Arron Wang & Brent Daniels-Soles
Term Project Part II
05/15/2019

Background

For the second round of the benchmark, we decided to compare two DBMS systems, instead of focusing on fine tuning a single DB instance. The reason we decided to go this route is:

- It allowed us to see how different DBMS's can perform from a basic install.
- See how each would perform for a basic user.
- Give an idea of the potential tradeoffs of each system given certain parameters.

This will help us both moving forward with analyzing/critiquing a particular database, should we need one in industry.

The databases we chose to analyze and benchmark against each other, were Postgres and MySQL.

System Research

The following information is taken from the office Postgres and MySQL docs.

- Postgres docs: https://www.postgresql.org/docs/9.1/
- MySQL docs: https://dev.mysql.com/doc/

Indices

Postgres and MySQL both support indices on tuples within tables. However, there is a key distinction between the two. While Postgres and MySQL have B-Tree and Hash indexes in common, they start to diverge once other problems (such as spatial data) are presented. A brief overview of indexes types are listed below:

- Postgres:
    - B-Tree
    - Hash
    - GiST
        - Generalized Search Tree
    - GIN
        - High level abstraction for implemented indexes
- MySQL

    - B-Tree
    - Hash
    - R-Tree (used for spatial indices)

Joins/Query Optimization

It is known that joins can be expensive. Between the two systems, I had a hard time finding specifics on which joins Postgres implements, however was able to find which ones MySQL does. They are: nested loop join, index join, index merge join. Where they differ is in how they approach constructing a query plan when presented with join operation.

MySQL determines which join, or in other words the “effectiveness” of join base on how the data will be joined together. For instance, MySQL notes the “best” join type is “system” join. According to their documentation, this is a constant access join. MySQL then ranks the works as the “ALL” join, which means a sequential scan of the entire table.

Postres on the other hand constructs queries based off of what indices are defined for the tables involved in the relation, and from there, conducts a near-exhaustive search of various query plans in order to derive the best one. However, the near-exhaustive search is only conducted when querying smaller amounts of data, as there can be huge, system-wide ramifications if performing a near-exhaustive search on large data sets. When joining big data together, Postgres uses what is called a “genetic algorithm” to attempt to discern the best query plan for large/many join operations, these optimizations are mainly based upon heuristics which operate off of some form of randomization.

Buffer Pool/Manager

For this section, I reference the following articles:

- Postgres buffer manager: http://www.interdb.jp/pg/pgsql08.html

The Postgres buffer pool is constructed from what are called ‘slots’. This term is interchangeable with pages, as each slot holds 8KB of data; the same amount as a page in memory. For the algorithm that manages which pages are evicted, Postgres implements the the clock algorithm, and when reading large amounts of data, will allocate a Ring buffer on the side (basically an on-demand small amount of data to store temporary data).

Digging through the MySQL docs, I was unable to land on a definite answer for the default size of pages or the default size of memory MySQL is allocated. The docs mentioned this aspect is left up to user discretion. Despite this, the docs go on to detail MySQL uses a variation of the Least Recently Used algorithm. The LRU algorithm employed by MySQL uses a head and tail pointer to memory to track older/inactive pages not really used, and newer/active pages, respectively. Everything the the ‘old’ list is candidate for eviction when a need arises.

Experiment

Queries

Below, you will see the queries we ran against the data generated in part one of the project. For this part, we compared clustered vs nonclustered index with a combination of joins based on specific values, and joins based off of ranges of values.

The table below lists the queries ran, and the corresponding results are listed below the table, with appropriate tags above each of the images. We decided to use different sizes of data, in order to get an idea of how the respective systems perform when dealing with non-trivial sizes of data.

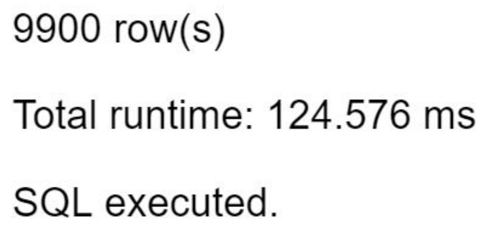
10MTUPLES = 10000 1MTUPLES=1000

Q1(clus. index 1% selection)	SELECT two,ten,stringu1 FROM tenktup1 WHERE unique2 BETWEEN 100 AND 9999;
Q2(clus. index 10% selection)	SELECT * FROM 10MTUPLES WHERE unique2 BETWEEN 792 AND 8764;
Q3(unclus. index 1% selection)	SELECT two,ten,stringu1 FROM tenktup1 WHERE unique1 BETWEEN 100 AND 9999;
Q4(unclus. index 1% selection)	SELECT * FROM 10MTUPLES WHERE unique1 BETWEEN 792 AND 8764;
Q5(clus. index)	SELECT * FROM 10MTUPLES, 1MTUPLES WHERE(10MTUPLES.unique2 = 1MTUPLES.unique2) AND (1MTUPLES.unique2 < 6689);
Q6(clus. index)	SELECT * FROM 10MTUPLES, 1MTUPLES, 1OOKTUPLES WHERE (1OOKTUPLES.unique2 = 1MTUPLES.unique2) AND (1MTUPLES.unique2 =10MTUPLES.unique2) AND (10MTUPLES.unique2 < 689)
Q7(unclus. index)	SELECT * FROM 10MTUPLES, 1MTUPLES WHERE (10MTUPLES.unique1 =

	1MTUPLES.unique1) AND (10MTUPLES.unique2 < 6689);
Q8(unclus. index)	SELECT * FROM 10MTUPLES, 1MTUPLES, 1OOKTUPLES WHERE (1OOKTUPLES.unique1 = 1MTUPLES.unique1) AND(1MTUPLES.unique1 = 10MTUPLES.unique1) AND (10MTUPLES.unique1 < 689)
Q9(clus. index)	SELECT AVG(hundred) FROM tenktup1;
Q10(clus. index)	SELECT count(four) FROM tenktup1 WHERE even100='123' GROUP BY unique2
Q11(unclus. index)	UPDATE 10MTUPLES SET unique1=5897 WHERE unique1=1491;
Q12(clus. index)	UPDATE 10MTUPLES SET unique2=5897 WHERE unique2=1491;

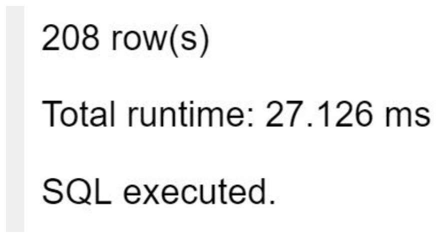
Clustered index vs. Non-Clustered Index (Postgres)

Q1:



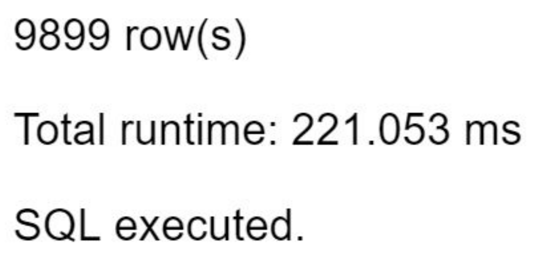
```
9900 row(s)

Total runtime: 124.576 ms

SQL executed.
```

Q2:



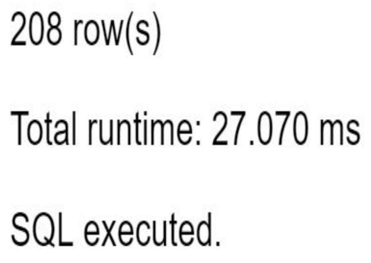
```
208 row(s)

Total runtime: 27.126 ms

SQL executed.
```

Q3:



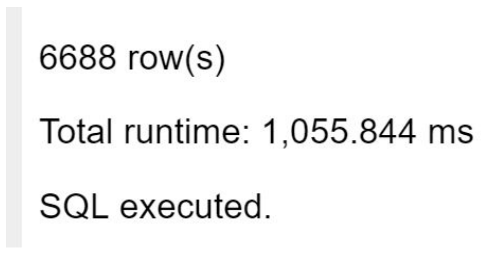
```
9899 row(s)

Total runtime: 221.053 ms

SQL executed.
```

Q4:



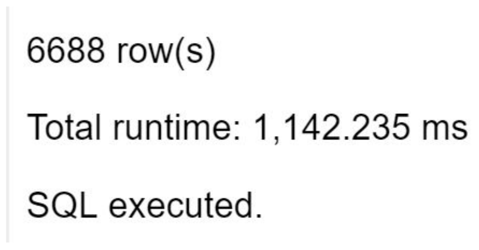
```
208 row(s)

Total runtime: 27.070 ms

SQL executed.
```

Clustered Index Join vs. Non-Clustered Index Join (Postgres)
Q5:



```
6688 row(s)

Total runtime: 1,055.844 ms

SQL executed.
```

Q7:

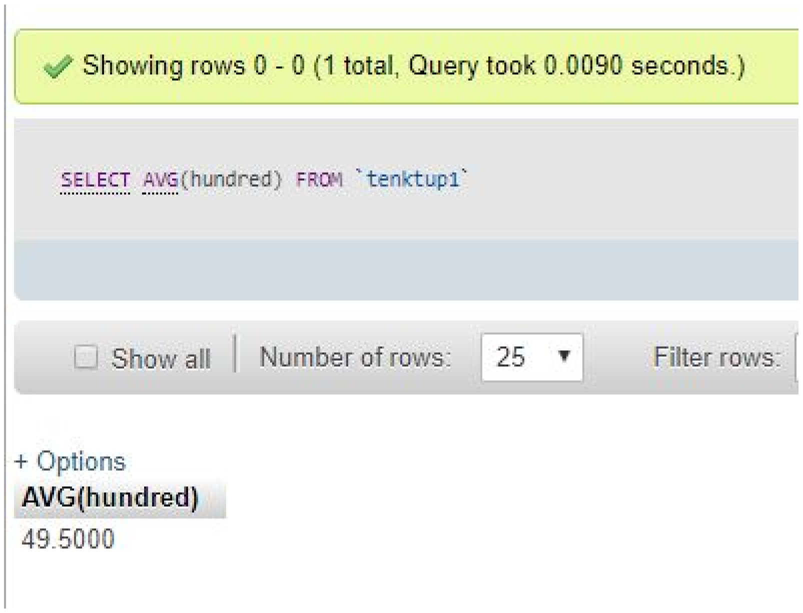


```
6688 row(s)

Total runtime: 1,142.235 ms

SQL executed.
```

Aggregate comparison: Clustered Index vs. Non-Clustered Index

Q9:

MySQL:

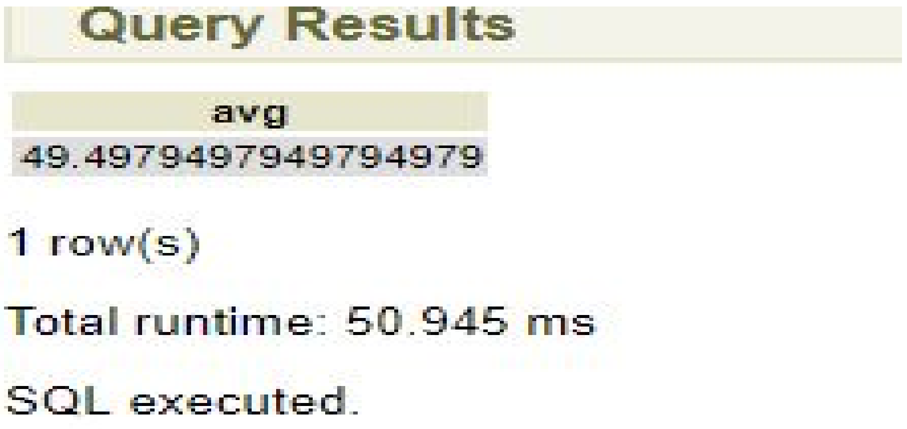


Showing rows 0 - 0 (1 total, Query took 0.0090 seconds.)

```
SELECT AVG(hundred) FROM `tenktup1`
```

AVG(hundred)
49.5000

Postgres



Query Results

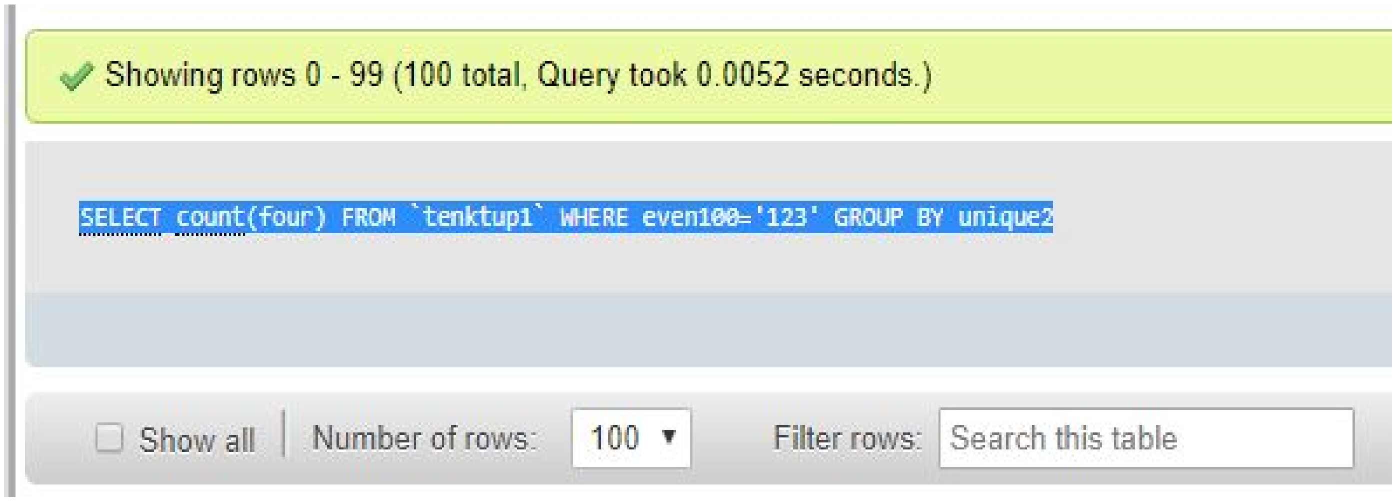
avg
49.4979497949794979

1 row(s)

Total runtime: 50.945 ms

SQL executed.

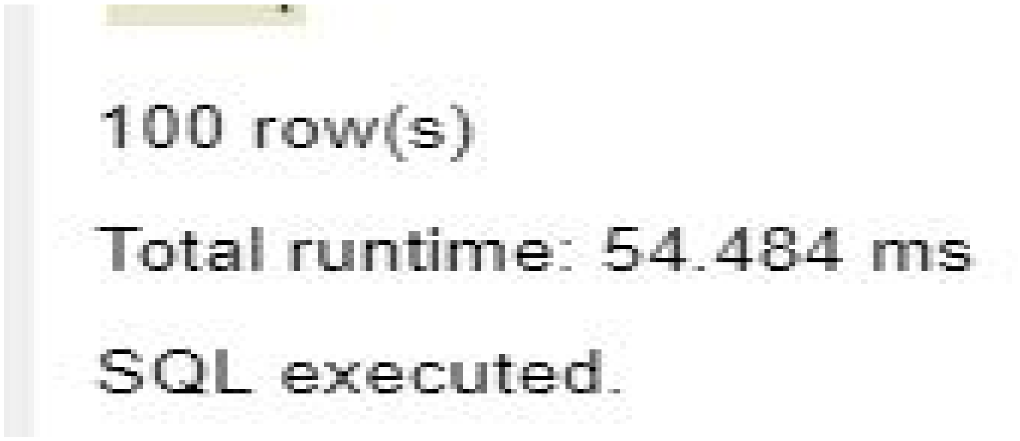
Q10:
MySQL



Showing rows 0 - 99 (100 total, Query took 0.0052 seconds.)

```
SELECT count(four) FROM `tenktup1` WHERE even100='123' GROUP BY unique2
```

Postgres



100 row(s)

Total runtime: 54.484 ms

SQL executed.

Lessons Learned

Through the process of research and experimentation with different queries, we were able to see what may seem like two very similar systems, behave differently (even if it was only marginal). The only way I can think of describing it, is like vanilla bean ice cream vs. only vanilla. They look, from the outside, more or less the same. However, as soon as you start to dig in, the taste (and in this case the way each of the respective systems are put together) are actually different.

Purely analyzing the results, taking into account we were not able to extensively test both systems against each other, it looks as though on average Postgres slightly edged out MySQL with regards to the queries and had overall a fairly consistent performance. Postgres has been know to be “battle tested”, but MySQL also has some street cred with powering some pretty popular systems. Over time, I do think these two will steadily become closer to one another in performance.

Couple things to note:

- We didn’t take into account other programs running on the machine while testing
- We were using localhost
    - In a cloud environment, would have to take into account network latencies