CS 487/587 Database Implementation

Spring 2019

Database Benchmarking Project - Part II

(Punam Rani Pal, Weiwei Chen)

Github link: https://github.com/waiwaixiaochen/CS587-Database-Benchmarking

1. **Which option / system (s) you will be working with and why you chose it (them) (20 pts)**

Systems

1) **MySQL:**

We choose it because It is a relational database system with client/server architecture which is easy to use and can be used to export and import databases from other applications. It also provides a great medium to store data and known for it's high availability feature. It has better performance on simple queries we use everyday, such as primary key lookups, range queries, etc.

2) **Google Cloud MySQL:**

We chose it because we wanted to compare the performance between a private MySQL database with a cloud MySQL database.It is the cloud version of MySQL. The database lives in the cloud and lets us take advantages of cloud deployment model.Cloud SQL is good for small application, easy to set up and convenient to use, and we don't need to worry about the disk/memory failure issues. It offers high performance and scalability.

3) **Google BigQuery:**

It is a columnar database which works well with structured and unstructured data. We chose it in order to compare the performance based on some general bench marked queries and then compare the features accordingly. BigQuery has several important features, It is serverless where computing resource can be spun up on-demand. It benefits users from zero server usage to full-scale usage without involving administrators and managing infrastructure. According to Google, BigQuery can scan Terabytes of data in seconds and Petabytes of data in minutes. For data ingestion, BigQuery allows you to

load data from Google Cloud Storage, or Google Cloud DataStore, or stream into BigQuery storage. We wanted to compare the performance of

2. System Research (30 pts)

(We chose) Option 1: Include research on your selected system - Describe types of indices, types of join algorithms, buffer pool size/structure, mechanisms for measuring query execution time

MySQL(for both local and cloud)

Types of Indices: Indices in MySQL are physical objects that are used to enforce uniqueness in a table. This will ensure there are no duplicate values in a table. Indices help in speeding up query processing and therefore the database performance**.**

Following are the types of indices:

- Clustered index
- Non-clustered index
- Primary Key
- Unique Key
- Normal Index
- Full Text Index

MySQL Joins:

- Inner join
- Outer join
- Left join
- Right join
- Nested loop join
- Block nested loop join

Buffer pool size/structure:

MySQL allocates buffers and caches to improve performance of database operations. We can improve MySQL performance by increasing the values of certain cache and buffer-related system variables and can also modify these variables to run MySQL on systems with limited memory.

- The InnoDB buffer pool is a memory area that holds cached InnoDB data for tables, indexes, and other auxiliary buffers. For efficiency of high-volume read operations, the buffer pool is divided into pages that can potentially hold multiple rows. For efficiency of cache management, the buffer pool is implemented as a linked list of pages; data that is rarely used is aged out of the cache, using a variation of the LRU algorithm.
- it is recommended that innodb_buffer_pool_size is configured to 50 to 75 percent of system memory.
- Buffer pool size must always be equal to or a multiple of innodb_buffer_pool_chunk_size (by default it is 128m)*innodb_buffer_pool_instances. If you configure innodb_buffer_pool_size to a value that is not equal to or a multiple of innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances, buffer pool size is automatically adjusted to a value that is equal to or a multiple of innodb_buffer_pool_chunk_size *innodb_buffer_pool_instances.

Mechanism for measuring query execution time:

We will use **Performance Schema** to measure the query execution time. Performance Schema can display event timer information in picoseconds to normalize timing data to a standard unit. We will give an example on the TENKTUP1 to show how to use this mechanism. The steps are as following:

- Make sure the statement and stage instrumentation is enabled by updating setup_instruments table.

```
mysql> UPDATE performance_schema.setup_instruments
    -> SET ENABLED = 'YES', TIMED = 'YES'
    -> WHERE NAME LIKE '%statement/%';
Query OK, 0 rows affected (0.06 sec)
Rows matched: 212  Changed: 0  Warnings: 0

mysql> UPDATE performance_schema.setup_instruments
    -> SET ENABLED = 'YES', TIMED = 'YES'
    -> WHERE NAME LIKE '%stage/%';
Query OK, 124 rows affected (0.00 sec)
Rows matched: 140  Changed: 124  Warnings: 0
```

- Make sure that events_statements_* and events_stages_* consumers are enabled.

```
mysql> UPDATE performance_schema.setup_consumers
    -> SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%events_statements_%';
Query OK, 1 row affected (0.00 sec)
Rows matched: 3  Changed: 1  Warnings: 0
```

```
mysql> UPDATE performance_schema.setup_consumers
    -> SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%events_stages_%';
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0
```

- Run the statement that we want to profile.

```
mysql> use mydatabase
Database changed
mysql> select * from TENKTUP1 where unique1 = 5;
+---------+---------+-----+------+-----+--------+------------+------------+----------------+---------------+---------+----------------+---------------+------------------
------------------------------------+------------------------------------------------------+------------------------------------------------------+
| unique1 | unique2 | two | four | ten | twenty | onePercent | tenPercent | twentyPercent | fiftyPercent | unique3 | evenOnePercent | oddOnePercent | stringu1
                                    | stringu2                                              | stringu4                                              |
+---------+---------+-----+------+-----+--------+------------+------------+----------------+---------------+---------+----------------+---------------+------------------
------------------------------------+------------------------------------------------------+------------------------------------------------------+
|       5 |    7655 |   0 |    2 |   9 |     10 |         35 |          5 |              4 |             1 |    7731 |            108 |           151 | HRLAAAAxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx | QBJAAAAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx | VVVVxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx |
+---------+---------+-----+------+-----+--------+------------+------------+----------------+---------------+---------+----------------+---------------+------------------
------------------------------------+------------------------------------------------------+------------------------------------------------------+
1 row in set (0.06 sec)
```

- Identify EVENT_ID of the statement by the table events_statements_history_long.

```
mysql> select EVENT_ID, TRUNCATE(TIMER_WAIT/1000000000000,6) as Duration, SQL_TEXT
    -> FROM performance_schema.events_statements_history_long WHERE SQL_TEXT LIKE '%5%';
+----------+----------+-----------------------------------------+
| EVENT_ID | Duration | SQL_TEXT                                |
+----------+----------+-----------------------------------------+
|       27 | 0.062918 | select * from TENKTUP1 where unique1 = 5 |
+----------+----------+-----------------------------------------+
1 row in set (0.04 sec)
```

- Query the table events_stages_history_long to retrieve the statement's stage events. Stages are linked to statements using event nesting. Each stage event record has a NESTING_EVENT_ID column that contains the EVENT_ID of the parent statement.

```
mysql> SELECT event_name AS Stage, TRUNCATE(TIMER_WAIT/1000000000000,6) AS Duration
    -> FROM performance_schema.events_stages_history_long
    -> WHERE NESTING_EVENT_ID=27;
+------------------------------------------------+----------+
| Stage                                          | Duration |
+------------------------------------------------+----------+
| stage/sql/starting                             | 0.006180 |
| stage/sql/Executing hook on transaction begin. | 0.000000 |
| stage/sql/starting                             | 0.000009 |
| stage/sql/checking permissions                 | 0.000004 |
| stage/sql/Opening tables                       | 0.014736 |
| stage/sql/init                                 | 0.000003 |
| stage/sql/System lock                          | 0.000008 |
| stage/sql/optimizing                           | 0.000557 |
| stage/sql/statistics                           | 0.001730 |
| stage/sql/preparing                            | 0.000381 |
| stage/sql/executing                            | 0.000000 |
| stage/sql/Sending data                         | 0.039120 |
| stage/sql/end                                  | 0.000003 |
| stage/sql/query end                            | 0.000000 |
| stage/sql/waiting for handler commit           | 0.000049 |
| stage/sql/closing tables                       | 0.000013 |
| stage/sql/freeing items                        | 0.000113 |
| stage/sql/cleaning up                          | 0.000000 |
+------------------------------------------------+----------+
18 rows in set (0.01 sec)
```

Google BigQuery:

Types of indices:

- BigQuery has no indices. It focuses on scanning and increases it performance by scanning the relevant columns in relevant partition.

Types of joins:

- Inner join
- Cross join
- Full join (outer)
- Left join
- Right join

Buffer pool size/structure:

In BigQuery the allocation of memory is done by arranging different clusters into partitions where the relevant data get stored. To improve the performance it tries to keep the related data to the same cluster in order to save time from scanning clusters in the different zones. It has a good caching capacity and performs better when the exact query runs later. Total size of the resources depends on the number of partitions and the total number of clusters a partition has. Generally, we get an unbounded resource size using cloud and we can easily increase and decrease the number of resources.

Mechanism for measuring query execution time:

For BigQuery, we only use the basic output schema which demonstrates the total execution time. When we need to compare performance of BigQuery and MySQL, we compare the total execution time for both system.

Note: When we compare local MySQL and Google Cloud MySQL, we use the mechanism of performance schema. When we compare MySQL and BigQuery, we use the total execution time.

3. Performance Experiment Design (80 pts)

Create 4 performance experiments that you will run (**We actually planned 7 performance experiments here**).

1. **Compare the performance when scanning the tuple (for Option1- multiple systems)**

   Experiment specifications:

   i. This test is to check the performance when scanning the whole table in three different systems(local MySQL, Google Cloud MySQL, Google BigQuery)

   ii. Use the tables ONEKTUP, TENKTUP1

   iii. We will use queries **SELECT * FROM ONEKTUP, SELECT * FROM TENKTUP1**

   Iv. The parameter (size of the table) changes

   v. The performance order we expect is: for the table with small size, the performance of the three different system may be very close. But for the table with large size, the performance(best to worst) might be: local MySQL > Google Cloud MySQL > Google BigQuery

2. **Compare queries with/without indices (for Option 1- multiple systems and Option 2- one system)**

   Experiment specifications:

   i. This test is to compare the the performance of queries with/without index in one system and among the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery)

   ii. Use the table TENKTUP1

   iii. We will divide this part into two sections. For the first section: we compare the queries by using a table in one system, that is in the same system, check the performance between using queries **with clustered index** and the queries **without index (use Wisconsin Bench queries 4 and 2)**, and between using queries **with non-clustered index** and the queries **without index (use Wisconsin Bench queries 6 and 2)**. And we also check the performance between using queries **with clustered index** and the queries **with non-clustered index (use Wisconsin**

Bench queries 4 and 6). Then we switch to other two systems and do the same comparisons as what we do in the first system. For the second section, we compare the performance among the three systems when we are using the same query for the same table.

iv. The parameters change when we are comparing queries in the same system (Option 2), we will need to change the parameters to change if it is index based query or non-indexed query, or clustered indexed or non-clustered indexed query

v. The performance order we expect is: For the same system, queries with index > without index, clustered index > non-clustered index. For different systems, Google Cloud MySQL > Google BigQuery > local MySQL for the same query

3. **Compare different join algorithms across systems (for Option 1- multiple systems and Option 2- one system)**

   Experiment specifications:

   i. This test is to compare the performance of different join algorithms in the same system and among the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery). **Note that MySQL resolves all joins using a nested-loop join.**

   ii. Use the tables ONEKTUP, TENKTUP1 and TENKTUP2

   iii. We will use the same two or three tables to do join in one system and among three different systems (we compare the same join in different systems). Our plan is to do joins between one big table and one small table**(use queries such as SELECT * FROM ONEKTUP, TENKTUP1 WHERE ONEKTUP.unique1 = TENKTUP1.unique1)**, or both tables are big **(use Wisconsin Bench queries 15- join between two big tables)** or all the three tables **(use Wisconsin Bench queries 11- join between three tables)** in the same system, then compare the performance. Then do the same queries in other two systems to compare the performance

   iv. Parameters change when we use different sizes of tables to do the join in the same system (for Option 2)

   v. The performance order we expect is: Joins on more tables may take more execution time. Joins on the tables with similar **big** sizes may take more time. Join on one small sized table and one large sized table may perform better. With the same query, the performance in the three

different systems will also depend on the sizes of the tables and whether the predicates are indices. If the predicates are indices, MySQL should always performs better than BigQuery since BigQuery does not support indices.

4. **Compare queries with different selectivities (for both Option 1- multiple systems and Option 2- one system)**

   Experiment specifications:

   i. This test is to compare the performance of different selections in one system or among the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery)

   ii. Use the table TENKTUP1

   iii. First we will **use Wisconsin Bench queries 1 (1% selection) and 2 (10% selection), and the query such as SELECT * FROM TENKTUP1 WHERE unique2 between 792 AND 2791 (20% selection),** run those queries in each of the three systems, and check the performance of the queries in the same system, then compare the same query in different systems

   iv. The parameters change when we are comparing queries in the same system (Option 2), we will need to change the parameters by changing the selectivity

   v. The performance order we expect is: For the same system, the queries with smaller selectivity will take less time. For different systems with the same query, when the selectivity is small, the performance(best to worst) may be: Google BigQuery > Google Cloud MySQL > local MySQL, but if the selectivity is high, the performance(best to worst) may be: local MySQL > Google Cloud MySQL > Google BigQuery

5. **Compare aggregation operation on different systems (for Option 1- multiple systems)**

   Experiment specifications:

   i. This test is to compare the performance of aggregation operation among the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery)

ii. Use the table TENKTUP1

iii. We will use **Wisconsin Bench queries 21 (minimum aggregate function with 100 partitions) and 22 (sum aggregate function with 100 partitions), and the query such as SELECT tenPercent, count(*) FROM TENKTUP1 GROUP BY tenPercent,** run those queries in each of the three systems, and compare the same query in different systems

iv. No parameters changed in this test

v. The performance order we expect is: if the predicates are indices, MySQL will perform better than BigQuery since BigQuery doesn't support indices. If the predicates are non-index, then the performance will depend on the size of the table, if the size is small, MySQL will perform better than BigQuery, if the size is large, BigQuery will perform better. For the performance of local MySQL and Google Cloud MySQL, Google Cloud MySQL will be better

6. **Compare insertion, deletion and update operations in different systems (for Option 1-multiple systems)**

   Experiment specifications:

   i. This test is to compare the performance of insertion, deletion and update operations to the same table in the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery)

   ii. Use the table TENKTUP1

   iii. We will use **Wisconsin Bench queries 26 (insert one tuple) and 27 (delete one tuple), and Wisconsin Bench queries 28 (update key attribute) and 32 (update indexed non-key attribute),** run those queries in each of the three systems, and compare the same query in different systems

   iv. No parameters changed in this test

   v. The performance depends on the predicates whether they are indices or non-indices. If they are indices, the performance order will be: Google Cloud MySQL > local MySQL > Google BigQuery. If they are non-indices, the performance order will be: Google BigQuery > Google Cloud MySQL > local MySQL

7. **Compare the performance of complex joins (for Option 1- multiple systems and Option 2- one system)**

   Experiment specifications:

   i. This test is to compare the performance of complex joins to the same two or three tables in one system and among the three different systems (local MySQL, Google Cloud MySQL, Google BigQuery)

   ii. Use the tables ONEKTUP, TENKTUP1, TENKTUP2

   iii. Instead of using basic joins as we stated in Experiment 3**,** we will add selectivity to them. We will use queries such as **SELECT * FROM ONEKTUP, TENKTUP1 WHERE ONEKTUP.unique1 = TENKTUP1.unique1 AND ONEKTUP BETWEEN 0 AND 49 (join along with 5% selectivity), SELECT * FROM TENKTUP1, TENKTUP2 WHERE TEKTUP1.unique2 = TENKTUP2.unique2 AND TENKTUP2.unique2 BETWEEN 792 AND 1791 (join along with 10% selectivity), SELECT * FROM TENKTUP1, TENKTUP2 WHERE TEKTUP1.unique2 = TENKTUP2.unique2 AND TENKTUP2.unique2 BETWEEN 892 AND 2891 (join along with 20% selectivity),** run those queries in each of the three systems, and compare the same query in different systems

   iv. The parameters change when we are comparing queries in the same system (Option 2), we will need to change the parameters, that is change the selectivity

   v. The performance order we expect is: For the same system, the lower the selectivity, the better the performance. For different systems of the same query, the performance(best to worst) order will be: Google Cloud MySQL > local MySQL > Google BigQuery

4. Lessons Learned:

We have planned to explore performances of local MySQL, Cloud MySQL and Google BigQuery. Working with those systems will give us an understanding of database management in cloud as well as in a private database. Since BigQuery use columnar database storage and it would be interesting to see how it would behave differently with MySQL which uses row bases storage. Learning the different algorithms that a query optimizer would pick when running a query will help us to get familiar with how to design our experiment query and optimize the query.

For the experiments part, we tried several queries for each test, and got some ideas on the performance of three different systems. Following are the observations we have got while designing the query plans based on our study of the systems.

1) Scanning a larger table might performs better in MySQL local machine and cloud MySQL than BigQuery given the buffer pool size large enough to hold the table and then it would scan everything at once given the size of the buffer pool. In BigQuery, there may be a possibility that the parts of the table is stored in the different partitions and to scan the whole table it must go through the entire nodes in a cluster.
2) While doing some experiments we observed that when we scan table with less tuples then BigQuery performs better or almost similar to the MySQL. In some cases it also depends on the workload on the cloud which might delay the processing time.
3) Another point we have noted while comparing the performance of local MySQL and Cloud MySQL, where the latter sometimes took more time. It may due to the overhead of processing the output for display.
4) MySQL supports indexes. The queries with the predicate conditions matches with the index will always perform better. Since BigQuery does not support index it might take longer time to process based on the selectivity factor.
5) We also think that if our predicate condition is very specific, for example, when the selectivity is 1, the MySQL will always perform better.
6) Join queries with no indexes might perform well in BigQuery because it support all joins. Hash/Merge join is mostly preferred on the equi-join conditions than nested-loop joins. Since nested-loop join scanned each tuple with outer with every tuple with inner and given a large table size, it will perform worst.

We have also learned about the properties of the systems that we selected. Knowing the pros and cons of each system will help us to design queries that will better suit the features of the systems. Optimizing the query plan is important for obtaining a better query that takes the advantage of the all privileges that will help us to get the result faster and improve performance. We became familiar with how joins works and what factors drive a system to choose a better join plan for the algorithm.

MySQL and BigQuery both have their advantages and disadvantages. We will try to explore all the factors that we have considered and implement it to see the results and compare with our assumption. Performing the experiments will help us to get a deeper understanding of how a system behaves and what query algorithm it selects to perform better.