LOG8415E: ADVANCED CONCEPTS OF CLOUD COMPUTING

ASSIGNMENT 3

Presented to
AMIN NIKANJAM, PH. D,
Department of Computer Engineering and Software Engineering

By
MATHIEU BROSSEAU

Ecole Polytechnique de Montréal
Monday October 28th 2024

1. Benchmarking MySQL with sysbench.

I run those commands on the 3 instances of the cluster:

```
# Use sysbench to benchmark the instance
"sudo apt-get install sysbench -y",
f"sudo sysbench /usr/share/sysbench/oltp_read_only.lua --mysql-db=sakila --mysql-user=\"{MYSQL_USER}\" --mysql-password=\"{MYSQL_PASSWORD}\" prepare",
f"sudo sysbench /usr/share/sysbench/oltp_read_only.lua --mysql-db=sakila --mysql-user=\"{MYSQL_USER}\" --mysql-password=\"{MYSQL_PASSWORD}\" run > {benchmark_file}",
```

Then I scp the result to my local output folder:

```
# Download the benchmark results
os.makedirs("output", exist_ok=True)
local_file_path = f"./output/sysbench_results_{instance_name}.txt"
scp_command = f"scp -o StrictHostKeyChecking=no -i {self.key_wrapper.key_file_path} ubuntu@{public_ip}:{benchmark_file} {local_file_path}"
os.system(scp_command)
print(f"Downloaded benchmark results to {local_file_path}")
```

Results:

<table>
<tr>
<td>Master</td>
<td><pre>
SQL statistics:
    queries performed:
        read: 119154
        write: 0
        other: 17022
        total: 136176
    transactions: 8511   (850.93 per sec.)
    queries: 136176 (13614.83 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: 8511

Latency (ms):
         min: 0.82
         avg: 1.17
         max: 21.65
         95th percentile: 1.96
         sum: 9972.17
</pre></td>
</tr>
</table>

<table>
<tr><td>Worker1</td><td>

```
SQL statistics:
    queries performed:
        read:                            118272
        write:                           0
        other:                           16896
        total:                           135168
    transactions:                        8448   (844.60 per sec.)
    queries:                             135168 (13513.59 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0005s
    total number of events:              8448

Latency (ms):
         min:                                    0.82
         avg:                                    1.18
         max:                                  102.60
         95th percentile:                        1.96
         sum:                                 9975.28
```

</td></tr>
<tr><td>Worker2</td><td>

```
SQL statistics:
    queries performed:
        read:                            114016
        write:                           0
        other:                           16288
        total:                           130304
    transactions:                        8144   (814.16 per sec.)
    queries:                             130304 (13026.52 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0012s
    total number of events:              8144

Latency (ms):
         min:                                    0.83
         avg:                                    1.23
         max:                                  180.63
         95th percentile:                        1.96
         sum:                                 9976.91
```

</td></tr>
</table>

The sysbench results are very similar on each instance.

2. Implementation of The Proxy pattern.

```python
nodes = {
    'master_host': master_ip,
    'worker1_host': worker1_ip,
    'worker2_host': worker2_ip,
}

# Function to connect to MySQL
def connect_to_mysql(host_ip):
    connection = pymysql.connect(
        host=host_ip,
        port=mysql_port,
        user=mysql_user,
        password=mysql_password,
        database=database
    )
    return connection

# Function to determine whether it's a read or write query
def is_write_query(query):
    write_keywords = ['INSERT', 'UPDATE', 'DELETE', 'CREATE', 'ALTER', 'DROP']
    query_upper = query.strip().upper()
    return any(query_upper.startswith(keyword) for keyword in write_keywords)

# Function to ping a node
def ping_node(host):
    """Ping a server and return its response time in milliseconds."""
    try:
        result = subprocess.run(
            ["ping", "-c", "1", host],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )
        response_line = result.stdout.split("\n")[1]
        time_ms = float(response_line.split("time=")[-1].split(" ")[0])
        return time_ms
```

```python
@app.post('/query')
def proxy_query(data: QueryRequest):
    query = data.query
    implementation = data.implementation

    if not query:
        return HTTPException(status_code=400, detail="No query provided")

    if is_write_query(query):
        receiver = 'master_host'
    else: # Read query
        if implementation == 1: # Direct hit
            receiver = 'master_host'
        elif implementation == 2: #Random between the workers
            receiver = random.choice(['worker1_host', 'worker2_host'])
        else: # Customized (least ping time)
            ping_times = [
                {"node": node, "ping": ping_node(nodes[node])}
                for node in nodes
            ]
            receiver = min(ping_times, key=lambda x: x["ping"])["node"]

    connection = connect_to_mysql(nodes[receiver])
    try:
        with connection.cursor() as cursor:
            cursor.execute(query)
            if query.strip().upper().startswith('SELECT'):
                return {'status': 'success', 'receiver': receiver, 'data': cursor.fetchall()}
            else:
                connection.commit()
                return {'status': 'success', 'receiver': receiver}
    except Exception as e:
        return HTTPException(status_code=500, detail=str(e))
    finally:
        connection.close()
```

The proxy app receives a request with query and implementation in the body. It checks if it's a read or write request by looking for keywords in the query. If it's a write request, it sets the receiver as the master node. If it's a read request, it uses the 'implementation' value to determine what action to do. After connecting to the correct instance and executing the query, it returns a simple response with status success and the name of the receiver for analysis.

3. Implementation of The Gatekeeper pattern.

```python
app = FastAPI()

IP_FORWARD = os.getenv('IP_FORWARD')
PORT_FORWARD = int(os.getenv('PORT_FORWARD', 8000))
PORT_APP = int(os.getenv('PORT_APP', 8000))

class QueryRequest(BaseModel):
    query: str
    implementation: int

@app.post("/query")
def handle_request(data: QueryRequest):
    try:
        query = data.query
        implementation = data.implementation
        if not query or not implementation:
            raise HTTPException(status_code=400, detail="Missing arguments in request body")
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid JSON payload")

    try:
        trusted_host_url = f"http://{IP_FORWARD}:{PORT_FORWARD}/query"
        trusted_host_response = requests.post(trusted_host_url, json={"query": query, "implementation": implementation})
        trusted_host_response.raise_for_status()
        return JSONResponse(content=trusted_host_response.json(), status_code=trusted_host_response.status_code)
    except requests.RequestException as e:
        raise HTTPException(status_code=502, detail=f"Error forwarding request: {str(e)}")

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=PORT_APP)
```

I also set up a firewall to improve security on both instances on top of the security groups:

```
"sudo apt install -y ufw",
"sudo ufw allow 22",
"sudo ufw allow 8000",
```

I set up the exact same app on the gatekeeper and on the trusted host. The internet facing gatekeeper takes the trusted host private ip and port as env parameters, while the internal facing trusted host takes the proxy private ip and port as env parameters. It takes in the requests, makes sure it has “implementation” and “query” in its body request, then creates a new request using those parameters. That way, we are 100% sure that we don’t send malicious stuff that could be in the request to our proxy.

4. Benchmarking the clusters.

```python
def send_query(self, query, implementation, url):
    try:
        response = requests.post(url, json={"query": query, "implementation": implementation})
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        return {"error": str(e)}
```

```python
def load_test(self):
    gatekeeper_ip = self.get_public_ip(self.inst_wrapper.instances[5]["InstanceId"])
    url = f"http://{gatekeeper_ip}:{PORT_APP}/query"
    execution_times = {}
    dispersion_of_reads_dict = {}

    for implementation in range(1, 4): # 1: direct, 2: random, 3: ping
        dispersion_of_reads = {}
        read_success = 0
        write_success = 0
        print(f"Running benchmark for implementation {implementation}...")
        initial_time = time.time()
        for _ in range(N_REQUESTS):
            query = self.generate_read_query()
            result = self.send_query(query, implementation, url)
            if "error" not in result:
                receiver = result["receiver"]
                if receiver in dispersion_of_reads:
                    dispersion_of_reads[receiver] += 1
                else:
                    dispersion_of_reads[receiver] = 1
                read_success += 1
            else:
                print(f"Read error: {result['error']}")
                break

        for _ in range(N_REQUESTS):
            query = self.generate_write_query()
            result = self.send_query(query, implementation, url)
            if "error" not in result:
                write_success += 1
            else:
                print(f"Write error: {result['error']}")
                break
```

```python
    execution_times[implementation] = time.time() - initial_time
    dispersion_of_reads_dict[implementation] = dispersion_of_reads
    print(f"Read success: {read_success}/{N_REQUESTS}")
    print(f"Write success: {write_success}/{N_REQUESTS}")

# output the data to ./output/benchmark_results.txt
with open("./output/benchmark_results.txt", "w") as f:
    f.write(f"Execution times: {execution_times}\n")
    f.write(f"Dispersion of reads: {dispersion_of_reads_dict}\n")
```

It's pretty basic, but I made a sample read and a sample write request. For each implementation, I send 1000 read and write requests. I save the total time to make both + the repartition of read requests for each implementation. I then write the data to a txt file.

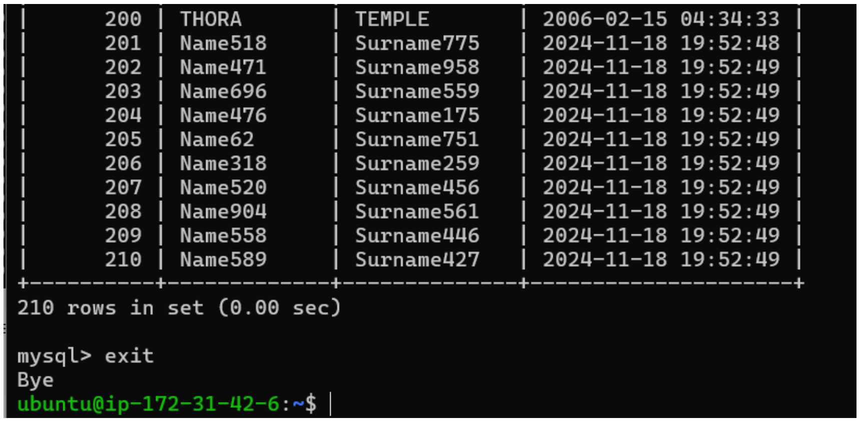
Results:

```
Press y to benchmark the results...y
Running benchmark for implementation 1...
Read success: 1000/1000
Write success: 1000/1000
Running benchmark for implementation 2...
Read success: 1000/1000
Write success: 1000/1000
Running benchmark for implementation 3...
Read success: 1000/1000
Write success: 1000/1000
Press y to clean up...
```

```
Execution times: {1: 211.8894443511963, 2: 208.42263746261597, 3: 216.1906440258026}
Dispersion of reads: {1: {'master_host': 1000}, 2: {'worker2_host': 516, 'worker1_host': 484}, 3: {'worker1_host': 495, 'master_host': 320, 'worker2_host': 185}}
```

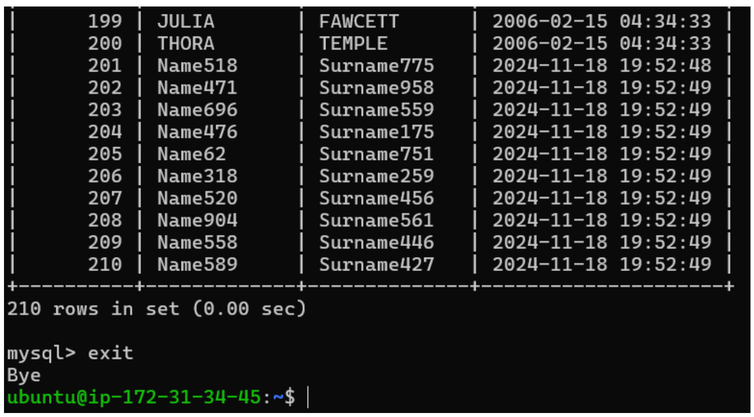
Execution times were 211, 208 and 216s, which is pretty similar. On dispersion of reads, for 1 and 2 it's pretty basic, but for 3 I was surprised that worker1 handled almost half of the requests.

I also made a run with only 10 requests to show that the replication works:

Master: 

```
|        200 | THORA        | TEMPLE       | 2006-02-15 04:34:33 |
|        201 | Name518      | Surname775   | 2024-11-18 19:52:48 |
|        202 | Name471      | Surname958   | 2024-11-18 19:52:49 |
|        203 | Name696      | Surname559   | 2024-11-18 19:52:49 |
|        204 | Name476      | Surname175   | 2024-11-18 19:52:49 |
|        205 | Name62       | Surname751   | 2024-11-18 19:52:49 |
|        206 | Name318      | Surname259   | 2024-11-18 19:52:49 |
|        207 | Name520      | Surname456   | 2024-11-18 19:52:49 |
|        208 | Name904      | Surname561   | 2024-11-18 19:52:49 |
|        209 | Name558      | Surname446   | 2024-11-18 19:52:49 |
|        210 | Name589      | Surname427   | 2024-11-18 19:52:49 |
+----------+------------+-------------+---------------------+
210 rows in set (0.00 sec)

mysql> exit
Bye
ubuntu@ip-172-31-42-6:~$
```

One of the worker: 

```
|        199 | JULIA        | FAWCETT      | 2006-02-15 04:34:33 |
|        200 | THORA        | TEMPLE       | 2006-02-15 04:34:33 |
|        201 | Name518      | Surname775   | 2024-11-18 19:52:48 |
|        202 | Name471      | Surname958   | 2024-11-18 19:52:49 |
|        203 | Name696      | Surname559   | 2024-11-18 19:52:49 |
|        204 | Name476      | Surname175   | 2024-11-18 19:52:49 |
|        205 | Name62       | Surname751   | 2024-11-18 19:52:49 |
|        206 | Name318      | Surname259   | 2024-11-18 19:52:49 |
|        207 | Name520      | Surname456   | 2024-11-18 19:52:49 |
|        208 | Name904      | Surname561   | 2024-11-18 19:52:49 |
|        209 | Name558      | Surname446   | 2024-11-18 19:52:49 |
|        210 | Name589      | Surname427   | 2024-11-18 19:52:49 |
+----------+------------+-------------+---------------------+
210 rows in set (0.00 sec)

mysql> exit
Bye
ubuntu@ip-172-31-34-45:~$
```

5. Describe clearly how your implementation works.

Scenario:

```python
if not self.retrieve_instance(INSTANCE_NAME_1):
    #1: create DB cluster (3 t2.micro instances)
    self.create_and_list_key_pairs()
    self.create_default_security_group()
    self.create_named_instance(INSTANCE_NAME_1, INSTANCE_TYPE_MICRO)
    self.create_named_instance(INSTANCE_NAME_2, INSTANCE_TYPE_MICRO)
    self.create_named_instance(INSTANCE_NAME_3, INSTANCE_TYPE_MICRO)
    for instance in self.inst_wrapper.instances:
        #2: setup DB cluster with replication (1 manager, 2 workers) and benchmark with sysbench
        print(f"Installing MySQL on instance {instance['InstanceName']}...")
        self.setup_and_benchmark_mysql(instance["InstanceId"], instance["InstanceName"])
else:
    self.retrieve_instance(INSTANCE_NAME_2)
    self.retrieve_instance(INSTANCE_NAME_3)
if not self.retrieve_instance(INSTANCE_NAME_PROXY):
    #3: implement Proxy pattern (1 instance t2.large) with 3 implementations
    self.create_named_instance(INSTANCE_NAME_PROXY, INSTANCE_TYPE_LARGE)
    self.setup_proxy(self.inst_wrapper.instances[3]["InstanceId"])
#4: implement Gatekeeper pattern (2 t2.large instances)
if not self.retrieve_instance(INSTANCE_NAME_TRUSTED_HOST):
    self.create_named_instance(INSTANCE_NAME_TRUSTED_HOST, INSTANCE_TYPE_LARGE)
    self.setup_gatekeeper(self.inst_wrapper.instances[4]["InstanceId"], self.get_private_ip(self.inst_wrapper.instances[3]["InstanceId"]))
if not self.retrieve_instance(INSTANCE_NAME_GATEKEEPER):
    self.create_named_instance(INSTANCE_NAME_GATEKEEPER, INSTANCE_TYPE_LARGE)
    self.setup_gatekeeper(self.inst_wrapper.instances[5]["InstanceId"], self.get_private_ip(self.inst_wrapper.instances[4]["InstanceId"]))
#5: add security groups for the instances
self.create_security_groups()
#6: benchmark the results (1000 reads, 1000 writes for each implementation)
if input("Press y to benchmark the results...") in ["y", "Y"]:
    self.load_test()
#7: clean up
if input("Press y to clean up...") in ["y", "Y"]:
    self.cleanup()
```

First I create a key pair for the ssh connections and a default security group allowing ssh from my computer that I use to set up every instance. I start by creating the 3 instances for the cluster. I use ssh to execute commands on them. I use common commands that install mysql, sets up a user and the sample database, then run sysbench. Then, depending on the master or the workers, I use different commands to set up replication. For the master status, I output the data to a file that I scp back to my computer so I'm able to use that data to set up the workers.

Then for the 3 other instances (proxy, trusted host and gatekeeper), I scp the FastAPI app to the instance, then install the necessary packages for it to work. I launch the app giving it the private IPs that are required by it to redirect traffic. Once all instances are setup, I create a 2nd security group for each type of instance in order to allow traffic from the required configuration.

sg-0360a62fea9d26050	TP3-SG-d25ee17d	vpc-00c5986cce8a120b4	Instances security
sg-0ad5dc037e75f5f4c	TP3-SG-DBCLUSTER-ccedc85a	vpc-00c5986cce8a120b4	DB Cluster security
sg-07ae61ee81cebb02b	TP3-SG-GATEKEEPER-f93d920e	vpc-00c5986cce8a120b4	Gatekeeper security
sg-0231ecd0929816830	TP3-SG-PROXY-8d4be954	vpc-00c5986cce8a120b4	Proxy security
sg-001cf5e3872524b4b	TP3-SG-TRUSTEDHOST-0495d141	vpc-00c5986cce8a120b4	Trusted Host security

The first security group is the default. It allows ssh access from my computer only. It is added to every instance created so I can run commands and deploy apps to them. The other security groups follow those rules:

- DBCLUSER only allows tcp3306 and icmp requests from Proxy private ip (internal facing)
- GATEKEEPER allows tcp8000 from all (internet facing)
- PROXY only allows tcp8000 from trusted host private ip (internal facing)
- TRUSTEDHOST only allows tcp8000 from gatekeeper private ip (internal facing)

The gatekeeper receives requests from the internet on port 8000. It validates the output, sends it to the trusted host that revalidates it and sends it to the proxy. The request is composed of a query and an implementation number. The proxy sends read requests to the instances following the implementation constraint. The write requests are sent to the master and replicated on the workers.
Finally, I run the benchmarking that sends requests to the public IP of the gatekeeper, gather the results and cleanup the created resources.

6. Summary of results and instructions to run your code.

The 3 proxy implementations had similar results, but I think that the 3rd implementation (ping all servers and sending the request to the one with least response time) would be best in a real-world context where read requests could take a much longer time. Even though it executed more code and logic, it still had similar results to the other 2 implementations. It also had an uneven distribution of requests, which shows that some instances are just more responsive than others. It would be interesting to compare data from sending it to the most responsive to sending it to the least responsive. It would also be interesting to use a much bigger sample read request to see the impact on the results.

To run my code, create a AWS_access.txt file containing CLI information inside /tp3. Then go to the project root and run “demo3.ps1” script. After running it, benchmark data for sysbench and DB cluster will be in tp3/output/ .

Link to Repo: https://github.com/mathieubrs1/LOG8415_final_assignement/tree/main
Link to video: https://www.youtube.com/watch?v=Ir-bXn2SJVI