

Polytechnique Montréal

LOG8415E - Advanced Cloud Computing Concepts

Cloud Design Patterns: Implementing a DB Cluster

Ayah Farhat - 2153946

Fall 2024

Table of contents

1	Introduction	1
2	Benchmarking MySQL with sysbench	1
3	Implementation of The Proxy pattern	3
3.1	FastAPI-Based Proxy Server	3
4	Implementation of The Gatekeeper pattern.	4
4.1	Gatekeeper Implementation	4
4.2	Trusted Host Implementation	5
5	Benchmarking the clusters.	5
5.1	Objective	5
5.2	Testing Setup	5
6	Description of the Implementation	6
6.1	Implementation Workflow	6
6.1.1	Infrastructure Setup	6
6.1.2	EC2 Instance Launch	6
6.1.3	MySQL Installation and Configuration:	7
6.1.4	Replication Setup:	7
6.1.5	Securing the Instances	7
6.1.6	FastAPI Service Deployment	8
6.2	Execution Flow	8
7	Summary of results	9
7.1	Observations	9
7.2	Conclusion	10
8	Instructions to run the code	10

1 Introduction

The objective of this lab was to deploy a scalable MySQL cluster on AWS with master-slave replication, a layered security approach with a gatekeeper, a trusted host and a proxy that routes requests to the mysql cluster. It required configuring secure communication between instances and the deployment of FastAPI applications on each instance. Finally, some automated requests of read and write operations on the Sakila database were sent to the gatekeeper using 3 different proxy hit patterns. The results were benchmarked to be able to compare between the 3 solutions. The entire solution was end-to-end automated to ensure that every step was documented and can be replicated.

2 Benchmarking MySQL with sysbench

The initial setup involved creating three t2.micro instances: one for the Manager node and two for Worker nodes. MySQL was installed on all three instances, and the Sakila sample database was loaded to provide a standardized dataset for testing. To validate the correctness of the setup, Sysbench was used to perform read-only benchmarks.

1. Instance Setup:

- Created three t2.micro AWS instances: one Manager and two Workers
- Configured Ubuntu and applied system updates

2. MySQL Installation:

- Installed MySQL Server on all instances using:

```
sudo apt update
sudo apt install mysql-server -y
```

- Configured a common user (sysbench_user) for consistency

3. Sakila Database Setup:

- Installed Sakila sample database on each instance
- Loaded schema and data using provided SQL files:

```
mysql -u root -p < /path/to/sakila-schema.sql
mysql -u root -p < /path/to/sakila-data.sql
```

4. Sysbench Installation:

- Installed sysbench on all instances:

```
sudo apt install sysbench -y
```

5. Preparing and running Benchmark:

- Prepared the database using the oltp_read_only.lua script:

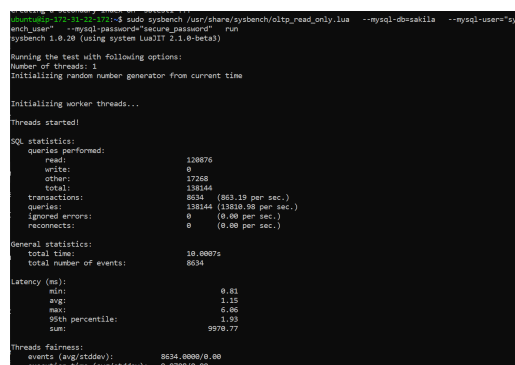
```
sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
--mysql-db=sakila --mysql-user="sysbench_user" \
--mysql-password="secure_password" prepare
```

- Performed a read-only benchmark on the Sakila database:

```
sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
--mysql-db=sakila --mysql-user="sysbench_user" \
--mysql-password="secure_password" run
```

6. Benchmark Results:

- SQL Statistics: 120,876 read queries, 863.19 transactions per second
- Latency: Minimum 0.81 ms, Average 1.15 ms, Maximum 6.06 ms
- Thread Fairness: Consistent performance across threads



```
Running the test with following options:
number of threads: 1
initializing random number generator from current time

Initializing worker threads...
Threads started!

SQL statistics:
queries performed:
  read:          120876
  write:         0
  other:         17268
  total:        138144
transactions:    8634 (863.19 per sec.)
queries:        138144 (13813.98 per sec.)
ignored errors: 0 (0.00 per sec.)
reconnects:     0 (0.00 per sec.)

General statistics:
total time:      10.0007s
total number of events: 8634

Latency (ms):
min:            0.81
avg:            1.15
max:            6.06
95th percentile: 1.93
sum:            9970.77

Threads fairness:
events (avg/stddev): 8634.0000/0.00
execution time (avg/stddev): 9.9708/0.00
```

Figure 1: Sysbench Benchmark results

The benchmarking results confirmed correct MySQL installations, proper Sakila database loading, and established baseline performance metrics for the cluster.

3 Implementation of The Proxy pattern

The proxy pattern implemented in the deployed system provides a middleware layer for routing of database queries between a manager node and worker nodes. This FastAPI-based proxy application allows both read and write operations to be handled dynamically. Three modes were developed: Direct Hit, Random Selection, and Customized Selection.

3.1 FastAPI-Based Proxy Server

Key components of the FastAPI application include:

- **Routing Modes Implementation:**
 - **Direct Hit:** All requests routed to the manager node. The write requests are always routed to the manager.
 - **Random:** Read requests distributed randomly among worker nodes. Using `random.choice()` to select worker for read requests.
 - **Customized:** Routes read requests to the node with least latency. Uses `socket.create_connection()` to check if a TCP port 3306 (already open to allow communication between the mySql instances) is open and return latency in ms. decided to do it this way instead of using PING to avoid exposing any unnecessary ports that may imply security risks.
- **Instance Discovery:** Within the FastAPI, Boto3 was used to find the ip address of the manager mysql instance and the 2 worker instances to be able to route the traffic to them based on the proxy pattern.
- **Key Endpoints:**
 - **/read:** when a client makes a request containing the keyword "SELECT"
 - **/write:** when a client makes a request containing the keywords "INSERT", "UPDATE", "DELETE", "REPLACE", "MERGE"
 - **/set_mode/{mode}:** to be able to change the proxy hit patterns, the options are "direct", "customized" and "random"

- **/health:** useful when debugging to make sure the instance is running and healthy

4 Implementation of The Gatekeeper pattern.

The gatekeeper serves as the entry point to the MySQL cluster, validating and routing incoming client requests to the appropriate Trusted Host instance. It is designed to ensure secure communication and enable dynamic interaction with the proxy-based database infrastructure without exposing it directly.

4.1 Gatekeeper Implementation

The Gatekeeper, implemented as a FastAPI service serves as the external-facing layer with the following responsibilities:

- **Request Validation:** Validates incoming requests for format and required fields.
- **Request Forwarding:** Forwards validated requests to the Trusted Host.
- Dynamic instance discovery using AWS EC2 APIs like in the proxy.
- JSON request parsing and validation.
- Key Endpoints:
 - **/process:** Accepts client requests, validates them, and forwards them to the Trusted Host at the `/process` endpoint. This centralizes request handling while maintaining modularity.
 - **/set_mode/mode:** Enables switching of routing modes (direct, random, or customized) by forwarding mode change requests to the Trusted Host's `/set_mode/mode` endpoint.
 - **/health:** Exposes the health status of the gatekeeper, useful for monitoring and diagnostics

4.2 Trusted Host Implementation

The Trusted Host, another FastAPI service on an internal instance, processes requests from the Gatekeeper and routes them to the Proxy Server. Its responsibilities include:

- **Query Classification:** The Trusted Host determines whether a query is a read or write operation based on its structure. Read operations (e.g., SELECT) are routed to the proxy's /read endpoint with a GET request, the query passed as a parameter in the url, while write operations (e.g., INSERT, UPDATE) are sent to /write with a POST request, the query is passed in the body of the request.
- **Routing Mode Management:** Requests to change the proxy's routing mode are forwarded from the Trusted Host to the proxy server.
- **Health Monitoring:** The Trusted Host provides its health status through a dedicated /health endpoint as well.

5 Benchmarking the clusters.

5.1 Objective

The benchmarking phase was conducted to evaluate the performance of the MySQL cluster under different query routing strategies implemented in the gatekeeper. The primary objective was to compare the response times for read and write queries across three modes: direct, random, and customized. The results for the benchmark include

- Average time for read and write operations
- Total time to process 1000 read and 1000 write requests sent to the gatekeeper /process endpoints

5.2 Testing Setup

- **Request Types:**
 - Read Query: SELECT first 10 records from sakila.actor
 - Write Query: INSERT new record into sakila.actor
- **Routing Modes:**

- Direct: All requests to Manager node
- Random: Reads to random Worker, writes to Manager
- Customized: Reads to Worker with least latency, writes to Manager

6 Description of the Implementation

6.1 Implementation Workflow

6.1.1 Infrastructure Setup

- **Key Pair Creation:**
 - Generated EC2 key pair (mysql-cluster-key) for secure SSH access
 - Private key saved locally with strict permissions
- **Security Groups:**
 - MySQL Cluster security group: Allows traffic only from Proxy Server security group on port 3306 and SSH traffic on port 22 from my IP address.
 - Proxy Server security group: Allows traffic from Trusted Host security group on port 8080 and SSH traffic on port 22 from my IP address.
 - Trusted Host security group: Accessible only from Gatekeeper security group and SSH traffic on port 22 from my IP address.
 - Gatekeeper security group: Handles external requests and communicates with Trusted Host, accessible on port 8080 from any IP address and SSH traffic on port 22 from my IP address.

6.1.2 EC2 Instance Launch

Instance Creation:

- MySQL Manager: Master node for write operations and replication setup.
- MySQL Workers: Two slave nodes for read replication.
- Proxy Server: Routes database queries to the appropriate nodes.

- Gatekeeper: Handles external requests and manages routing modes.
- Trusted Host: Acts as a secure intermediary between the gatekeeper and the proxy.

All instances are launched using the Ubuntu AMI (ami-005fc0f236362e99f).

6.1.3 MySQL Installation and Configuration:

- MySQL is installed on the manager and worker nodes, and the Sakila sample database is imported.
- The password for the root user was configured.
- A user was created in order to be able to access the data using the credentials when making the read and write requests through the proxy.

6.1.4 Replication Setup:

- A replication user is created on the manager node with privileges to replicate data.
- The script captures the manager node's log file and position, used for configuring the slaves.
- Worker nodes are configured as slaves, using the manager's log details to synchronize their data.

6.1.5 Securing the Instances

To enhance the security of the Gatekeeper and Trusted Host instances, the following steps are implemented:

- Firewall (IPTables) Configuration:
 - Flushes existing rules to start with a clean slate.
 - Gatekeeper allows external traffic on port 8080 for client requests and allows SSH traffic on port 22. Drops all other incoming traffic.
 - Trusted Host allows traffic on port 8080 only from the Gatekeeper's IP address and allows SSH traffic on port 22. Drops all other incoming traffic.
 - Saves the rules persistently to ensure they remain after reboot.

- Removing Unused Services:
 - This could possibly be implemented this but no risky services installed were found.
- Disabling Unnecessary Ports: Configures the firewall (UFW) to deny all incoming traffic by default, allow SSH (port 22) for administrative access, allow port 8080 for the Gatekeeper (enables public access), for the Trusted Host (restricts access only to the Gatekeeper).

6.1.6 FastAPI Service Deployment

- Services: Proxy Server, Gatekeeper, Trusted Host
- FastAPI applications deployed on respective instances
- Dependencies installed and services started using uvicorn
- Logs redirected towards a dedicated log file on the instance to be able to review and analyse when debugging and if there were issues

6.2 Execution Flow

1. Client sends request to Gatekeeper
2. Gatekeeper validates and forwards to Trusted Host
3. Trusted Host processes and forwards to Proxy Server
4. Proxy Server routes query based on mode
5. MySQL Cluster handles read/write operations and returns response

7 Summary of results

```
Benchmark Results:
Mode: direct
  Avg Read Time: 0.174390 seconds
  Avg Write Time: 0.184164 seconds
  Total Read Time: 174.39 seconds
  Total Write Time: 184.16 seconds
-----
Mode: random
  Avg Read Time: 0.173706 seconds
  Avg Write Time: 0.184345 seconds
  Total Read Time: 173.71 seconds
  Total Write Time: 184.34 seconds
-----
Mode: customized
  Avg Read Time: 0.174777 seconds
  Avg Write Time: 0.183379 seconds
  Total Read Time: 174.78 seconds
  Total Write Time: 183.38 seconds
-----
```

Figure 2: Benchmarking results

The benchmarking results, as shown in the table below and the screenshot above, summarize the average and total execution times for read and write operations under each routing mode:

Mode	Avg Read Time (s)	Avg Write Time (s)	Total Read Time (s)	Total Write Time (s)
Direct	0.174390	0.184164	174.39	184.16
Random	0.173706	0.184345	173.71	184.34
Customized	0.174777	0.183379	174.78	183.38

Table 1: Benchmark Results for Different Routing Modes

7.1 Observations

- **Direct Mode:** Read and write operations were consistent, with slightly lower response times compared to other modes. Direct routing eliminates the overhead of selecting a worker node, but it also means that the same instance is handling all the read and write requests, potentially slowing down response time.
- **Random Mode:** Randomly distributing read requests among worker nodes showed comparable performance to the direct mode. Although slightly better results were expected since the requests were being distributed between the instances.
- **Customized Mode:** Although this mode dynamically selects the worker node with the least latency, the additional latency measure-

ments slightly increased the average read time. The overall performance was nearly identical to the other modes.

7.2 Conclusion

The benchmark results reveal minimal differences in performance across the three routing modes which was surprising as the assumption was that a more strategic approach like the customized model would yield better results. While the direct mode is the fastest for scenarios requiring low overhead, random and customized modes excel in load distribution and adaptability. Since this implementation was a fairly simple read/write request API, it cannot be said for certain which method is most optimal. The choice of mode depends on the application requirements, such as performance, scalability, and fault tolerance.

8 Instructions to run the code

To run the code, you have to make sure that your aws credentials are configured on your machine. You can then install the packages boto3, paramiko, requests and httpx. Finally, you need to trigger the `mysql_cluster_script.py` script. This script will take care of triggering all necessary python scripts to create the key pair, the security groups, the instances, set up the mysql instances with replication between the master and the slaves, deploy the fast apis onto the appropriate instances, run the benchmarking requests for all three proxy hit configurations and provide you with the needed benchmarking metrics concerning the experiment.

Github Link for code base

Google drive link for video demo