LOG8415
Advanced Concepts of Cloud Computing

Vlad Drelciuc (1941366)
Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada

1 Project Setup and Running the Code

1.1 Prerequisites

The source code for this assignment can be found here: `https://github.com/vdrelciuc/LOG8415-projet`

Before using this project, make sure to follow these steps:

1. Install Python 3
2. Install project dependencies:

   ```
   pip install boto3
   ```

3. Install AWS CLI
4. Configure your AWS Access Key:

   ```
   aws configure
   ```

5. Replace **SUBNET_ID** in **cluster_benchmark.py** and **proxy_deploy.py** with your own AWS Subnet (select one with CIDR 172.31.0.0/20)

1.2 Standalone MySQL Server Benchmark Setup

To benchmark your standalone MySQL Server instance, simply run:

```
python standalone_benchmark.py
```

Once the script is executed, you can SSH into your Standalone instance. You will find the benchmark output saved under **/home/ubuntu/results.txt**

1.3 MySQL Cluster Benchmark Setup

To benchmark your MySQL Cluster, simply run:

```
python cluster_benchmark.py
```

Once the script is executed, you can SSH into your Manager Node instance. You will find the benchmark output saved under **/home/ubuntu/results.txt**

1.4 Proxy Setup

To implement the Proxy Cloud Pattern, you will require the same infrastructure as the MySQL Cluster Benchmark (1 manager node + 3 data nodes) + a Proxy instance.

If you haven't already done so, start by creating the MySQL Cluster by running:

```
python cluster_benchmark.py
```

To create the Proxy instance, simply run the following:

```
python proxy_deploy.py
```

Once the entire infrastructure has been deployed, copy the SSH private key that you used to create the cluster EC2 instances (by default: vockey.pem) from your local machine to the Proxy:

```
scp -i /local/path/to/vockey.pem ubuntu@<PROXY_PUBLIC_IP>:LOG8415-projet/key.pem
```

Then, SSH into the Proxy instance and navigate to **/home/ubuntu/LOG8415-projet/**
From there, you can run the following proxy strategies:

- Direct Hit:

  ```
  python3 app.py direct [sql_query_between_double_quotes]
  ```

- Random:

  ```
  python3 app.py random [sql_query_between_double_quotes]
  ```

- Custom:

  ```
  python3 app.py custom [sql_query_between_double_quotes]
  ```

A few remarks:

- The Direct Hit strategy will always target the master node.
- The Random strategy will always target a random worker node.
- The Custom strategy will target the node with the lowest average ping (can be either the master or one of the workers).
- The Proxy will scan the SQL query for a "SELECT" statement, an indicator of a read-only request. If no "SELECT" statement is found, the app will automatically shift to a Direct Hit strategy, as write requests (INSERT, DELETE, ALTER, etc.) can only be processed by the master node.

2 Benchmarking MySQL standalone vs. MySQL Cluster

For this experiment, I used Sysbench to benchmark the performance of MySQL standalone vs a MySQL Cluster using NBD. I used a read+write test to get an overall understanding of their performance difference. Some other experiment parameters include a table size of 100k items, 6 threads and a 60 seconds time limit. Each test has been run 3 times and Table 1 presents an overview of the average results.

Table 1: Average Benchmark Results for MySQL Standalone vs. MySQL Cluster

	MySQL Standalone	MySQL Cluster
SQL queries performed:		
read:	364448	135478
write:	104128	38708
other:	52064	19354
total:	520640	193540
transactions:	26032	9677
Latency (ms):		
min:	3.80	21.90
avg:	13.83	37.21
max:	705.61	106.25

We notice that the standalone instance processed almost 2.7 times the total amount of queries of the cluster in the same time frame (60 seconds). We also notice that the average latency of a standalone request is almost 5.75 times smaller than the one of a cluster request. Both of these results can be explained by the fact that a standalone instance processes everything efficiently on a single machine. On the opposite, the cluster spreads the work across 4 different machines (1 master + 3 worker nodes), which introduces overhead work (for routing the requests) as well as network latency.

These results may make the standalone instance look more appealing at the first glance, but it's important to say that the cluster would scale a lot better if the tables were larger, or if the system had more queries per second to treat. In that case, the standalone system would bottleneck faster than the cluster, especially in a scenario with many more reads than writes. Furthermore, the cluster brings another major benefit, which is data redundancy (NBD was set to have 3 replicas of the data, which means each worker instance holds its own copy of the data).

3 Implementation of The Proxy pattern

I implemented the Proxy cloud pattern on a different machine running on the same network as the MySQL cluster. The overall network configuration is:

```
NODE      : INTERNAL DNS
master    : ip-172-31-2-1.ec2.internal
worker1   : ip-172-31-2-2.ec2.internal
worker2   : ip-172-31-2-3.ec2.internal
```

```
worker3   : ip-172-31-2-4.ec2.internal
proxy     : ip-172-31-2-5.ec2.internal
```

The Proxy hosts a Python app that takes 2 parameters: a strategy and an SQL query. The strategy can be "direct", "random" or "custom". The SQL query must be between double quotes. I have only tested SELECT and INSERT statements, but other statements such as DELETE or ALTER should also work.

- Selecting the "direct" strategy will simply forward the SQL query directly to the master node. This is done via an SSH tunnel from proxy to master. From there, the request is processed locally in MySQL. The "direct" strategy can be used for any type of query since the master node can process everything (reads and writes).
- Selecting the "random" strategy will forward the SQL query to a random worker node. This is also done via an SSH tunnel from proxy to master, but this time, the request is processed on the selected worker node via port binding. The "random" strategy can only be used for SELECT queries since worker nodes can only read data, but not write it.
- Selecting the "custom" strategy will forward the SQL query to the node with the lowest average ping from the proxy. This could either be the master or any of the worker nodes. Once again, we are using an SSH tunnel from proxy to master and port binding to select the desired node. My implementation of the "custom" strategy only supports SELECT queries, since we don't know whether the lowest latency node will be master or a worker, and therefore cannot guarantee that a write query could be processed if it fell on a worker.