Highly Available Database Architectures in AWS

Santa Clara, California | April 23th – 25th, 2018
Mike Benshoof, Technical Account Manager, Percona
Hello, Percona Live Attendees!

What this talk is meant to be...

• High level overview of a highly available (HA) database solution
  - What is it and why do we need it?
  - General concepts

• Examples of HA architectures using different AWS components
  - EC2, RDS, Aurora, and ProxySQL

• General best practices from a design and application standpoint
  - High level considerations of issues and planning for failure
What this talk is meant to be...
- High level overview of a highly available (HA) database solution
  - What is it and why do we need it?
  - General concepts
- Examples of HA architectures using different AWS components
  - EC2, RDS, Aurora, and ProxySQL
- General best practices from a design and application standpoint
  - High level considerations of issues and planning for failure

What this talk is not meant to be...
- A deep dive into AWS or MySQL internals
  - Won't be any mention of provisioned IOPS or buffer pool size
- A listing of several benchmarks with a recommendation of which is “best”
  - Benchmarks can be misleading, your application is unique
- A description of a “silver bullet” architecture that will fit every use case
  - There is no single solution
So let’s dig in...

What is a highly available database solution?

An architecture that is designed to continue to function normally in the event of hardware or network failure within the system.
So let’s dig in...

In practice, this generally translates to some level of automatic failover that generally results in some level (however brief) of downtime.
What does it look like?

- Application servers sending R/W traffic to primary database
- Failover database in the background - unused
- Some synchronization mechanism between primary and failover
What does it look like?

- Primary database fails!!
What does it look like?

- R/W traffic is re-routed to the failover node
- No application changes are needed, but some level of retry logic is recommended
Some general concepts...

- **Virtual Endpoint**
  - Application connects to an alias and not the physical servers
  - This allows the endpoint to handle the routing to backend resources
  - Some examples
    - Load balancer (physical or logical)
    - DNS
    - Floating IP address
Some general concepts...

- **Virtual Endpoint**
  - Application connects to an alias and not the physical servers
  - This allows the endpoint to handle the routing to backend resources
  - Some examples
    - Load balancer (physical or logical)
    - DNS
    - Floating IP address

- **Synchronization**
  - Data is kept in sync between primary and failover resources
  - Can be synchronous or asynchronous, but done automatically in real-time
  - Some examples
    - MySQL Replication (async)
    - Block level replication (sync)
    - Clustering solution - i.e. Galera (sync)
Let’s take this to the cloud...

AWS Components at our disposal

• Elastic Compute Cloud (EC2)
  - *Self managed MySQL instances, generally built on Linux AMI*
  - *Highly customizable / flexible*
Let’s take this to the cloud...

AWS Components at our disposal

- Elastic Compute Cloud (EC2)
  - *Self managed* MySQL instances, generally built on Linux AMI
  - *Highly customizable / flexible*

- Relational Database Service (RDS)
  - *Can run* MySQL native or Aurora (or other engines such as SQL Server, Postgres, Oracle)
  - *Less flexible, but fully managed* (point-and-click snapshots, replicas, etc)
Let’s take this to the cloud...

AWS Components at our disposal

• Elastic Compute Cloud (EC2)
  - Self managed MySQL instances, generally built on Linux AMI
  - Highly customizable / flexible

• Relational Database Service (RDS)
  - Can run MySQL native or Aurora (or other engines such as SQL Server, Postgres, Oracle)
  - Less flexible, but fully managed (point-and-click snapshots, replicas, etc)

• Miscellaneous Building Blocks
  - Elastic Load Balancer (ELB)
  - Route 53 (DNS failover strategies)
  - Elastic IP (virtual IP that can be assigned to EC2 instances)
So Many Choices!

- The options are endless!
- Here are the solutions we’ll discuss
  - Percona XtraDB Cluster on EC2
  - RDS for MySQL
  - Amazon Aurora
Percona XtraDB Cluster (PXC)

Percona XtraDB Cluster

- Percona Server for MySQL
- Galera Cluster (for replication)
  - Synchronous replication
  - Transaction based replication
    - Transaction is verified locally
    - Certified as valid on other nodes before local commit
- Can read/write to any node in the cluster
  - Preferred architecture
    - Write to single node, read from any node
    - Software load balancer for HA
PXC Use Cases

- Need the ability for multi-node writing
  - Ideally architected to avoid collisions
  - i.e. each node writes to dedicated schema/tables
PXC Use Cases

- Need the ability for multi-node writing
  - Ideally architected to avoid collisions
  - I.e. each node writes to dedicated schema/tables
- Require consistent reads
  - Application requires additional read replicas
  - Application cannot tolerate any replica lag
PXC Use Cases

- Need the ability for multi-node writing
  - Ideally architected to avoid collisions
  - I.e. each node writes to dedicated schema/tables
- Require consistent reads
  - Application requires additional read replicas
  - Application cannot tolerate any replica lag
- Maximum data durability
  - Guarantee transactions are remotely received
PXC Use Cases

- Need the ability for multi-node writing
  - Ideally architected to avoid collisions
  - I.e. each nodes writes to dedicated schema/tables
- Require consistent reads
  - Application requires additional read replicas
  - Application cannot tolerate any replica lag
- Maximum data durability
  - Guarantee transactions are remotely received
- Require cross-WAN (region) synchronous replication
  - Will add latency to writes (business decision)
PXC in AWS

EC2 Based deployment
- 3 base Linux AMI instances
PXC in AWS

EC2 Based deployment

- 3 base Linux AMI instances
- Nodes located in different AZs
  - Mitigates split-brain from AZ failure
PXC in AWS

EC2 Based deployment

- 3 base Linux AMI instances
- Nodes located in different AZs
  - Mitigates split-brain from AZ failure
- Provisioned IOPs or local storage
  - I3 instances with local NVMe
    - Note - relies on PXC for redundancy
  - GP2 not suitable for high throughput
PXC in AWS

EC2 Based deployment

- 3 base Linux AMI instances
- Nodes located in different AZs
  - Mitigates split-brain from AZ failure
- Provisioned IOPs or local storage
  - I3 instances with local NVMe
    - Note - relies on PXC for redundancy
  - GP2 not suitable for high throughput
- Cross region supported, higher write latency
  - Same for multiple VPCs - supported, but with potential latency increase
So how do we route??

Enter ProxySQL…

- Layer 7 software load balancer
So how do we route??

Enter ProxySQL…

- Layer 7 software load balancer
- Monitors backend nodes
  - Handles failed nodes transparently
  - Configurable retries
So how do we route??

Enter ProxySQL…

- Layer 7 software load balancer
- Monitors backend nodes
  - Handles failed nodes transparently
  - Configurable retries
- Potential for advanced routing
  - Read/write splitting
  - Table/schema based routing
So how do we route??

Enter ProxySQL…

- Layer 7 software load balancer
- Monitors backend nodes
  - Handles failed nodes transparently
  - Configurable retries
- Potential for advanced routing
  - Read/write splitting
  - Table/schema based routing
- Run locally or own layer
  - Local preferred for fewer app servers (< 10)
  - Use ELB for HA when separate layer
And finally the full stack...

- App servers point to ProxySQL behind ELB
- ProxySQL configured with
  - Writes pointed to single PXC node
  - Reads pointed to all three nodes in the cluster
- In the event of primary failure:
  - Write traffic shifted to another PXC node
  - Reads continue to be sent to all healthy nodes
RDS for MySQL / Amazon Aurora
Relational Database Service (RDS)

- Fully managed RDBMS, built on AWS components
  - EC2 instances
  - EBS volumes
Relational Database Service (RDS)

- Fully managed RDBMS, built on AWS components
  - EC2 instances
  - EBS volumes
- Operational features
  - Snapshots (restoring from snapshots)
  - Point-in-time recovery
  - On-demand replicas
Relational Database Service (RDS)

- Fully managed RDBMS, built on AWS components
  - EC2 instances
  - EBS volumes
- Operational features
  - Snapshots (restoring from snapshots)
  - Point-in-time recovery
  - On-demand replicas
- Availability features
  - Multi A/Z with failover (MySQL)
  - Automatic replica promotion (Aurora)
  - Master DNS endpoint (Virtual endpoint)
RDS Use Cases

- Desire (or need) fully managed DBaaS
  - Limited DBA staff
  - Developer/Application focused DBA staff
RDS Use Cases

- Desire (or need) fully managed DBaaS
  - Limited DBA staff
  - Developer/Application focused DBA staff
- Variable (but predictable) traffic patterns
  - Add/remove replicas quickly as needed
  - I.e. every weekend, traffic triples so add 2 new replicas each Friday evening, terminate each Monday morning (saves $)
RDS Use Cases

- **Desire (or need) fully managed DBaaS**
  - Limited DBA staff
  - Developer/Application focused DBA staff

- **Variable (but predictable) traffic patterns**
  - Add/remove replicas quickly as needed
  - I.e. every weekend, traffic triples so add 2 new replicas each Friday evening, terminate each Monday morning (saves $)

- **Application tolerates stale reads**
  - Can’t guarantee consistent reads, replication lag is possible
RDS Use Cases

- Desire (or need) fully managed DBaaS
  - Limited DBA staff
  - Developer/Application focused DBA staff
- Variable (but predictable) traffic patterns
  - Add/remove replicas quickly as needed
  - I.e. every weekend, traffic triples so add 2 new replicas each Friday evening, terminate each Monday morning (saves $)
- Application tolerates stale reads
  - Can’t guarantee consistent reads, replication lag is possible
- Non-complex design required
  - Standard read/write 90/10 traffic pattern, single node writes, read pool
Aurora vs RDS for MySQL

Aurora

- Shared backend storage
  - Replicated across multiple A/Z
  - Workload optimized by role (write master vs read replica)
- Cluster endpoints provided
  - Writer endpoint (DNS failover)
  - Reader endpoint (DNS round-robin)
- Hot-spare (replica) promoted to master is accessible
Aurora vs RDS for MySQL

Aurora

- Shared backend storage
  - Replicated across multiple A/Z
  - Workload optimized by role (write master vs read replica)
- Cluster endpoints provided
  - Writer endpoint (DNS failover)
  - Reader endpoint (DNS round-robin)
- Hot-spare (replica) promoted to master is accessible

RDS for MySQL

- Independent storage
  - Block level replication for standby
  - MySQL async for replica
- Writer endpoint managed by DNS
  - DNS Failover to hot-spare in different A/Z
- Hot-spare is EBS block level replication and not accessible
High Availability in RDS / Aurora

- Endpoints are unique DNS entries
  - Provided by RDS
  - Can be fronted by Route 53 CNAME as needed
High Availability in RDS / Aurora

- **Endpoints are unique DNS entries**
  - Provided by RDS
  - Can be fronted by Route 53 CNAME as needed

- **RDS for MySQL**
  - Writer DNS is failover only to hot-spare
  - Hot-spare isn’t reachable under normal operation
    - *i.e. can’t use as a slave for long queries, etc*
  - On master failure, potential for slight (~1 minute) data loss
High Availability in RDS / Aurora

- Endpoints are unique DNS entries
  - Provided by RDS
  - Can be fronted by Route 53 CNAME as needed

- RDS for MySQL
  - Writer DNS is failover only to hot-spare
  - Hot-spare isn’t reachable under normal operation
    - i.e. can’t use as a slave for long queries, etc
  - On master failure, potential for slight (~1 minute) data loss

- Aurora
  - Writer DNS is failover to active replica
  - Replica can be normal read-only replica in pool
  - On master failure, near instant failover with no data loss
RDS for MySQL Architecture

- App servers point to ProxySQL behind ELB
- ProxySQL configured with
  - Writes pointed DNS of primary
  - Reads pointed to pool of read replicas
- In the event of primary failure, write traffic shifted to RDS standby, read traffic still sent to replica pool
Aurora Architecture

- App servers point to ProxySQL behind ELB*
- ProxySQL configured with
  - Writes pointed to primary
  - Reads pointed to pool of read replicas
- In the event of primary failure, write traffic shifted to replica automatically promoted, read traffic still sent to replica pool

* Note: Requires ProxySQL 2.0+ due to Aurora using innodb_read_only for replicas
So why do we add ProxySQL to the RDS stack when DNS already handles failover??
More than just failover...

ProxySQL adds flexibility to the stack far beyond DNS failover:

- Rule based Read/Write splitting
  - Inspect queries, send plain selects to read-only hostgroup
  - Simplifies application logic
More than just failover...

ProxySQL adds flexibility to the stack far beyond DNS failover:

- Rule based Read/Write splitting
  - Inspect queries, send plain selects to read-only hostgroup
  - Simplifies application logic
- Seamlessly handles query retry on failover - i.e.


More than just failover...

ProxySQL adds flexibility to the stack far beyond DNS failover:

- Rule based Read/Write splitting
  - Inspect queries, send plain selects to read-only hostgroup
  - Simplifies application logic
- Seamlessly handles query retry on failover - i.e.
  


- Route queries for sharding
  - Inspect query for schema/table name, route to specific hostgroup
More than just failover...

- Dynamic rewrites / routing
  - At runtime, transparently rewrite or route problem queries
  - Leverages the query rules engine
More than just failover...

- **Dynamic rewrites / routing**
  - At runtime, transparently rewrite or route problem queries
  - Leverages the query rules engine
- **Route queries for sharding**
  - Inspect query for schema/table name, route to specific hostgroup
  - Can be done dynamically if new shards are added
More than just failover...

- **Dynamic rewrites / routing**
  - At runtime, transparently rewrite or route problem queries
  - Leverages the query rules engine

- **Route queries for sharding**
  - Inspect query for schema/table name, route to specific hostgroup
  - Can be done dynamically if new shards are added

- **More intelligent Query Cache**
  - Selectively cache queries by fingerprint
  - Query cache purging outside of mysql in background
  - No application changes needed to call external cache (i.e. memcache)
For your reading pleasure...

RDS / Aurora Details:
- https://aws.amazon.com/rds/mysql/faqs/

PXC

ProxySQL
- https://github.com/sysown/proxysql/wiki
Summary

- Multiple options for HA in AWS
  - PXC on EC2 (consistent reads, data durability, multi-region support)*
  - RDS / Aurora (fully managed, general workload/pattern, elastic)
- Need HA at each level
  - Application, load balancer (routing), database
  - Several AWS components available
- ProxySQL acts as intelligent (layer 7) load balancer
  - Dynamic rules/routing
  - Query / connection retry logic
  - Run locally or behind ELB
Thank You Sponsors!!

- Alibaba Cloud
- Microsoft
- DataCore
- facebook
- Grafana
- VividCortex
- Shannon Systems
- TIMESCALE
- Yelp
- Box
- SolarWinds
- Dynimize
- TwiDB
- PingCAP
- Altinity
- ObjectRocket
- MySQL
- Navicat
- GitHub
- cPanel
- BLAZINGDB
- Google Cloud
- datanami
- HPCwire
- Packt
- ODBMS.org
- ENTERPRISETECH
Rate My Session

Introducing gh-ost: triggerless, painless, trusted online schema migrations

11:20 → 12:10, Matterhorn 2

TAP THE SESSION

Rate & Review

Tap a star to rate

Feedback (optional)

SUBMIT