LOG8415E : Advanced Concepts of Cloud Computing

Final Project: Scaling Databases and Implementing Cloud Design Patterns

Handover documentation

Huard Eric, 2228242

github repository : https://github.com/ehuard/LOG8415E_Project
(made public only after december 29th)

presentation video: https://youtu.be/b4A5zQb1Bhk

December 27th, 2023

I) Introduction

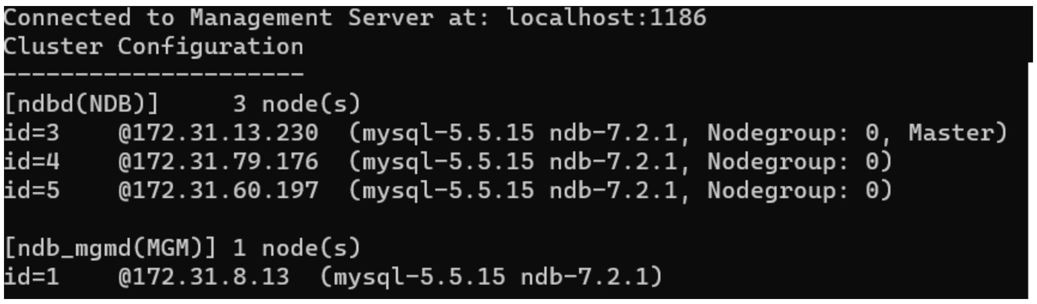
With the growing significance of cloud computing, numerous challenges have arisen, prompting the development of reusable solutions known as cloud design patterns. The aim of this project is to establish a MySQL Cluster on Amazon EC2 and integrate two specific design patterns: the Proxy pattern and the Gatekeeper pattern.

Through this project, I gained practical experience working with a distributed database, specifically MySQL Cluster. I set it up and conducted a benchmark against the standalone version of MySQL. Subsequently, I acquired knowledge on implementing a strategy discussed in class, commonly employed for distributing data across VM instances. Lastly, delving into the Gatekeeper pattern allowed me to further understand a security-enhancing strategy discussed in class, involving the addition of a broker layer.

II) Benchmarking MySQL and MySQL Cluster

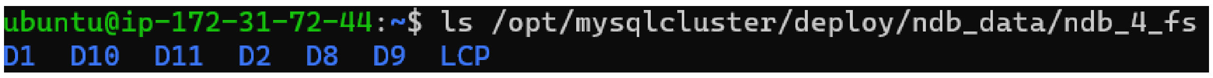
For the first part of this assignment, we had to setup MySQL on a standalone EC2 instance and MySQL Cluster on a cluster made of one master instance and three workers. All the instances are `t2.micro` instances.
The instanciation of the different virtual machines is done in `setup/main.py`. All the commands used to setup the standalone version can be found in `setup/setup_mysql_standalone.sh`. In order to setup the cluster, we have to inject the ip-adresses or private dns names of the other instances of the cluster in some configuration files. Thus, the first few commands can be found in `setup/setup_master.sh` and `setup/setup_slaves.sh`. The next commands are dynamically generated from functions defined in `setup/utils_create_scripts.py`.

After all the commands have been run, we can see that all the nodes are connected to form the cluster (by running `ndb_mgm -e show`) and that the data, initially only present on the master, has been replicated to the workers:



```
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     3 node(s)
id=3    @172.31.13.230  (mysql-5.5.15 ndb-7.2.1, Nodegroup: 0, Master)
id=4    @172.31.79.176  (mysql-5.5.15 ndb-7.2.1, Nodegroup: 0)
id=5    @172.31.60.197  (mysql-5.5.15 ndb-7.2.1, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @172.31.8.13  (mysql-5.5.15 ndb-7.2.1)
```



```
ubuntu@ip-172-31-72-44:~$ ls /opt/mysqlcluster/deploy/ndb_data/ndb_4_fs
D1  D10  D11  D2  D8  D9  LCP
```

Considering that this project is primarily aimed at gaining initial experience with MySQL Cluster and design patterns, it's worth noting that I didn't extensively delve into the configuration of MySQL. For instance, the username utilized is simply "root", and if required, the password is also set as "root." It's crucial to emphasize that in a real deployment scenario, a database would necessitate significantly heightened security measures. I have however added a firewall to accept traffic only with the other instances of the cluster and with the proxy (SSH can still be done from anywhere but it needs the encryption key).

In order to perform the benchmark between MySQL standalone and MySQL Cluster, I used the Sakila sample database. The benchmark is done using the benchmarking tool Sysbench. The commands used for this benchmark can be found in tests/orm_banchmark.py. I am running an olt read-write benchmark on 4 threads with a table size set to 1 000 000 and an unlimited number of requests. The results are presented in the table below.

Read-Write Benchmark		
	Cluster	Standalone
Execution time (s)	10.0085 s	10.0074 s
# read queries	55 916	30 198
# write queries	15 976	8 628
total number of queries	79 880	43140
queries/second	7 971.71	4 309.90
# transactions	3 994	2 157
transactions/second	398.99	215.49
mean latency (ms)	10.02 ms	18.55 ms
max latency (ms)	32.18 ms	73.68 ms

In every aspect, the cluster version performs better. It performs 85% more queries in the same time as the standalone version. The same goes for the number of transactions. In terms of latency, the standalone version has 85% more latency on average but it can be more than double of the cluster's latency on some transactions.

On the read only benchmark, the cluster outperforms the standalone implementation by about 20%.

Read Only Benchmark		
	Cluster	Standalone
Execution time (s)	9.9962 s	10.0688 s
# read queries	86 534	67 186
total number of queries	98 896	76 784
queries/second	9 882.92	7 672.57
# transactions	6 181	4799
transactions/second	617.68	479.54
mean latency (ms)	6.47 ms	8.33 ms
max latency (ms)	12.83 ms	15.82 ms

III) Proxy implementation

The next step of this project was to implement the Proxy design pattern. Since we have data replication between the master and the datanodes in the cluster, the proxy can route requests and provides read scalability on a relational database. Three implementations were required for this pattern:

- Direct hit: incoming requests are directly forwarded to MySQL master node and there is no logic to distribute data.
- Random: randomly choose a slave node on MySQL cluster and forward the request to it.
- Customized: measure the ping time of all the servers and forward the message to one with less response time.

This only applies to the read requests. All the write requests are handled by the manager and replicated on its workers.

The proxy is implemented via a Flask application running on a `t2.large` instance. The code used to add the proxy can be found in `setup/utils_proxy.py`. I used mysql-connector-python to connect to the database. It could also have been done by using pymysql. The flask application is running on port 5000, a non privileged port used by default by Flask.

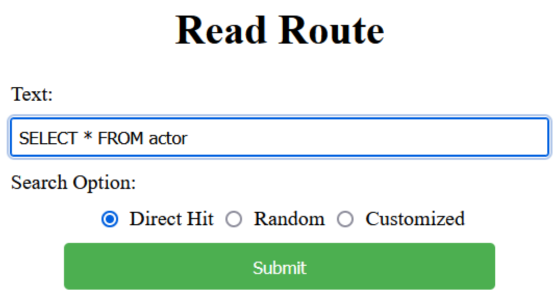
Note that for the customized mode, I ping the workers just before sending the request. I think it could be better if we pinged every few milliseconds and stored the results of these pings somewhere. Indeed, since we have to wait for the return of the 3 pings, doing them before every request can cause some latency. I did not implement this idea to keep it simple and due to lack of time.

Since the proxy has the port 5000 open and an application running, I added a firewall so it can only connect through SSH (and the encryption key is stored on my computer) or communicate with the cluster and the trusted host.

IV) Gatekeeper implementation

Lastly, the implementation of the Gatekeeper pattern was required. This pattern outlines a method for brokering access to storage. It represents a standard security best practice aimed at minimizing the attack surface of system roles. This is achieved by internal communication limited to other roles within the pattern. The Gatekeeper pattern involves two roles engaged in the gatekeeping process. An Internet-facing web role manages user requests, but the Gatekeeper, being cautious, distrusts all incoming requests. It validates the input and operates with partial trust. In the event of a successful attack on the web role by a hacker, no sensitive data is stored there.

The internet-facing instance of the gatekeeper is a t2.large instance with a Flask application running and listening on port 80. This application can be accessed by anyone from anywhere: it is the only entry point of our system. A user can acces it by typing the address in its web browser, for instance http://ec2-3-235-249-218.compute-1.amazonaws.com/read. He can then type a SQL query and select the routing mode (direct-hit, random or customized). In order to perform a write request, like an INSERT query, the user needs to change the “read” in the previous link by “write”.



SELECT * FROM actor

[✓ Direct Hit] Random Customized

To set up this application, I send the 3 files `templates/read.html`, `templates/write.html` and `template/result.html` on the virtual machine. The commands used to generate the code and launch the application can be found in `setup/utils_gatekeeper.py` and in `setup/main.py`.

The second part of the Gatekeeper pattern is the trusted host. It is a t2.large instance with a Flask application running on port 5000 that will receive a request, perform some security and sanity checks, and forward the request to the proxy if no problem is detected. If a problem is detected, it directly responds to the internet-facing app.
Numerous procedures are possible for validating and sanitizing incoming requests. Due to time constraints, I prioritized the implementation of specific simple measures. These include ensuring the trusted host validates the request's routing mode, confirming the query adheres to proper SQL syntax, and verifying the absence of any UNION operator. This precaution is taken to prevent potential issues such as the creation of a temporary table with an

overwhelming number of rows, leading to system crashes or inaccessibility. Notably, I emphasized the importance of designing the system to facilitate the seamless addition of new checks and sanitization methods The commands used to generate the code and launch the application can be found in setup/utils_gatekeeper.py and in setup/main.py.

V) How to run the code

First of all, you will need to get your own access keys to Amazon Web Services. You specifically need :

- aws_access_key_id;
- aws_secret_access_key;
- aws_session_token.

These need to be properly configured on your computer at `/.aws/credentials`.
Once all keys are set up, run `pip install -r requirements`, eventually in a virtual environment if you prefer . It will verify and install all needed libraries.

Once the steps above are done, you can run `python3 ./setup/main.py` in order to setup all the instances. This will take some time (more than 5 minutes), so you can go grab a cup of coffee in the meantime. Be careful, it requires you to be in the project directory to run properly. It also needs you to have the write permission. You will also need to make sure the key created has the right permissions to connect to the instances using SSH. If it fails because you have too large permissions, please change them and run the program again : it will not delete the existing key and you will be able to connect with no issue.

To get the benchmark results, run `python3 ./tests/perform_benchmarks.py`. It will form the benchmarks on the distant machines and fetch them in your local directory (in /tests/mark_results) after one minute.

In order to access the application, look at the addresses printed at the end of the initialization. They sould look like that http://ec2-3-235-249-218.compute-1.amazonaws.com/read and http://ec2-3-235-249-218.compute-1.amazonaws.com/write. On these pages, you can either submit a read or a write request, depending on which page your are currently on.

It is possible that some parts of the initialization went wrong. Namely, you may have to connect through SSH to the gatekeeper, trusted_host and proxy instances and run these commands:

- on the gatekeeper:
    - sudo pip3 install flask requests
    - sudo python3 flask_app.py
- on the trusted host:
    - pip3 install flask requests sqlfluff
    - python3 flask_app.py
- on the proxy:

- pip3 install flask ping3 mysql-connector-python
- python3 flask_app.py

The public dns of these instances can be found in data.json. They are also printed during initialization.