LOG8415e

Advanced Concepts of Cloud Computing

Scaling Databases and Implementing Cloud Patterns

Final Project

Autumn 2022

By:
Jeremy Hage

Presented to:
Vahid Majdinasab

Friday, December 23, 2022

Github link https://github.com/jehag/8415.git

1 Benchmarking MySQL stand-alone vs. MySQL Cluster

To compare the performance of MySQL stand-alone against MySQL Cluster, it was required to install and use Sysbench on both of the MySQL servers. It is a good comparison tool since it can run the same tests on both systems to compare their performance. The tests ran were a read-write test, a read-only test, and a write-only test. Here are the commands used to run those tests:

```
sudo sysbench oltp_read_write --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    prepare
sudo sysbench oltp_read_write --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    --num-threads=6 --max-time=60 --max-requests=0 run

sudo sysbench oltp_read_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    prepare
sudo sysbench oltp_read_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    --num-threads=6 --max-time=60 --max-requests=0 run

sudo sysbench oltp_write_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    prepare
sudo sysbench oltp_write_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --mysql_storage_engine=ndbcluster \
    --num-threads=6 --max-time=60 --max-requests=0 run
```

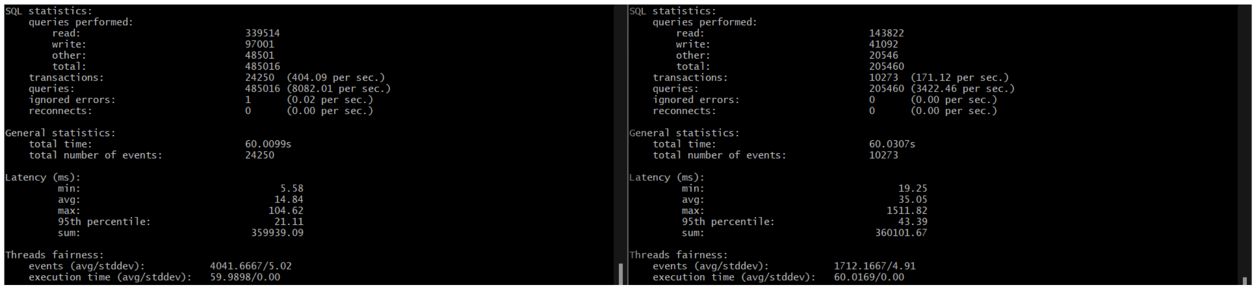
Those 6 commands were ran on the cluster MySQL, which explains the MySQL storage engine argument being ndbcluster. The following 6 commands were run on the stand-alone MySQL, and are similar to the commands ran on the cluster, with the exception of the MySQL storage engine argument which is missing, since the stand-alone does not need to specify this field.

```
sudo sysbench oltp_read_write --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root prepare
sudo sysbench oltp_read_write --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --num-threads=6 --max-time=60 \
    --max-requests=0 run

sudo sysbench oltp_read_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root prepare
sudo sysbench oltp_read_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --num-threads=6 --max-time=60 \
    --max-requests=0 run

sudo sysbench oltp_write_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root prepare
sudo sysbench oltp_write_only --table-size=999999 --db-driver=mysql \
    --mysql-db=sakila --mysql-user=root --num-threads=6 --max-time=60 \
    --max-requests=0 run
```

After running these commands, the console showed the result of the test done on both servers. This is the results side by side, by test. On the left is the stand-alone server, while on the right is the cluster server.



```
SQL statistics:
    queries performed:
        read:                            339514
        write:                           97001
        other:                           48501
        total:                           485016
    transactions:                        24250  (404.09 per sec.)
    queries:                             485016 (8082.01 per sec.)
    ignored errors:                      1      (0.02 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0099s
    total number of events:              24250

Latency (ms):
         min:                                    5.58
         avg:                                   14.84
         max:                                  104.62
         95th percentile:                       21.11
         sum:                               359939.09

Threads fairness:
    events (avg/stddev):           4041.6667/5.02
    execution time (avg/stddev):   59.9898/0.00
```

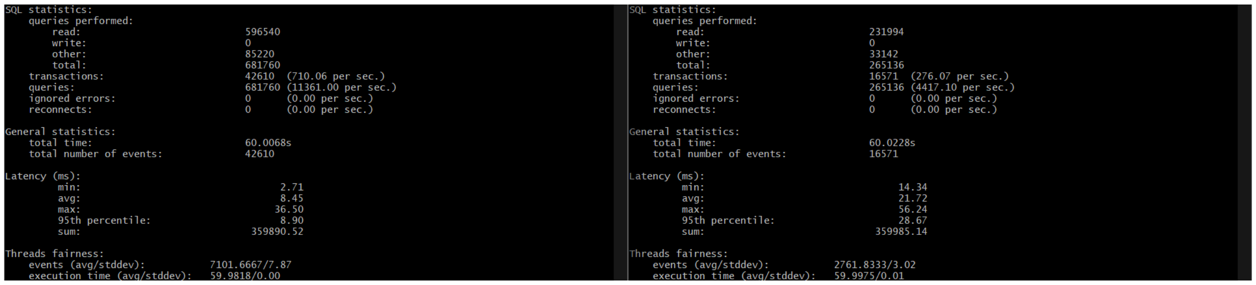
```
SQL statistics:
    queries performed:
        read:                            143822
        write:                           41092
        other:                           20546
        total:                           205460
    transactions:                        10273  (171.12 per sec.)
    queries:                             205460 (3422.46 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0307s
    total number of events:              10273

Latency (ms):
         min:                                   19.25
         avg:                                   35.05
         max:                                 1511.82
         95th percentile:                       43.39
         sum:                               360101.67

Threads fairness:
    events (avg/stddev):           1712.1667/4.91
    execution time (avg/stddev):   60.0169/0.00
```

Figure 1: Read-write console results



```
SQL statistics:
    queries performed:
        read:                            596540
        write:                           0
        other:                           85220
        total:                           681760
    transactions:                        42610  (710.06 per sec.)
    queries:                             681760 (11361.00 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0068s
    total number of events:              42610

Latency (ms):
         min:                                    2.71
         avg:                                    8.45
         max:                                   36.50
         95th percentile:                        8.90
         sum:                               359890.52

Threads fairness:
    events (avg/stddev):           7101.6667/7.87
    execution time (avg/stddev):   59.9818/0.00
```

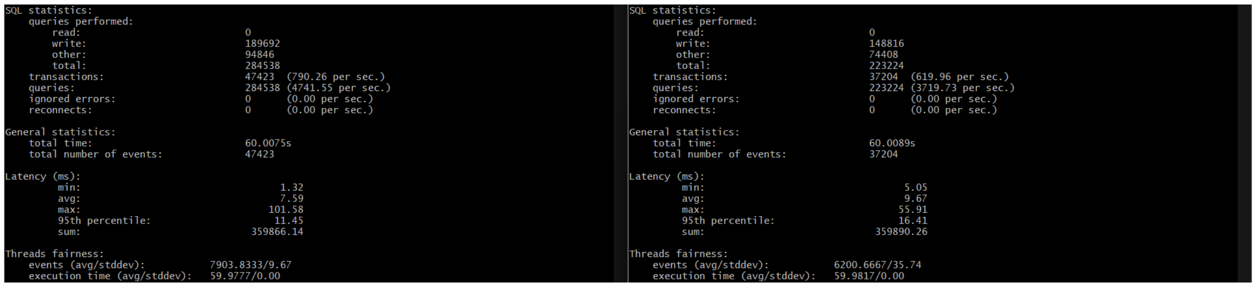
```
SQL statistics:
    queries performed:
        read:                            231994
        write:                           0
        other:                           33142
        total:                           265136
    transactions:                        16571  (276.07 per sec.)
    queries:                             265136 (4417.10 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0228s
    total number of events:              16571

Latency (ms):
         min:                                   14.34
         avg:                                   21.72
         max:                                   56.24
         95th percentile:                       28.67
         sum:                               359985.14

Threads fairness:
    events (avg/stddev):           2761.8333/3.02
    execution time (avg/stddev):   59.9975/0.01
```

Figure 2: Write-only console results



```
SQL statistics:
    queries performed:
        read:                            0
        write:                           189692
        other:                           94846
        total:                           284538
    transactions:                        47423  (790.26 per sec.)
    queries:                             284538 (4741.55 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0075s
    total number of events:              47423

Latency (ms):
         min:                                    1.32
         avg:                                    7.59
         max:                                  101.58
         95th percentile:                       11.45
         sum:                               359866.14

Threads fairness:
    events (avg/stddev):           7903.8333/9.67
    execution time (avg/stddev):   59.9777/0.00
```

```
SQL statistics:
    queries performed:
        read:                            0
        write:                           148816
        other:                           74408
        total:                           223224
    transactions:                        37204  (619.96 per sec.)
    queries:                             223224 (3719.73 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0089s
    total number of events:              37204

Latency (ms):
         min:                                    5.05
         avg:                                    9.67
         max:                                   55.91
         95th percentile:                       16.41
         sum:                               359890.26

Threads fairness:
    events (avg/stddev):           6200.6667/35.74
    execution time (avg/stddev):   59.9817/0.00
```

Figure 3: Write-only console results

From those outputs, the important bit is the number of transactions done in each test. Since both tests had the same amount of time to run and that they had similar configurations, the number of transactions is directly correlated to which system is faster and therefore displays the best performance. In all 3 tests, we can see that the standalone server is ahead of the cluster server by a significant number of transactions. In the write-only test, the cluster processed roughly 78% as many transactions as the stand-alone (37204/47423). This number gets even bigger for the read-only test at 39% (16571/42610) and the read-write test at 42% (10273/24250). It is clear that the cluster is much slower than the stand-alone. There could be multiple reasons as to why the cluster completed fewer transactions than the stand-alone. Some possible reasons include:

- Network latency: The cluster nodes might be located on different servers or in different regions, which could increase network latency and slow down query processing.

- Replication lag: If the cluster is set up with replication, there might be a lag between the updates made on the primary node and their propagation to the secondary nodes. This could slow down queries that need to access data on the secondary nodes.
- Overhead of cluster management: The overhead of managing a cluster, such as maintaining inter-node communication and coordinating transactions, could also contribute to slower performance.

2 Implementation of The Proxy pattern

The proxy pattern was implemented using python. It uses the following imports: random, pymysql, sshtunnel, and pythonping. To implement the proxy pattern, I'm using an SSH tunnel to communicate with the subordinate nodes and choose which one to send the messages to. On top of that, I use pymysql to connect to the cluster and send the queries to the cluster. With those 2 elements, I implemented the 3 proxy implementations:

- The direct implementation does not use the SSH tunnel. It sends the queries to the MySQL server on the master after connecting with mypysql.
- The randomized implementation uses the SSH tunnel between the master and a subordinate node chosen at random from the 3 using the random module. It then sends the queries to the MySQL server on the master after connecting with mypysql. The request sent to the master will be forwarded to the subordinate picked by the SSH tunnel.
- The customized implementation uses the SSH tunnel between the master and a subordinate node chosen by pinging all the subordinates and picking the one with the lowest ping, using pythonping. It then sends the queries to the MySQL server on the master after connecting with mypysql. The request sent to the master will be forwarded to the subordinate picked by the SSH tunnel.

3 Describe clearly how your implementation works

The MySQL cluster is built using a Terraform script. This script deploys:

- 1 security group called public that has all ports open to the internet.
- 1 security group called private that has selected ports open to the private subnet.
- 1 master instance on the private security group that runs the management server and the MySQL-cluster server.
- 3 subordinate instances on the private security group that runs the data nodes.
- 1 proxy instance on the public security group that runs the proxy script.
- 1 stand-alone instance on the public security group that runs the MySQL server.

The management server on the master controls data replication on the data nodes on the subordinates. The MySQL cluster server on the master acts as an entry point for queries into the cluster database. The proxy sends queries to the MySQL cluster server that in turn relays those queries to the data nodes. On initialization, the data nodes know from their config where the management server is to connect to it. The management server knows from its config where to look for the data node and the MySQL server API. The management server has the responsibility of keeping track of what is available and what is not. If one of the data nodes stops responding, the management node will stop sending requests to it and will act as if it did not exist. Because of this, the cluster is more resilient to data node failure. This is not the case for the stand-alone server, for example, since if it goes down, the database is no longer accessible. The same applies to the master and the proxy, as they are also single points of failure.

4 Summary of results and instructions to run your code

In summary, a cluster can be slower than a single instance database. It does not mean that the single instance database is better, as there are other things that one might choose a cluster for, such as resilience. The infrastructure created in this lab is not perfect, but it gave me insight into some key elements that a database cluster should have:

- Replicated data nodes to improve resiliency and consistency.
- Multiple entry points prevent single points of failure.
- Strategies to access the data efficiently and to split the load across the cluster.
- A secure network to prevent data leaks and misuse.

To run the code, you need to install awscli first. This can be done throught this link:
https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

Once this is done, you need to install terraform:
https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli

The next step is to setup the aws credentials of your account:
https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/setup-credentials.html

After doing all of this, the last step is to run the main.tf file. To do so, enter these commands in a command line at the same location as the main.tf file:

```
terraform init
terraform apply
```

Then enter yes in the terminal to approve the changes to be made.
Your cluster should be deployed and ready to use.

To use the proxy, head into the proxy instance and locate the proxy.py file. Run the following commands there:

```
sudo su
source base/bin/activate
python proxy.py
```

The reason we log in the sudo user is that the pythonping module requires sudo privileges to run.