SQLite vs. PostgreSQL

Aidan Reidel and Evan Hackett

Which systems and Why?

We chose to use PostgreSQL and SQLite because we have the most experience with these two database systems. We were interested in how these two systems would perform against each other considering their differing design philosophies.

- Postgres is designed in a way that it is highly configurable and customizable
- SQLite is designed to be simple and lightweight above all

Goals of our Benchmark

The goal of our benchmark design is to investigate the performance impacts that come out of the different designs and implementations of SQLite and PostgreSQl.

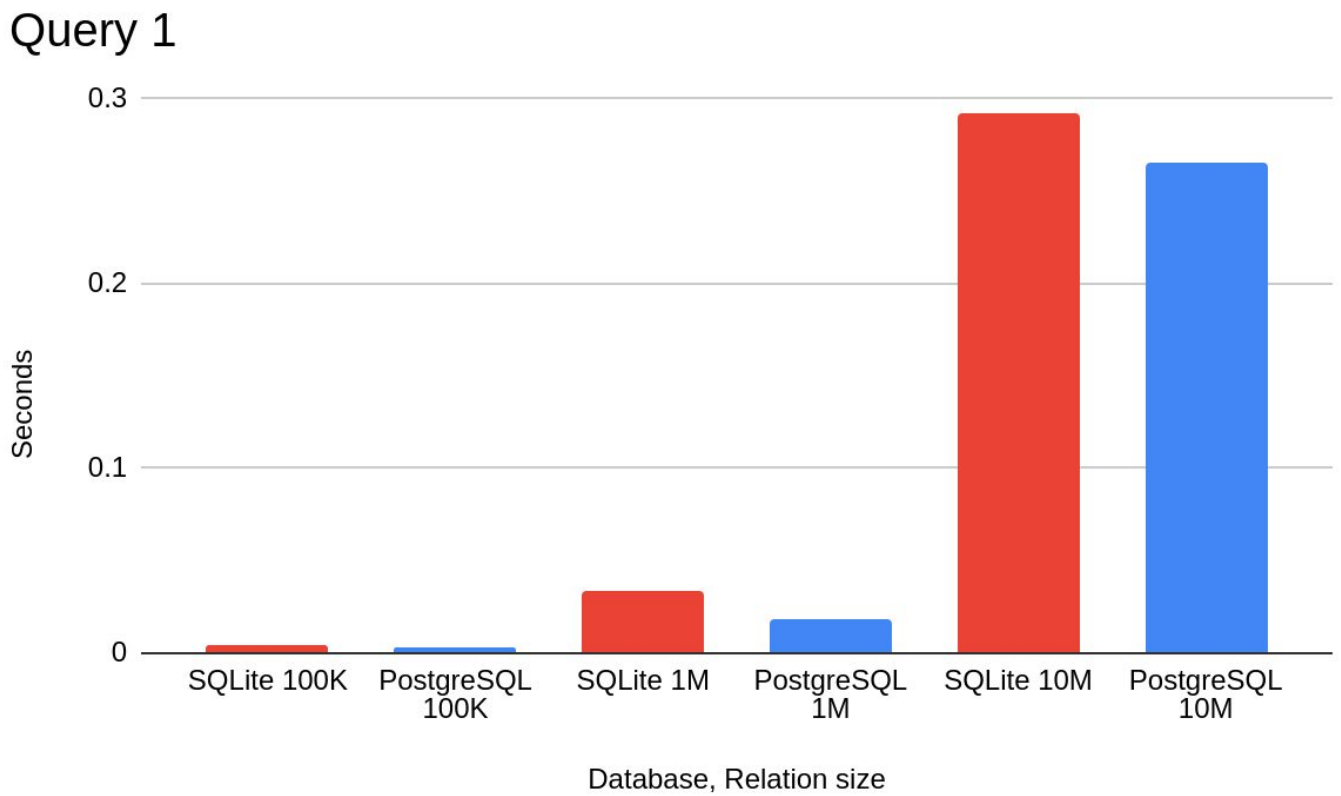
From the SQLite website: “SQLite emphasizes economy, efficiency, reliability, independence, and simplicity” [1]. Where database systems like PostgreSQL emphasize control, concurrency and scalability among other things.

[1]: https://sqlite.org/whentouse.html

Scaled Select Comparison Description

- How does scaling up a database affect the performance of selections with and without an index?
- We will use a 100,000 row ~20mb, and a 1,000,000 row ~200mb, and 10,000,000 row ~2gb, relations
- We expect the results to be mostly the same, except maybe PostgreSQL will perform better when the relation gets very large. SQLite docs state that SQLite doesn't perform well when the data reaches "big data" scale, but we aren't running tests with that much data.
- With index: `INSERT INTO TMP SELECT * FROM TABLE WHERE onePercent = 1;`
- No index: `INSERT INTO TMP SELECT * FROM TABLE WHERE two = 1;`

Scaled Select Comparison Graph



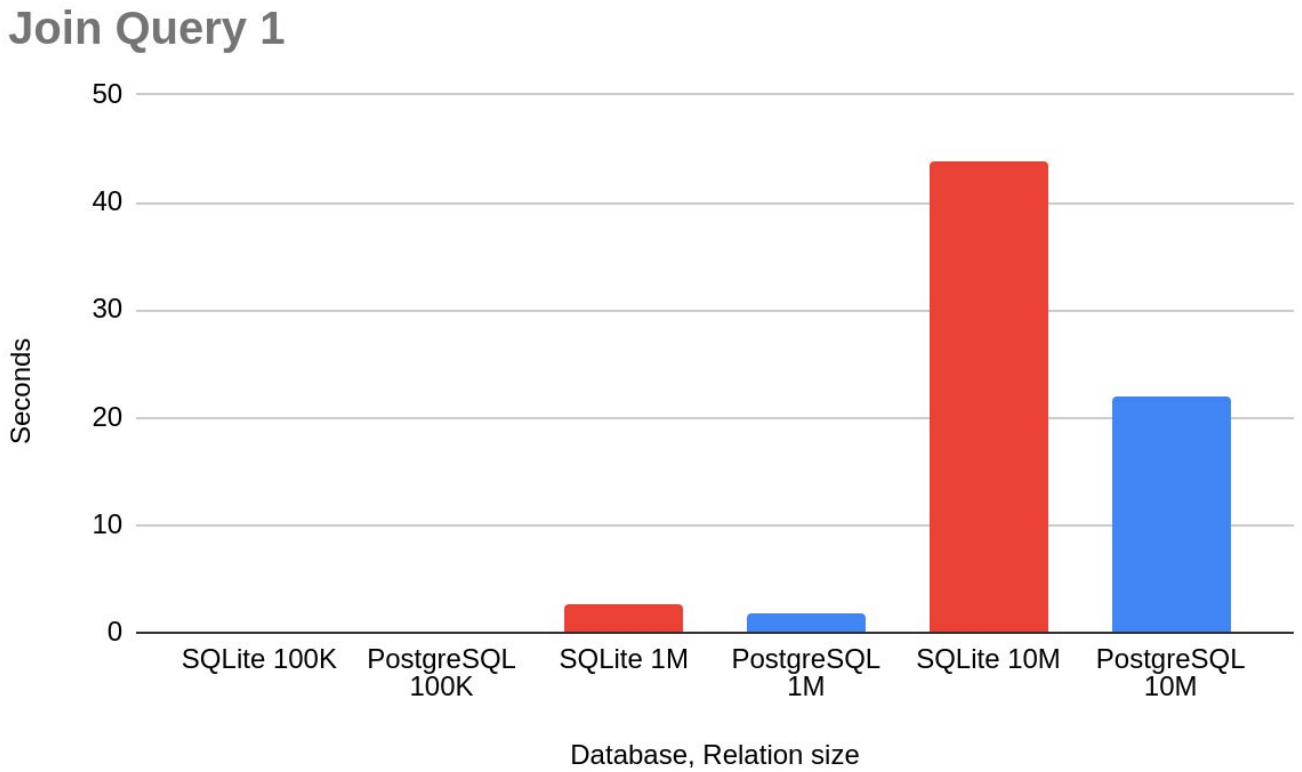
Scaled Select Comparison Results

- Results were pretty much what we expected
  - Performance was mostly the same, with PostgreSQL coming out a little faster
- PostgreSQL was faster even when the relation size was small, not just when the relation scaled up
- We didn’t see as big of a difference on the large relation as we expected, but we weren’t too surprised because we knew we weren’t at “big data” scale
- For more data see Table 1 at the end of this presentation

Scaled Join Comparison Description

- This is an extension of the scaled select where we are comparing a join instead of selects
- We expect similar results 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 see PostgreSQL scaling better.
- `INSERT INTO TMP SELECT t1.unique1 FROM TABLE t1, TABLE t2 WHERE t1.unique1 = t2.unique2;`

Scaled Join Comparison



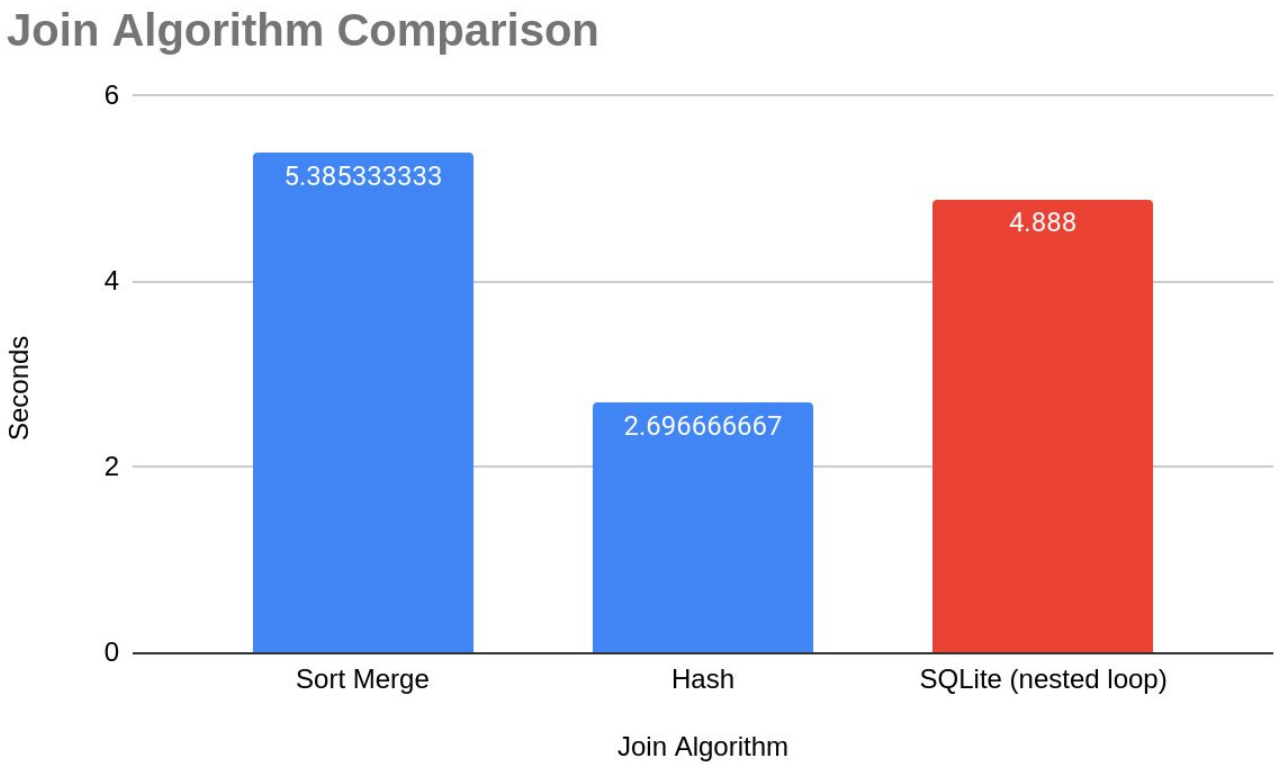
Scaled Join Comparison Results

- As expected, PostgreSQL performed much better on the larger relation size.
  - This is most likely due to SQLite using nested loop join and Postgres using Hash join for this query
- SQLite performed quite well until the relation got large, in which case Postgres ended up being twice as fast
- For more data see Table 2 at the end of this presentation

Join Algorithm Comparison Description

- We were interested in comparing the performance of SQLite's nested loop join compared to PostgreSQL's Hash, Sort Merge and nested loop joins.
- `INSERT INTO TMP SELECT t1.unique1 FROM TENMTUP t1, TENMTUP t2 WHERE (t1.tenPercent = 2) AND (t1.unique1 = t2.unique2);`
- Disable PostgreSQL hash join: `SET enable_hashjoin TO FALSE;`
- Disable PostgreSQL merge join: `SET enable_mergejoin TO FALSE;`

Join Algorithm Comparison



Join Algorithm Comparison Results

- SQLite’s nested loop join is surprisingly fast considering it is a loop join
  - Postgres’ loop join was not shown in the graph because we had to stop it because it was running too long (still running after 15 min)
  - We suspect something special is going on under the hood of SQLite. It must not be a traditional loop join
- Overall Postgres’ Hash join algorithm performed the best for the queries we tested
- SQLite’s loop join did outperform Postgres’ sort merge
  - Although our query wasn’t necessarily a good query for flexing sort merge, so this should be taken with a grain of salt
- For more data see Table 3 at the end of this presentation

Concurrent Writers Comparison

- 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.
- We planned on using the unix `time` command to time how long it would take 50 (or whatever) concurrent writers to update the database
- Unfortunately, we did not quite finish this experiment because we had trouble getting our PostrgreSQL script to work
    - The script would hang after updating the rows, which messed up the `time` commands measurement
    - In the future, this could probably be resolved by recording the time from within the script instead of relying on the unix command (so it wouldn't matter if it hangs at the end)

SQLite Concurrent Writers Script

```
const sqlite3 = require('sqlite3').verbose()

const db_filename = './project.db'
const db = new sqlite3.Database(db_filename)

// number of concurrent writers
const NUM_WRITERS = 1

db.parallelize(function() {
  for (let i = 0; i < NUM_WRITERS; i++) {
    const query = `UPDATE ONEMTUP SET string4 = ? WHERE fiftypercent = 1`
    db.run(query, 'test', i)
  }
})

db.close()
```

PostgreSQL Concurrent Writers Script

```
const { Pool } = require('pg')

// number of concurrent writers
const NUM_WRITERS = 1

const pool = new Pool({
  max: NUM_WRITERS
})

let count = 0

// This still doesn't quite work
// If we can't figure this out use js time instead!

for (let i = 0; i < NUM_WRITERS; i++) {
  const query = `UPDATE ONEMTUP SET string4 = $1 WHERE fiftypercent = $2`

  pool.query(query, ['test', 1], () => {
    count++
    if (count === NUM_WRITERS) pool.end()
  })
}
```

Concurrent Writers Results

- Since we didn’t finish this, we don’t have results to report
- We are pretty sure though that PostgreSQL would perform better as more concurrent writers attempt to update the database
- Overall it was good experience getting some practice writing scripts that interact with a database driver library
    - We learned a lot about “pooling connections” and other aspects not talked about in our previous database course

Conclusion

- Overall PostgreSQL was more performant than SQLite in pretty much every test we ran
- We still like SQLite though, if anything we like it more now, due to it being simpler to set up, simpler to interact with, and simpler to write scripts for (in our experience)
- We imagine the vast customization options that PostgreSQL has, as well as its superior performance, would make it more suitable for projects of a certain scale
    - But anything that doesn’t reach that scale is probably better off with SQLite!

Lessons Learned

- We had a hard time figuring out what kind of queries to use to compare these two SQL based relational databases. It took us a while 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.
- One issue we encountered was when we originally imported the csv file into sqlite, all of our columns had type TEXT. We didn't know that, so our queries were returning confusing results.
- Writing a script that performs concurrent writes on a database is more difficult than we thought when we set out
- When we designed our queries we forgot to insert them into temp tables which gave us more varied results than we expected. When we realized this we got more consistent results.

Table 1: Raw Data for Selection Experiment

100K				
	SQLite 100K Q1	PostgreSQL 100K Q1	SQLite 100K Q2	PostgreSQL 100K Q2
Trial 1	0.004	0.015	0.092	0.061
Trial 2	0.003	0.003	0.065	0.05
Trial 3	0.004	0.004	0.066	0.05
Trial 4	0.004	0.003	0.068	0.047
Trial 5	0.004	0.002	0.066	0.047
Average	0.004	0.003333333333	0.06666666667	0.049
1M				
	SQLite 1M Q1	PostgreSQL 1M Q1	SQLite 1M Q2	PostgreSQL 1M Q2
Trial 1	0.763	0.044	0.84	0.5
Trial 2	0.033	0.018	0.553	0.491
Trial 3	0.034	0.018	0.553	0.513
Trial 4	0.033	0.019	0.546	0.501
Trial 5	0.033	0.017	0.624	0.504
Average	0.03333333333	0.01833333333	0.5743333333	0.5016666667
10M				
	SQLite 10M Q1	PostgreSQL 10M Q1	SQLite 10M Q2	PostgreSQL 10M Q2
Trial 1	3.707	0.293	6.944	4.145
Trial 2	0.289	0.264	5.098	4.113
Trial 3	0.295	0.266	5.046	4.083
Trial 4	0.287	0.264	5.269	4.053
Trial 5	0.292	0.265	5.121	4.048
Average	0.292	0.265	5.162666667	4.083

Table 2: Raw Data for Scaled Join Experiment

100K		
	SQLite 100K Q1	PostgreSQL 100K Q1
Trial 1	0.811	0.142
Trial 2	0.173	0.146
Trial 3	0.171	0.148
Trial 4	0.174	0.133
Trial 5	0.174	0.136
Average	0.1736666667	0.1413333333
1M		
	SQLite 1M Q1	PostgreSQL 1M Q1
Trial 1	9.965	1.866
Trial 2	2.568	1.881
Trial 3	2.548	1.825
Trial 4	2.573	1.9
Trial 5	2.575	1.871
Average	2.563666667	1.872666667
10M		
	SQLite 10M Q1	PostgreSQL 10M Q1
Trial 1	44.395	20.864
Trial 2	42.321	21.763
Trial 3	44.741	22.397
Trial 4	43.763	21.823
Trial 5	41.421	22.442
Average	43.819	21.99433333

Table 3: Raw Data for Join Algorithm Experiment

	Sort Merge	Hash	Nested Loop	SQLite (nested loop)
1	14.05	2.664	Too Long	4.864
2	5.341	2.675	Too Long	4.893
3	5.235	2.702	Too Long	4.893
4	5.364	2.713	Too Long	4.898
5	5.451	2.669	Too Long	4.878
Average	5.385333333	2.696666667	Too Long	4.888