AEROSPIKE

SUMMIT '19

SIGNAL

Rebuilding on a Strong Foundation: from Cassandra to Aerospike, One Year On

Jason Yanowitz
EVP, Chief Technology Officer
Signal Digital

Overview of talk

- **Introduction (2 min) ← We’re here**
- **Cassandra to Aerospike (10 min)**
- **The next generation of our data model (26 min)**

Overview of talk

- Introduction (2 min)
- **Cassandra to Aerospike (10 min) ← We’re here**
- The next generation of our data model (26 min)

Cassandra failures and mitigations, a non-random sampling

- **We mutated most of our data. This makes Cassandra sad.**
  - Moved to async writes to ensure it wasn't on the critical path for reads
- **Couldn't handle batch workloads**
  - Build moar and bigger rings
- **Couldn't maintain data quality and performance**
  - More nodes -> more gossip -> more overhead
  - Reduce quorum requirements on reads for some use cases
- **Found data was stochastically globally replicated**
  - Hire consultants. Run Repairs.
  - Towards the end, we couldn't even run repair successfully ($10k in data xfer/day!)
- **Too many rings. So many $.**
  - Added a cache in front of it for some use cases

Executing the migration

- **There are three major components to prepare**
  - Changing our app
  - Becoming operationally facile with Aerospike
  - Migrating the data (Aerospike Client Services)
- **Actual mechanics**
  - Snapshot Cassandra
  - Run Cassandra -> Aerospike Tool (Juggernaut)
  - Enter Dual write mode (catchup on buffered writes)
  - Test, test, test
  - Move read traffic to Aerospike, one region at a time

Main Takeaways from the Migration

- **What went well**
  - Advice from others
  - Time to completion—from contract signed to live in production was ~100 days (~3 staff years of labor, including testing)
  - Scope stayed largely stable
  - Test datasets, test servers
  - Paying Aerospike Client Services to do the throwaway work
  - > 66% OpEx savings (servers, storage, data transfer)

- **Areas for Improvement**
  - Focused on Juggernaut performance before correctness of migration logic
  - Ran migration 2.5 times
  - Testing after migration was hard

Overview of talk

- Introduction (2 min)
- Cassandra to Aerospike (10 min)
- **The next generation of our data model (26 min) ← We're here**

Measure n Times, Cut Once (n > 1)

- **~6 weeks, 3 engineers, 1 product person to ask:**

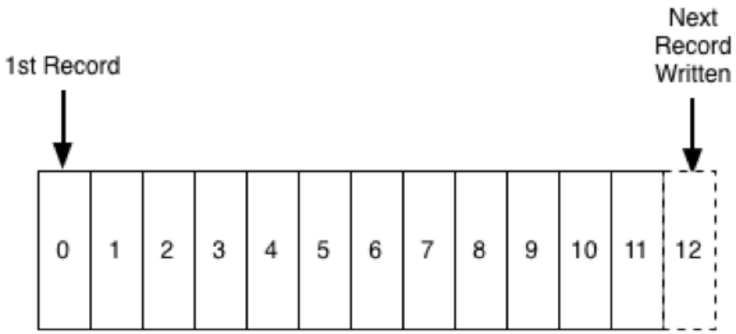
“Can Signal systematically simplify its data model in 2019, given staffing, product, financial and engineering constraints?”

- **Answer: Yes!**
- **DB Learnings (YMMV!)**
  - AWS Neptune
  - Neo4j
- **Design learnings**
  - Event sourcing
  - Logical Monotonicity via Conflict-Free Replicated Data Types (CRDTs)

SIGNAL | AEROSPIKE SUMMIT ‘19

Event Sourcing

- **Model all data changes as events and persist them in an immutable log**
- **Solves**
  - Provenance questions
    - Where did this data come from?
    - Why does it look this way?
  - How data changes over time
  - Point-in-time recovery



- **Does not solve the biggest headaches of distributed systems**
  - CAP theorem (Consistency, Availability, Partition-Tolerance) concerns
  - At-least once messaging semantics
  - Performance loss from coordination (if it’s even possible)

Moving Towards Operational Sanity

- **CALM: Consistency as Logical Monotonicity. Work by Hellerstein, Alvaro *et al***
- **Allows for deterministic outcomes on top of non-deterministic systems**

“Does the program produce the outcome we expect despite any race conditions that might arise?”— Hellerstein and Alvaro, 2019

- **Models semantics of minimizing coordination in distributed systems**

"In many cases, however, coordination is not a necessary evil, it is an incidental requirement of a design decision.” — Hellerstein and Alvaro, 2019

But will it blend?

- **In terms of expressiveness, if you can provide a total ordering for events, logical monotonicity can solve all problems in PTIME!**
- **The CAP theorem was a negative result; an impossibility proof.**
- **CALM goes the other way and carves out the set of “which programs can be consistently computed while remaining available under partition.”**

CRDTs – type of logical monotonicity

- **Relatively recent (~10 years ago)**
- **Data structures that meet three criteria***
  - Associative $(A \otimes B) \otimes C = A \otimes (B \otimes C)$
  - Commutative $A \otimes B = B \otimes A$
  - Idempotent $A \otimes A = A$
- **Two distribution methods**
  - State-based
  - Op-based
- **As with any system that is CALM, these are *Strongly Eventually Consistent***
  - **Strong** in a mathematical sense

* Pedantry Disclaimer: There are other parts of the formal definition, but this is the important part for our purposes.

Why CRDTs for us?

Op	Current System			Proposed Schema with CRDT structures		
	Associative	Commutative	Idempotent	Associative	Commutative	Idempotent
Write identifier vertex	🚫	🚫	🚫	✓	✓	✓
Write vertex ctime / mtime	N/A	N/A	N/A	✓	✓	✓
Write edge	🚫	🚫	🚫	✓	✓	✓
Write device vertex	🚫	🚫	✓	✓	✓	✓
Write device ctime / mtime	🚫	🚫	✓	✓	✓	✓
Delete vertex	🚫	🚫	🚫	✓	✓	✓
Delete edge	🚫	🚫	🚫	✓	✓	✓

But how to do this? An Easy Example: Grow Only Set

- **Supports two operations**
  - Add(X)
  - Present?(X)
- **Add(1), Add(2), Present?(1) => true, Present?(3) => false, Add(1)**
  - Elements: {1,2}
- **Is it a CRDT?**
  - Associative ({1} + {2}) + {3} = {1} + ({2} + {3}) = {1,2,3}
  - Commutative {1} + {2} = {2} + {1} = {1,2}
  - Idempotent {1} + {1} = {1}

How to delete?

How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), **Delete(x)**, Present?(x)

```
Adds:        { }
Deletes:     { }
```


How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**

```
Adds:         { 1, 2 }
Deletes:      { }
```


How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**
- **Present?(2) `# => true`**

```
Adds:          { 1, 2 }
Deletes:       {  }
```


How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**
- **Present?(2) `# => true`**
- **Delete(2)**

```
Adds:       { 1, 2 }
Deletes:    { 2 }
```

How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**
- **Present?(2) `# => true`**
- **Delete(2)**
- **Present?(2) `# => false`**

```
Adds:           { 1, 2 }
Deletes:        { 2 }
```


How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**
- **Present?(2) # `=> true`**
- **Delete(2)**
- **Present?(2) # `=> false`**
- **Add(2)**

```
Adds:           { 1, 2 }
Deletes:        { 2 }
```


SIGNAL | AEROSPIKE SUMMIT '19

How to delete?

- **Keep 2 Grow Only Sets**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x), Delete(x), Present?(x)
- **Add(1), Add(2)**
- **Present?(2) # `=> true`**
- **Delete(2)**
- **Present?(2) # `=> false`**
- **Add(2)**
- **Present?(2) # `=> false`**

```
Adds:          { 1, 2 }
Deletes:       { 2 }
```

 | Proprietary & Confidential |

What to do?

- **We haven't captured causality, so values latch**
- **We need partial ordering (*happens before* relationship)**
- **Vector clocks would work but are a hassle**
- **We're gonna cheat**

What to do?

- **We haven't captured causality, so values latch**
- **We need partial ordering (*happens before*)**
- **Vector clocks would work but are a hassle**
- **We're gonna cheat**
- **Operations**
  - Add(X, **time**)
  - Delete(X, **time**)
  - Present?(X)

How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,**time**), Delete(x,**time**), Present?(x)

```
Adds:          { }  # this is now a Map
Deletes:       { }  # so is this
```


How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
- **Present?(2) `# => true`**

```
Adds:          { 1: 103, 2: 103 }
Deletes:       { }
```


How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
- **Present?(2)** `# => true`
- **Add(1,100)**

```
Adds:       { 1: 103, 2: 103 }
Deletes:    { }
```


How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
- **Present?(2) `# => true`**
- **Delete(2,102)**

```
Adds:          { 1: 103, 2: 103 }
Deletes:       { 2: 102 }
```

 | Proprietary & Confidential |

SIGNAL | AEROSPIKE SUMMIT '19

How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
- **Present?(2) # `=> true`**
- **Delete(2,102)**
- **Present?(2) # `=> true`**

```
Adds:           { 1: 103, 2: 103 }
Deletes:        { 2: 102 }
```


SIGNAL | AEROSPIKE SUMMIT '19

How to delete?

- **Keep 2 Grow Only Sets, together, they form a Last Write Wins (LWW) Set**
  - [AddGrowOnlySet. DeleteGrowOnlySet]
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
- **Present?(2) # `=> true`**
- **Delete(2,102)**
- **Present?(2) # `=> true`**
- **Delete(2,104)**
- **Present?(2) # `=> false`**

```
Adds:          { 1: 103, 2: 103 }
Deletes:       { 2: 104 }
```


I lied. A little.

- **We can collapse the two sets into a single one**
  - GrowOnlySet of `{ element: (time, visible?) }`
  - Add(x,time), Delete(x,time), Present?(x)
- **Add(1,103), Add(2,103)**
  - `{ 1: (103, true), 2: (103, true) }`
- **Present?(2) `# => true`**
- **Delete(2,104)**
  - `{ 1: (103, true), 2: (104, false) }`
- **Present?(2) `# => false`**

SIGNAL | AEROSPIKE SUMMIT '19

Back to our problem…

- **Using CALM design principles we can build a DGAF (Distributed Graphs Are Fun) system**
- **What's a graph?**
  - Vertices = LWWSet()
  - Edges = LWWSet()
  - Graph = [Vertexes, Edges]
- **What's a vertex?**
  - (id, data)
- **What's an edge?**
  - (vertexId0, vertexId1)

How to do this in Aerospike (v4)

- **Namespace: Vertices**
- **ids are a tuple (org, identifierType, identifierValue)**

Bins:

```
id:                [org, idType, idVal]# same as PK for the record
```


How to do this in Aerospike (v4)

- **Namespace: Vertices**
- **ids are a tuple (org, identifierType, identifierValue)**
- **vtime is a specially crafted unsigned long**
  - First 63 bits are nanoseconds since epoch time, last bit is `recordVisible?`

Bins:

```
id:          [org, idType, idVal]# same as PK for the record
vtime:       [vtime]             # guess why this is a list
```


How to do this in Aerospike (v4)

- **Namespace: Vertices**
- **ids are a tuple (org, identifierType, identifierValue)**
- **vtime is a specially crafted unsigned long**
  - First 63 bits are nanoseconds since epoch time, last bit is `recordVisible?`
- **edges are, conceptually, tuples: (vertexId0, vertexId1)**
  - This record is always vertexId0, so we only need to keep track of the other vertexIds

Bins:

```
id:            [org, idType, idVal]        # same as PK for the record
vtime:         [vtime]                     # it's actually a sorted list
edges:         { otherVertexId: [vtime] } # LWWSet of otherVertexIds
```


How to do this in Aerospike (v4)

- **Namespace: Vertices**
- **ids are a tuple (org, identifierType, identifierValue)**
- **vtime is a specially crafted unsigned long**
  - First 63 bits are nanoseconds since epoch time, last bit is `recordVisible?`
- **edges are, conceptually, tuples: (vertexId0, vertexId1)**
  - This record is always vertexId0, so we only need to keep track of the other vertexIds
- **eventIds help reconcile against the immutable event store.**

Bins:

```
id:          [org, idType, idVal]       # same as PK for the record
vtime:       [vtime]                    # sorted list of length 1
edges:       {otherVertexId: [vtime]}   # LWWSet of otherVertexIds
eventIds:    ['eventId1'…'eventIdN']
```


SIGNAL | AEROSPIKE SUMMIT '19

How to do a write

- **Let's leverage Aerospike's design for performing atomic updates on records.**
- **This exploits one of the design strategies of CALM—push coordination to the smallest possible bound.**
- **Obvious approach: use a UDF (Lua) that defines the necessary business logic**
- **Clever approach: nested operations on Complex Data Types (CDTs).**
  - Keep vtimes as an ordered list of length 1 and trim to the largest value.
  - Adding an edge is now adding to the ordered list and removing the lowest value

Bins:

```
id:           [org, idType, idVal]        # same as PK for the record
vtime:        [vtime]                     # always trimmed to highest value
edges:        {otherVertexId: [vtime]}    # LWWSet of otherVertexIds
eventIds:     ['eventId1'…'eventIdN']
```


What about GC?

- **How do you ever purge everything from these sets?**
- **You define a “quiesce” time—the maximum amount of time you expect it to take for a message to get processed**
  - I’d pick a reasonable multiple of your expected MTTR
  - Let’s pick 3 days because it straddles a weekend and seems absurd.
- **Now you need to find what needs to be GC’d**
  - Scan the db in place
  - ETL the db and use a data warehouse
- **Define a new event type for GC and emit them for every deleted item that’s sufficiently old**
  - GC events apply **iff** vtime on the item in Aerospike matches vtime in the event you emit

What’s next?

- We’re in the midst of Phase 1 which will build out the main pieces of the pipeline, stand it up in a “shadow” mode and provide immediate business value for analytics
- Then we’ll roll through additional phases, each delivering incremental business value and improving our facility with these abstractions and new operational requirements.
- None of this could happen without having a much more stable data plane (i.e., Aerospike) .

Questions?
Comments.
Disagreements!

