LOG8415

Advanced Concepts of Cloud Computing

Final Project : Scaling Databases and Implementing Cloud Design Patterns

By :

Samy Cheklat – 1937812

Presented to:

Vahid Majdinasab

Thursday, December 28th 2023

The YouTube link is: https://youtu.be/Te97zOKcuSg

The GitHub link is: https://github.com/ikrash3d/TP1-LOG8415-TF

1 Introduction

This lab assignment involves practical application of container technology on *AWS* to host web applications. The tasks are deploying custom containers on instances, installing *Docker* and running the necessary *Flask* applications. The report will go over how scaling databases and cloud design patterns were implemented using those technologies.

2 Deployment of scaling databases and cloud design patterns

To simplify the deployment of our application, I've opted to utilize *Terraform*. This tool, categorized as infrastructure as code, enables me to articulate cloud resources through configuration files. Upon the successful creation of our *EC2* virtual machines, security groups were created to manage the input and the output for each instance. Upon launching our *Docker* image on the instance, a script facilitates the automatic execution of my different *Flask* applications.

3 Benchmarking *MySQL Stand-Alone* vs *MySQL Cluster*

For the *Stand-Alone*, *MySQL* was installed within that *EC2* instance. By using *Sakila*, a *MySQL* data model that you can open within *MySQL* to examine the database structure, were able to perform Online Transaction Processing (OLTP) benchmark. The benchmark was prepared and ran through commands where the table size, threads and time was specified.

```
# Prepare a MySQL database for an OLTP (Online Transaction Processing) benchmark
sudo sysbench /usr/share/sysbench/oltp_read_write.lua prepare --db-driver=mysql --mysql-host=ip-172-31-47-224.ec2.
internal --mysql-db=sakila --mysql-user=root --mysql-password --table-size=1000000

# Run the OLTP benchmark
sudo sysbench /usr/share/sysbench/oltp_read_write.lua run --db-driver=mysql --mysql-host=ip-172-31-47-224.ec2.internal
--mysql-db=sakila --mysql-user=root --mysql-password --table-size=1000000 --threads=6 --time=60 --events=0

# Cleanup the OLTP benchmark
sudo sysbench /usr/share/sysbench/oltp_read_write.lua cleanup --db-driver=mysql --mysql-host=ip-172-31-47-224.ec2.
internal --mysql-db=sakila --mysql-user=root --mysql-password --table-size=1000000 --threads=6 --time=60 --events=0
```

Figure 1 : OLTP benchmark

For the *Cluster* the same process was applied. Although the *Cluster* was acting as a manager where the writes requests are handled by the manager and replicated on its workers, while the read requests were processed by the workers. For that, a *config.ini* file was established in the *Manager*. In this file, the private *IP* address of the *Manager* is defined and so are the private *IP* addresses of *Worker-0, Worker-1* and *Worker-2*.

To establish the connection between each worker and the *Manager*, in each *Woker*, the following command: **"ndbd -c ip-172-31-47-224.ec2.internal:1186**" is ran. This command is the process that is used to handle all the data in tables using the *NDB Cluster* storage engine. The *IP* address provided is the private

domain name server (DNS) of the *Manager*. This is how the requests are forwarded or replicated depending on the type of requests. As for the benchmark, the library *sysbench* was used.

4 Implementing the Proxy pattern

Implementing the proxy pattern was quite a task. The implemented proxy pattern serves as a robust solution for distributing *SQL* queries across a network of worker instances. Leveraging the *Flask* web framework, *AWS SDK* for Python (*Boto3*), and *SSH Tunnel* libraries, the system effectively manages the execution of *SQL* queries on a distributed architecture. The system utilizes *Boto3* to configure an *AWS EC2* client, allowing seamless interaction with *AWS* services.

The *Proxy* has one endpoint where the requests are filtered based on their query type. The request contains the query type and the *SQL* query as query parameters.

Each query type is handled differently. If the query is type of "direct_hit", the request is forwarded to the *Manager*. If the query is type of "random", a random worker is then chosen to process the query. Finally, if the query type is type of "customized", all the workers are pinged. The worker with the smallest response time is then chosen to process the *SQL* query. All those requests are forwarded through the *SSHTunnelForwarder* to the chosen instance. A private key is used to encrypt the data and the communication port is specified. The tunnel is then used to connect to the *Sakila* database that runs in *MySQL*.

```python
# Function to select the worker instance with the least response time
def select_worker_with_least_response_time(workers):
    min_ping_time = float('inf')
    selected_worker = None

    for worker in workers:
        worker_ip_address = worker['PublicIpAddress']
        ping_time = measure_worker_ping_time(worker_ip_address)

        if ping_time < min_ping_time:
            min_ping_time = ping_time
            selected_worker = worker

    return selected_worker

# Function to get a random worker instance
def get_random_worker(workers):
    if not workers:
        return None
    return random.choice(workers)
```

***Figure 2* : Random worker and Fastest node method**

5 Implementation of The Gatekeeper pattern

The *Gatekeeper* is basically our entry door. This application runs as a *Flask* application. Same goes for the *Trusted* Host. The task was to forward the request from the *Gatekeeper* to our *Trusted Host*. Although, for that some regulations for our *Trusted Host* were needed. Firstly, communication was established through

an *SSH Tunnel*. That way we could deny some “man in the middle” attack. Because *SSH Communication* requires a private key, the data transferred through that tunnel is encrypted. Furthermore, another layer of security was added with the regulation of inputs and outputs to the *Trusted Host*.

To add that extra layer of security, only communications from the *Gatekeeper* was accepted to the *Trusted Host*. If someone other than the *Gatekeeper* tried to contact the *Trusted Host*, the connection would simply time out after a while. To allow both *HTTP* requests and *SSH* communication, in the *Terraform* file, a security group was specifically designed for the *Trusted Host* where the private *IP* address of the *Gatekeeper* was specified. Also, only port 80 and 22 were opened, the two ports required for the *SSH Tunnel Forwarder*.

Figure 3 : Security group of the Trusted Host

6 Description of how the implementation works

As for the resources, the *Stand-Alone, Manager* and *Workers* all run in a *t2.micro* instance. For the *Trusted Host, Gatekeeper* and *Proxy*. They all run in a *t2.large* instance. All the instances run a distribution of *Ubuntu*. The *Gatekeeper, Proxy* and *Trusted Host* are all *Flask* applications that run in a *Docker* container in their respective instances. As for the *Workers*, the *Manager* and the *Stand-Alone*, they all run a process of *MySQL* within the *Ubuntu* distribution.

As for the implementation, the flow goes like this. The *Gatekeeper* is the entry point of the whole infrastructure. He forwards the requests to the *Trusted Host*. A request has a *query_type* and an *SQL* query as query parameters. The *Trusted Host* is configurated in a way where only the *Gatekeeper* can communicate with the *Trusted Host*. After that, the *Trusted Host* forwards the requests to the *Proxy*. The *Proxy* then looks at the type of *SQL* query.

Each query needs a type and is handled differently. If the query is type of “direct_hit”, the request is forwarded to the *Manager*. If the query is type of “random”, a random worker is then chosen to process the query. Finally, if the query type is type of “customized”, all the workers are pinged. The worker with the smallest response time is then chosen to process the *SQL* query. All those requests are forwarded through

the *SSHTunnelForwarder* to the chosen instance. A private key is used to encrypt the data and the communication port is specified.

After the forwarding is done, the *SQL* request is then processed by the chosen instance. The response then follows the same path but backwards this time. The *Gatekeeper* then showcases the result of the *SQL* query.

As for the *Stand-Alone*, the purpose of this instance was to simply benchmark the difference between a *MySQL Cluster* and *MySQL.*

7 Summary of results and instructions to run the code

Summary of results:

As for the benchmarking of the *MySQL Stand-Alone* and *MySQL Cluster*, for the *Cluster*, 435 840 queries were performed. A total of 21 792 transactions were performed with an average time of 363.15 transactions per second. As for the queries, 435 840 were performed with an average time of 7272.91 queries per second. As for the general statistics, it took a total time of 60 seconds for a total of 21 792 events. As for the latency, the average was 16.52 milliseconds (ms).

For the *Stand-Alone*, 292 640 queries were performed. A total of 14 632 transactions were performed with an average time of 243.77 transactions per second. As for the queries, 292 640 were performed with an average time of 4875.41 queries per second. As for the general statistics, it took a total time of 60 seconds for a total of 14 632 events. As for the latency, the average was 24.61 ms.

By looking at the two instances, we see that the *Cluster* executes transactions faster than the *Stand-Alone*. The *Cluster* executes the transactions 48.85% faster than the *Stand-Alone*. As for the queries, the *Cluster* executes queries 48.91% faster than the *Stand-Alone*. Another distinction between the two is the average latency. The *Cluster* seems to respond faster than the *Stand-Alone* with a response time 32.87% faster.

The full details are shown in the two pictures down below.

```
SQL statistics:
    queries performed:
        read:                            305088
        write:                           87168
        other:                           43584
        total:                           435840
    transactions:                        21792  (363.15 per sec.)
    queries:                             435840 (7262.91 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0071s
    total number of events:              21792

Latency (ms):
         min:                                    4.42
         avg:                                   16.52
         max:                                  589.24
         95th percentile:                       27.17
         sum:                                359939.02

Threads fairness:
    events (avg/stddev):           3632.0000/11.86
    execution time (avg/stddev):   59.9898/0.00

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

Figure 4 : SQL statistics of the MySQL Cluster Manager

```
Initializing worker threads...

Threads started!

SQL statistics:
    queries performed:
        read:                            204848
        write:                           58528
        other:                           29264
        total:                           292640
    transactions:                        14632  (243.77 per sec.)
    queries:                             292640 (4875.41 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0214s
    total number of events:              14632

Latency (ms):
         min:                                   12.82
         avg:                                   24.61
         max:                                  405.05
         95th percentile:                       38.25
         sum:                                360023.08

Threads fairness:
    events (avg/stddev):           2438.6667/10.77
    execution time (avg/stddev):   60.0038/0.00

ubuntu@ip-172-31-47-5:~/sakila-db$
```

Figure 5 : SQL statistics of the SQL Stand-Alone server

Following the implementation of the *Trusted Host* and *Gatekeeper*, we can test the security of our infrastructure. After being connected to the *Gatekeeper*, hitting the */health* route, we can see that our *Trusted Hust* is up and running. We can also see that sending a request with a query type and a *SQL* query through the *Gatekeeper* returns the proper information.

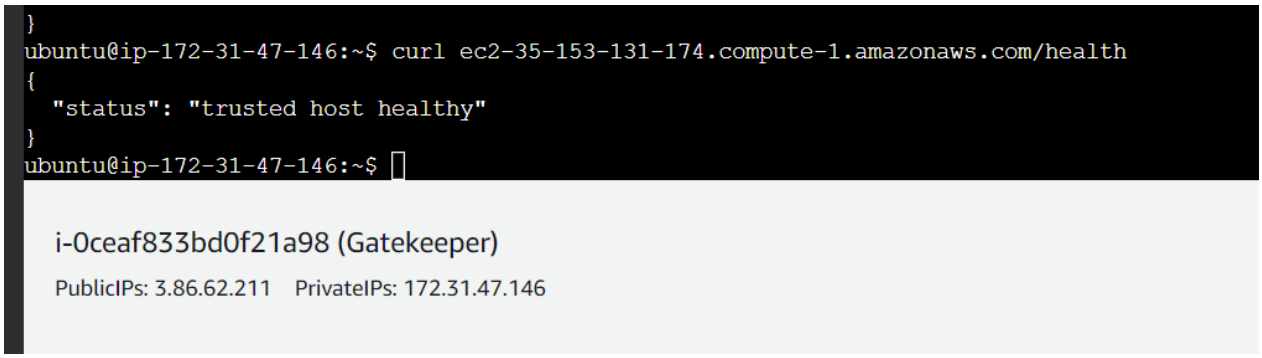

Figure 6: Hitting Trusted Host's /health within Gatekeeper

Figure 7 : Hitting Trusted Host's /query within Gatekeeper

Although this result is not the same if the *Trusted Host* is targeted with a machine that is not the *Gatekeeper*. For testing purposes, through *Postman*, locally, I have targeted the public DNS of the *Trusted Host*. Both endpoints, */health* and */query*, were timed out. Therefore, our security group for the *Trusted Host* is working well. Although, if we try and target the */query* endpoint of our *Gatekeeper*, we are met with the proper response.

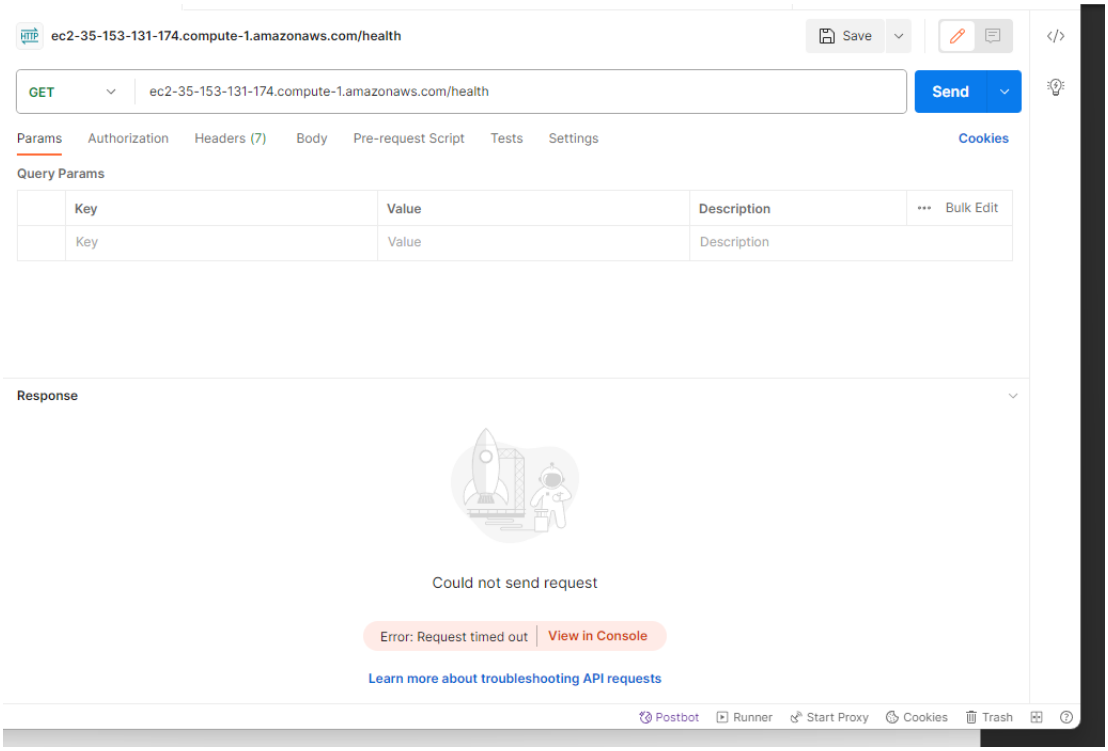

Figure 8: Hitting Trusted Host's /health on Postman locally

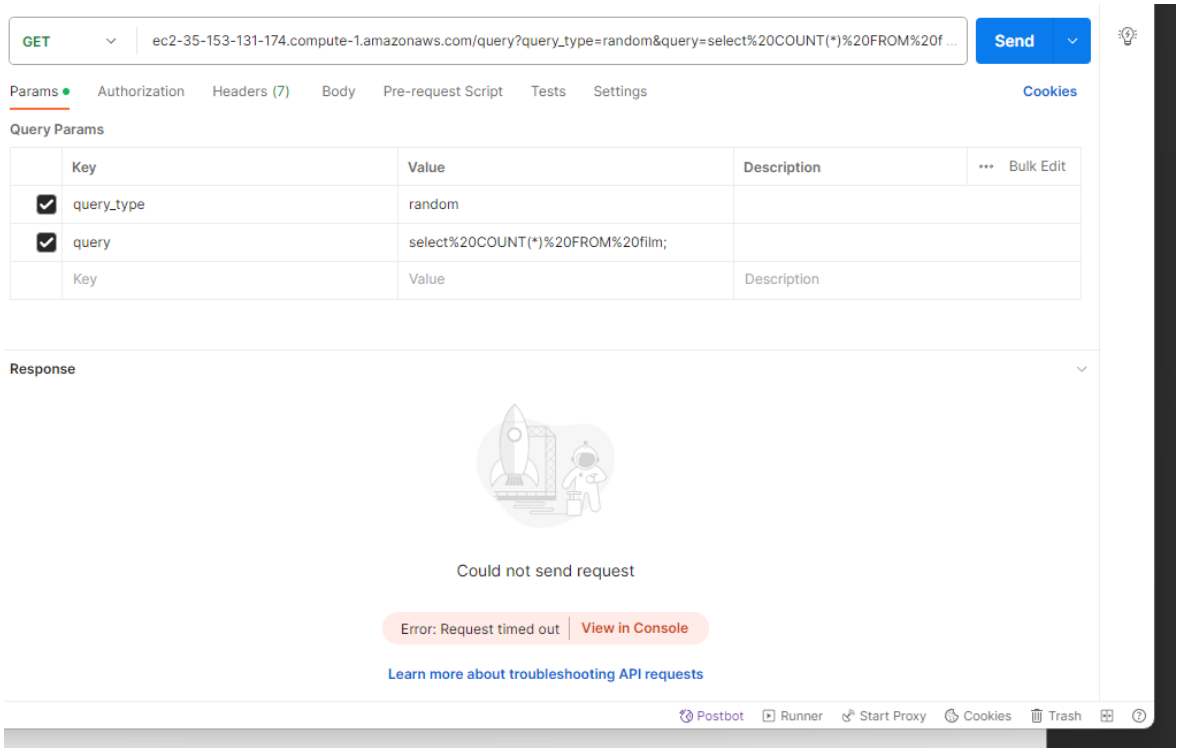

Figure 9: Hitting Trusted Host's /query on Postman locally

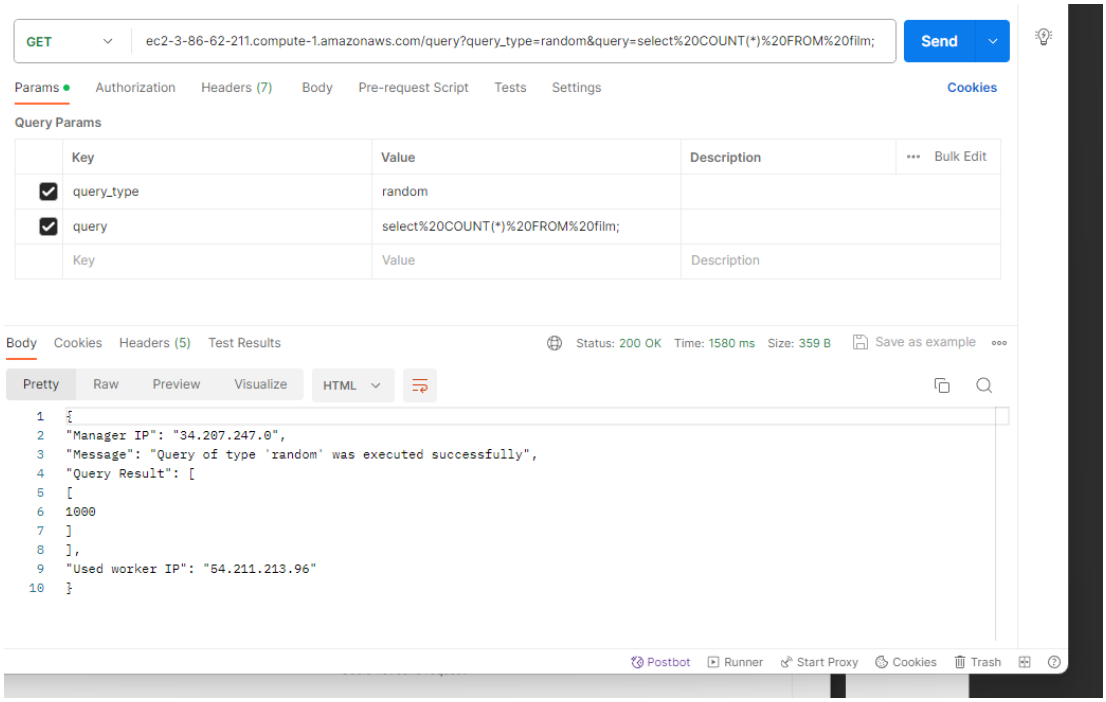

Figure 10: Hitting Gatekeeper's /query on Postman locally

As for the queries, all returns what they should return. A query type of type "random" returns a random worker public IP address. As for the query type "direct_hit" we see that *User worker IP* is the same as *Manager IP*. Also the *SQL* query is processed as it should. Depending on the *SQL* query provided in the request, the answers is returned accordingly.

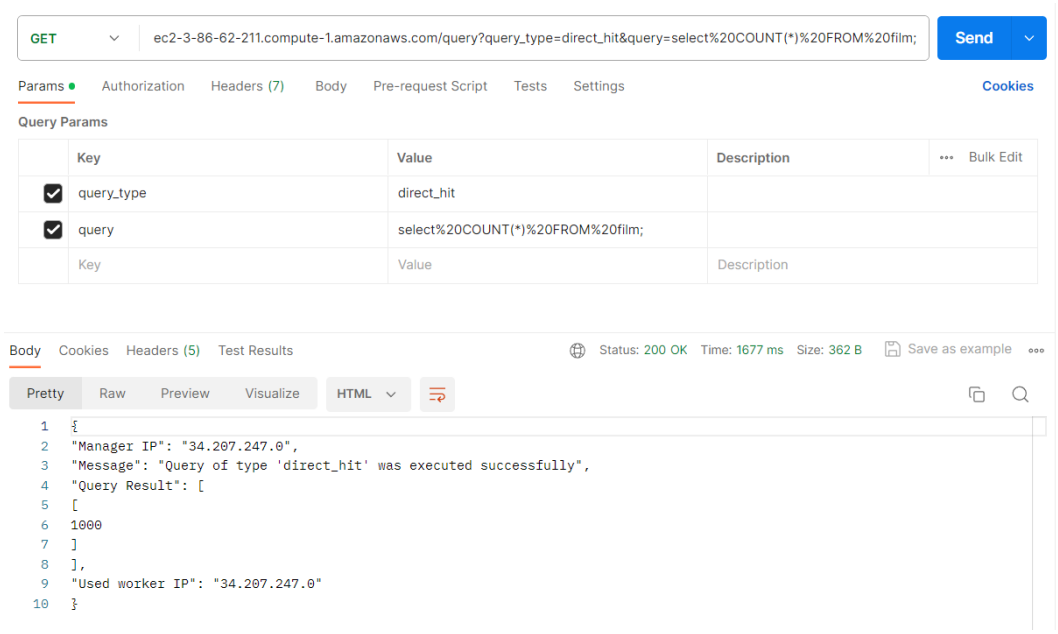

Figure 11: Hitting Gatekeeper's /query with query type "direct_hit" on Postman locally

Instruction to the run the code:

1. Clone the repo through https://github.com/ikrash3d/TP1-LOG8415-TF or download the zip file and extract the project.
2. On *AWS* in the *EC2* service, create a ***RSA*** type ***.pem*** key. After creating the key download it.
3. In each of these folders : ***/gatekeeper, /proxy*** and ***/trusted_host***, add the ***.pem*** key you created.
4. Create a python file named ***aws_creds.py***. In each file, manually, add the two variables with their respective values. The variables are : ***AWS_ACCESS_KEY*** and ***AWS_SECRET_ACCESS_KEY***
5. After this file is created, you will need to add this file in these folders: ***/gatekeeper***, ***/proxy*** and ***/trusted_host***
6. In a bash terminal, go in the ***/scripts*** folder. Launch the creation of the project with the following command "**bash run.sh**"
7. Follow the instructions provided in the terminal. Provide your ***AWS_ACCESS_KEY*** and ***AWS_SECRET_ACESS_KEY***.