CS487 Project Part III

by David Djernaes, Grady Ku

Systems selection

- The systems we decided on were PostgreSQL and MySQL (both hosted on Google Cloud Platform)
- PostgreSQL was our baseline choice system due to our pre-existing familiarity with the system and MySQL since it is a close analogue as a relational database system. Our goal was to see if there was clear advantage in choosing one system over another.
- A unique characteristic of PostgreSQL:
  → has specialized infrastructure that can be used to implement more advanced indexes
  (e.g. GiST, SP-GiST, GIN, and BRIN).
- A unique characteristic of MySQL:
  → there is no default allocated size for buffer pool buffers (the size must be configured manually before deployment via InnoDB).

Our Approach to the benchmark

- What were our goals?
- Using the Wisconsin Benchmark as a starting point for exploration, we created four tests to measure the perceived strengths and weaknesses of each system in the following ways (with a primary focus on performance):
  - the same set of queries were run on the two relational database systems
  - both systems were run as virtual machines on the Google Cloud Platform
  - the query optimizer for each system was allowed to choose the best option for execution
  - the final query execution time was the average of five trials (after removal of the highest and lowest times of the five)
  - the queries were designed to focus on the strengths of each system and to see if there were appreciable differences in query execution time

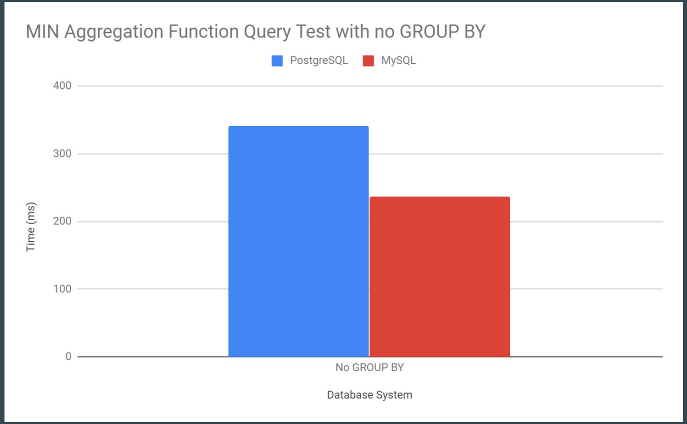
Test #1: Aggregation Function (MIN)

Description of the test: Compare the performance between PostgreSQL and MySQL database systems when queries are run with, and without, the GROUP BY clause. For the queries themselves, we used a slightly modified variant of the Wisconsin Benchmark queries #20 and #21. To keep things simple, we chose to test the query on a non-indexed attribute because the performance of an indexed query was not the main purpose of this test.

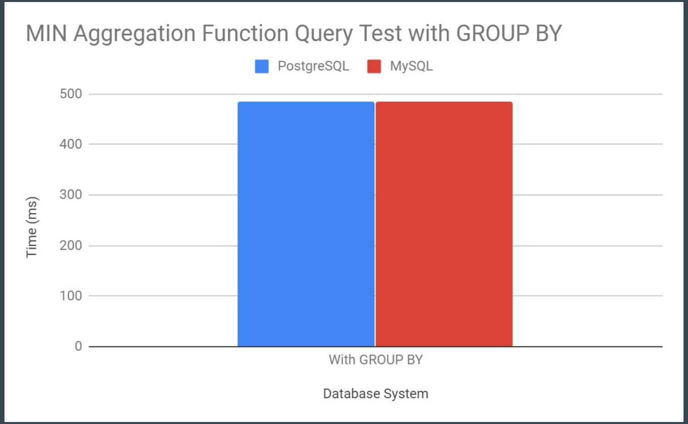
Expected results: While MySQL is known to excel with read-heavy workloads, MySQL is also known to struggle a bit when it comes to working with large relation sizes. Because we are testing the query on the million tuple table, we expected MySQL to perform slightly worse than PostgreSQL due to the relatively large table size.

Test #1: Results

A.



B.



Queries implemented:

A.
```
SELECT MIN(milliontup.unique3)
FROM milliontup;
```

B.
```
SELECT MIN(milliontup.unique3)
 FROM milliontup
 GROUP BY milliontup.onePercent;
```

Test #1: Conclusions/Insights

We originally predicted MySQL to perform slightly worse than PostgreSQL, but it appears that the opposite is true. MySQL performed noticeably better than PostgreSQL for the query without the GROUP BY clause. However, after including a GROUP BY clause, both systems performed relatively equally with MySQL edging out the victory by a margin of 0.3 milliseconds.

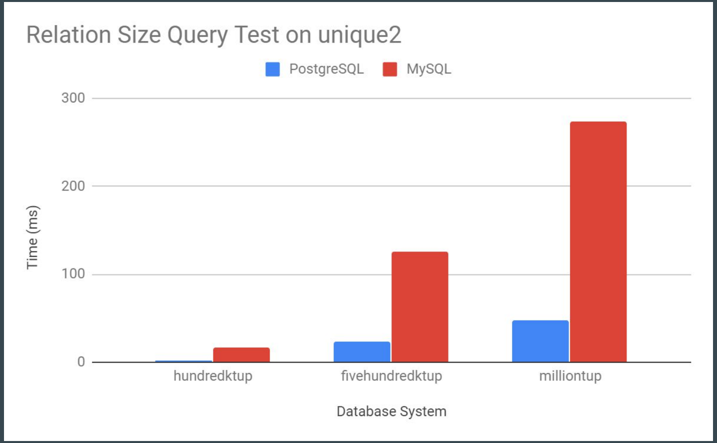
Based on these test results, it is clear that we underestimated just how much better MySQL performs when the query itself is read-only and simple. We put a little too much weight on the large table size being the downfall of MySQL. However, it is also important to note how much MySQL's performance is affected by the inclusion of a GROUP BY clause. We believe this is because the GROUP BY clause puts the query over the threshold from simple to slightly complicated for MySQL. Because PostgreSQL is known to handle complex queries better than MySQL, this reason seems fairly plausible.

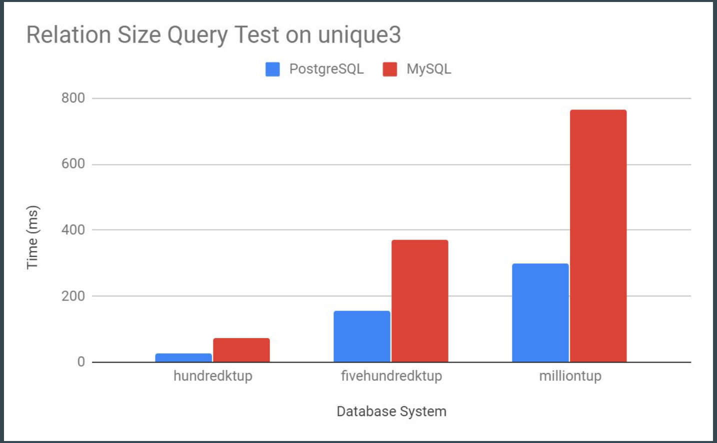
Test #2: Relation Size

Description of the test: Compare the performance between PostgreSQL and MySQL database systems when queries are executed on increasing relation sizes. For the relation sizes, we chose to test the queries on the *hundredktup*, *fivehundredktup*, and *milliontup* relations. The *hundredktup* table was selected to be used as the baseline. The *fivehundredktup* table was selected to test the impact of an increase in table size by a factor of 5. The *milliontup* table was selected to test the impact of an increase in table size by a factor of 10.

Expected results: Due to the read-only nature of the test, we expected MySQL to exhibit better times than PostgreSQL for the *hundredktup* and *fivehundredktup* relations. However, we expected PostgreSQL to eke out the win on the *milliontup* relation because of the fairly large increase in tuple count. In addition, we also expected the first set of queries to exhibit better times overall than the second set due to the first set utilizing the clustered index.

Test #2: Results





Queries implemented:

A.
```
SELECT * FROM hundredktup1
WHERE unique2 BETWEEN 45000 AND 55000;

SELECT * FROM fivehundredktup
WHERE unique2 BETWEEN 245000 AND 295000;

SELECT * FROM milliontup
WHERE unique2 BETWEEN 450000 AND 550000;
```

B.
```
SELECT * FROM hundredktup1
WHERE unique3 BETWEEN 45000 AND 55000;

SELECT * FROM fivehundredktup
WHERE unique3 BETWEEN 245000 AND 295000;

SELECT * FROM milliontup
WHERE unique3 BETWEEN 450000 AND 550000;
```

Test #2: Conclusions/Insights

The results for this test were very surprising. We originally expected MySQL to beat out PostgreSQL for the two smaller tables, but this was not the case as PostgreSQL completely ran away with this one on all table sizes & queries. I think the results can be attributed to PostgreSQL performing range queries much more efficiently than MySQL. However, we were indeed correct about the indexed query performing better than the non-indexed query as each query on the index exhibited remarkably faster performance times compared to the non-indexed query.

Based on the results of both this test and the previous test, MySQL seems to be the better choice over PostgreSQL if the queries being used are simple and return only a single tuple. Otherwise, PostgreSQL should be the default system to use.

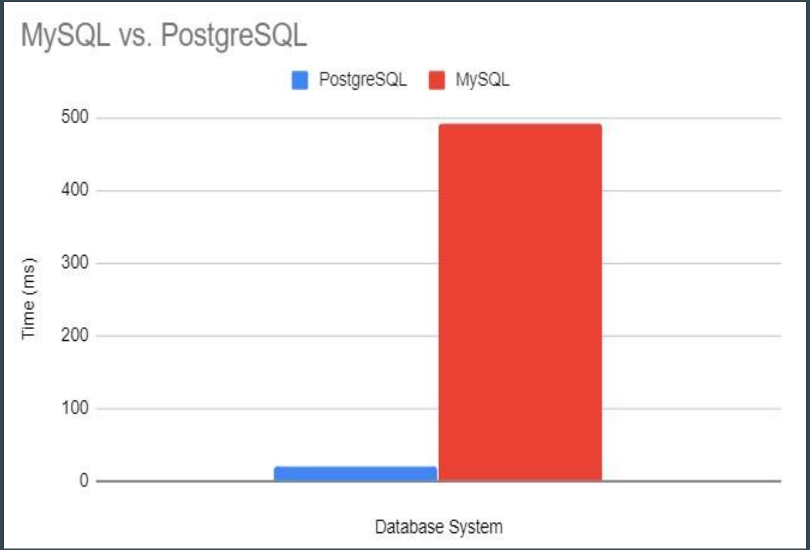
Test #3: To-disk versus To-screen

Description of the test: Compare the performance between PostgreSQL and MySQL database systems when queries output to the screen versus written to disk, via a temporary table. The same selectivity level will tested for each system. The relations tested were the onektup and tenktup sizes.

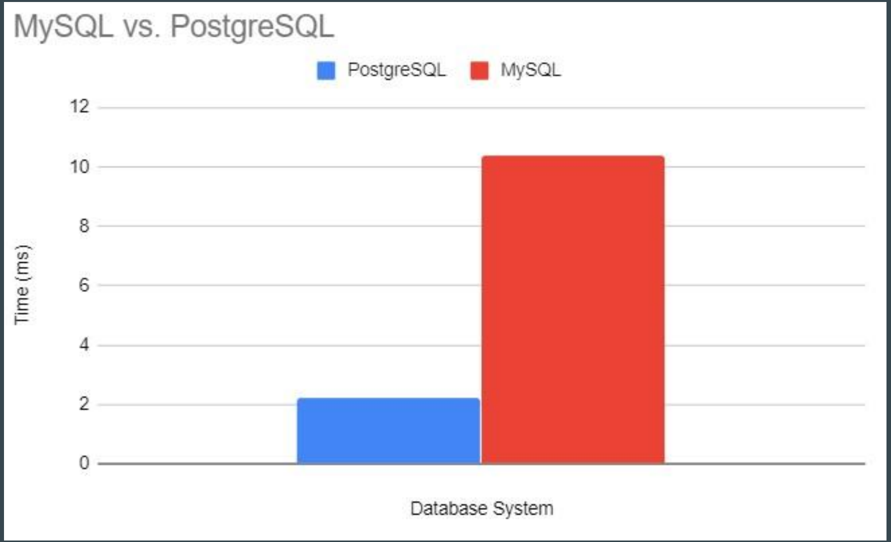
Expected results: Initially, we were going to test two levels of selectivity: a 10% output and a 20% output. This particular query seemed to have some issues with the larger table sizes and repeatedly crashed the system. Our solution was to limit the queries to the *onektup* and the *tenktup* tables and measure the performance on a 10% of the relation output. It was expected that MySQL will perform better for the 10% relation output since its general perception for being more optimized for read-heavy workloads. However, with PostgreSQL, the ability to handle concurrency better, so it should handle a larger write load with the output with greater efficiency. It will be interesting to note if there is a significant difference between the two systems. If there is not a significant difference, the output selectivity may be adjusted until differences appear.

Test #3: Results

A.



B.



Queries implemented:

A.

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2
INTO TMP FROM onektup H, tenktup T
WHERE H.tenpercent = T.tenpercent
AND T.unique2 < 100;
```

B.

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2
FROM onektup H, tenktup T
WHERE H.tenpercent = T.tenpercent
AND T.unique2 < 100;
```

Test #3: Conclusions/Insights

This test was an interesting one to manage. We had intended to query the larger datasets to compare performance between the two systems; however, when working with the *fivehundredktup* and *hundredktup* relations the query either stalled (and crashed the system) or took longer than we were willing to wait to provide a query analysis. This led to several iterations of this test and revising the query table size until the results were provided in a manageable amount of time.

The differences in processing speed were a significantly stark contrast with PostgreSQL being the winner by a large margin for both queries. The index on the *unique2* attribute seems to have made the most significant efficiency impact with this query, regardless of whether writing the output to disk or to the screen. While the queries tested were relatively simple, it shows that MySQL did not manage these well.

In the write-to-disk query there was such a large margin of efficiency difference we originally thought it was an error in our measurements, but we double-checked our findings and found that in writing to disk there was a 25X advantage and in writing to memory at 5X advantage with PostgreSQL over MySQL.

Test #4: Query Complexity

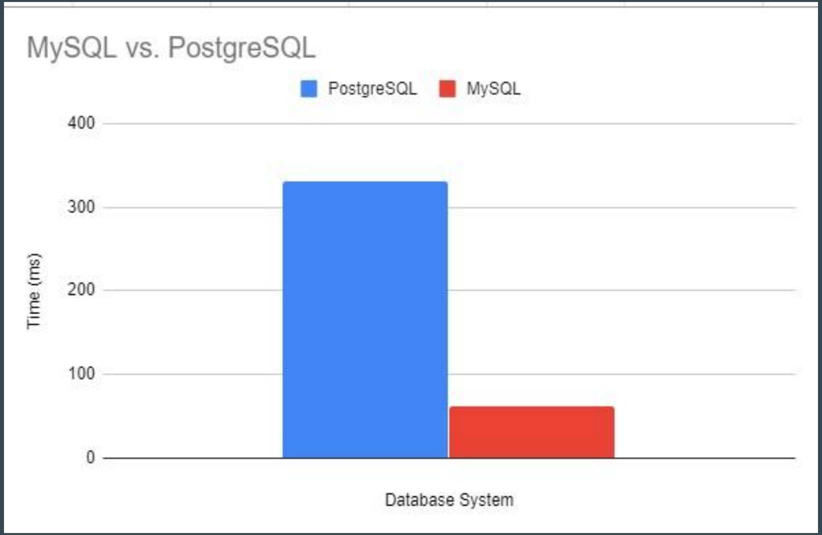
Description of the test: This test will measure the performance of the two relational database systems with how they manage queries of different complexity levels. They will first be measured with a simple query and then measured with how they manage a more complex join query. The same selectivity level will tested for each system. The relations tested were the onektup and tenktup sizes.

Expected results: It is expected that MySQL will perform better with the less complicated query and PostgreSQL will manage the more complicated query with better efficiency. These are the general performance assumptions for the two systems and this test will see if the differences are visible with this experiment.

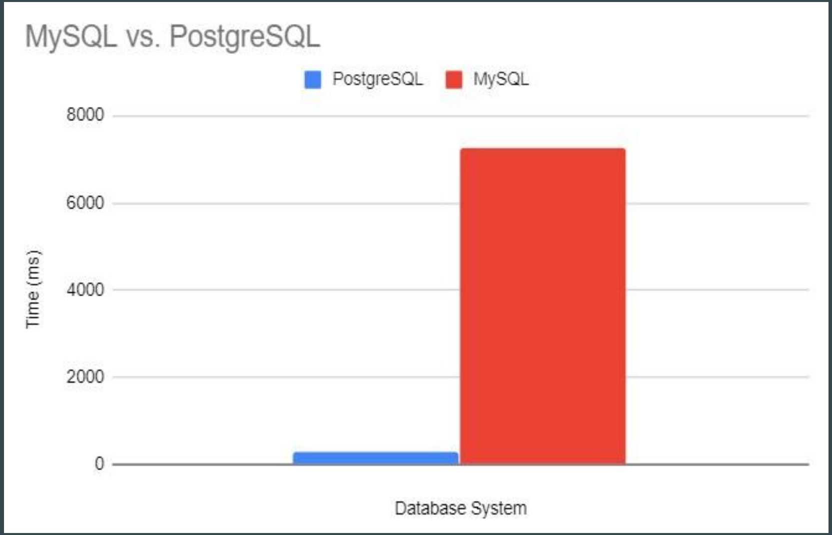
The queries are based on producing an output of 10% of the relation. The first query is a simple join on the two different *hundredktup* tables with the output finding matches on the *tenPercent* attribute. The second table relies on the same basic structure of the first query but joins *unique1* and *unique2* attributes of separate tables, as well as a third join predicate that only outputs *unique2* if it's less than 100.

Test #4: Results

A.



B.



Queries implemented:

A.
```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2
FROM hundredktup1 H, hundredktup2 T
WHERE H.tenpercent = T.tenpercent;
```

B.
```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2
FROM hundredktup1 H, hundredktup2 T
WHERE H.tenPercent = t.tenPercent
AND T.unique2 < 100  AND H.unique1 < T.unique2;
```

Test #4: Conclusions/Insights

The surprising contrast in efficiencies of the two systems could not have been more plain. For the first query that was the simple join base on 10% of the relation, MySQL outperformed by a wide margin. The query was produced on average in 62ms with MySQL, while Postgres required, on average, 329ms. Greater than 5X performance with this query. However, with the more complicated query, MySQL, did not fare as well.

The second query with the additional joins that made use of the index on *unique2* tilted the efficiency to Postgres. With MySQL, this query took approx 7,275ms, whereas Postgres completed the query in 298ms--a difference of approximately 25X.

It is of note, that our prediction of which database system would do better with which query was correct; however, we still found the wildly differing efficiencies to be surprising. With that said, ensuring the selection predicate was on the index for this query, and knowing that Postgres makes use of this efficiency, the fact that it outperformed MySQL was not as much of a shock.

Lessons learned

- In regards to benchmarking on GCP's SQL systems, we learned that there can be frequent disruptions from Google's side of the platform and as a result, frequent reboots of each system must be done. In the future, we now know to set aside some extra time for running queries on GCP's SQL systems.
- Additionally, we learned that GCP's databases execute queries at a blazingly fast speed (much faster than we had ever anticipated). Because of this, we realized that for future benchmark projects involving GCP SQL systems, the queries that are tested should be more complex and be performed on larger tables.
- A couple of our original queries failed to complete execution in a reasonable amount of time (i.e. we did not know when the query would finish). Because of this, we had to modify some of our original queries to compensate for this issue. However, in the process, we also learned that choosing to include or exclude the index from a query can determine whether a query will successfully execute or not.

Project Summary

Is one system better than the other? An internet search of whether PostgreSQL or MySQL is better yields many links to sites with great marketing copy touting the respective system’s modularity and scalability benefits, ease of management, and a steady stream of content regarding their superiority of their product. Nevertheless, it is of interest to be able to query the systems in a way that truly illustrates the relative strengths and weaknesses of each relational database system. In this project we tested both systems with a series of queries that stressed each system based on selectivity size, index based searches, aggregate outputs, and join predicates of increasing complexity.

For the most part, MySQL performed less well than PostgreSQL. While it testing was not an exhaustive study; out of the twelve tests, 9 of the 11 tests showed PostgreSQL was significantly faster in returning queries than MySQL, with one test showing a tie. The results of this test indicates that depending on the expected query types one might need to perform, it is very important to use a database management system that is optimized for the particular needs of its end users.

Appendix

The queries in this project were run in both MySQL and PostgreSQL. The data below shows the average of 5 separate runs of the query with the highest and lowest times removed. The remaining three times were then averaged and the result was used as a comparison of the query performance for each system.

The table relations used all had the same set of attributes but varied in size from one thousand rows to one million rows.
For reference, the query included at the top of each table is provided in Postgres syntax. The time displayed is in milliseconds.

QUERY SET 1

// Based on Wisconsin Benchmark Query 20

```
SELECT MIN(t.unique3) FROM milliontup t;
```

PostgreSQL	**MySQL**
340.5	**237.2**

// Based on Wisconsin Benchmark Query 21

```
SELECT MIN(M.unique3) FROM milliontup M GROUP BY M.onepercent;
```

PostgreSQL	**MySQL**
485.6	**485.3**

QUERY SET 2

SELECT * FROM hundredktup1 h WHERE h.unique2 BETWEEN 45000 AND 55000

PostgreSQL	**MySQL**
2.3	**17.2**

SELECT * FROM fivehundredktup F WHERE F.unique2 BETWEEN 245000 AND 295000

PostgreSQL	**MySQL**
23.5	**125.8**

SELECT * FROM milliontup M WHERE M.unique2 BETWEEN 450000 AND 550000;

PostgreSQL	**MySQL**
47.9	**273.8**

// Second set of queries:

SELECT * FROM HUNDREDKTUP1 WHERE unique3 BETWEEN 45000 AND 55000

PostgreSQL	**MySQL**
26.3	**73.6**

SELECT * FROM FIVEHUNDREDKTUP WHERE unique3 BETWEEN 245000 AND 295000

PostgreSQL	**MySQL**
157.1	**370.5**

SELECT * FROM MILLIONTUP WHERE unique3 BETWEEN 450000 AND 550000

PostgreSQL	**MySQL**
299.8	**764.8**

QUERY SET 3

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2 INTO TMP FROM onektup H, tenktup T
        WHERE H.tenpercent = T.tenpercent AND T.unique2 < 100;
```

PostgreSQL	MySQL
19.1	4922.4

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2 FROM onektup H, tenktup T
        WHERE H.tenpercent = T.tenpercent AND T.unique2 < 100;
```

PostgreSQL	MySQL
2.2	10.4

QUERY SET 4

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2 FROM hundredktup1 H, hundredktup2 T
        WHERE H.tenpercent = T.tenpercent AND T.unique2 < 100;
```

PostgreSQL	MySQL
323.9	62.2

```
SELECT H.unique1, H.stringu1, T.unique2, T.stringu2 FROM hundredktup1 H, hundredktup2 T
        WHERE H.tenPercent = t.tenPercent AND T.unique2 < 100 AND H.unique1< T.unique2;
```

PostgreSQL	MySQL
297.4	7274.6