LOG8415E: Advanced Concepts of Cloud Computing

Cloud Design Patterns: Implementing a DB Cluster

Presented to :
NIKANJAM, Amin
MAJDINASAB, Vahid
Department of Software Engineering

By :
MUDENGE, Marco

Polytechnique de Montréal
Tuesday December 3rd, 2024

Introduction

This report presents the implementation and analysis of a MYSQL database cluster through its design and deployment incorporating key cloud design patterns on Amazon EC2. The lab assignment focuses on creating a resilient, secure, and efficient database architecture by implementing the proxy and gatekeeper patterns. These patterns enhance load distribution and robust access control mechanisms, enhancing the overall scalability and security of the system.

The first phase of the lab involved setting up a standalone MySQL database on three EC2 instances and deploying the Sakila sample database. Sysbench, our benchmarking tool, was used to validate the installation and evaluate the performance of each instance. The next phase extended these capabilities to implement a distributed MySQL cluster, where the Proxy pattern was employed to optimize read/write operations, and the Gatekeeper pattern was integrated to fortify the security framework.

Benchmarking MySQL with Sysbench

Benchmarking MySQL is done by using Sysbench to evaluate the performance of the MySQL servers in both standalone and clustered configurations. The benchmarking steps include:

- Installing Sysbench using “*sudo apt-get install sysbench -y*”
- Running the benchmarks to test read-only performance using “*sudo sysbench /usr/share/sysbench/oltp_read_only.lua --mysql-db=sakila --mysql-user="USER" --mysql-password="PASSWORD" run*”
- Retrieving the results to provide insights on database performance

Implementation of the Proxy pattern

The Proxy pattern improves scalability and performance by managing data replication and routing requests based on operation type. A master database handles write operations, while workers handle read operations.

The proxy is made aware of the cluster’s instances addresses during its set up through SSH. As for the cluster’s instances, their security group is updated to only allow traffic from the proxy once it has been created. The known cluster’s IP addresses are then used by an on-instance Python application to redirect queries accordingly.

The Proxy’s logic is deployed through a *FastAPI* Python application using SSH. The three implementations are used to redirect traffic as follows:

- Direct hit: All write and read requests are directly forwarded to the master node.
- Random: Read requests are forwarded to a worker node chosen randomly.
- Customized: Read requests are forwarded to the node (master or worker) with the lowest ping.

Please see file *./logic/proxy_logic.py* for its complete implementation.

Implementation of The Gatekeeper Pattern¸

The Gatekeeper pattern enforces security by isolating sensitive operations:

- Roles:
    - Gatekeeper: The Internet-facing instance validates all incoming requests.
    - Trusted Host: An internal instance that processes validated requests from the Gatekeeper.

- Security Measures:
    - Requests from external clients are validated and sanitized by the Gatekeeper.
    - Validated requests are forwarded to the Trusted Host for database operations.
    - The trusted host communicates securely with the Proxy for routing operations to the MySQL cluster.

In the implementation, the gatekeeper and the trusted host have distinct security groups. Those security groups are updated after the instance's creation. The trusted host security group only allow traffic from the gatekeeper and to the proxy. The proxy's security group only allow traffic from the trusted host.

The Gatekeeper's logic is also deployed over SSH at startup with Python's *FastAPI*. Please see file *./logic/gatekeeper_logic.py* for its complete implementation.

Benchmarking the cluster

Cluster benchmarking involves sending 1000 read and write requests to evaluate the performance of the Proxy implementations:

- **Direct hit**: Measure baseline performance by directly forwarding requests to the manager.
- **Random distribution**: Evaluate load balancing and response times when requests are distributed randomly across worker nodes.
- **Customized routing**: Assess the efficiency of routing based on response times. Results are expected to highlight the trade-offs between simplicity, load distribution, and a slight optimization of performance.

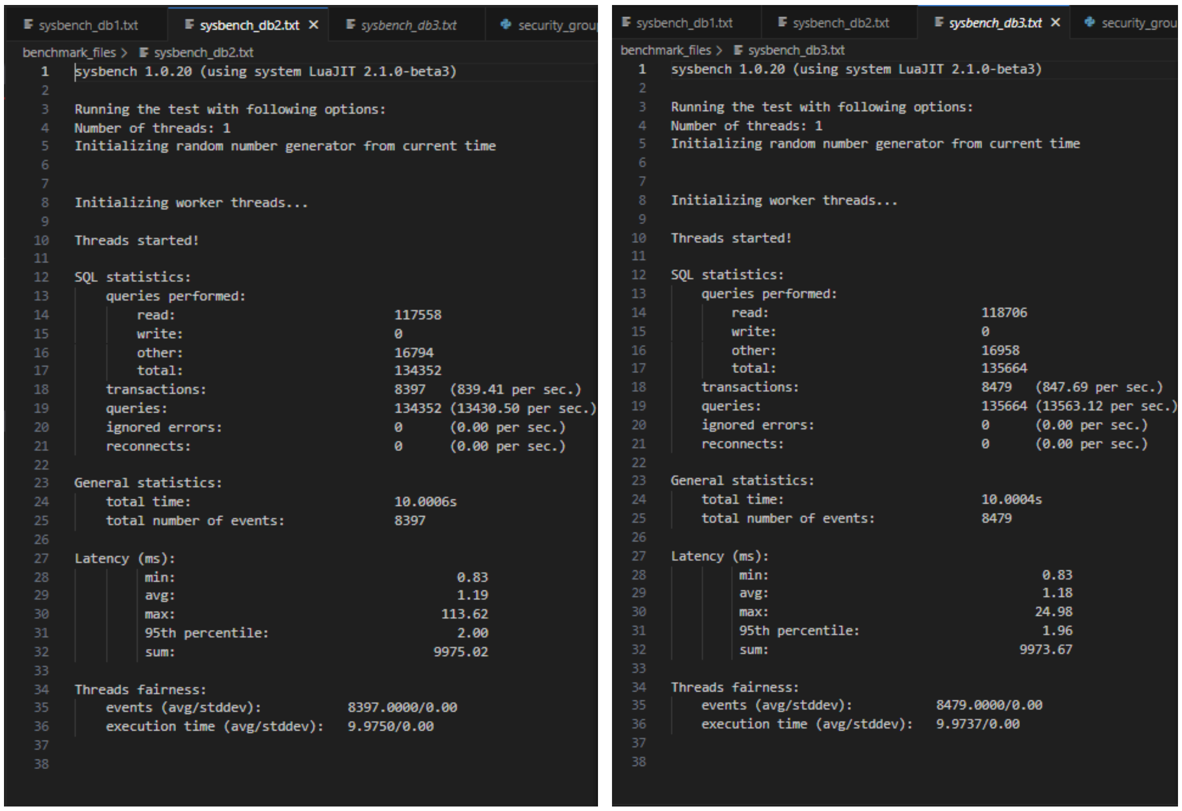
Standalone benchmarking results with Sysbench

Before benchmarking the completed MySQL cluster, a local benchmark on standalone MySQL instances was conducted. These benchmarks to evaluate the baseline performance of the system and also ensure the correctness of the initial setup.

The following metrics are collected for each instance:

- **SQL statistics**: Information about the queries.
- **Latency**: The average time taken to complete a transaction.

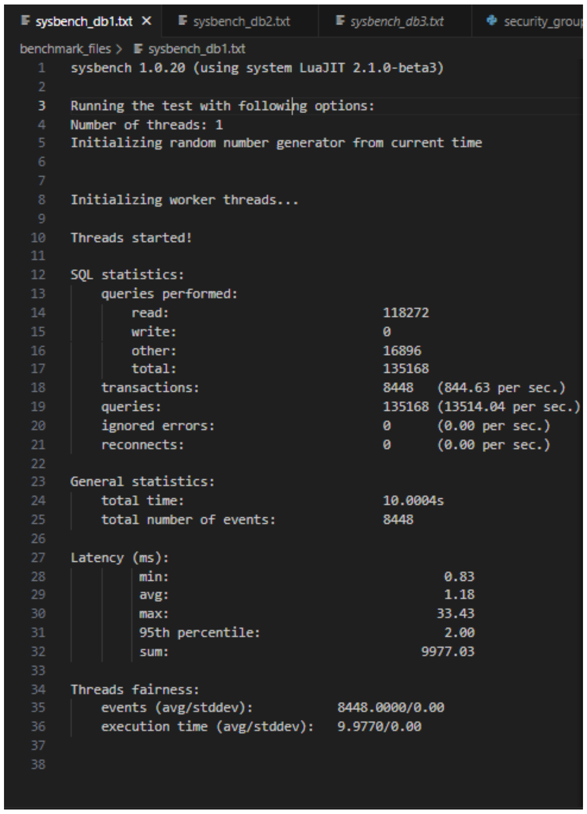
Please see files in *./benchmark_file* directory for complete results.


sysbench_db1.txt
sysbench_db2.txt
sysbench_db3.txt
security_grou
benchmark_files > sysbench_db2.txt
benchmark_files > sysbench_db3.txt

```
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

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:                            117558
        write:                           0
        other:                           16794
        total:                           134352
    transactions:                        8397   (839.41 per sec.)
    queries:                             134352 (13430.50 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0006s
    total number of events:              8397

Latency (ms):
         min:                                    0.83
         avg:                                    1.19
         max:                                  113.62
         95th percentile:                        2.00
         sum:                                 9975.02

Threads fairness:
    events (avg/stddev):           8397.0000/0.00
    execution time (avg/stddev):   9.9750/0.00
```

```
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

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:                            118706
        write:                           0
        other:                           16958
        total:                           135664
    transactions:                        8479   (847.69 per sec.)
    queries:                             135664 (13563.12 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0004s
    total number of events:              8479

Latency (ms):
         min:                                    0.83
         avg:                                    1.18
         max:                                   24.98
         95th percentile:                        1.96
         sum:                                 9973.67

Threads fairness:
    events (avg/stddev):           8479.0000/0.00
    execution time (avg/stddev):   9.9737/0.00
```

Figure 1: Baseline benchmarking results for worker nodes



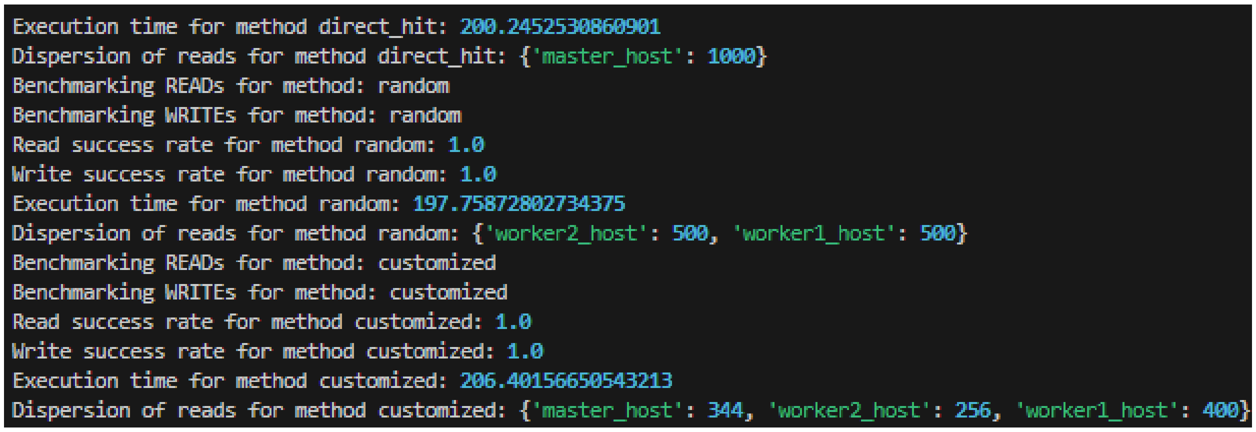
```
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

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:                            118272
        write:                           0
        other:                           16896
        total:                           135168
    transactions:                        8448   (844.63 per sec.)
    queries:                             135168 (13514.04 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0004s
    total number of events:              8448

Latency (ms):
         min:                                    0.83
         avg:                                    1.18
         max:                                   33.43
         95th percentile:                        2.00
         sum:                                 9977.03

Threads fairness:
    events (avg/stddev):           8448.0000/0.00
    execution time (avg/stddev):   9.9770/0.00
```

Figure 2: Baseline benchmarking results for master node

Benchmarking with different implementation

The purpose of these benchmarks is to evaluate the performance of the MySQL cluster using three different proxy implementation methods: Direct Hit, Random, and Customized. It helps analyze how each implementation handles read and write operations under a workload of 1000 read requests and 1000 write requests.

Total execution time (reads and writes) was measured for each method along with a distribution of routing decisions. The figure below displays the results. We can see the dispersion of queries varying from one implementation to another as expected. The execution time from the customized implementation is slightly higher. This can be attributed to the added ping logic and the fact that the SQL queries used are executed fast enough to nullify this method's advantages.



```
Execution time for method direct_hit: 200.2452530860901
Dispersion of reads for method direct_hit: {'master_host': 1000}
Benchmarking READs for method: random
Benchmarking WRITEs for method: random
Read success rate for method random: 1.0
Write success rate for method random: 1.0
Execution time for method random: 197.75872802734375
Dispersion of reads for method random: {'worker2_host': 500, 'worker1_host': 500}
Benchmarking READs for method: customized
Benchmarking WRITEs for method: customized
Read success rate for method customized: 1.0
Write success rate for method customized: 1.0
Execution time for method customized: 206.40156650543213
Dispersion of reads for method customized: {'master_host': 344, 'worker2_host': 256, 'worker1_host': 400}
```

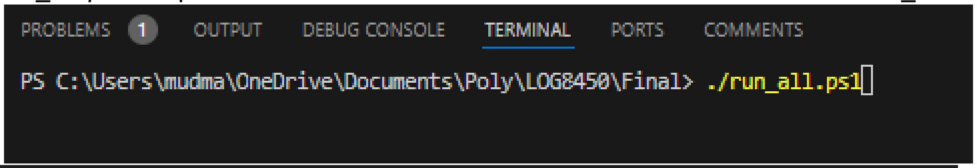
Figure 3: Benchmarking results for different implementations

Instructions to run code

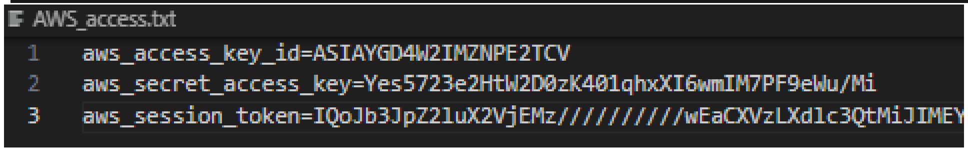
The source code to this lab can be found the following address:

https://github.com/marcomudenge/AWS_MySQL_Cluster_Cloud_Designing

To install requirements, create the AWS environment and run benchmarks, simply execute the *run_all.ps1* script. The AWS access information should be stored in the AWS_access.txt file.



```
PS C:\Users\mudma\OneDrive\Documents\Poly\LOG8450\Final> ./run_all.ps1
```



```
aws_access_key_id=ASIAYGD4W2IMZNPE2TCV
aws_secret_access_key=Yes5723e2HtW2D0zK401qhxXI6wmIM7PF9eWu/Mi
aws_session_token=IQoJb3JpZ2luX2VjEMz//////////wEaCXVzLXdlc3QtMiJIMEY
```

Instructions to view demo

The demo to this lab can be found at the following address:

https://www.youtube.com/@Marcom2024