Aidan Reidel
Evan Hackett

CS 487/587 Database Implementation
Winter 2020
Database Benchmarking Project - Part II

1. We chose to use sqlite and postgresql as our systems to benchmark. We chose these systems simply because they are the main databases we have experience using.
2. System Research
    a. Option 1: Research on our selected systems
        i. SQLite only uses nested loop join - https://www.sqlite.org/optoverview.html

        To measure query execution time:
        SQLite: `.timer on` will turn on the timer which will display the runtime after the query executes.
        PostgreSQL: `\timing on` is the equivalent command

        SQLite supports both clustered and non-clustered indexes - https://stackoverflow.com/questions/29293799/does-sqlite-support-indexes

        All SQLite indexes are B-trees - https://sqlite.org/btreemodule.html

        PostgreSQL supports both clustered and non-clustered indexes and offers more options for index types (defaults to B-tree) - https://www.postgresql.org/docs/9.5/indexes-types.html

        SQLite only supports one writer at a time - https://sqlite.org/whentouse.html

3. Performance Experiment Design
    a. Updating rows with multiple concurrent writers
        i. This experiment seeks to expose SQLite's weakness of handling multiple writers.
        ii. We will use a single 10,000 row (~10mb) table.
        iii. Queries:
            1. `UPDATE table1 SET string4 = "hello" WHERE onepercent = X`

            The key here is to execute the writes simultaneously such that

SQLite will queue them up and PostgreSQL will execute some in parallel.

So we will run the above query multiple times in a row (using a client program) with different values of X.

iv. We expect PostgreSQL to be able to outperform SQLite by updating multiple rows at the same time. SQLite will queue the writers up, each one waiting for the previous one to finish.

b. Scaled database selection statements
   i. We will be testing how scaling up a database affects the performance of selections across both of our databases both with and without an index.
   ii. We will use a 10,000 row (~10mb), a 100,000 row (~100mb) and a 1,000,000 row (~1gb)
   iii. Queries:
      1. `SELECT * FROM tableName WHERE onepercent = 10;`
      2. `SELECT * FROM tableName WHERE two = 1;`
      3. We will establish an index on the one percent relation for this query
   iv. We expect the results to be mostly the same, except maybe PostgreSQL will perform better when the relation gets very large. We know according to the SQLite docs that SQLite doesn't perform well when the data reaches "big data" scale, but we aren't running tests with that much data. It will be interesting to see if SQLite scales poorly or not when scaling from ~10MB to ~1GB.

c. Scaled database Join Statements -> Continuation of c.
   i. What performance issue are you testing?
   ii. We will use a 10,000 row (~10mb), a 100,000 row (~100mb) and a 1,000,000 row (~1gb)
   iii. Queries:
      1. `select * from table1 as t1, table1 as t2 WHERE t1.unique1 = t2.unique1;`
   iv. We essentially expect the same as the previous test, although we think SQLite may not scale as well with joins since it only supports nested loop joins. Since PostgreSQL supports other join algorithms, we could see PostgreSQL scaling better here.

d. Compare join algorithm performance across systems
   i. This experiment tests how SQLite's lack of hash joins and merge joins could negatively affect its performance compared to PostgreSQL which does support hash joins and merge joins.
   ii. We will use a 10,000 row (~10mb) table
   iii. Queries:
      1. `select * from table1 as t1, table1 as t2 WHERE t1.unique1 = t2.unique1;`

Aidan Reidel
Evan Hackett

2. `Select * from table1 as t1 table2 as t2 WHERE (t1.unique1 < 1000) AND (t1.unique1 = t2.unique2)`
3. `Select * from table1 as t1 table2 as t2 WHERE t1.unique1 = t2.unique2`

iv. We expect PostgreSQL to perform better due to it supporting multiple join algorithms

4. Lessons Learned: Include lessons learned or issues encountered (20 pts)
    a. We had a hard time figuring out what kind of queries to use to compare these two SQL based relational databases. It took us awhile to find info on how each type of database performs in different conditions. It then took us a fair amount of research to see what kind of affects these differences would make.