Designing and building a distributed data store in Go

3 February 2018

Matt Bostock
Who am I?

Platform Engineer working for Cloudflare in London.

Interested in distributed systems and performance.
Building and designing a distributed data store
What I will (and won't) cover in this talk

MSc Computer Science final project
TIMBALA
DURABLE TIME-SERIES DATABASE
It ain't production-ready

Please, please, don't use it yet in Production if you care about your data.
What's 'distributed'?

"A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages."

-- Wikipedia
Why distributed?

Survive the failure of individual servers

Add more servers to meet demand
Fallacies of distributed computing

The network is reliable.
Latency is zero.
Bandwidth is infinite.
The network is secure.
Topology doesn't change.
There is one administrator.
Transport cost is zero.
The network is homogeneous.
Use case

Durable long-term storage for metrics
Why not use 'the Cloud'?

- On-premise, mid-sized deployments
- High performance, low latency
- Ease of operation
Requirements
Sharding

The database must be able to store more data than could fit on a single node.
Replication

The system must replicate data across multiple nodes to prevent data loss when individual nodes fail.
High availability and throughput for data ingestion

Must be able to store a lot of data, reliably
Operational simplicity
Interoperability with Prometheus

Reuse Prometheus' best features

Avoid writing my own query language and designing my own APIs

Focus on the 'distributed' part
By the numbers

Cloudflare's OpenTSDB installation (mid-2017):

- 700k data points per second
- 70M unique timeseries
Minimum Viable Product (MVP)?
How to reduce the scope?

Reuse third-party code wherever possible
Milestone 1: Single-node implementation

Ingestion API

Query API

Local, single node, storage
Milestone 2: Clustered implementation

1. Shard data between nodes (no replication yet)

2. Replicate shards

3. Replication rebalancing using manual intervention
Beyond a minimum viable product

Read repair

Hinted handoff

Active anti-entropy
To the research!

NUMA
Data/cache locality
SSDs
Write amplification
Alignment with disk storage, memory pages
mmap(2)
Jepsen testing
Formal verification methods
Bitmap indices
xxHash, City hash, Murmur hash, Farm hash, Highway hash
Back to the essentials

Coordination

Indexing

On-disk storage format

Cluster membership

Data placement (replication/sharding)

Failure modes
Traits (or assumptions) of time-series data
Immutable data

No updates to existing data!

No need to worry about managing multiple versions of the same value and copying (replicating) them between servers
Simple data types; compress well

Don't need to worry about arrays or strings

Double-delta compression for floats

Tension between write and read patterns

Continuous writes across majority of individual time-series

Occasional reads for small subsets of time-series across historical data

Writing a Time Series Database from Scratch (https://fabxc.org/tsdb/)
Prior art

Amazon's Dynamo paper

Apache Cassandra

Basho Riak

Google BigTable

Other time-series databases
Coordination

Keep coordination to a minimum

Avoid coordination bottlenecks
Cluster membership

Need to know which nodes are in the cluster at any given time

Could be static, dynamic is preferable

Need to know when a node is dead so we can stop using it
Memberlist library

I used Hashicorp's Memberlist library

Used by Serf and Consul

SWIM gossip protocol
Indexing
Could use a centralised index

Consistent view; knows where each piece of data should reside

Index needs to be replicated in case a node fails

Likely to become a bottleneck at high ingestion volumes

Needs coordination, possibly consensus
Could use a local index

Each node knows what data it has
Data placement (replication/sharding)
Consistent hashing

Hashing uses maths to put items into buckets

Consistent hashing aims to keep disruption to a minimum when the number of buckets changes
Consistent hashing: example

n = nodes in the cluster

1/n of data should be displaced/relocated when a single node fails

Example:

- 5 nodes
- 1 node fails
- one fifth of data needs to move
Consistent hashing algorithms

Decision record for determining consistent hashing algorithm (https://github.com/mattbostock/timbala/issues/27)
Consistent hashing algorithms

First attempt: Karger et al (Akamai) algorithm


Second attempt: Jump hash


Jump hash implementation

```go
func Hash(key uint64, numBuckets int) int32 {
    var b int64 = -1
    var j int64

    for j < int64(numBuckets) {
        b = j
        key = key*2862933555777941757 + 1
        j = int64(float64(b+1) * (float64(int64(1)<<31) / float64((key>>33)+1)))
    }

    return int32(b)
}
```

Partition key

The hash function needs some input

The partition key influences which bucket data is placed in

Decision record for partition key (https://github.com/mattbostock/timbala/issues/12)
Replicas

3 replicas (copies) of each shard

Achieved by prepending the replica number to the partition key
On-disk storage format

Log-structured merge
LevelDB
RocksDB
LMDB
B-trees and b-tries (bitwise trie structure) for indexes
Locality-preserving hashes
Use an existing library

Prometheus TSDB library  
(https://github.com/prometheus/tsdb)

Cleaner interface than previous Prometheus storage engine

Intended to be used as a library

Writing a Time Series Database from Scratch  
(https://fabxc.org/tsdb/)
Architecture

No centralised index (only shared state is node metadata)

Each node has the same role

Any node can receive a query

Any node can receive new data

No centralised index, data placement is determined by consistent hash
Testing

- Unit tests
- Acceptance tests
- Integration tests
- Benchmarking
Unit tests
Data distribution tests

How even is the distribution of samples across nodes in the cluster?

Are replicas of the same data stored on separate nodes?
Distribution of samples when replication factor is 3 across a cluster of 5 nodes:

Node 0: ######### 19.96%; 59891 samples
Node 1: ######### 19.99%; 59967 samples
Node 2: ########## 20.19%; 60558 samples
Node 3: ######### 19.74%; 59212 samples
Node 4: ########## 20.12%; 60372 samples

Summary:
Min: 59212
Max: 60558
Mean: 60000.00
Median: 59967
Standard deviation: 465.55
Total samples: 300000

Distribution of 3 replicas across 5 nodes:
0 nodes: 0.00%; 0 samples
1 nodes: 0.00%; 0 samples
2 nodes: 0.00%; 0 samples
3 nodes: #FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 100.00%; 100000 samples

Replication summary:
Min nodes samples are spread over: 3
Max nodes samples are spread over: 3
Mode nodes samples are spread over: [3]
Mean nodes samples are spread over: 3.00
Data displacement tests

If I change the cluster size, how much data needs to move servers?

```plaintext
=== RUN   TestHashringDisplacement
293976 unique samples
At most 19598 samples should change node
15477 samples changed node

293976 unique samples
At most 21776 samples should change node
16199 samples changed node
--- PASS: TestHashringDisplacement (4.33s)
```
Data displacement failure

Too much data was being moved because I was sorting the list of nodes alphabetically.
Jump hash gotcha

"Its main limitation is that the buckets must be numbered sequentially, which makes it more suitable for data storage applications than for distributed web caching."

Jump hash works on buckets, not server names

Conclusion: Each node needs to remember the order in which it joined the cluster
Acceptance tests

Verify core functionality from a user perspective
Integration tests

Most effective, least brittle tests at this stage in the project

Some cross-over with acceptance tests

Docker compose for portability, easy to define
Benchmarking

Benchmarking harness using Docker Compose
pprof

`go tool pprof`

or

`go get github.com/google/pprof`

Go Diagnostics (https://tip.golang.org/doc/diagnostics.html)
pprof CPU profile

pprof --dot http://localhost:9080/debug/pprof/profile | dot -T png | open -f -a /Applications/Preview.app
Gauging the impact of garbage collection

GOGC=off

[golang.org/pkg/runtime/](https://golang.org/pkg/runtime/)
Microbenchmarks

$ go test -benchmem -bench BenchmarkHashringDistribution -run none ./internal/cluster
goos: darwin
goarch: amd64
pkg: github.com/mattbostock/timbala/internal/cluster
BenchmarkHashringDistribution-4    2000000    954 ns/op    544 B/op    3 allocs/op
PASS
ok      github.com/mattbostock/timbala/internal/cluster    3.303s

[golang.org/pkg/testing/#hdr-Benchmarks](https://golang.org/pkg/testing/#hdr-Benchmarks)
Failure injection

Stop nodes

Packet loss, re-ordering, latency using tc (Traffic Control)

www.qualimente.com/2016/04/26/introduction-to-failure-testing-with-docker/
Conclusions

- Greatest challenge in distribution systems is anticipating how they will fail and lose data
- Make sure you understand the tradeoffs your Production systems are making
Use dep, it's awesome 🤘

github.com/golang/dep (https://github.com/golang/dep)
More information

- **Timbala architecture documentation** ([https://github.com/mattbostock/timbala/blob/master/docs/architecture.md](https://github.com/mattbostock/timbala/blob/master/docs/architecture.md))
- **Designing Data-Intensive Systems** ([https://dataintensive.net/](https://dataintensive.net/))
- **OK Log blog post** ([https://peter.bourgon.org/ok-log/](https://peter.bourgon.org/ok-log/))
- **Notes on Distributed Systems for Young Bloods** ([https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/](https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/))
- **Achieving Rapid Response Times in Large Online Services** ([https://www.youtube.com/watch?v=1-3Ah7Fxsc](https://www.youtube.com/watch?v=1-3Ah7Fxsc))
- **Jepsen distributed systems safety research** ([https://jepsen.io/talks](https://jepsen.io/talks))
- **Writing a Time Series Database from Scratch** ([https://fabxc.org/tsdb/](https://fabxc.org/tsdb/))
- **Failure testing with Docker** ([https://www.qualimente.com/2016/04/26/introduction-to-failure-testing-with-docker/](https://www.qualimente.com/2016/04/26/introduction-to-failure-testing-with-docker/))
- **SWIM gossip protocol paper** ([https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf](https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf))
Thank you

Matt Bostock
@mattbostock (http://twitter.com/mattbostock)