Analyzing MySQL Database Performance
Hayley Zorkic & Paloma June

Section 1: Overview

The aim of this project was to measure the write performance of a MySQL database system. We tried improving the write performance by performing batched inserts (small, medium, large batch sizes), tuning system variables, and adding hardware resources (CPUs, RAM, Network). For each experiment and control, we obtained the average time of 8 runs so we could compare them to each other (Figure 3).

We started by configuring a Compute Engine instance (Virtual Machines) with the following specifications: us-central1 and us-central1-a regions, 50GB SSD, and n1-standard-8 series and machine type. After sampling 500 rows, we wrote a python script that collected the maximum character length of each column which we used to define the column lengths. To create our actual database, we wrote a Data Definition Language file (DDL file) which defines the schema to create a load_testing database with a table called Person. Finally, we increased the scale of the project by loading 1 million records using python scripts.

Section 2: Baseline run (Milestone 1)

For our baseline experiment, we found the time it took to execute each row in a 1,000,000 row csv individually, committing 5000 rows at a time into our table. We connected to our MySQL connector and used the timeit module. As shown in Figure 1, we developed a for loop that called the `execute()` function on each row in the dataset, but only committed after every 5000 rows. The average runtime over 8 runs was 16 minutes long.

This code was slow, and needed improvements to run efficiently. The code may have been slowed down by running the "try", "execute", and "finally" statements one million times in addition to limited computing resources, and/or committing patterns.

```
sql = ("INSERT INTO Person (first_name, last_name, company_name, address, city, county, state, zip, phone1, phone2, email, web) \
       VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)")


count = 0
for rec in result:

    rec = tuple(rec)
    try:
        cursor = connection.cursor()
        cursor.execute(sql, rec)
        count += 1
        if count % 5000 == 0:
            connection.commit()
            print(str(str(count) + " records inserted successfully into Person table"))
            cursor.close()

    except mysql.connector.Error as error:
        print(rec)
        print("Failed to insert record into Person table {}".format(error))

    finally:
        if connection.is_connected() & count == len(result):
            connection.close()
            print("MySQL connection is closed")
```

Figure 1: Experiment 1 Code Snippet. The basic logic goes as follows: insert every row individually, commit every 5000 executions.

Section 3: Batch runs (Milestone 2)

For our next 3 experiments we found the time it took to execute and commit data in batches of 5000, 50000, 500000 rows of data. We connected to our MySQL connector and called the timeit module. As shown in Figure 2, we created a loop that first spliced our data into rows of 5000, 50000, or 500000, converted the slice into a list of tuples. We then call the `executemany()` function and batch commit each slice (of respective batch size) until all rows were inserted.

```
df = pd.read_csv('us-1000000.csv', sep=',', header=0)
#df = pd.read_csv('us-500.csv', sep=',', header=0)
df = df.astype(str)
sql = ("INSERT INTO Person (first_name, last_name, compa
```

```
def split_dataframe(df, split_size):
    splits = list()
    num_splits = math.ceil(len(df) / split_size)
    for i in range(0, len(df), split_size):
        df_subset = df.iloc[i:i + split_size]
        splits.append(df_subset)
    return splits
```

```
split = split_dataframe(df, 5000)
```

```
%%timeit
import mysql.connector
connection = mysql.connector.connect(
  host="10.128.0.4",
  user="root",
  password="delties",
  database="load_testing",
  autocommit=False
)

print(connection)

try:
    r = 0
    for batch in split:

        result = []
        for index, row in batch.iterrows():
            result.append(tuple(row))

        cursor = connection.cursor()
        cursor.executemany(sql, result)
        connection.commit()
        r = r + cursor.rowcount
        print(r, "records inserted successfully into Person table")
        cursor.close()

except mysql.connector.Error as error:
    print("Failed to insert record into Person table {}".format(error))

finally:
    if connection.is_connected():
        connection.close()
        print("MySQL connection is closed")
```

Figure 2: Experiment 2/3/4 Code Snippet. The basic logic goes as follows: split dataframe into a list of tuples of the batch commit size, execute the batch at the same time, then commit the batch at the same time.

This produced a 18 fold improvement in time compared to the control experiment, which is a significant improvement. The average runtime for batch commits of 5,000 over 8 runs was 1 minute 17 seconds long. The average runtime for batch commits of 50,000 over 8 runs was 1 minute 13 seconds long. The average runtime for batch commits of 500,000 over 8 runs was 1 minute 15 seconds long. While batching records created a significant reduction in run time, altering the batch size itself did not provide a substantial difference in the runtime. From these experiments, we conclude that there is a limit to how much we can reduce runtime by optimizing the software alone.

Section 4: Hardware upgrade runs (Milestone 2)

There are computational limits to our commands and at a certain point, we have to increase the physical hardware to improve runtimes.

For the next experiment we increased the compute power by: increasing the packet allowance from the default 16MB to 512MB; changing the machine series to C2 from N1; increasing the CPU quota to 30; and increasing the RAM to 120 GB. For the final experiment, in addition to these alterations, we configured a high-bandwidth server.

For our updated hardware experiments, we found the time it took to batch execute and commit 500000 rows of data. We connected to our MySQL connector and called the timeit module. As shown in Figure 2, we developed a for loop that collected a list of rows of data as tuples with a length 500000. We then call the `executemany()` function and batch committed each slice until all 1 million rows were inserted.

After the hardware changes, we did not observe a substantial change in runtime. The average runtime for batch commits of 500,000 with our upgraded hardware over 8 runs was 1 minute 13 seconds long. The average runtime for batch commits of 500,000 with our upgraded hardware and increased bandwidth over 8 runs was 1 minute 10 seconds long. Compared to the standard hardware experiments, the updated hardware experiments were slightly faster. The high bandwidth experiment was the fastest.

Section 5: Potential future improvements

Experiment	Modifications	Average Run Time Across 8 Runs
1	5,000 grouped commit inserts, base hardware	16 minutes
2	5,000 batched commit inserts, base hardware	1 minute 17 seconds
3	50,000 batched commit inserts, base hardware	1 minute 13 seconds
4	500,000 batched commit inserts, base hardware	1 minute 15 seconds
5	500,000 batched commit inserts, c2-standard-30 machine	1 minute 13 seconds
6	500,000 batched commit inserts, high-bandwidth server	1 minute 10 seconds

Figure 3: Table with Experiment Times

The results of our experiment can be summed up in Figure 3. After performing 6 experiments, we concluded that implementing batch committing produces significant improvements in run time and increasing the bandwidth of the server improved runtime mildly.

To expand on this study, we would increase the combination of changes made to both the hardware and code in order to attempt to speed the run time up. We could run experiments using no batching, batching at 5000, batching at 50000, and batching at 500000 with the base hardware, improved hardware, and increased bandwidth- 12 experiments total. This could help us identify deeper hardware/software improvement trends and better quantify how much improving the hardware actually improves runtime.

In addition to gaining a better understanding of how hardware and software affect runtime, we believe that running the same 12 experiments on a database with multiple tables. For example, we could create separate tables with cities, relationships, housing costs, etc. that relate to each other through foreign keys and primary keys. This experiment could help us observe if various software or hardware changes improved runtime in more complex database architectures.