SQLite
CS582 Project Phase 2

•••

Alex Robic and Matt Bundas

SQLite - An Overview

- SQL-type RDBMS, known for being minimalist, concise, local, yet powerful

”SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.” - sqlite.org

- Used when you are dealing with local, non-enormous, single-writer data

- Not open source, but in the public domain

https://www.sqlite.org/images/sqlite370_banner.gif

SQLite Optimization - Finding the Fastest Plan

- SQLite constructs graphs whose nodes are tables, and edges are operations such as joins, with estimated cost along them.

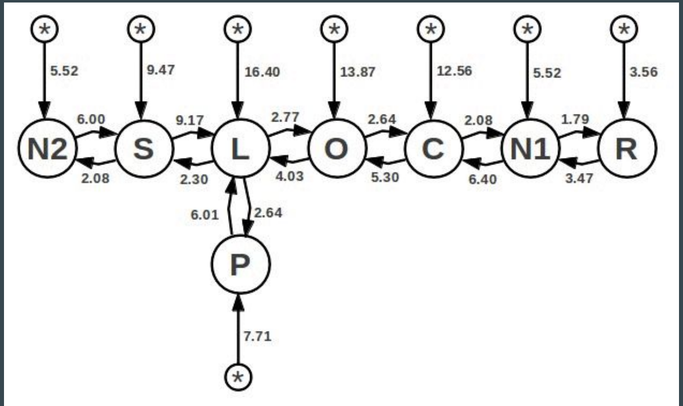
- Original SQLite (pre 2014) used a Nearest Neighbor heuristic to find best plan.

- Modern SQLite uses “N Nearest Neighbor” (N3).

- Typically uses N = 1 for simple queries, N = 5 for two-way joins and N = 10 for larger joins.

- In sample graph, 1NN finds path R-N1-N2-S-C-O-L-P with cost 36.92. N=8 finds path with cost 29.78. Actual optimal solution is path with cost 27.38. These costs are logarithmic

- For queries which make use of many OR terms in WHERE clause, SQLite may do a full-scan.



https://www.sqlite.org/queryplanner-ng.html

A sample cost graph used in determining query execution plan

SQLite Optimization - Other Ideas

- Stores “rowid”, which a sorted integer index. This is the value SQLite uses instead of primary key, unless primary key is already an integer. Allows primary key to be NULL.

- Optimizes BETWEEN statements, such as e1 between e2 and e3 into virtual tables. Where it evaluates e1 >= e2 AND e1 <= e3. Behind the scenes it allows SQLite to only load e1 once.

- Optimizes OR statements by rewriting them as a single IN statement, or as a series of joins. Helps to optimize usage of indexes.

- When more than one indice can be used, optimizer picks option with least amount of work.

- Many more clever tricks with regards to WHERE clauses, covering indices, subquery flattening, co-routines, automatic indexing, min/max optimization.

- Does seem to temporarily store tables created during queries.

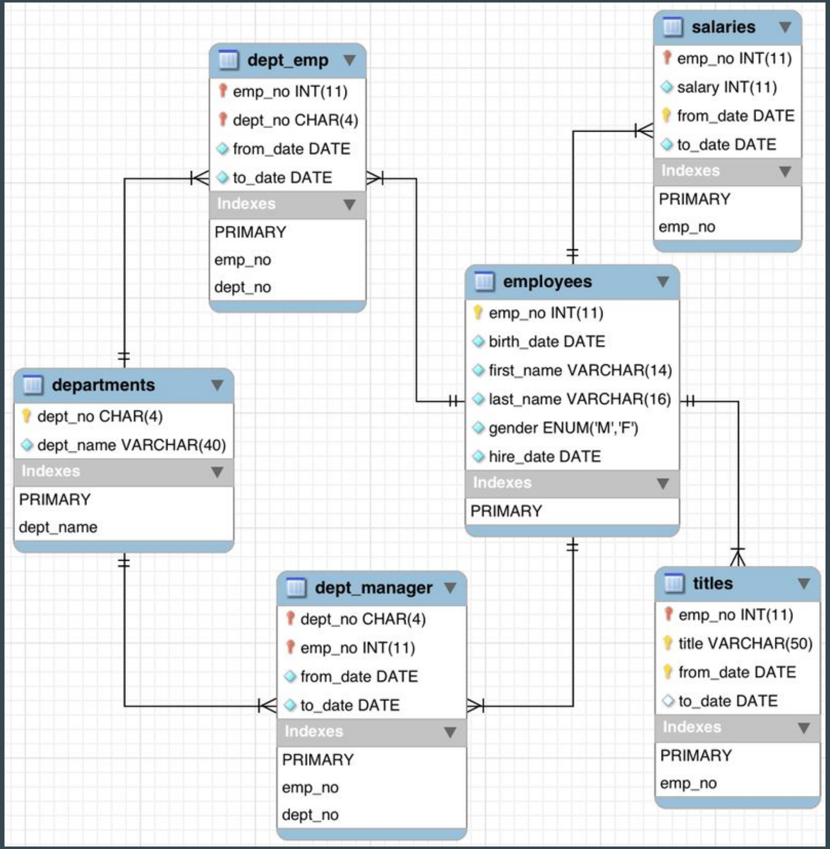
Database Used

- An artificially generated database representing a company's departments, department managers, employees and their titles an salaries.

- Provided by MySQL as a large sample database.

- Contains 6 tables, ~4 million entries, about 200 MB.

- https://github.com/datacharmer/test_db



dept_emp
emp_no INT(11)
dept_no CHAR(4)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no
dept_no

salaries
emp_no INT(11)
salary INT(11)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no

departments
dept_no CHAR(4)
dept_name VARCHAR(40)
Indexes
PRIMARY
dept_name

employees
emp_no INT(11)
birth_date DATE
first_name VARCHAR(14)
last_name VARCHAR(16)
gender ENUM('M','F')
hire_date DATE
Indexes
PRIMARY

dept_manager
dept_no CHAR(4)
emp_no INT(11)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no
dept_no

titles
emp_no INT(11)
title VARCHAR(50)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no

https://dev.mysql.com/doc/employee/en/sakila-structure.html

Methods Used

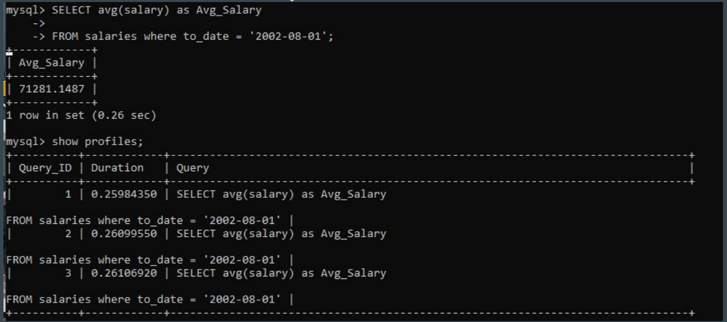
- Testing done on common remote server

- MySQL database constructed using “SOURCE employees.sql”

- SQLite database constructed using “.read employees.sql” on personal computer to create .db file, then transferred to remote server

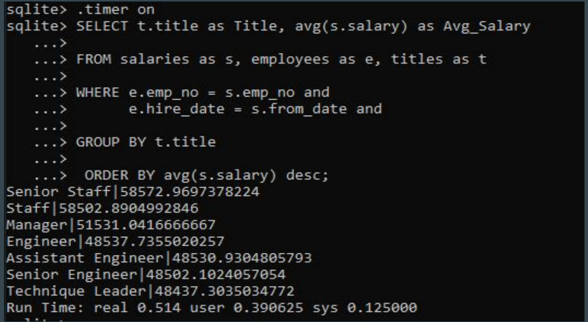
- Ran each test 8 times, calculated the average.

MySQL



```
mysql> SELECT avg(salary) as Avg_Salary
    ->
    -> FROM salaries where to_date = '2002-08-01';
+------------+
| Avg_Salary |
+------------+
| 71281.1487 |
+------------+
1 row in set (0.26 sec)

mysql> show profiles;
+----------+------------+-----------------------------------------------------------------------------+
| Query_ID | Duration   | Query                                                                       |
+----------+------------+-----------------------------------------------------------------------------+
|        1 | 0.25984350 | SELECT avg(salary) as Avg_Salary

FROM salaries where to_date = '2002-08-01' |
|        2 | 0.26099550 | SELECT avg(salary) as Avg_Salary

FROM salaries where to_date = '2002-08-01' |
|        3 | 0.26106920 | SELECT avg(salary) as Avg_Salary

FROM salaries where to_date = '2002-08-01' |
+----------+------------+-----------------------------------------------------------------------------+
```

SQLite



```
sqlite> .timer on
sqlite> SELECT t.title as Title, avg(s.salary) as Avg_Salary
   ...>
   ...> FROM salaries as s, employees as e, titles as t
   ...>
   ...> WHERE e.emp_no = s.emp_no and
   ...>       e.hire_date = s.from_date and
   ...>
   ...> GROUP BY t.title
   ...>
   ...>  ORDER BY avg(s.salary) desc;
Senior Staff|58572.9697378224
Staff|58502.8904992846
Manager|51531.0416666667
Engineer|48537.7355020257
Assistant Engineer|48530.9304805793
Senior Engineer|48502.1024057054
Technique Leader|48437.3035034772
Run Time: real 0.514 user 0.390625 sys 0.125000
```

Overview Of Queries

- Adapted the themes of the sample queries to the employees database.

- Include joins, sorts, aggregations, filtering, ranging.

- Provided indexes on attributes used in queries which aren’t there by default.

- Times range from 4s to 0.0033s.

Query 1

Average Starting Salary of Each Title

```
SELECT t.title as Title, avg(s.salary) as Avg_Salary

FROM salaries as s, employees as e, titles as t

WHERE e.emp_no = s.emp_no and
            e.hire_date = s.from_date and
            t.emp_no = e.emp_no

GROUP BY t.title
ORDER BY avg(s.salary) desc
```

No Additional Indexes

Query Execution Plan No Indexes:

1) Scan titles using automatic primary index
2) Search employees using automatic index on rowid
3) Search salaries using automatic index on emp_no and from_date
4) Do group by with B-Tree
5) Perform aggregation
6) Do order by with B-Tree

With Additional Indexes

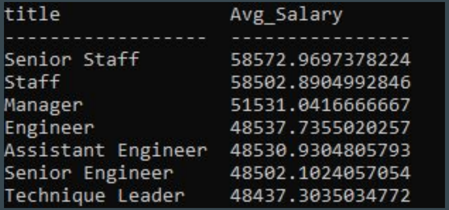
Indexes Added:

from_date on salaries
hire_date on employees

Query Execution Plan No Indexes:

1) Scan titles using automatic primary index
2) Search employees using rowid
3) Search salaries using automatic index on emp_no and from_date
4) Do group by with B-Tree
5) Perform aggregation
6) Do order by with B-Tree

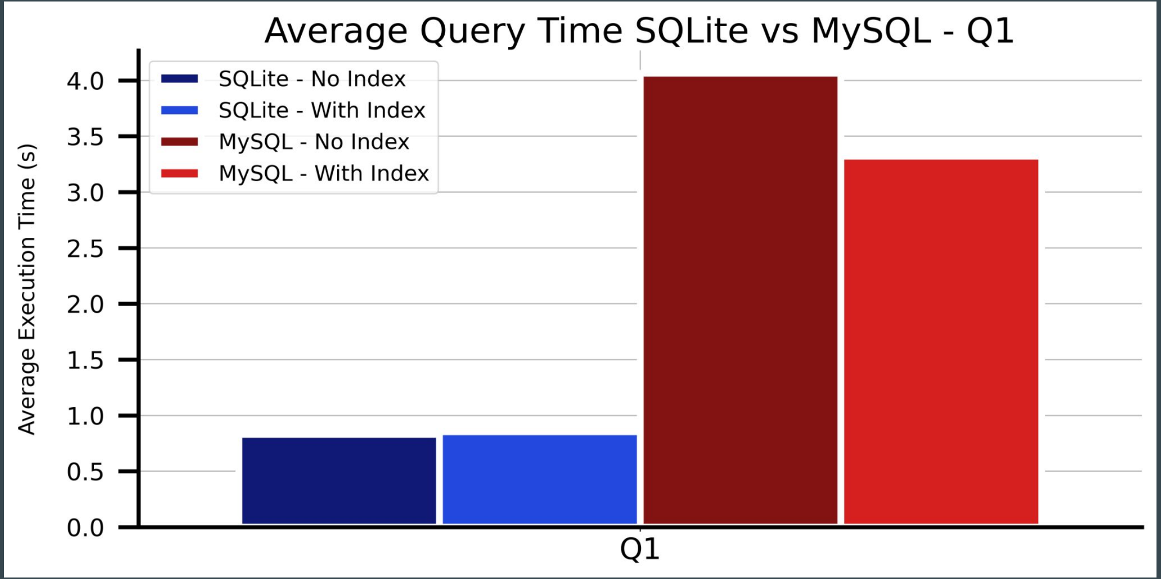
Query 1 - Results



```
title              Avg_Salary
------------------ ----------------
Senior Staff       58572.9697378224
Staff              58502.8904992846
Manager            51531.0416666667
Engineer           48537.7355020257
Assistant Engineer 48530.9304805793
Senior Engineer    48502.1024057054
Technique Leader   48437.3035034772
```

SQLite No Index - 0.83s
SQLite With Index - 0.85s

MySQL No Index - 4.06s
MySQL With Index - 3.32s



```
SELECT t.title as Title, avg(s.salary) as Avg_Salary

FROM salaries as s, employees as e, titles as t

WHERE e.emp_no = s.emp_no and e.hire_date = s.from_date and   t.emp_no = e.emp_no

GROUP BY t.title
ORDER BY avg(s.salary) desc
```

Query 2

Number of Hires Made Under Each Manager

```
SELECT dm.emp_no as Manager, count(de.emp_no) as Hires

FROM dept_manager as dm

JOIN departments as d ON dm.dept_no = d.dept_no
JOIN dept_emp as de on de.dept_no = d.dept_no

 WHERE de.from_date < dm.to_date and
 de.from_date > dm.from_date

 GROUP BY dm.emp_no

 ORDER BY count(de.emp_no) desc;
```

No Additional Indexes

Query Execution Plan No Indexes:

1. Scan dept_manager with automatic primary index
2. Search department with automatic index on dept_no
3. Search dept_emp with automatic on dept_no
4. Filter data based on condition
5. Use B-Tree for Group By
6. Perform Aggregation
7. Use B-Tree for Order By

With Additional Indexes

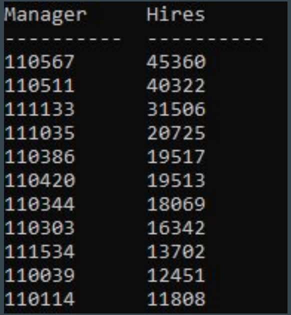
Indexes Added:

from_date and to_date on dept_manager
from_date and to_date on dept_emp

Query Execution Plan With Indexes:

1. Scan dept_manager with automatic primary index
2. Search department with automatic index on dept_no
3. Search dept_emp with automatic on dept_no
4. Filter data based on condition
5. Use B-Tree for Group By
6. Perform Aggregation
7. Use B-Tree for Order By

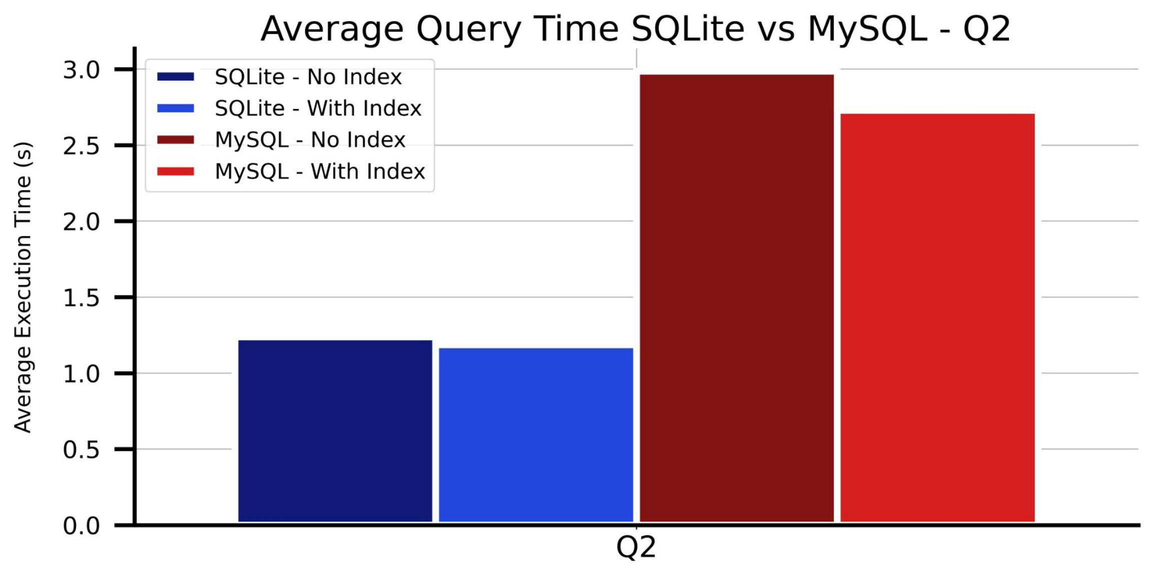
Query 2 - Results



```
Manager     Hires
----------  ----------
110567      45360
110511      40322
111133      31506
111035      20725
110386      19517
110420      19513
110344      18069
110303      16342
111534      13702
110039      12451
110114      11808
```

SQLite No Index - 1.24s
SQLite With Index - 1.19s

MySQL No Index - 2.99s
MySQL With Index - 2.73s



```
SELECT dm.emp_no as Manager, count(de.emp_no) as Hires

FROM dept_manager as dm

JOIN departments as d ON dm.dept_no = d.dept_no JOIN dept_emp as de on de.dept_no = d.dept_no

 WHERE de.from_date < dm.to_date and de.from_date > dm.from_date

 GROUP BY dm.emp_no ORDER BY count(de.emp_no) desc
```

Query 3

Average Salary of Employees Named Manu

```
SELECT avg(s.salary) as Avg_Salary

FROM employees as e, salaries as s

WHERE e.first_name = 'Manu' and e.emp_no = s.emp_no
```

No Additional Indexes

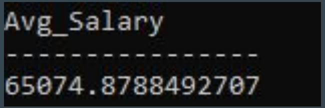
Query Execution Plan No Indexes:
1) Scan Employees for “Manu”
2) Search Salaries With Primary Index on emp_no
3) Perform Aggregation

With Additional Indexes

Indexes Added:
first_name on employees

Query Execution Plan With Indexes:
1) Search Employees with Index on first_name
2) Search Salaries With Primary Index on emp_no
3) Perform Aggregation

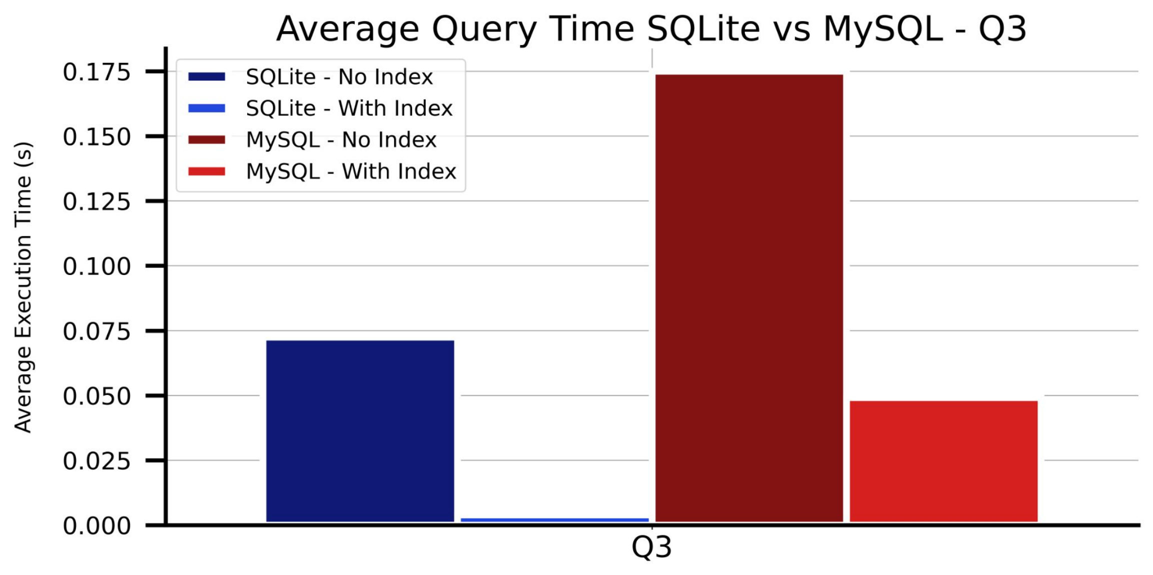
Query 3 - Results



```
Avg_Salary
----------------
65074.8788492707
```

SQLite No Index - 0.0725s
SQLite With Index - 0.0039s

MySQL No Index - 0.175s
MySQL With Index - 0.049s



```
SELECT avg(s.salary) as Avg_Salary

FROM employees as e, salaries as s

WHERE e.first_name = 'Manu' and e.emp_no = s.emp_no
```

Query 4

Information of Employees With Salary Between 100,000 and 150,000

```
SELECT DISTINCT e.emp_no, e.first_name, e.last_name

FROM employees as e

JOIN salaries as s on e.emp_no = s.emp_no

WHERE s.salary >= 100000 and s.salary <= 150000;
```

No Additional Indexes

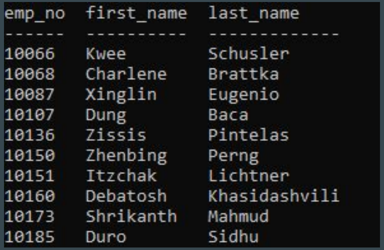
Query Execution Plan No Indexes:
1) Scan employees table
2) Search salaries that meet criteria with automatic primary index on emp_no

With Additional Indexes

Index Added:
salary on salaries

Query Execution Plan With Indexes:
1) Search salaries that meet criteria with index on salaries
2) Search employees using automatic index on rowid

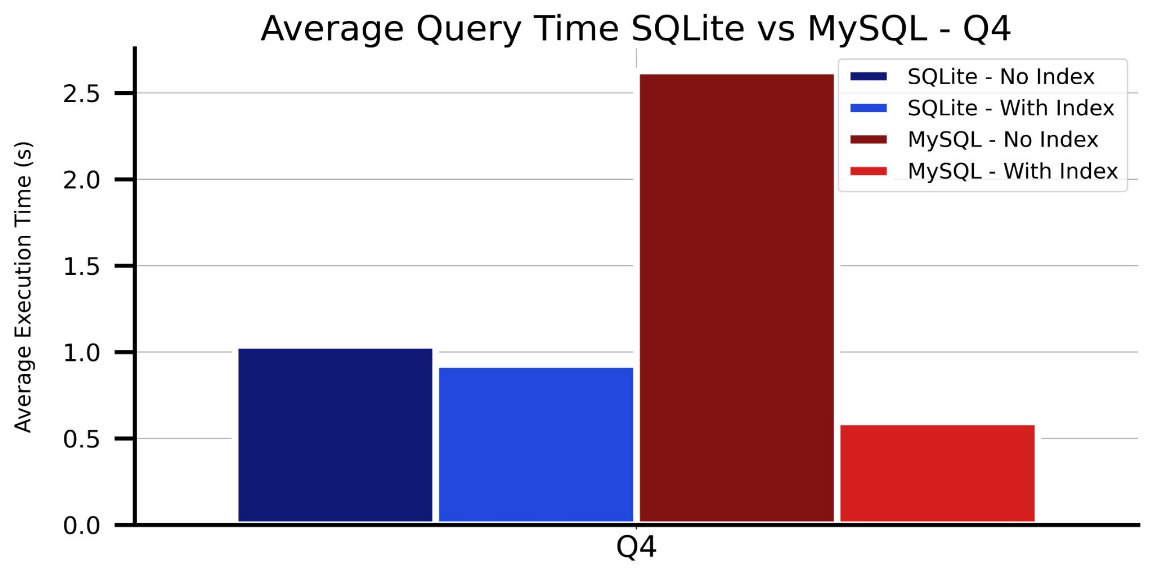
Query 4 - Results



```
emp_no  first_name  last_name
------  ----------  ------------
10066   Kwee        Schusler
10068   Charlene    Brattka
10087   Xinglin     Eugenio
10107   Dung        Baca
10136   Zissis      Pintelas
10150   Zhenbing    Perng
10151   Itzchak     Lichtner
10160   Debatosh    Khasidashvili
10173   Shrikanth   Mahmud
10185   Duro        Sidhu
```

SQLite No Index - 1.04s
SQLite With Index - 0.92s

MySQL No Index - 2.63s
MySQL With Index - 0.60s



SELECT DISTINCT e.emp_no, e.first_name, e.last_name

FROM employees as e

JOIN salaries as s on e.emp_no = s.emp_no

WHERE s.salary >= 100000 and s.salary <= 150000;

Query 5

Average Salary of Salaries Which Expire on 2002-08-01

```
SELECT avg(salary) as Avg_Salary

FROM salaries where to_date = '2002-08-01';
```

No Additional Indexes

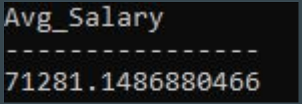
Query Execution Plan No Indexes:
1) Scan salaries table for rows that meet criteria
2) Perform aggregation

With Additional Indexes

Index Added:
to_date on salaries

Query Execution Plan With Indexes:
1) Search salaries with index on to_date
2) Perform aggregation

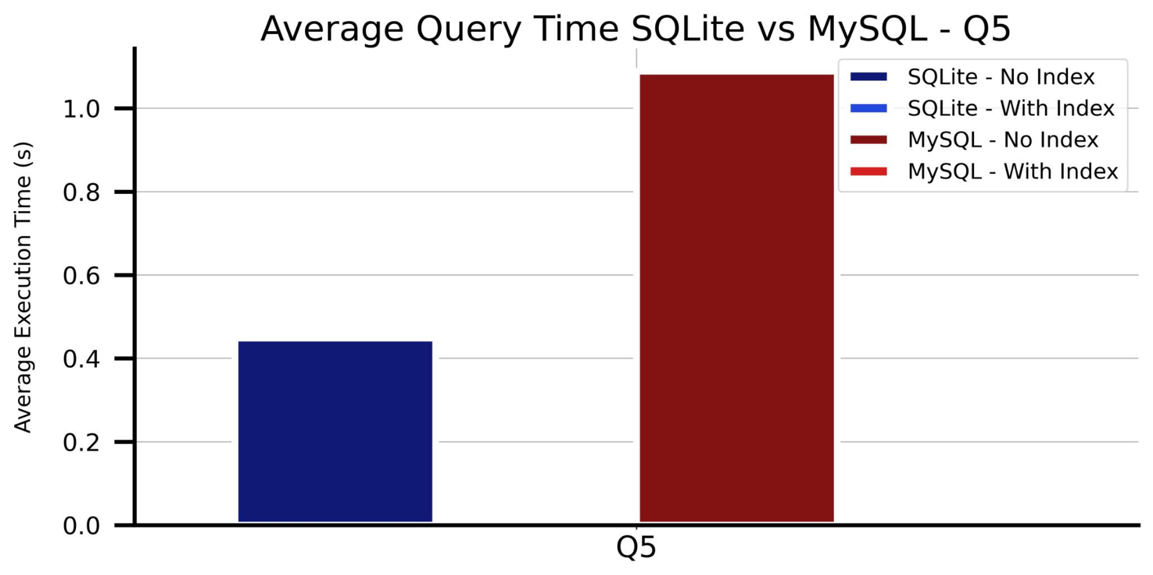
Query 5 - Results



```
Avg_Salary
----------------
71281.1486880466
```

SQLite No Index - 0.45s
SQLite With Index - 0.0033s

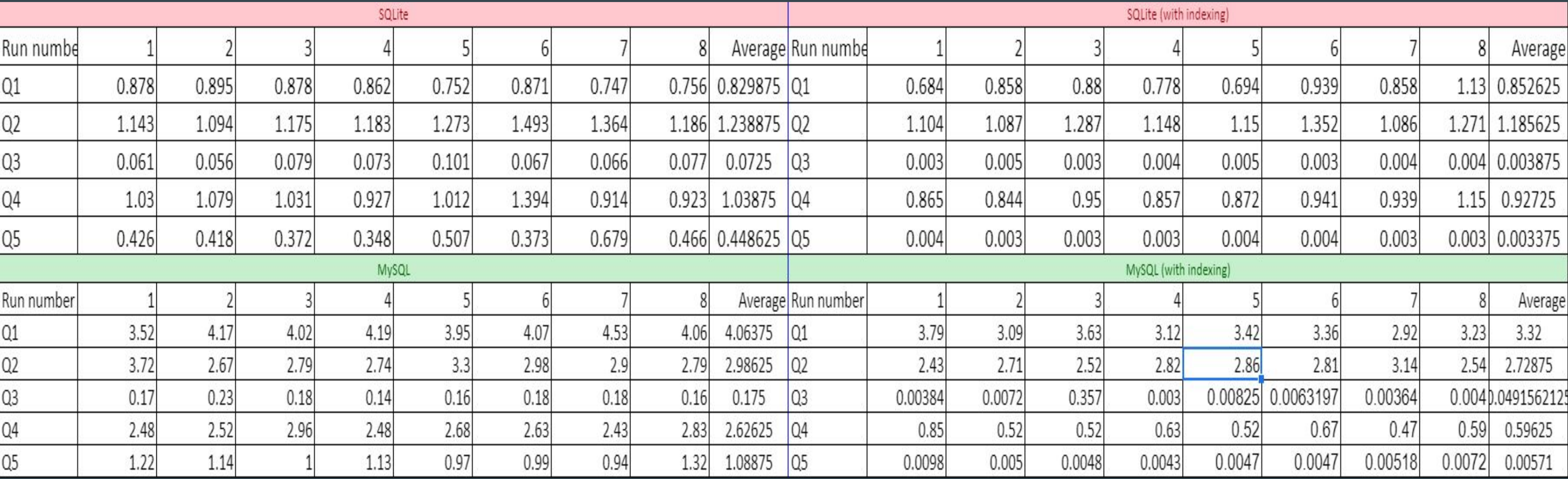
MySQL No Index - 1.09s
MySQL With Index - 0.0057s



SELECT avg(salary) as Avg_Salary

FROM salaries where to_date = '2002-08-01';

Data from all queries



SQLite

Run numbe	1	2	3	4	5	6	7	8	Average
Q1	0.878	0.895	0.878	0.862	0.752	0.871	0.747	0.756	0.829875
Q2	1.143	1.094	1.175	1.183	1.273	1.493	1.364	1.186	1.238875
Q3	0.061	0.056	0.079	0.073	0.101	0.067	0.066	0.077	0.0725
Q4	1.03	1.079	1.031	0.927	1.012	1.394	0.914	0.923	1.03875
Q5	0.426	0.418	0.372	0.348	0.507	0.373	0.679	0.466	0.448625

SQLite (with indexing)

Run numbe	1	2	3	4	5	6	7	8	Average
Q1	0.684	0.858	0.88	0.778	0.694	0.939	0.858	1.13	0.852625
Q2	1.104	1.087	1.287	1.148	1.15	1.352	1.086	1.271	1.185625
Q3	0.003	0.005	0.003	0.004	0.005	0.003	0.004	0.004	0.003875
Q4	0.865	0.844	0.95	0.857	0.872	0.941	0.939	1.15	0.92725
Q5	0.004	0.003	0.003	0.003	0.004	0.004	0.003	0.003	0.003375

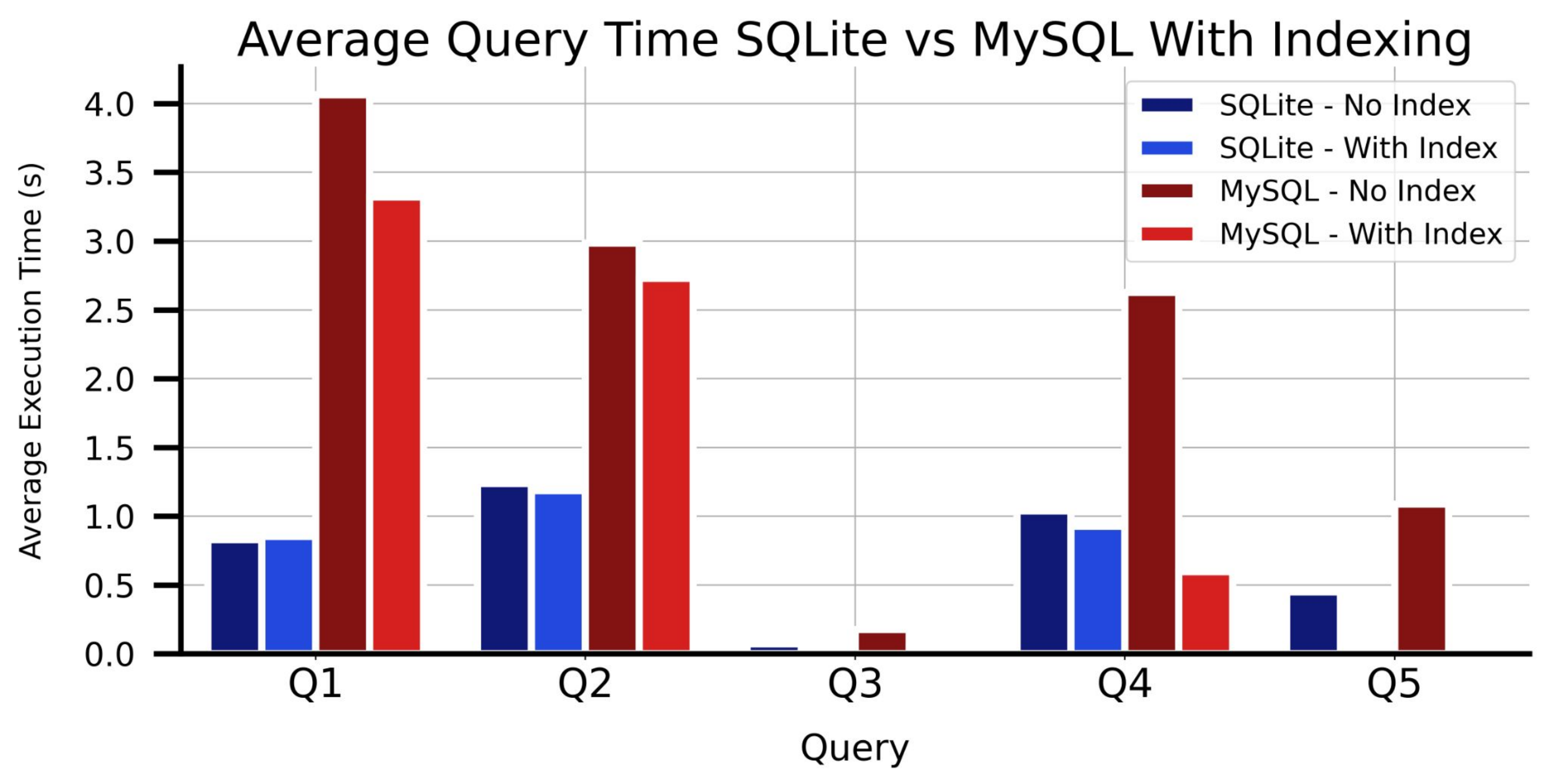
MySQL

Run number	1	2	3	4	5	6	7	8	Average
Q1	3.52	4.17	4.02	4.19	3.95	4.07	4.53	4.06	4.06375
Q2	3.72	2.67	2.79	2.74	3.3	2.98	2.9	2.79	2.98625
Q3	0.17	0.23	0.18	0.14	0.16	0.18	0.18	0.16	0.175
Q4	2.48	2.52	2.96	2.48	2.68	2.63	2.43	2.83	2.62625
Q5	1.22	1.14	1	1.13	0.97	0.99	0.94	1.32	1.08875

MySQL (with indexing)

Run number	1	2	3	4	5	6	7	8	Average
Q1	3.79	3.09	3.63	3.12	3.42	3.36	2.92	3.23	3.32
Q2	2.43	2.71	2.52	2.82	2.86	2.81	3.14	2.54	2.72875
Q3	0.00384	0.0072	0.357	0.003	0.00825	0.0063197	0.00364	0.004	0.0491562125
Q4	0.85	0.52	0.52	0.63	0.52	0.67	0.47	0.59	0.59625
Q5	0.0098	0.005	0.0048	0.0043	0.0047	0.0047	0.00518	0.0072	0.00571

Total Results



Other Speed Comparisons

Time to create database:

SQLite - 16s

MySQL - 19s

Drop From Salaries where Salary > 125,000:

SQLite No Index- 0.417s

SQLite Index on Salaries - 0.002s

MySQL - 0.65s

MySQL With Index on Salaries - 0.03s

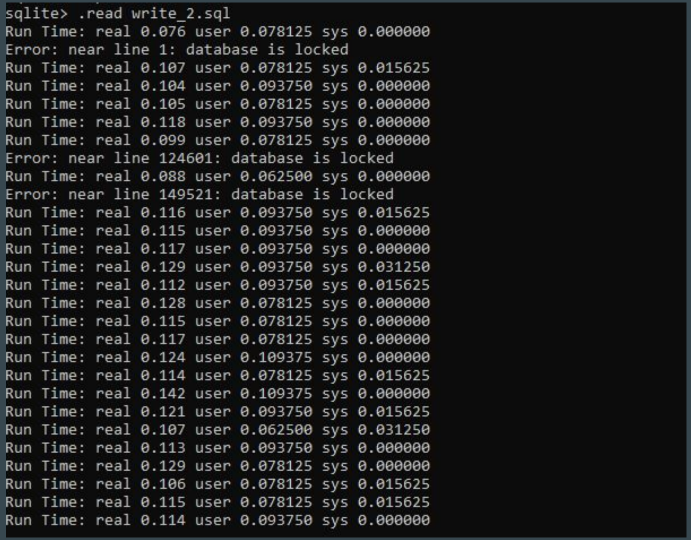
Create index salary on salaries:

SQLite - 1.95s

MySQL - 2.5s

SQLite Demos

SQLite Concurrent Writing:



```
sqlite> .read write_2.sql
Run Time: real 0.076 user 0.078125 sys 0.000000
Error: near line 1: database is locked
Run Time: real 0.107 user 0.078125 sys 0.015625
Run Time: real 0.104 user 0.093750 sys 0.000000
Run Time: real 0.105 user 0.078125 sys 0.000000
Run Time: real 0.118 user 0.093750 sys 0.000000
Run Time: real 0.099 user 0.078125 sys 0.000000
Error: near line 124601: database is locked
Run Time: real 0.088 user 0.062500 sys 0.000000
Error: near line 149521: database is locked
Run Time: real 0.116 user 0.093750 sys 0.015625
Run Time: real 0.115 user 0.093750 sys 0.000000
Run Time: real 0.117 user 0.093750 sys 0.000000
Run Time: real 0.129 user 0.093750 sys 0.031250
Run Time: real 0.112 user 0.093750 sys 0.015625
Run Time: real 0.128 user 0.078125 sys 0.000000
Run Time: real 0.115 user 0.078125 sys 0.000000
Run Time: real 0.117 user 0.078125 sys 0.000000
Run Time: real 0.124 user 0.109375 sys 0.000000
Run Time: real 0.114 user 0.078125 sys 0.015625
Run Time: real 0.142 user 0.109375 sys 0.000000
Run Time: real 0.121 user 0.093750 sys 0.015625
Run Time: real 0.107 user 0.062500 sys 0.031250
Run Time: real 0.113 user 0.093750 sys 0.000000
Run Time: real 0.129 user 0.078125 sys 0.000000
Run Time: real 0.106 user 0.078125 sys 0.015625
Run Time: real 0.115 user 0.078125 sys 0.015625
Run Time: real 0.114 user 0.093750 sys 0.000000
```

SQLite Typing - Live

Conclusion

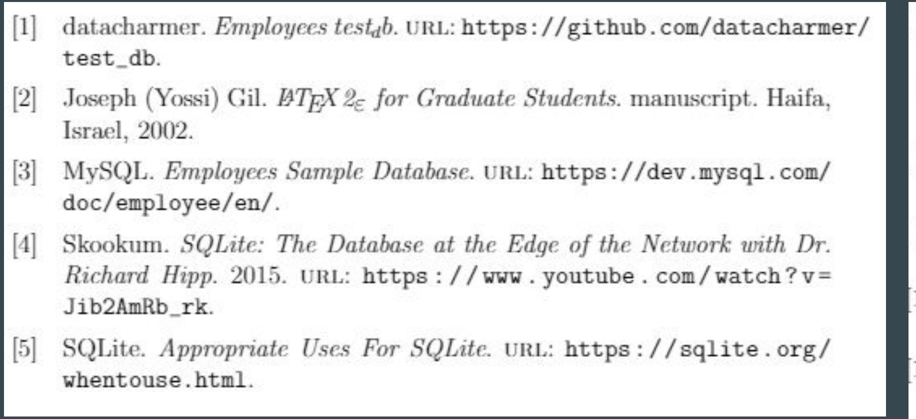
- In the five queries, SQLite was 2-13 times faster than MySQL.

- In other tests, SQLite are closer in performance, but with SQLite edging MySQL out.

- Increased performance comes from lack of overhead, simplistic code, dealing with a single .db file.

- Comes through on its selling point.

Resources



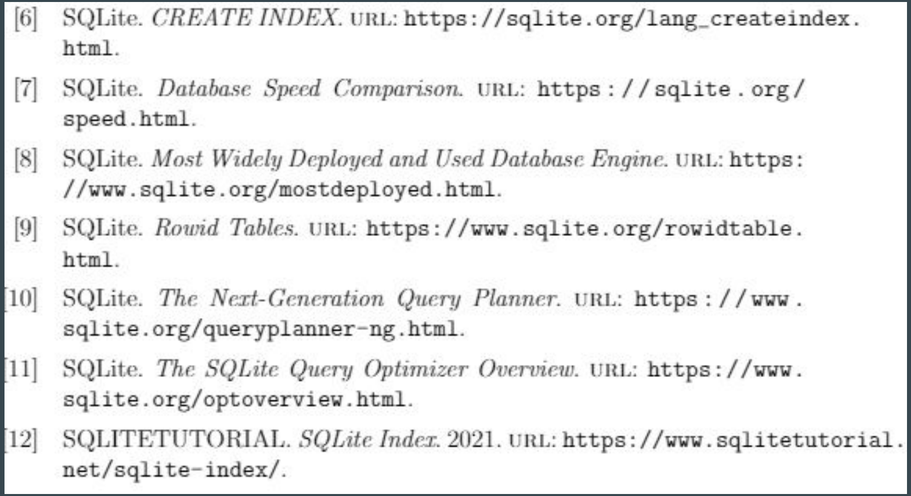
[1] datacharmer. *Employees test$_d$b*. URL: https://github.com/datacharmer/test_db.

[2] Joseph (Yossi) Gil. *$\LaTeX\,2_\varepsilon$ for Graduate Students*. manuscript. Haifa, Israel, 2002.

[3] MySQL. *Employees Sample Database*. URL: https://dev.mysql.com/doc/employee/en/.

[4] Skookum. *SQLite: The Database at the Edge of the Network with Dr. Richard Hipp*. 2015. URL: https://www.youtube.com/watch?v=Jib2AmRb_rk.

[5] SQLite. *Appropriate Uses For SQLite*. URL: https://sqlite.org/whentouse.html.



[6] SQLite. *CREATE INDEX*. URL: https://sqlite.org/lang_createindex.html.

[7] SQLite. *Database Speed Comparison*. URL: https://sqlite.org/speed.html.

[8] SQLite. *Most Widely Deployed and Used Database Engine*. URL: https://www.sqlite.org/mostdeployed.html.

[9] SQLite. *Rowid Tables*. URL: https://www.sqlite.org/rowidtable.html.

[10] SQLite. *The Next-Generation Query Planner*. URL: https://www.sqlite.org/queryplanner-ng.html.

[11] SQLite. *The SQLite Query Optimizer Overview*. URL: https://www.sqlite.org/optoverview.html.

[12] SQLITETUTORIAL. *SQLite Index*. 2021. URL: https://www.sqlitetutorial.net/sqlite-index/.