Cloud Design Patterns - Final Assignment

LOG8415: Advanced Concepts of Cloud Computing

Amin Nikanjam
Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
amin.nikanjam[at]polymtl.ca

Student:
Arina Lazarenko
2410751

December 3, 2024

Contents

1 Introduction

This report explains the setup of a MySQL database cluster on Amazon EC2 using two cloud design patterns: Proxy and Gatekeeper. The objective was to build a system capable of handling requests efficiently while ensuring robust security. Performance was tested through benchmarking, and the entire process was automated with scripts to streamline deployment and testing.

2 MySQL

The installation and configuration of MySQL were automated using a Bash script to ensure a consistent setup across all instances. The process began by updating the system and installing MySQL, followed by modifications to the configuration file. Key settings such as `server-id` for replication and `bind-address` to allow external connections were added. Binary logging was enabled to support replication of the `sakila` database.

The MySQL service was then started and secured. The script automated the process of setting a root password, removing default users and test databases, and flushing privileges to apply changes. To test the setup, the Sakila database was downloaded, extracted, and imported into the MySQL server.

To verify the functionality of the MySQL servers, benchmarking was conducted using Sysbench. The steps included:

1. **Installing MySQL:** Three `t2.micro` EC2 instances were set up as standalone MySQL servers.
2. **Deploying the Sakila Database:** The Sakila databese was successfully installed to manager and workers instances.
3. **Replication:** The replication was done by setting MASTER role to the manager instance and SLAVE role to workers' instances by using the script below: A MASTER was created

```
mysql -u root -p$ROOT_PASSWORD -e "
CREATE USER 'proxy'@'%' IDENTIFIED WITH mysql_native_password BY 'proxy';
GRANT SELECT ON sakila.* TO 'proxy'@'%';
FLUSH PRIVILEGES;

CHANGE MASTER TO
    MASTER_HOST = '$MANAGER_IP',
    MASTER_USER = 'replica_user',
    MASTER_PASSWORD = '$REPLICA_PASSWORD',
    MASTER_LOG_FILE = '$MANAGER_LOG_FILE',
    MASTER_LOG_POS = $MANAGER_LOG_POSITION;

START SLAVE;
```

Figure 1: SQL script that was run on workers' instances

with permissions to read and write data, while a SLAVE user was granted only to read data from the Sakila database.

```
mysql -u root -p$ROOT_PASSWORD -e "
CREATE USER 'replica_user'@'%' IDENTIFIED WITH mysql_native_password BY '$REPLICA_PASSWORD';
GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'%';

CREATE USER 'proxy'@'%' IDENTIFIED WITH mysql_native_password BY 'proxy';
GRANT SELECT, INSERT, UPDATE, DELETE ON sakila.* TO 'proxy'@'%';
FLUSH PRIVILEGES;

START MASTER;
```

Figure 2: SQL script that was run on manager instance

4. Running Sysbench Tests:

   - The database was prepared for benchmarking and run by using commands

     ```
     sudo sysbench /usr/share/sysbench/oltp_read_only.lua --mysql-db=sakila
     --mysql-user="root" --mysql-password="\$ROOT_PASSWORD" prepare
     ```

     ```
     sudo sysbench /usr/share/sysbench/oltp_read_only.lua --mysql-db=sakila
     --mysql-user="root" --mysql-password="\$ROOT_PASSWORD" run
     ```

   - Results:

Figure 3: Worker 1 output

The benchmarking confirmed that all three servers responded quickly and processed requests effectively without errors. Figures 3-5

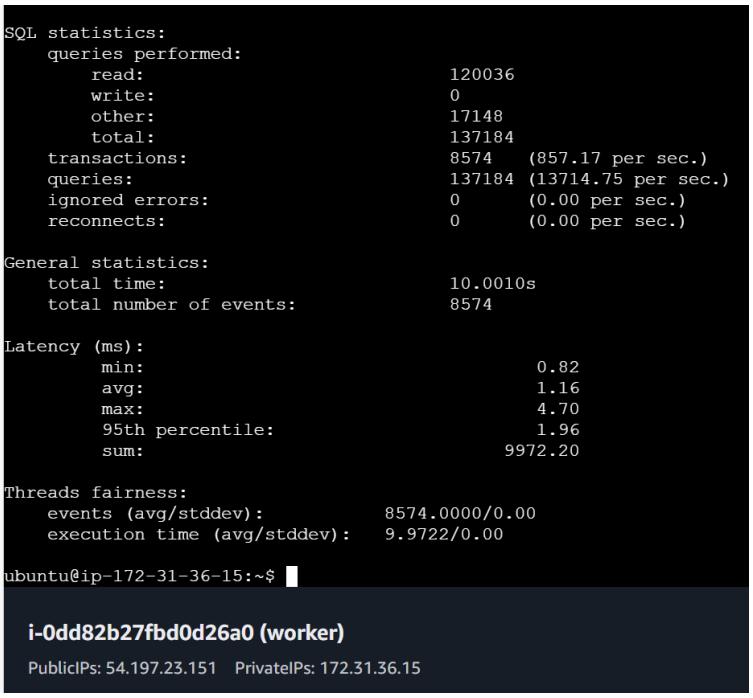

Figure 4: Worker 2 output

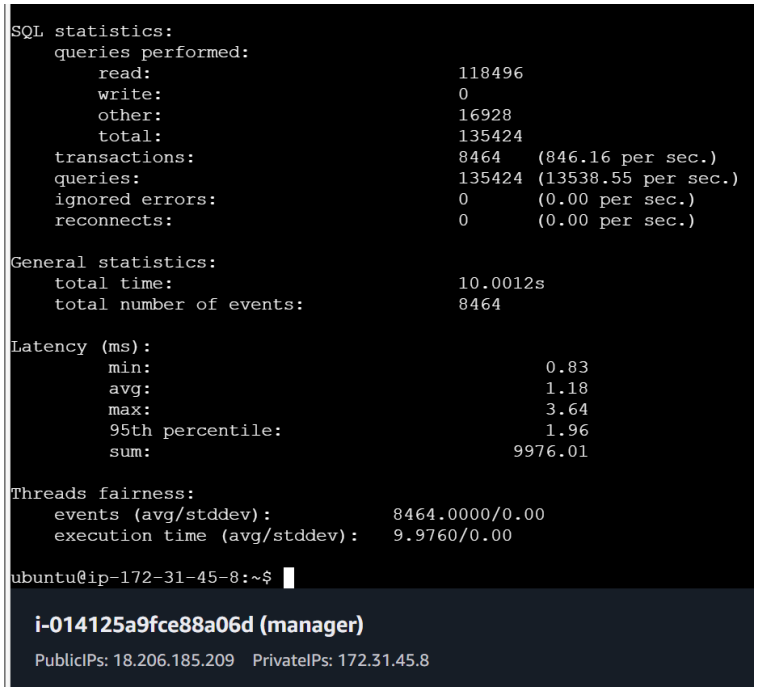

Figure 5: Manager output

3 Implementation of Proxy Pattern

The Proxy Pattern was implemented to manage query routing efficiently in a distributed MySQL database cluster. The key objective of the proxy was to route queries intelligently between the manager node and worker nodes based on the query type and routing strategy.

The Proxy (`proxy.py`) was designed as an intermediary that received all database queries from trusted host and determined the most appropriate node for execution. Its key features include:

- **Write Queries:** Always directed to the manager node. This ensures data consistency since the manager is the authoritative source for all write operations.
- **Read Queries:** Directed to worker nodes, leveraging read replicas to distribute the load and enhance scalability.

Routing Strategies:

- **Direct Hit:** In the Direct Hit strategy, all queries (both read and write) were sent directly to the manager node of the cluster. This approach bypassed the worker nodes entirely, relying solely on the manager for query execution. This strategy was implemented in the `direct_hit` function in `proxy.py`, which established a direct connection to the manager node using the `connect_to_mysql` helper function.
- **Random Distribution:** The Random Distribution strategy leveraged the worker nodes for handling read queries, spreading the workload across the cluster. Write queries continued to be directed to the manager node to maintain data integrity. This strategy was implemented in the `random_worker` function in `proxy.py`, where a worker node was selected randomly from a preconfigured list of workers. The function established a connection to the selected worker and executed the query.
- **Customized Distribution:** The Customized Distribution strategy introduced an adaptive mechanism to optimize query routing. In this strategy, read queries were directed to the worker node with the lowest response time. This was achieved by measuring the ping time for each worker node using the `measure_ping_time` function in `proxy.py`. The worker node with the shortest ping time was selected as the target for query execution. The `customized_worker` function handled this process, dynamically adapting to real-time conditions in the cluster.

4 Implementation of Gatekeeper Pattern

The Gatekeeper Pattern was implemented to provide an additional layer of security and isolation for the database cluster. It involved two key components: the Gatekeeper and the Trusted Host.

4.1 Gatekeeper

The Gatekeeper was designed to act as the external-facing component that handles incoming requests from users or clients. Its primary responsibilities include:

- Authentication: Before processing any request, the Gatekeeper validates the incoming request by checking for a password in the `X-Gatekeeper-Password` header. Unauthorized requests are immediately rejected with a 403 error code. This was implemented in the `authenticate` middleware function in the `gatekeeper.py` file.
- Request Validation: The Gatekeeper ensures that incoming requests contain the necessary fields, such as `operation` and `query`. Invalid requests are returned with a 400 error code.
- Routing Requests: After authentication and validation, the Gatekeeper forwards the request to the appropriate endpoint of the Trusted Host based on the operation type.

This component ensured that only validated and authorized requests reached the internal infrastructure, reducing the risk of attacks or misuse.

4.2 Trusted Host

The Trusted Host was implemented as an intermediary between the Gatekeeper and the Proxy. It is responsible for securely forwarding requests to the Proxy for processing. Its primary functions include:

- **Health Check:** Provides an endpoint to confirm that the Trusted Host is running and accessible.
- **Request Forwarding:** The Trusted Host receives validated requests from the Gatekeeper and routes them to the Proxy, ensuring that the Proxy is not exposed directly to external requests. This functionality is implemented in the `process_mode` and `forward_query` functions of the `trusted_host.py` file

4.3 Security Enhancements

To ensure the Trusted Host is secure:

- All unnecessary ports and services were disabled.
- IP restrictions were applied, allowing only the Gatekeeper to communicate with the Trusted Host.
- Sensitive credentials and configurations were securely managed using environment variables.

This setup allowed the Gatekeeper to act as a secure filter, preventing any unauthorized access to the Proxy and database nodes.

5 Benchmarking

The implementation was benchmarked under three routing strategies: Direct Hit, Random Distribution, and Customized Distribution. The benchmarking involved sending 1000 read and 1000 write requests to the database cluster for each strategy. The results measured the success rate, execution time, and the distribution of read queries among the nodes.

Routing strategy	Succsess WRITEs	Success READs
Direct Hit	1000/1000	1000/1000
Custom Distribution	1000/1000	1000/1000
Random Distribution	1000/1000	1000/1000

Table 1: Success queries results for each strategy

Routing strategy	Execution time (s)
Direct Hit	210.68
Custom Distribution	205.74
Random Distribution	213.09

Table 2: Execution time for each strategy

Routing strategy	Manager	Worker1	Worker2
Direct Hit	1000	-	-
Custom Distribution	-	472	528
Random Distribution	340	437	223

Table 3: Dispersion of reads for each strategy

The benchmarking results for the implementation highlight the performance differences across the three routing strategies: Direct Hit, Random Distribution, and Customized Distribution. All strategies achieved 100% success rates for both read and write queries, demonstrating the reliability of the system.

However, the execution times varied, with Direct Hit being the slowest at 210.68 seconds due to all queries being handled by the manager node. Random Distribution slightly improved performance (213.09 seconds) by distributing read queries almost evenly between Worker1 (472 reads) and Worker2 (528 reads). The Customized Distribution strategy achieved the fastest execution time at 205.74 seconds by dynamically routing read queries to the node with the lowest latency, resulting in a more balanced and optimized workload (Manager: 340 reads, Worker1: 437 reads, Worker2: 223 reads). The results are really close to each other, however with more various data Custom Distribution should show better results.

6 Conclusion

This project successfully demonstrated the implementation of distributed database architecture using the Proxy and Gatekeeper cloud design patterns. The Proxy efficiently managed query routing with three distinct strategies: Direct Hit, Random Distribution, and Customized Distribution. Among these, Customized Distribution proved to be the most effective, dynamically selecting the best worker node based on real-time performance metrics.

The Gatekeeper Pattern enhanced the system's security by isolating the external-facing components from internal ones. It ensured that only validated and authorized requests reached the internal infrastructure, reducing potential attack surfaces. The Trusted Host further strengthened

the architecture by securely forwarding requests to the Proxy, maintaining the integrity of the internal components.

The benchmarking results validated the reliability and scalability of the system, with all routing strategies achieving 100% success rates for both read and write queries. Execution times and query distribution highlighted the trade-offs between simplicity and performance, with Customized Distribution offering the best balance.

Future work could explore further enhancements, such as implementing adaptive load balancing or introducing additional layers of security.

7 Instructions to Run Code

1. **Configure AWS Credentials:**
   - Set up your AWS credentials on your local machine using the AWS CLI or environment variables.
2. **Edit the `globals.py` File:**
   - Open the `globals.py` file and fill in the constants with the appropriate relative paths required by your project.
3. **Insert AWS Credentials in `.aws` File:**
   - Open the `.aws` file and insert your AWS credentials in the following format:

     ```
     aws_access_key_id=[INSERT]
     aws_secret_access_key=[INSERT]
     aws_session_token=[INSERT]
     ```

4. **Make the `run_all.sh` Script Executable:**
   - In the terminal, run the following command to give execution permissions to the `run_all.sh` script:

     ```
     chmod +x run_all.sh
     ```

5. **Run the Bash Script:**
   - After making the script executable, run it using the following command:

     ```
     ./run_all.sh
     ```

6. **Check the Benchmarking Results:**
   - Once the script has completed running, the benchmarking results will be saved to a file named `benchmark_results.txt`. You can open or review this file for performance data.

8 Source

GitHub Repository: https://github.com/ArinaLazarenko/CloudComputingFinalProject