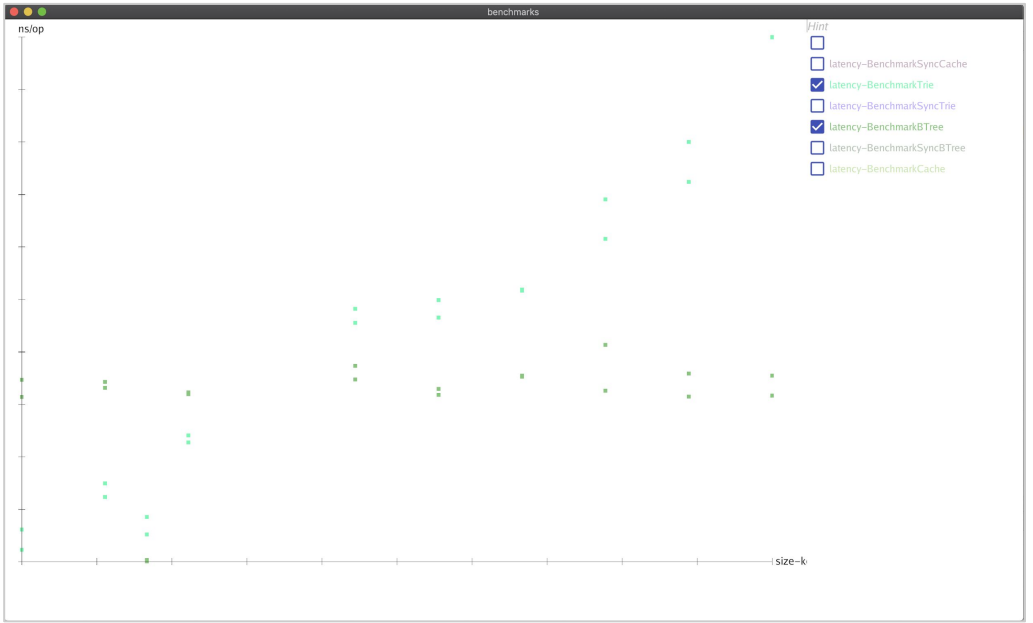
```
func TestComplexEvolutionScenario(t *testing.T) {

    scenario := Benchmark(Evolution().
        add(limit( iteration: 5)).
        add(num(pow( m: 10))).
        add(key(add( m: 5))).
        add(value(pow( m: 2))).
        create(),
         num: 2,  keySize: 2,  valueSize: 2)
```

```
// in-memory

func BenchmarkCache(b *testing.B) {
    executeBenchmarks(b, mem.CacheFactory)
}

func BenchmarkSyncCache(b *testing.B) {
    executeBenchmarks(b, mem.SyncCacheFactory)
}

func BenchmarkTrie(b *testing.B) {
    executeBenchmarks(b, mem.TrieFactory)
}

func BenchmarkSyncTrie(b *testing.B) {
    executeBenchmarks(b, mem.SyncTrieFactory)
}

func BenchmarkBTree(b *testing.B) {
    executeBenchmarks(b, mem.BTreeFactory)
}

func BenchmarkSyncBTree(b *testing.B) {
    executeBenchmarks(b, mem.SyncBTreeFactory)
}

// file

// BenchmarkFileStorage executes the benchmarks for the file storage
func BenchmarkFileStorage(b *testing.B) {
    executeBenchmarks(b, file.StorageFactory( path: "testdata/file"))
```

BENCHMARK VISUALISATION

```
file := flag.String( name: "file",  value: "benchmarks.txt",  usage: "")
...
oremi.Draw( title: "benchmarks",
    layout.Horizontal,
     width: 1400,
     height: 800,
    gatherBenchmarks(benchmarks))
```

```
collections := make(map[string]map[string]oremi.Collection)
collections["latency"] = make(map[string]oremi.Collection)
for label, benchmark := range graphs {
    collections["latency"][label] = benchmark.Extract(
        bench.Key,
        bench.Latency,
        bench.Include(map[string]float64{bench.Num: 1000}),
        bench.Exclude(map[string]float64{bench.Key: 16})).
        Color(colors.Get(label))
}
```



Gio

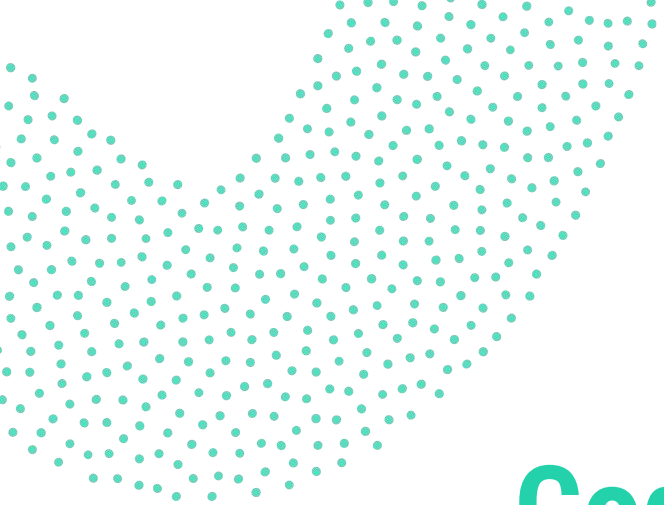
github/drakos74/oremi : a visualisation tool in go … based on [gio]

LET'S TRY IT OUT

- {IDE + go + internet}
- $ *git clone git@github.com:drakos74/lachesis.git*
- $ git checkout *key-value-storage*
- $ *go mod tidy*

- `$ cd lachesis/store/file/bash`
- `$ go test`
- `$ cd lachesis/store/benchmark`
- `go test -bench=. -benchmem`

`Go to … lachesis/store.benchmark/results`



Coding interlude ..

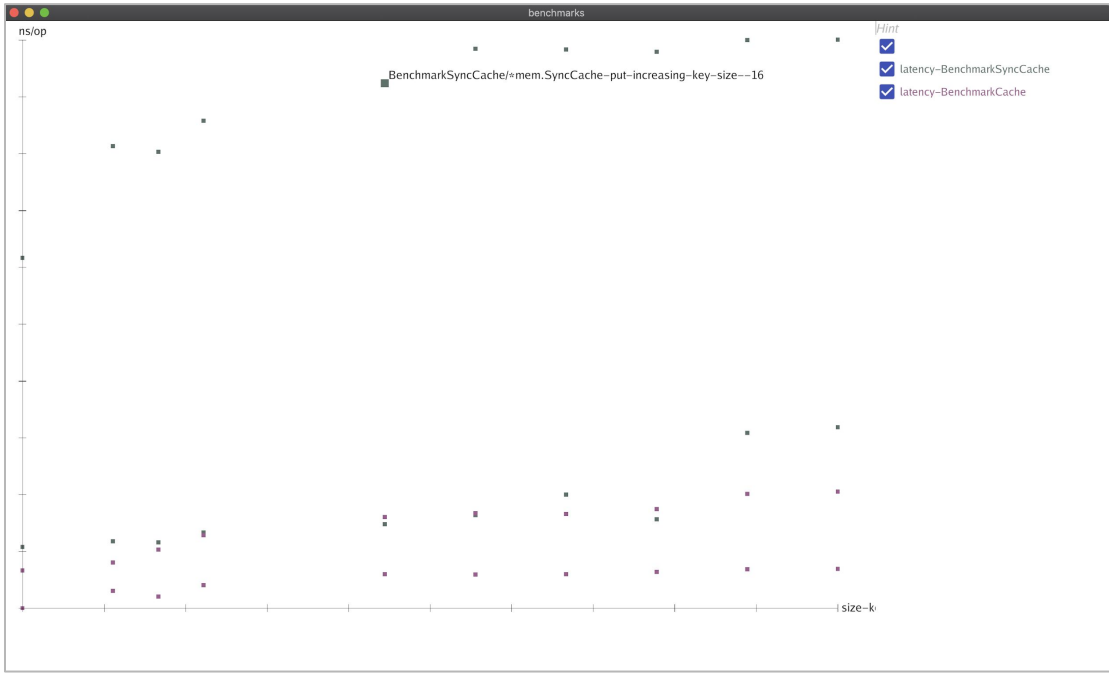
Get familiar with the testing and benchmarking tools

- Test bashDB
- Benchmark hashmap
- Benchmark Sync HashMap

{branch: *key-value-storage*}

Benchmark ‘map’

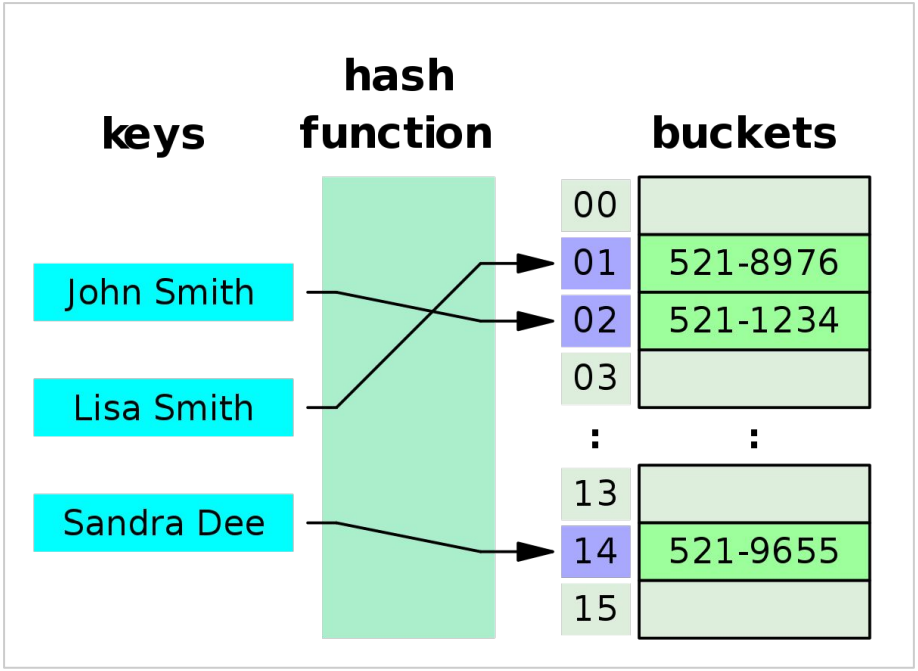
- Sync.Map in go is considerably slower than a simple map
- Especially for the ‘put’ operation
- another aspect is the handling of interface{} instead of concrete structs
- An extension of the ‘simple map’ with a RWLock is much faster … than sync.Map



INDEXING

HASHING

~ O(1) retrieval guarantee



QUERYING

Retrieve elements based on their properties rather than the key

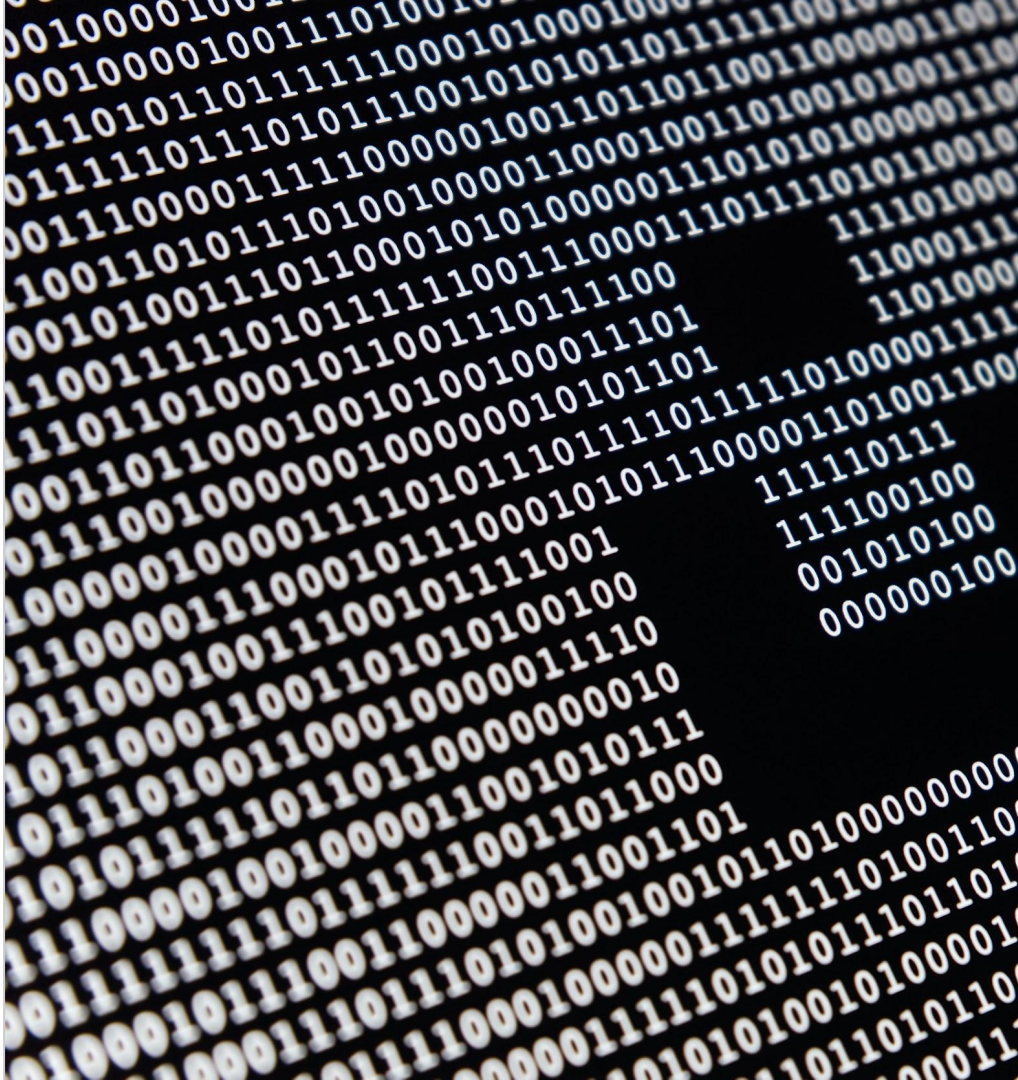
social-usage	i	fb	tw	t

RowKey	A|20151104
i:tw	@a
tw:10	180
tw:11	270
tw:12	240
t:tw	690

RowKey	B|20151104
i:tw	@b
i:fb	b
fb:10	600
fb:11	1200
fb:12	900
tw:10	90
t:fb	2700
t:tw	90

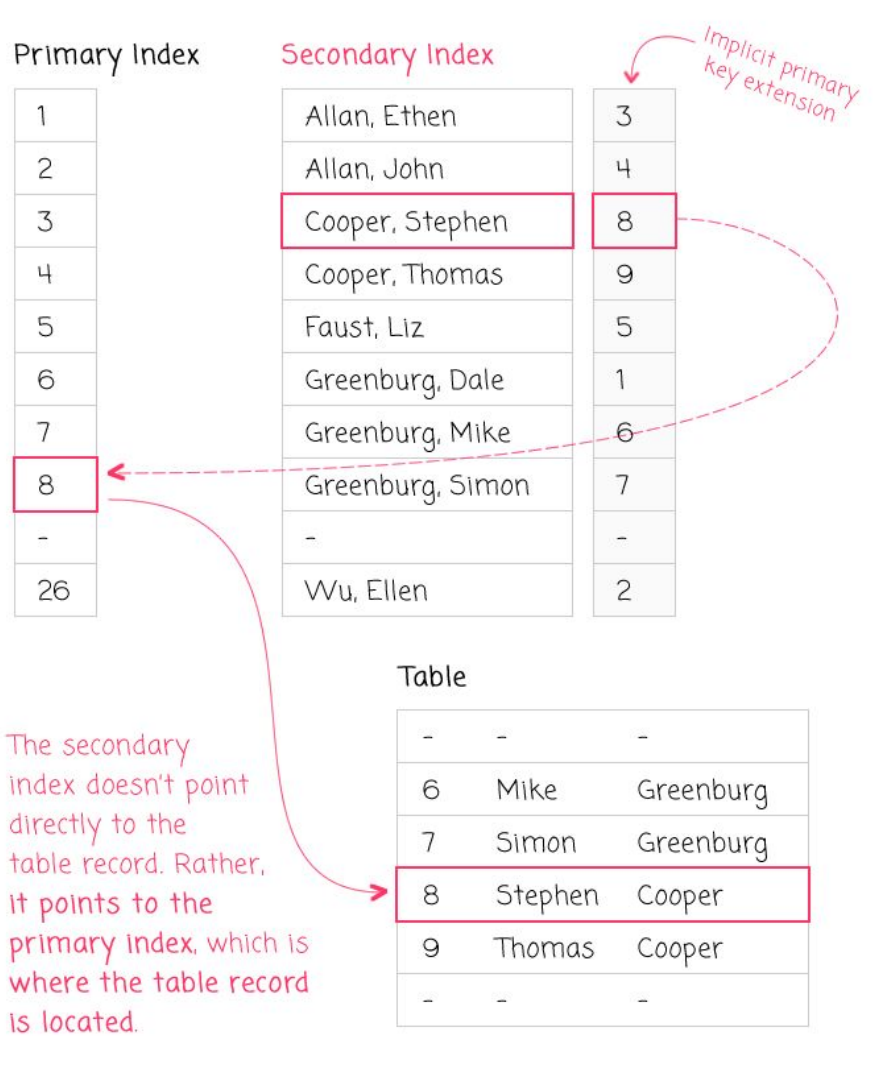
KEY SIZE

Key size affects querying performance ...randomly



INDEXES

Enhanced query features with minimal overhead



KEY DATA STRUCTURES

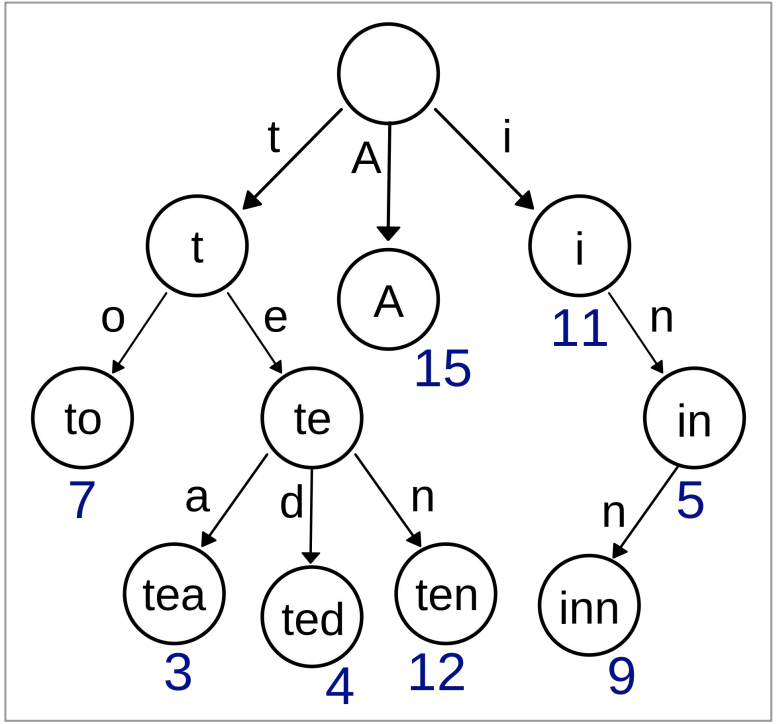
Given the above, a commonly used approach is to store indexes or the keys themselves in more performant data structures in terms of retrieval or query speed.

TRIE

An ordered tree data structure used to store a dynamic set or associative array.

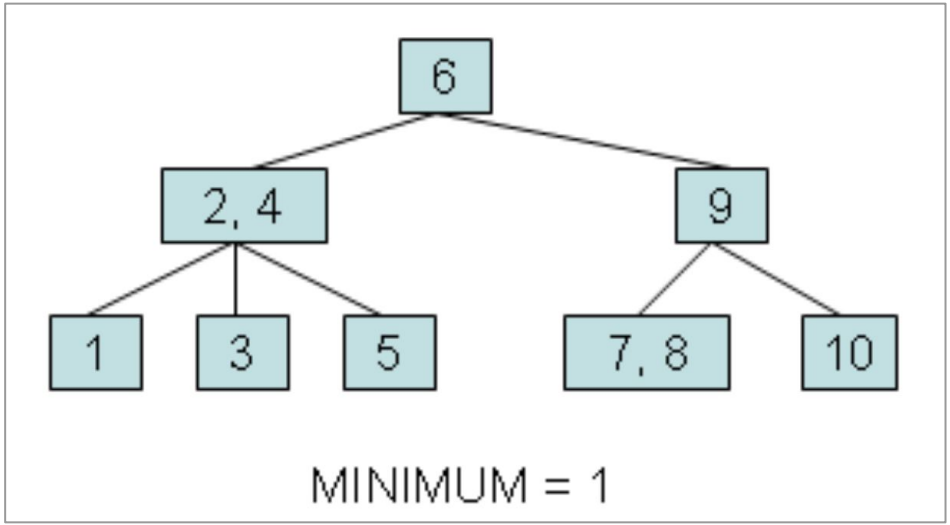
.. its position in the tree defines the key with which it is associated

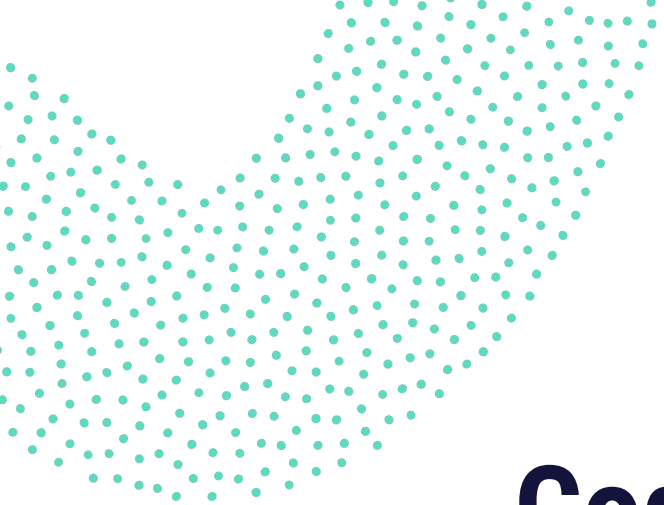
source: Wikipedia



B+ TREE

A self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time (source Wikipedia)





Coding interlude ..

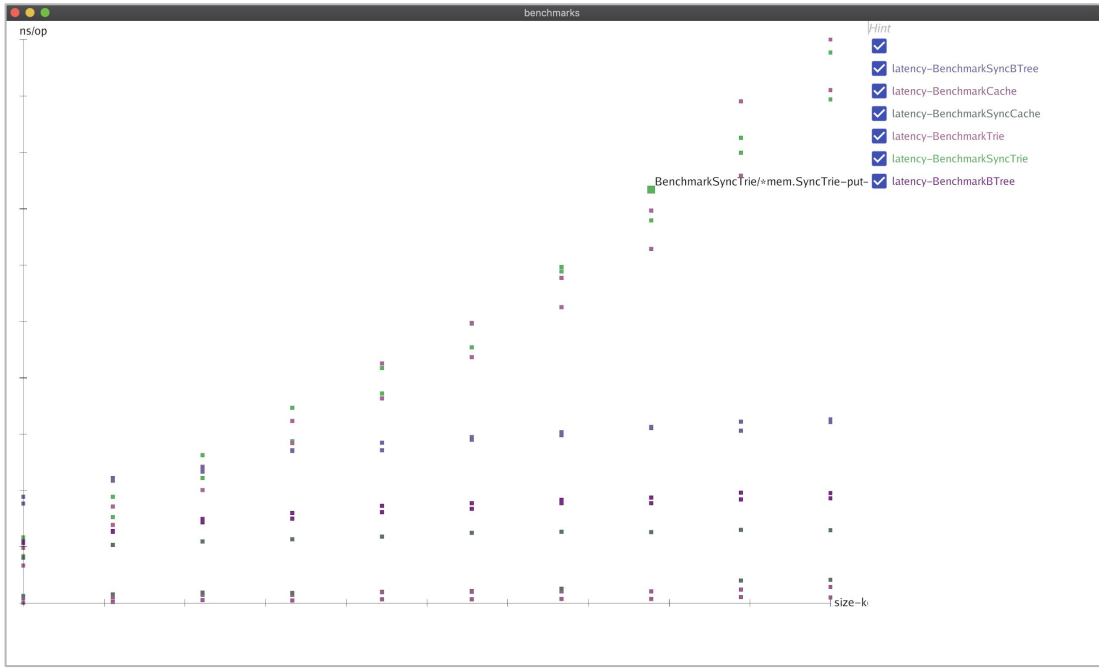
Investigate different indexing and keying structures

- Benchmark Trie
- Benchmark B+Tree

{branch: *key-value-indexes***}**

Trie v.s. BTree

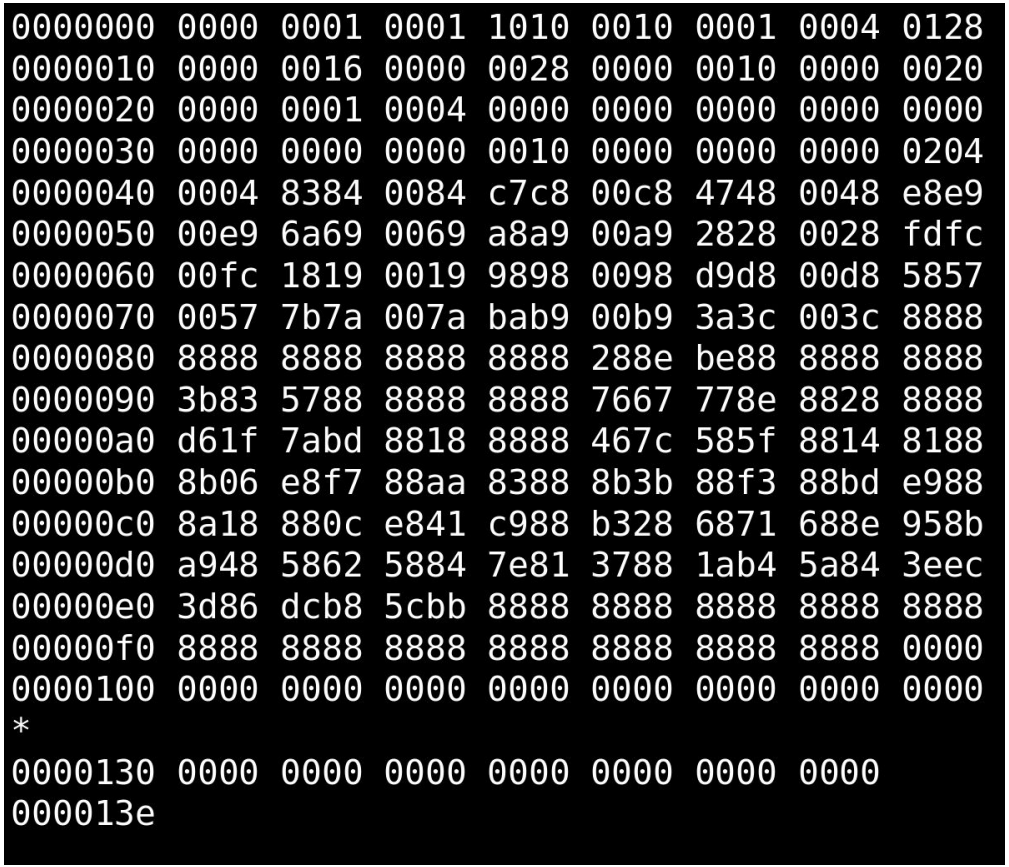
- A Trie performance degrades fast based on the key size
- A BTree is much more consistent in terms lookup speed for different keys
- For very small keys (~4 bytes) a Trie seems to perform better



PERSISTENT STORAGE

FILES AND BYTES

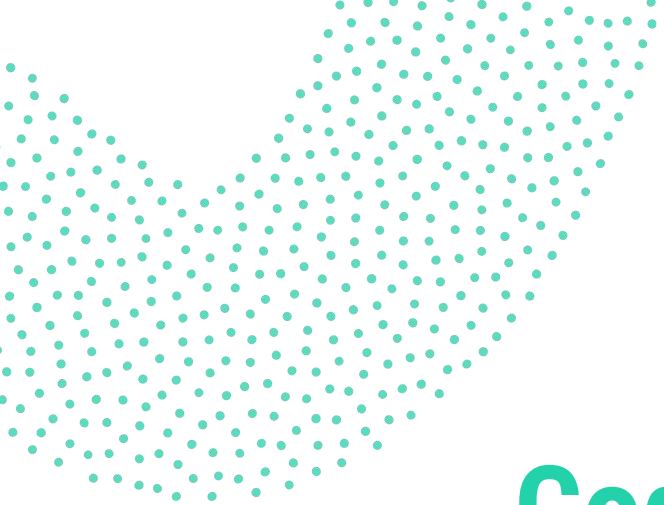
- Network
- I/O
- De-/Serialisation



PERFORMANCE

Performance trade-off options

- Keep files open
- Batch operations
- Use small files
- ...



Coding interlude ..

Experiment with file storage trade-offs

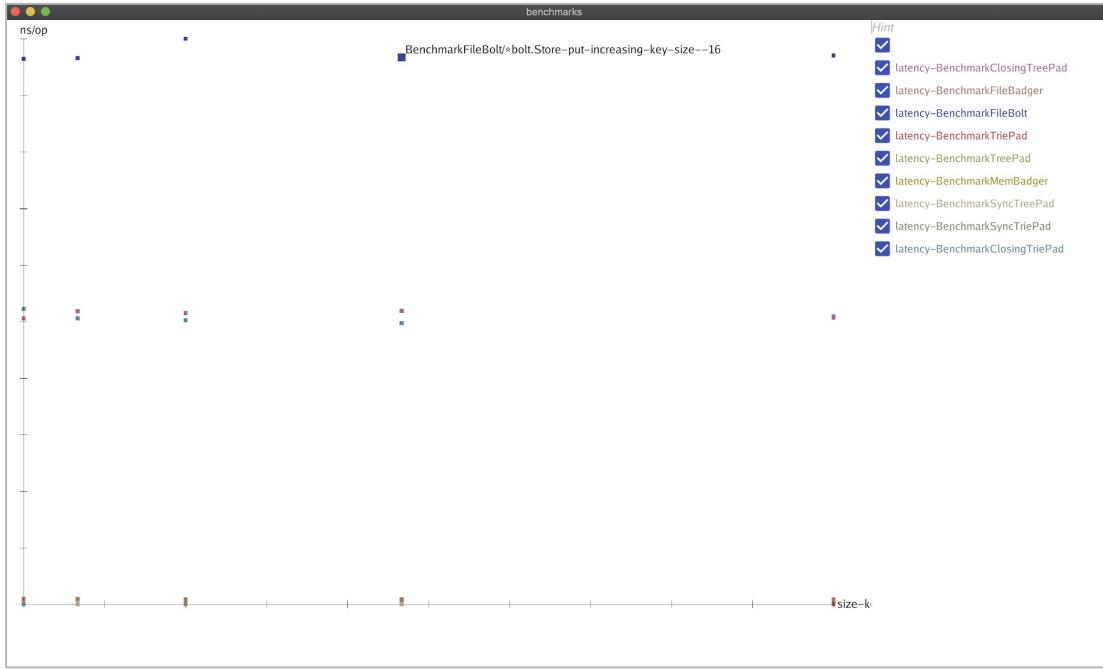
- Benchmark File storage with Trie
- Benchmark File storage with B+Tree
- Benchmark File storage without closing files
- Benchmark File storage with closing files

{branch: *key-value-file-storage*}

Who won ?

- Bolt performs worst in terms of writes (compared to Badger)

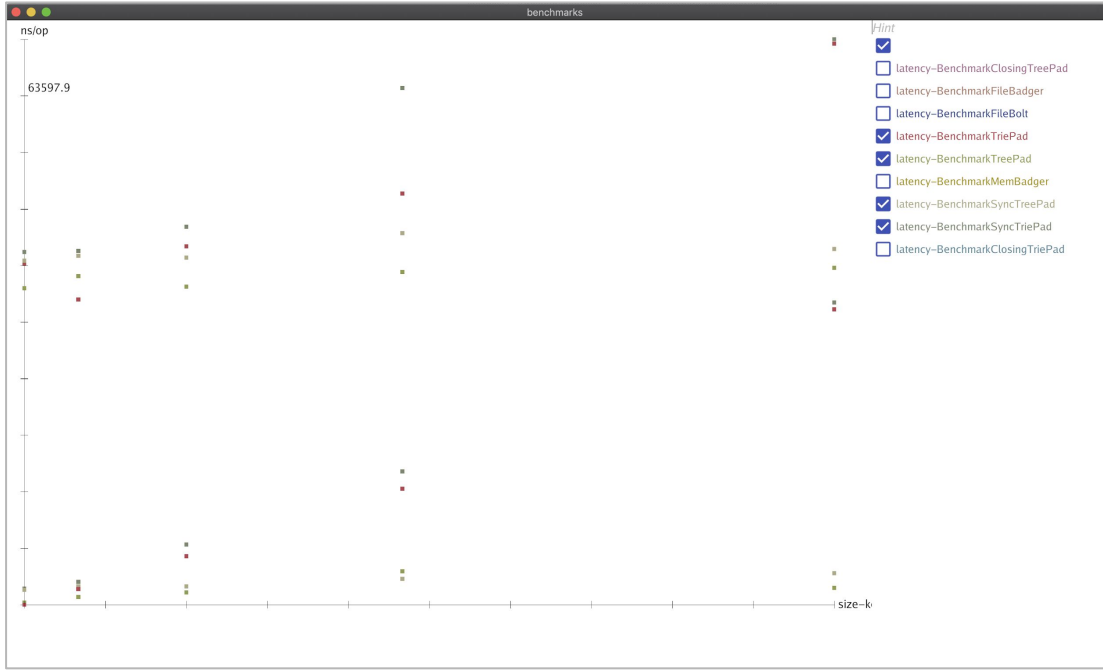
This makes sense, given that it's an *ordered* store. We would expect it to perform best for time-series range queries.



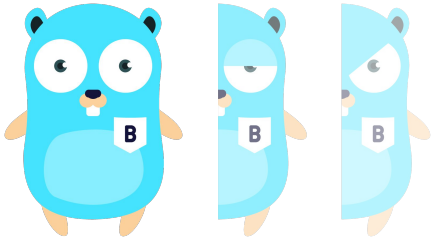
Who won ?

- The *non-closing* file storage performs almost as good as the simple map implementation.

But … we would need to handle file corruption scenarios.



SPLITTING THE WORK

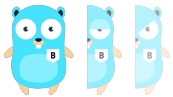


NETWORK SEGMENTATION

The Network struct implements the Storage interface

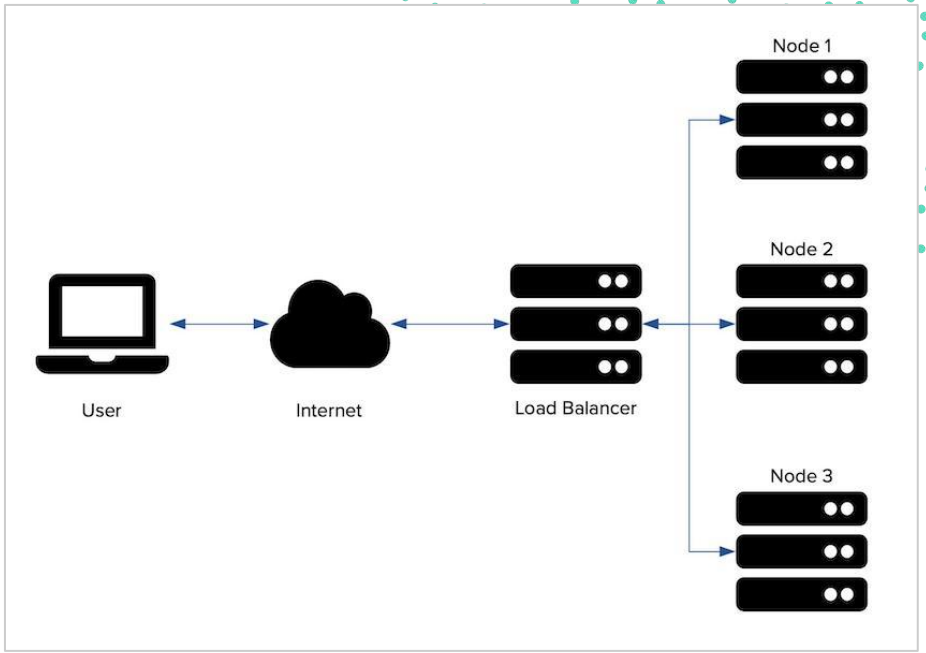
```
// Network emulates a cluster
// the network has a Storage f
type Network struct {
    Switch
    WorldClock
    nodes []Storage
    cnl   func()
}
```

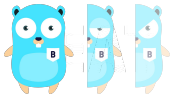
```
// apply properties to the members of the network
for i := 0; i < f.parallelism; i++ {
    node := f.nodeFactory(f.storage, f.protocol)

    // listen to internal cluster events
    go func() {...}()

    // listen to client events
    go func() {...}()

    // register node to the network interface
    route.Register(len(nodes))
    nodes = append(nodes, node)

}
```



SPLIT THE WORK

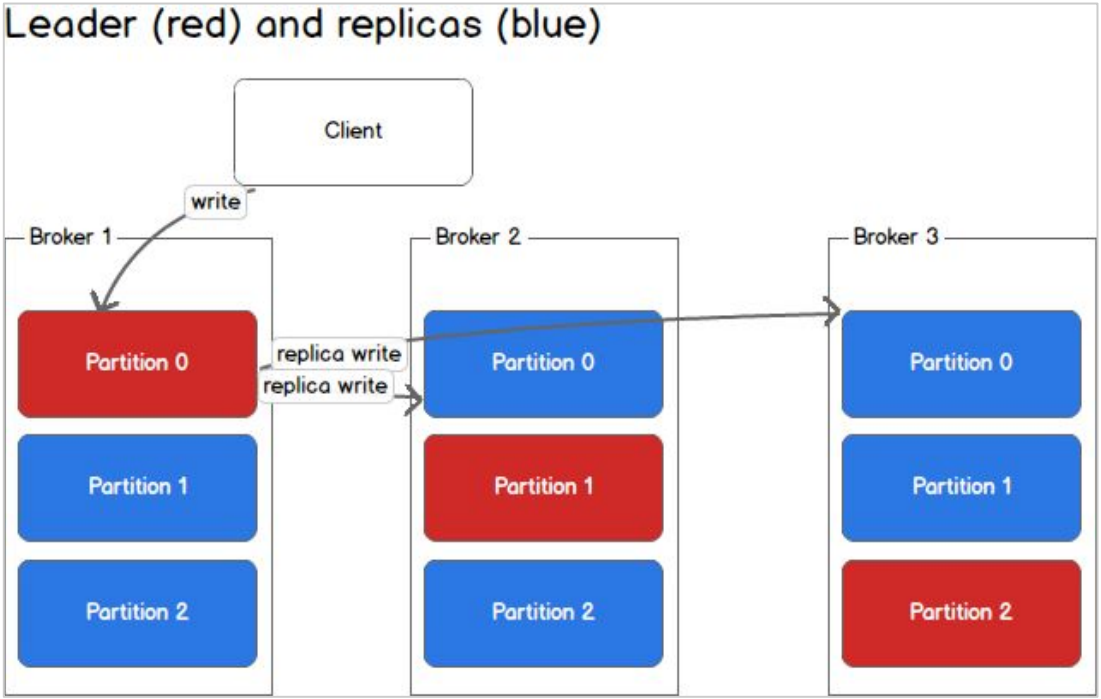
- Network switch / router
- Sharding
- Hashing



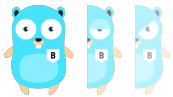


RESILIENCE TO FAILURE

Multiple copies of the same data to comply with the (high) availability guarantees



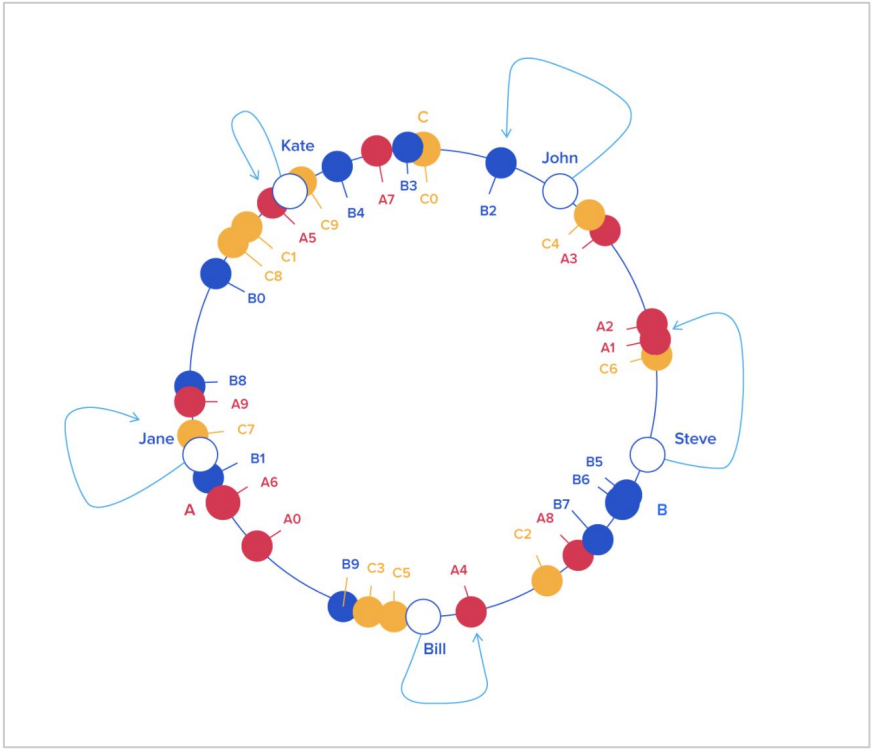
Kafka partitioning example

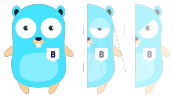


SHARE THE LOAD

Creating replicas needs a consistent routing algorithm

- Leader-Follower
- Consistent Hashing
- ...



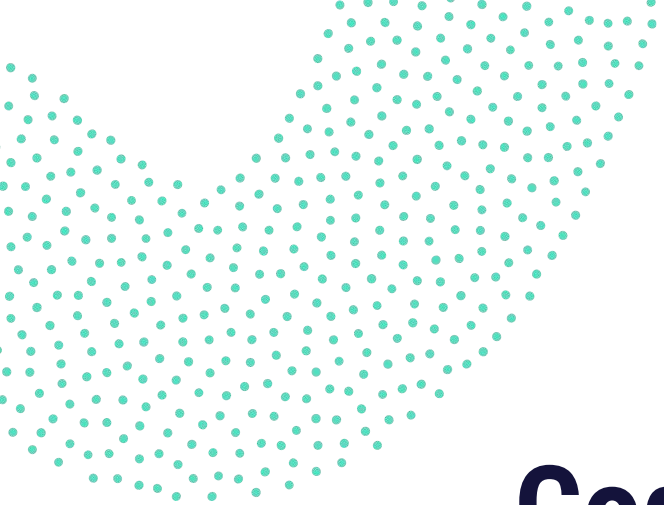


FAILURE SCENARIO(S)

Emulating failures by decorating at the network ‘Switch’ level

```
// NodeDown emulates the
type NodeDown struct {
    index      int
    iterations int
    duration   int
    Switch
}
```

```
// Route returns the node responsible for serving the current request
func (u *NodeDown) Route(key Key) ([]int, error) {
    ids, err := u.Switch.Route(key)
    liveIds := make([]int, 0)
    for _, id := range ids {
        if u.index >= 0 && u.index == id && u.iterations < u.duration {
            // we need to ignore this one e.g. node is down
            u.iterations++
        } else {
            liveIds = append(liveIds, id)
        }
    }
```



Coding interlude ..

Test the failure scenarios for different approaches

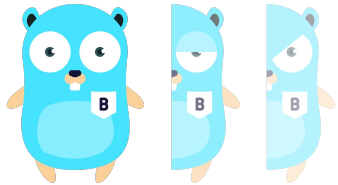
- Test single node cluster
- Test leader-follower
- Test sharding
- Test consistent hashing

{branch: *key-value-storage-split-work***}**

How to split ?

- Randomly ?
- Sharding ?
- Hashing ?
- Leader-Follower ?
- Consistent hashing ?

DISTRIBUTED CONSENSUS



2 PHASE COMMITS

- Prepare
- Commit

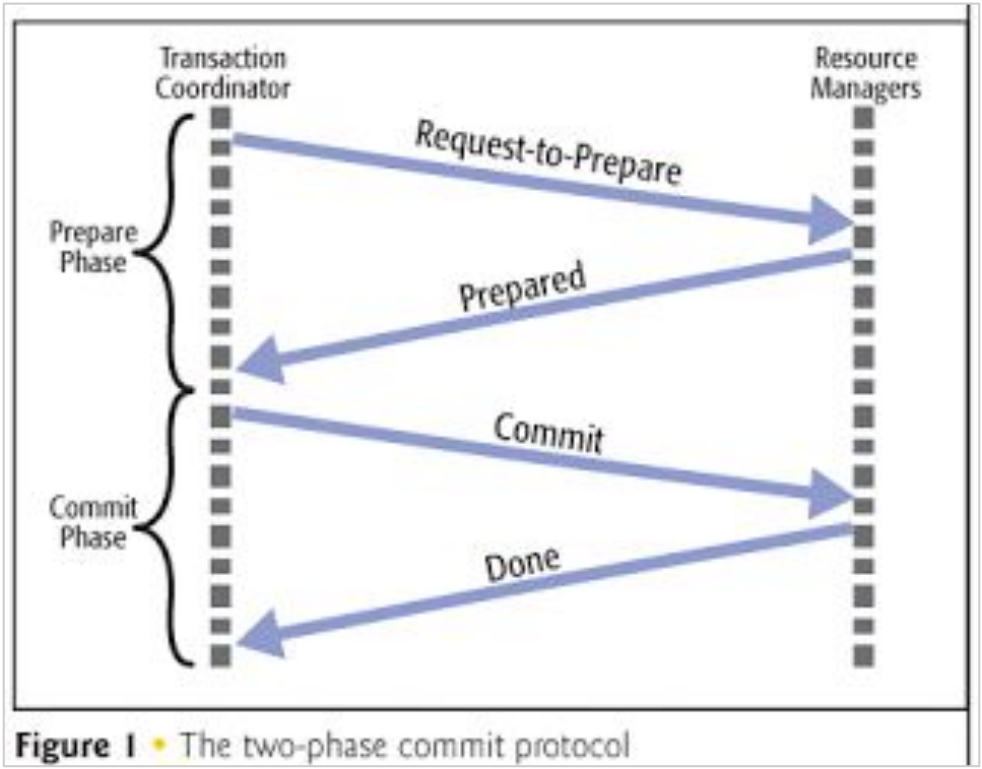


Figure 1 • The two-phase commit protocol

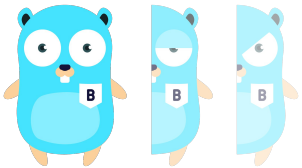
WRITE AHEAD LOG

Prepare the operations in a Log
… ready to commit and act on them.

EQUALITY VS AUTHORITY

- **Coordinator**
  Leader-Follower

- **Distributed**
  Peer-to-peer consensus



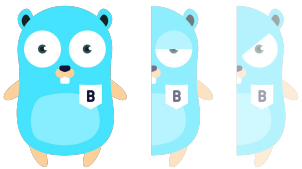
PAXOS

- **Proposer (phase1)**
  Prepare RPC call for a **key**

- **Acceptor (phase1)**
  Returns value for **key**

- **Proposer (phase2)**
  Checks the **values**
  (these can be the same or empty)

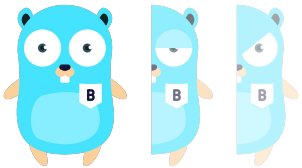
- **Acceptor (phase2)**
  Commits the proposed **value**



RAFT

- **Leader**
  Append-RPC call with **commit index** (phase 1)
  Commit-RPC call at index (phase 2)

- **Follower**
  Appends to Log at **commit index** (phase 1)
  Commits to Store (phase 2)



PROOF OF WORK…?

e.g. Bitcoin etc ...

- Would it make sense ?
  - What would this help with ?
  - What drawbacks would this produce ?

PROOF OF STAKE…?

… leader-election (?)

… hashing (?)

INTER-NODE COMMUNICATION

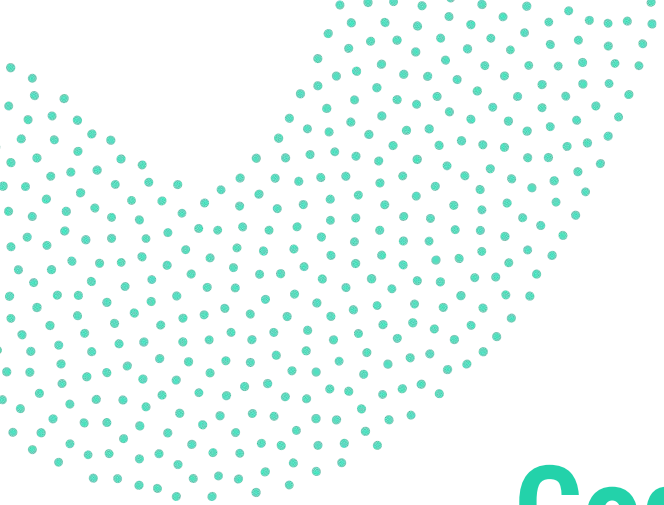
Emulating Inter-Node **`communication`** through central channels

```
// emulate the node internal protocol communication layer
go func() {
    for msg := range node.Cluster().Internal.out {
        // ignore empty messages
        if msg != Void {
            for _, n := range nodes {
                if msg.Source != n.Cluster().Internal.ID &&
                    (msg.RoutingID == 0 ||
                        n.Cluster().Internal.ID == msg.RoutingID) {
                    n.Cluster().Internal.in <- msg
                }
            }
        }
    }
}()
```

```
// Message represents an inte
// messages are used for the
type Message struct {   Katik
    ID        uint32
    Source    uint32
    RoutingID uint32
    Type      MsgType
    Signal    Signal
    Content   interface{}
    Err       error
}
```

PROTOCOL DEFINITION

```
// Protocol implements the internal cluster communication requirements,
// e.g. the proposers and acceptors interaction logic
func Protocol() network.ProtocolFactory {

    processor := network.ProcessorFactory(func(state *network.State, node *network.StorageNode, element store.Element) (rpc interface{}, wait bool) {
        ...
    })

    // follower phase 1 processing logic
    processor.Propose(func(state *network.State, storage store.Storage, msg interface{}) (interface{}, error) {
        ...
    })

    // leader phase1 processing logic
    processor.Promise(func(state *network.State, storage store.Storage, msg interface{}) (interface{}, error) {
        ...
    })

    // follower phase 2 processing logic
    processor.Commit(func(state *network.State, storage store.Storage, msg interface{}) (interface{}, error) {
        ...
    })

    // leader phase 2 confirmation from followers
    processor.Confirm(func(state *network.State, storage store.Storage, msg interface{}) (interface{}, error) {
        ...
    })

    return network.ConsensusProtocol(*processor)
}
```



Coding interlude ..

Test the failure scenarios for …
PAXOS & RAFT

{branch: *key-value-consensus***}**

Paxos v.s. ...

```
func newPaxosNetwork(event ...network.Event) store.StorageFactory {
    return network.Factory(event...).
        Router(lb.RandomPartition).
        Storage(mem.CacheFactory).
        Nodes( parallelism: 10).
        Protocol(Protocol()).
        Node(network.Node).
        Create()
}
```

- Paxos is simpler
  - No leader
  - Better distribution
- Paxos adds complexity for edge-case scenarios
  - Breaking ties
  - Network partitioning recovery
- Paxos is difficult to implement
  - … and test

... v.s. Raft

```
func newRaftNetwork(event ...network.Event) store.StorageFactory {
    return network.Factory(event...).
        Router(lb.LeaderFollowerPartition).
        Storage(mem.SyncCacheFactory).
        Nodes( parallelism: 10).
        Protocol(Protocol()).
        Node(network.Node).
        Create()
}
```

- Raft is fairly simple
  - Simple way to break ties (everyone follows the leader)
  - Simple way of synchronising / ordering calls (no need for system clock)
- Raft adds much more overhead on the leader
  - Although one can distribute the leadership based on hashing
  - Leader-Election overhead

WRAP UP

IT'S ALL ABOUT THE SPECS

- ✓ It's fairly straightforward to implement a key-value storage API.
- ✓ The complexity increases as one adds more guarantees for its usage.
- ✓ There are design options that behave well for some access models, but worse for others.
- ✓ Some industry solutions try to cover "everything" , some others choose to solve a single or only few well defines use-case(s)

LESSONS LEARNT

- Identify the requirements
- Read the docs
- Understand the trade-offs
- Design to switch for when the *requirements change*
- … *because most probably they will*

REFERENCES.01

- **Storage**

  https://www.freecodecamp.org/news/the-pros-and-cons-of-different-data-formats-key-values-vs-tuples-f526ad3fa964/

  https://blog.westerndigital.com/data-availability-vs-durability/#:~:text=Durability%2C%20on%20the%20other%20hand,is%20never%20lost%20or%20compromised.

- **Big Data**

  https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

- **Distributed Systems**

  https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

  https://bowenli86.github.io/2016/07/04/distributed%20system/zookeeper/ZooKeeper-Consistency-Guarantees/

- **Kafka partitioning**

  https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

- **Consistent hashing**

  https://www.toptal.com/big-data/consistent-hashing

- **Go datastores benchmark**

  https://dgraph.io/blog/post/badger-lmdb-boltdb/

- **SSD vs HDD**

  https://www.2ndquadrant.com/en/blog/tables-and-indexes-vs-hdd-and-ssd/

REFERENCES.02

- **Etcd**

  https://etcd.io/docs/v3.4.0/dev-guide/limit/

- **HBase**

  https://mapr.com/blog/guidelines-hbase-schema-design/

- **MongoDB**

  https://docs.mongodb.com/manual/core/write-performance/

- **Gio UI**

  https://gioui.org/

- **Paxos**

  https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

- **Raft**

  https://raft.github.io/

- **2 Phase Commits**

  https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222

THANK YOU!

Thanks to
the Beat Marketing & Design Team(s)
For all their help and support with the presentation

Vangelis Katikaridis | Sr. Backend Engineer

github.com/drakos74
Twitter: @vangkos

BEAT