#ASLI ENGINEERING

How databases are managed in production?

SWIPE

BY

ARPIT BHAYANI

Dissecting GitHub Outage

Understand how databases are managed

What Happened ? ProxySQL rollout

Database crashed → Automatic Failover

↓

New master also crashed

↓

New DB crashed ← Manual Failover

↓

Finally code reverted → Back to normal

This is a very common scenario, let's understand each phase in detail

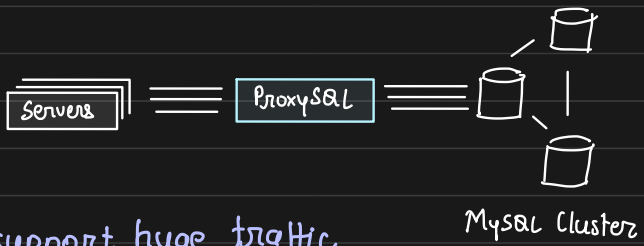
What is ProxySQL ?

ProxySQL sits as a proxy b/w the servers/clients and MySQL cluster.

Every connection and query goes through ProxySQL

Why do we need ProxySQL in production?

- Better connection handling
  - connection pool
  - connection multiplexing

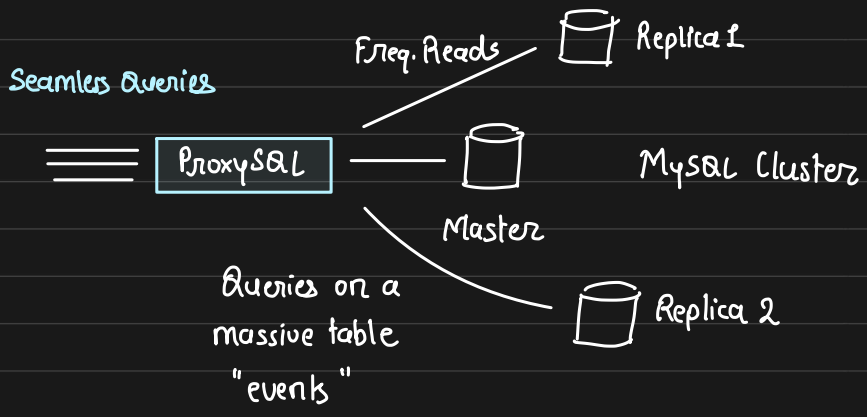


* fewer DB connections can support huge traffic

- Gatekeeper to enhance security and routing
  - route writes to primary
  - route reads to replicas

} Instead of API servers keeping the logic, ProxySQL does that seamlessly

* we configure rules and fire query. ProxySQL will automatically route the request to correct and intended node



- Caching
  - Given that ProxySQL is transparently sitting b/w servers and database, it is the best place to cache common queries

- Temporary access management
  - can create temporary users in DB and delete them after a few hours

↗ We keep our data secure + logging / tracing at one place

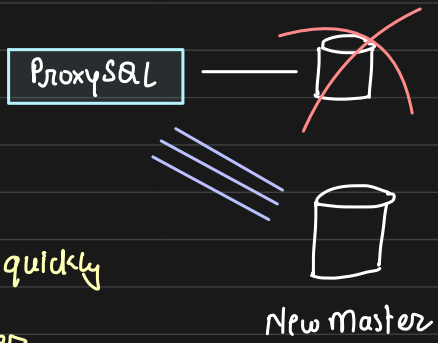
What failed?

GitHub teams wanted a feature and that was available on the latest ProxySQL version.

Hence DB team updated ProxySQL.

After a week, primary node crashed

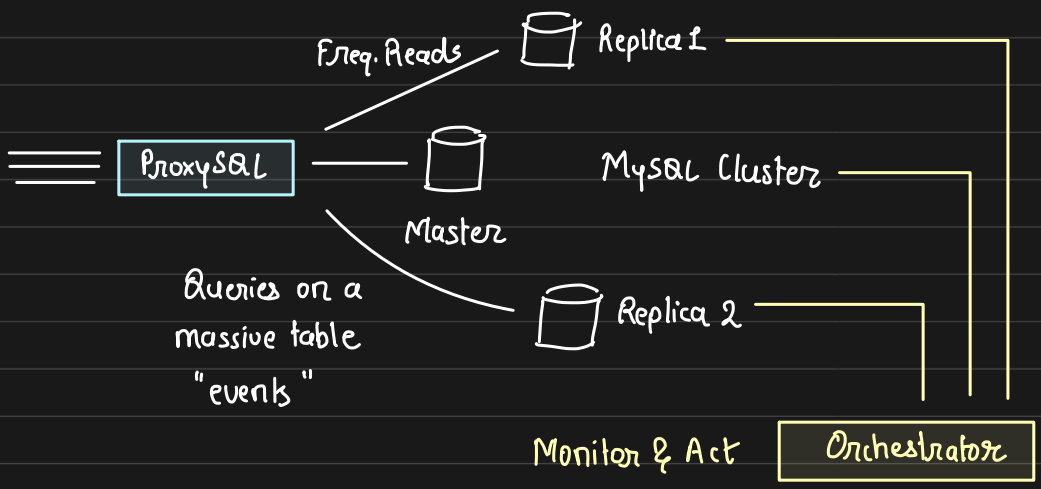
"orchestrator" detected the failure and quickly promoted a replica to be the new master



What is Orchestrator?

Tool to manage MySQL topology and ensure High Availability

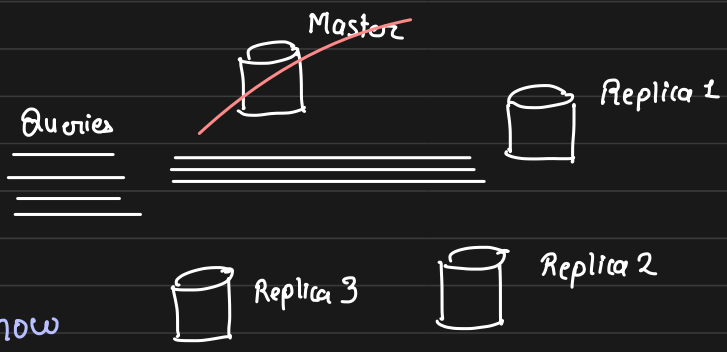
1. Discovery → topology + performance
   - fancy UI to see how "good" is replication
2. Recovery
   - Discover failures and perform automated recovery



* Key Feature: Anti-flapping

Prevents cascading failure by NOT doing automated failover

Cascading DB Failures



If a node goes down, orchestrator promotes a replica as new master

* The new master has to now bear the load, and high chances it would crash, leading to a cascaded failure

If the newly promoted master also goes down, Orchestrator would not promote another replica again until a cooloff period. → ANTI-FLAPPING

This would ensure NO cascading failure propagate further

Manual Failover

Because orchestrator is not spinning up new DB GitHub team hence had to do a manual failover

The new DB also starved for CPU and crashed!

So, how did they recover?

Revert the root cause

Because DB was not recovering, and new ones couldn't handle the incoming load, GitHub team reverted the code that required newer version of ProxySQL and down-versioned proxySQL as well.

This solved the issue and MySQL cluster started taking writes

What did we learn?

- how companies handle production database?
- ProxySQL and Orchestrator
- importance of anti-flapping policy
- when nothing works, full revert !!