Sharding:
DIY or Out of the Box Solution?

October 21, 2020

By Art van Scheppingen
art.vanscheppingen@messagebird.com

Agenda

- MessageBird & me
- Short sharding primer
- DIY sharding project
- Vitess sharding project
- DIY shards vs Vitess shards
- Conclusion

MessageBird & me

Who am I and who is MessageBird?

MessageBird

MessageBird is a cloud communications platform that empowers consumers to communicate with your business in the same way they communicate with their friends - seamlessly, on their own timeline and with the context of previous conversations.

For additional information visit:
www.messagebird.com

245+ Agreements
We have 245+ direct-to-carrier agreements with operators worldwide.

20,000+ Customers
Customers in over 60+ countries, across a great variety of industries.

350+ Employees
More than 350 employees speaking over 20 languages based in the Americas, Europe & Asia.

We’re hiring!

https://messagebird.com/careers

Short sharding primer

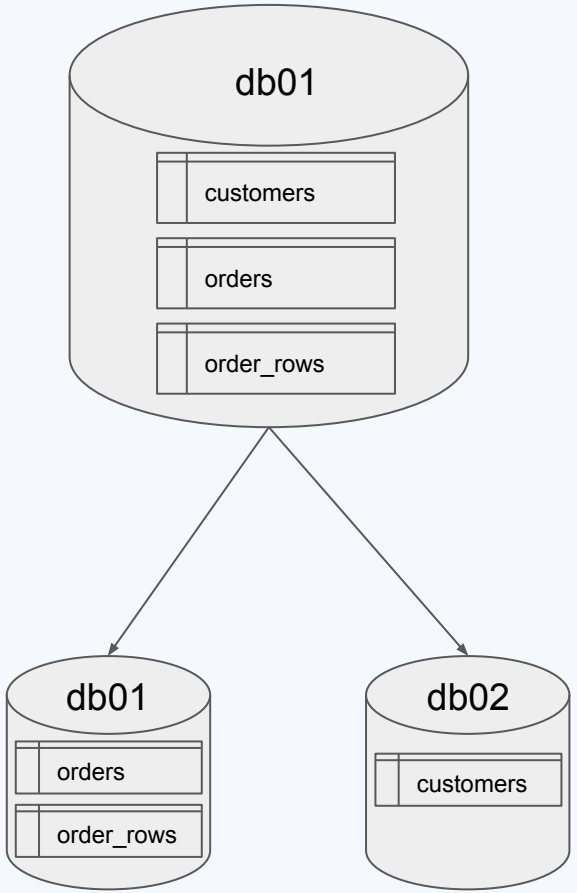
Just to be sure we’re all on the same page

What is (database) sharding?

- Breaking up a database into multiple smaller parts
- Not the same as a table partitioning:
  - Partitions are kept on the same database host
- Two types of sharding:
  - Functional sharding
  - Horizontal sharding

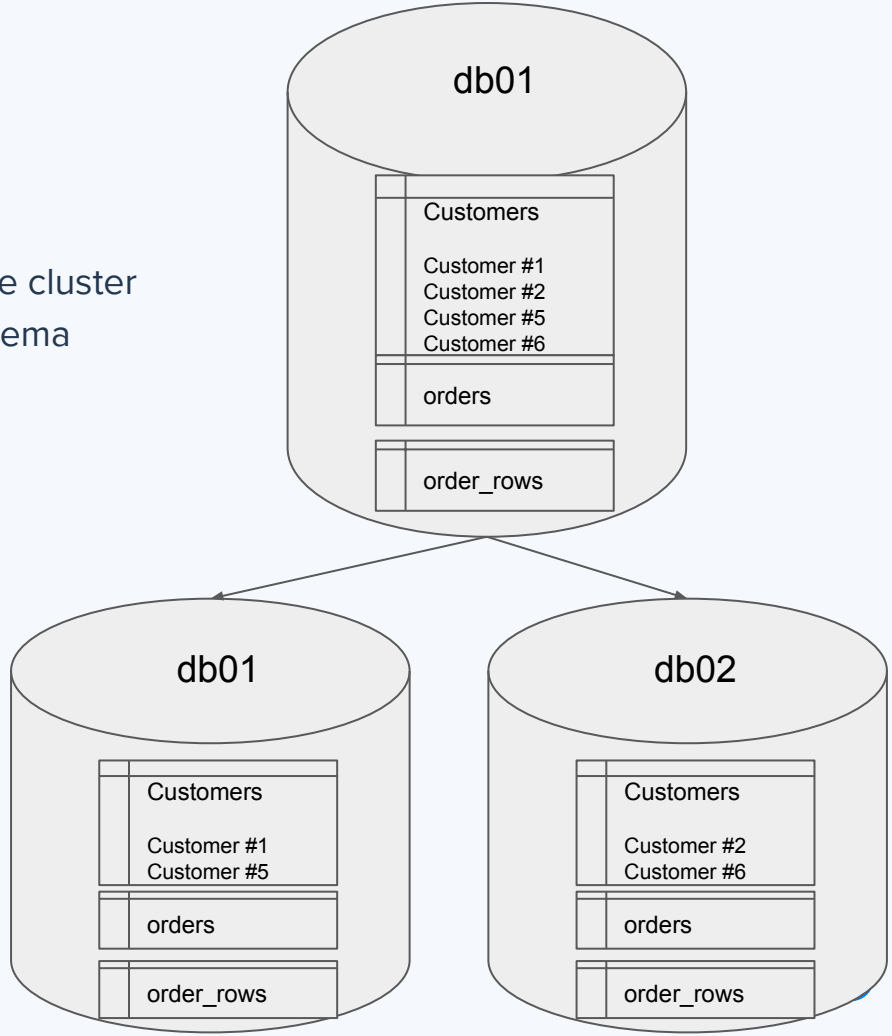
What is functional sharding?

- Functional sharding (a.k.a. Poor Man's Sharding)
  - Split a database by function
  - E.g. move "customers" table to a different database cluster
  - Typically shard schema contains with 1 or 2 tables
- Allows scaling to function
  - Vary in cpu/memory/diskspace
- No cross shard queries possible
  - You can't join customers and orders table
  - Two separate queries are necessary
- Data integrity
  - Foreign keys are not possible between shards
  - Integrity can't be guaranteed
- Shard outage
  - Makes part of your data inaccessible



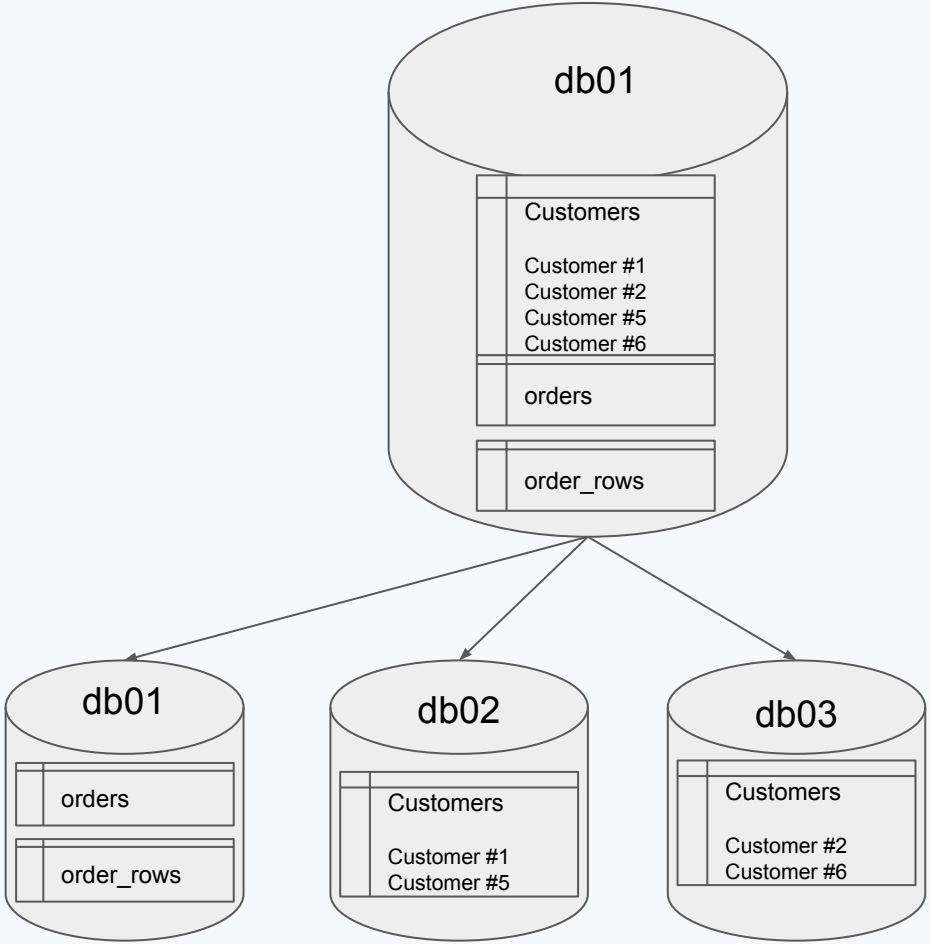
What is horizontal sharding?

- Horizontal sharding (a.k.a. True Sharding)
  - Split a database upon data
  - E.g. move “customer-X” to a different database cluster
  - Typically multiple identical shards with full schema
- Algorithm on a value in the data
  - Identifier, Creation date, (Geographic) location
- Dynamic sharding
  - Database to store pointers
- Allows scaling on even resources
  - Same number of CPUs, memory, etc
- No cross shard queries possible
  - You can’t select all rows from the customers table
- Data duplication
  - Certain tables need to be duplicated
  - Shared records need to be duplicated



Why not do both?

- Adds more complexity
- Less data duplication
- Allows scaling to function
- Allows scaling on even resources
    - Reshard when necessary



DIY Shards

Our own sharding solution

Problematic replication on short-term storage database

- Database primary is able to persist all messages
  - Sent from the MP workers in parallel
- Database replicas can’t keep up
  - Replication on MySQL is single threaded
  - Scaling up is no option (no faster CPUs in GCE)
  - 40% YoY growth

Possible solutions

- Write less
- Write smarter
- Enable parallel replication
- Shard our data

Possible solutions

- Write less: ***not possible***
- Write smarter: ***multi-insert statements requires code overhaul***
- Enable parallel replication: ***requires database migration***
- Shard our data: ***requires database migration (and connection logic)***

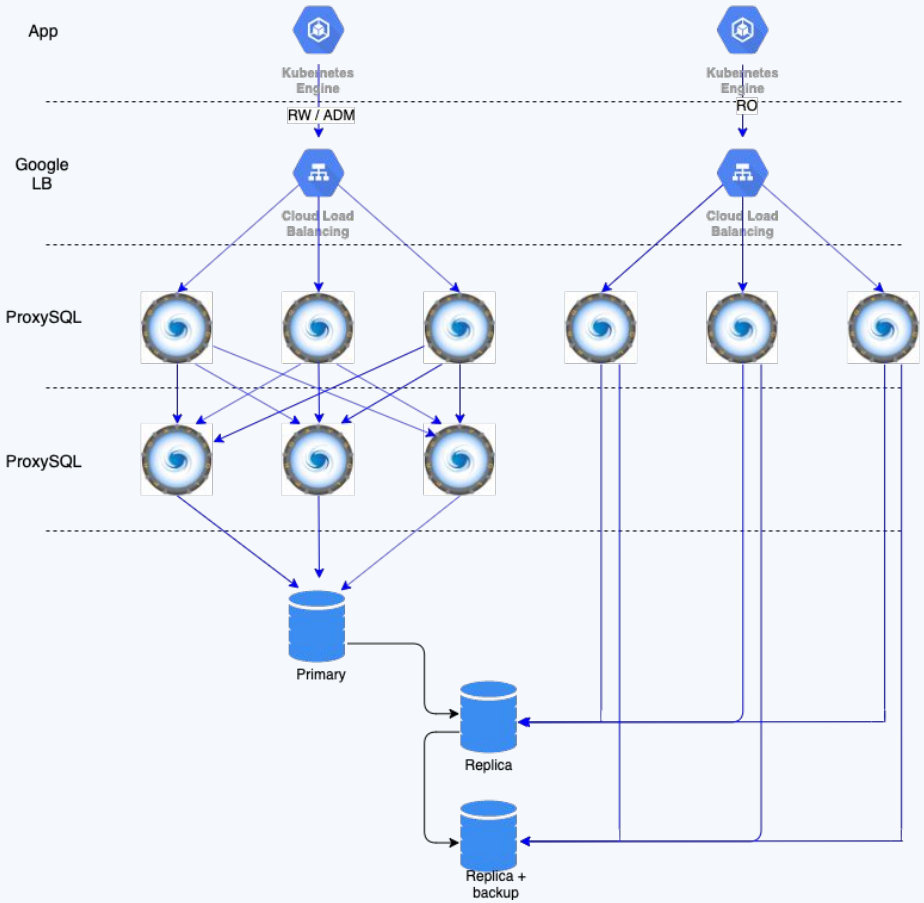
How do we shard in our DIY sharding?

- Sharding our messages
  - Message UUID as shard key
  - Random data distribution
  - No cross shard queries necessary
- Sharding our connections
  - For every shard open a connection
  - Write to shard connection based upon UUID
  - Read from shard connection based upon UUID

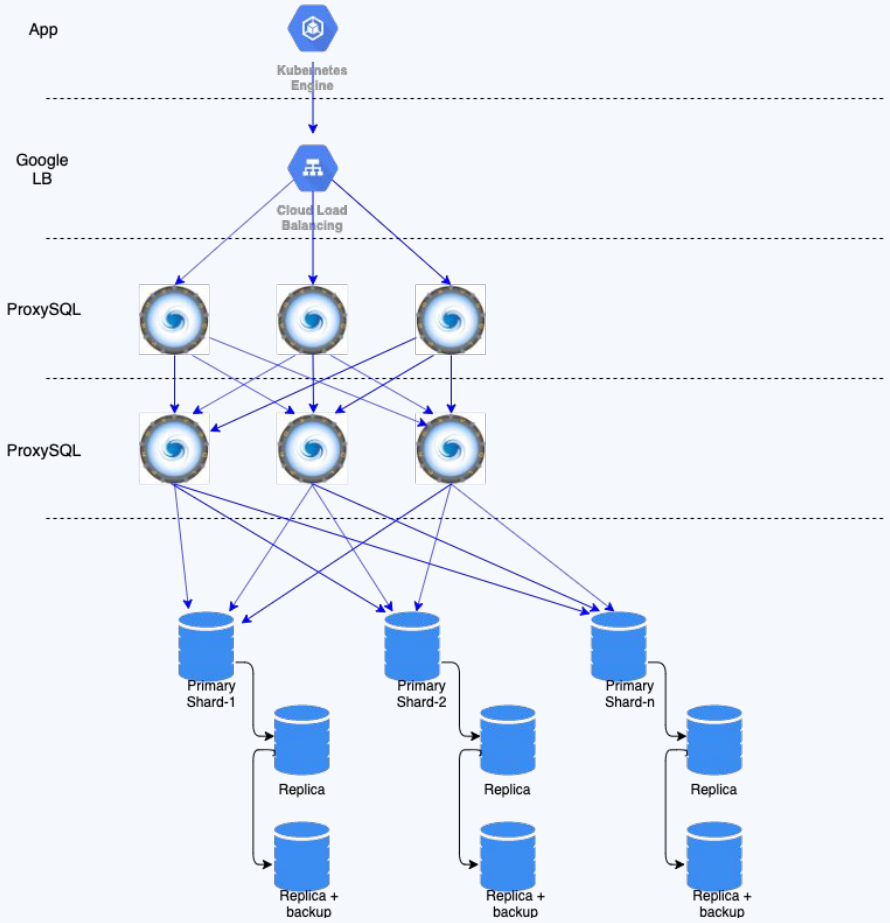
Sharding algorithm

- Algorithm definition
  - UUID v1 requires a bit of tinkering
  - Similar to “UUID % <shards>”
- Algorithm validity
  - Algorithm is valid for UUIDs between <start_date> and <end_date>
  - Allows us to switch algorithms and store data differently
  - Necessary for adding new shards
- Data retention
  - Data retention is 10 days
  - Schema and algorithm changes can cycle quickly
  - No shard splitting is necessary

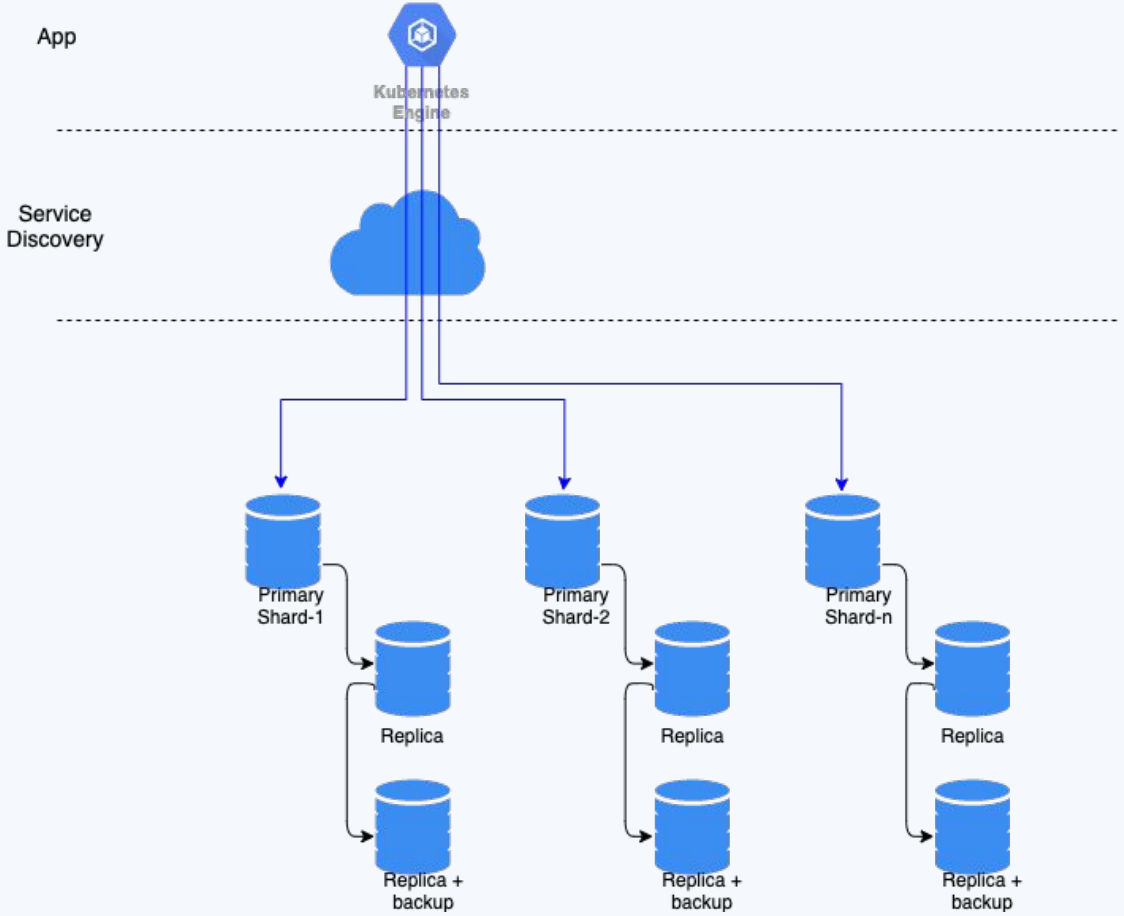
A typical database at Messagebird



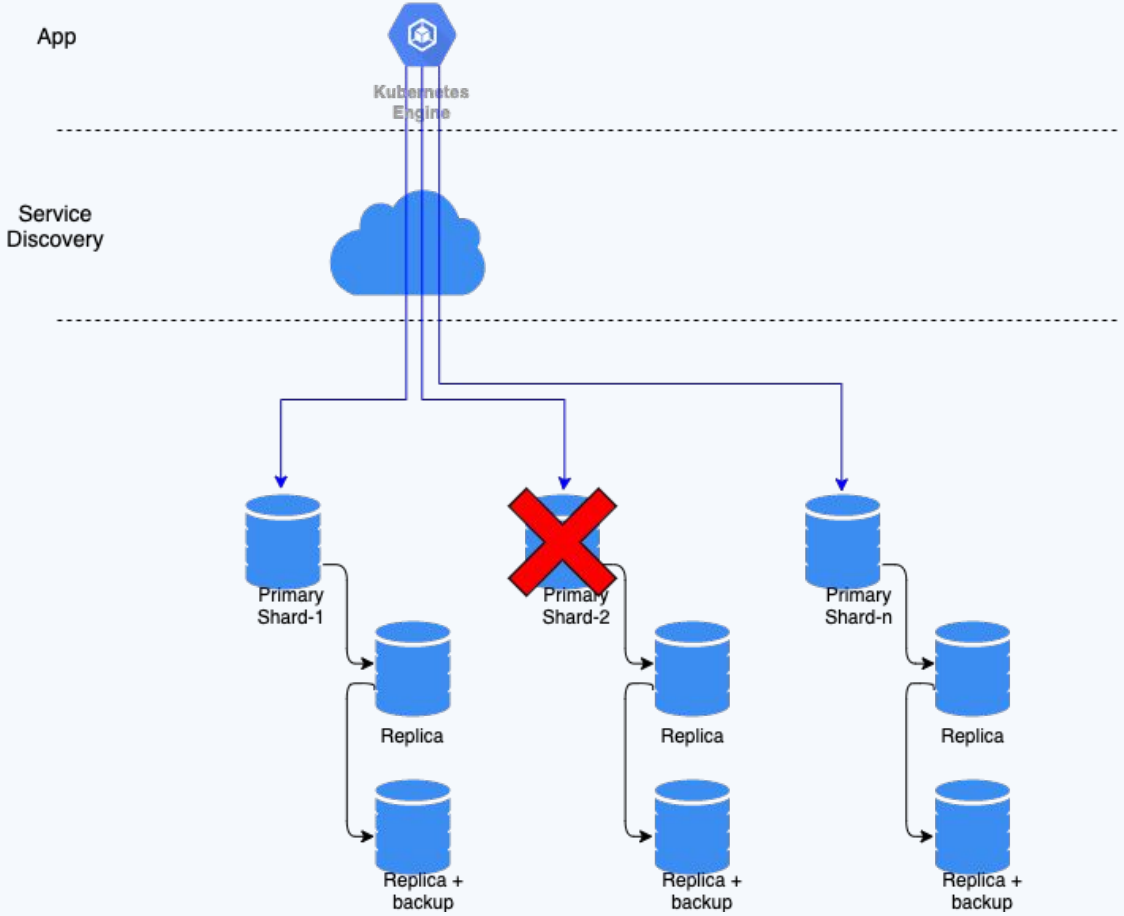
Let’s apply that multiple times for sharding!



How does that look from the application side?



What if a shard becomes unavailable?



What if a shard becomes unavailable?

- Part of our existing data is inaccessible
  - Updating existing data: not possible
  - Reading existing data: read from replica
  - Remember: not all our data is inaccessible!
- Inserting new data
  - Recalculate UUID
  - Write to shard connection based upon new UUID
  - Read from shard connection based upon new UUID
- In theory we would be able to always store messages

Did it work out as planned?

- Outages happened
  - Complete shard unavailability
  - Re-sharding algorithm
  - UUID recalculation
- Scalability
  - Adding new shards is relatively difficult
  - Adding a new shard requires a code change
- ProxySQL
  - ProxySQL is receiving many connections
  - Connection multiplexing helps (1 out of 5 connections)

Can we reuse it?

- Sharding algorithm is very (UUID) specific
  - Not usable as a template for other data types
- More “permanent” data
  - Shard splitting is not possible
  - Hot spots are difficult to fix
  - Asymmetric scaling isn’t possible
- Other issues
  - Cross shard querying is not possible
  - Cross shard joins are not possible
- Dependencies
  - ProxySQL is an essential component
  - Deployments need to be coordinated very carefully

Replacing the Archive

Are we able to use Vitess out of the box?

The Archive challenge

- Mid-term storage is being used for the following use cases:
  - Short term (near-realtime) monitoring of routes
  - “Quick” reference for messages < 7 days
  - Ship data to our Archive (6 months retention)
- Similar to the short-term storage: replication lag
  - Replication lag means no read-scaling
- The Archive is huuuge
  - 6TB maximum data size on mid-term storage (between 1 and 3 months of data)
  - ~13TB of data in cold-storage CloudSQL (about 6 months of data)

The Archive solution

- Split up functionality
  - Realtime data should be in a realtime database
  - All referencing should be done on the Archive
- Sharding is inevitable
  - 40% YoY growth
  - Small hands make great work

What boundaries were set?

- Simple access paths
  - On message identifier (UUID)
  - On customer identifier (int)
  - Within a certain date range (datetime)
- Simple aggregations
  - How many messages by X
- Everything else: analytics

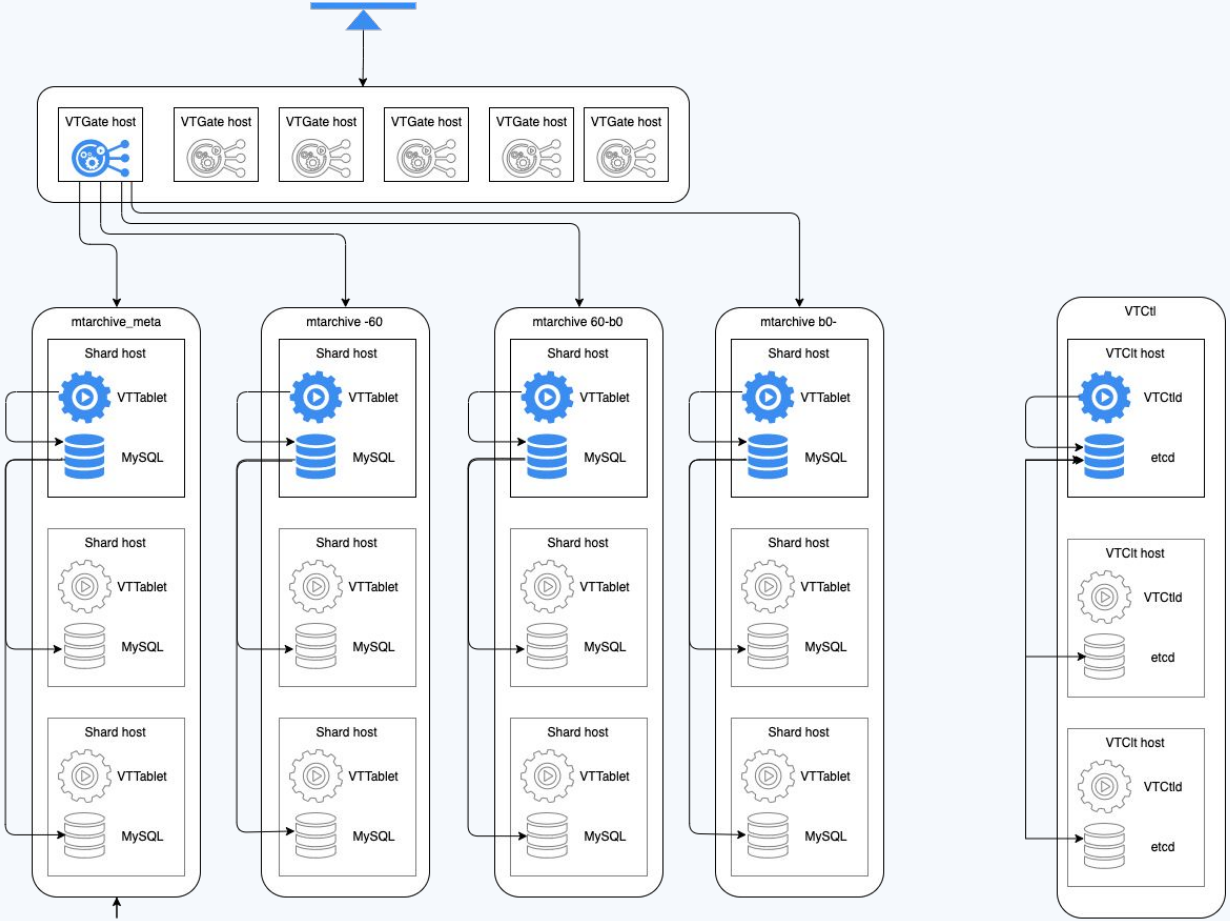
Why did we consider Vitess?

- Vitess promises
  - Transparent sharding
  - Shard splitting
  - Asymmetrical sharding
  - Materialized views (aggregates)
- Vitess uses MySQL as a foundation
- Vitess might also be suitable for sharding our other databases
- Vitess might also be suitable for multi-region data

Choosing the right solution for the Archive

- Kubernetes or VM install?
  - Vitess K8s are advised to be small (250GB max)
  - PoC is the MT Archive with almost 10TB of data
  - 10TB / 250GB = 40 shards
  - 1 shard consists out of 3 instances (master, replica, backup host)
  - 40 shards = 120 instances
  - 120 instances = 120 worker nodes
- VMs can be larger (500GB to 1TB)

Our Archive Vitess setup



What did we encounter during the PoC?

- Vitess Shard performance
  - Performance bottlenecks have been identified
  - InnoDB/MySQL have been tuned to our setup
  - Vitess also needs tuning (e.g. grpc thread pools)
- Productionalizing Vitess
  - Automation via ansible (provisioning + config)
  - Send Vitess metrics to Prometheus
  - Integrate backups
- Vitess Administration
  - Shard splitting
  - Schema migrations

Comparison between Vitess and DIY Shards

How different is it?

How do we perceive them?

DIY Shards

- **We are in control**
  - We “define” our sharding algorithm and store over shards
  - Algorithm tied to a specific time range
  - Algorithm change involves setting an end date to the previous algorithm

Vitess

- **We rely completely on a framework**
  - Vitess enforces us to make our schemas fit
  - We don’t have any influence on what goes on behind the scene
  - We simply point our queries to VTGate and then the magic happens

How do others (devs) perceive them?

DIY Shards

- **Fear of change**
  - A lot of work and risk to add shards
  - Shard operations have to be done in code

Vitess

- **Fear of unknown**
  - More moving components means more (possible) failures
  - Certain (specific) queries are not supported
  - If a shard fails, we can’t write that specific data

Operational: schema changes

DIY Shards

- Daily tables
- Application handles data truncation
- Application handles schema migrations

Vitess

- Vitess forces named columns
- Multiple schema versions can co-exist
- OSC tools work (per shard)
- OSC is tedious on 8+ shards
- OSC requires space

Operational: scaling out

DIY Shards

- Deploy new shard hosts
- Define new hosts in service discovery
- Deploy code with new algorithm
- Connections to new shards

Vitess

- Deploy new shard hosts
- Initiate shard split
- Move reads to new shards
- Move writes to new shards

Operational: MySQL upgrades

DIY Shards

- Upgrade replica hosts
- Initiate switchover via Orchestrator
- Upgrade old primary

Vitess

- Upgrade replica hosts
- Initiate switchover via Vitess (commandline)
- Upgrade old primary

Operational: sharding platform upgrades

DIY Shards

- Deploy new application version

Vitess

- Upgrade shard hosts
- Upgrade Vitess control hosts
- Upgrade VTGate hosts

(So far we performed only minor version upgrades)

Conclusion

Regrets? Words of caution?

Running Vitess on “iron”

- Allows us to keep things “as they are”
  - Easy integration into existing DBA tools
  - Less steep learning curve
- Pros:
  - We get “more” performance out of our hosts
  - We can allow larger data size
- Cons:
  - Less agile
  - Shard operations take longer and may time out
  - Hosts need to be repaired (when broken)

Is Vitess saving you time?

- Total time for DIY Shards was 3 sprints for 1 squad
  - Close collaboration between database engineers and developers
  - Simple problem, simple solution
- Total time for Vitess Archive is 5 months
  - 1 developer and 1 database engineer
  - Learning curve for Vitess (new components)
  - Incompatible queries
  - Vitess frequent updates + documentation

Questions?