SQLite: Past, Present, and Future

Tanisha Hegde

Structure

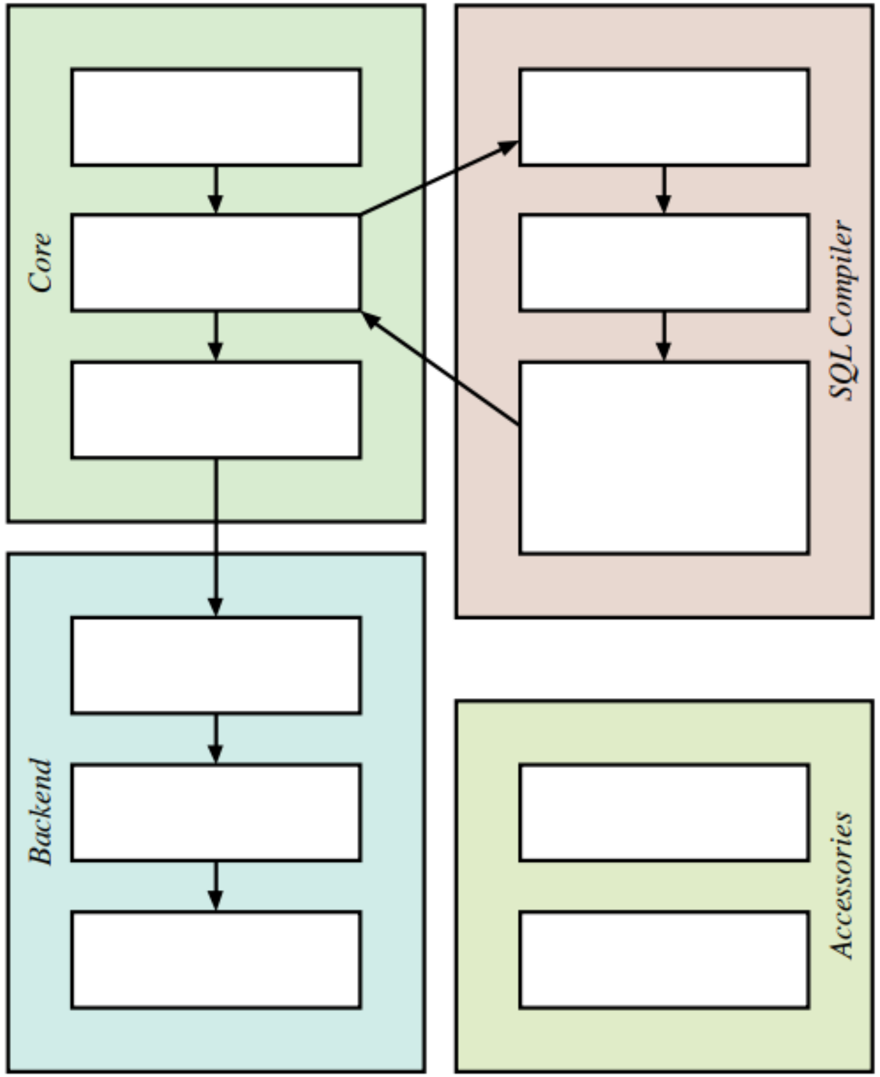
- Overview
- Architecture of SQLite
- Evaluation and Optimizations
    - SQLite with OLTP
    - SQLite with OLAP
    - SQLite and Blob I/O
- Conclusion

Overview

- SQLite is a popular embedded relational database management system (RDBMS).
- Is Lightweight and has a widespread use in various applications, including mobile devices, browsers, and desktop software.
- SQLite is primarily designed for fast online transaction processing (OLTP), employing row-oriented execution and a B-tree storage format.
- With the rise of data science and need to store data in CSV and JSON formats embeddable database engines are equipped making SQLite already popular (Kaggle) .

Architecture

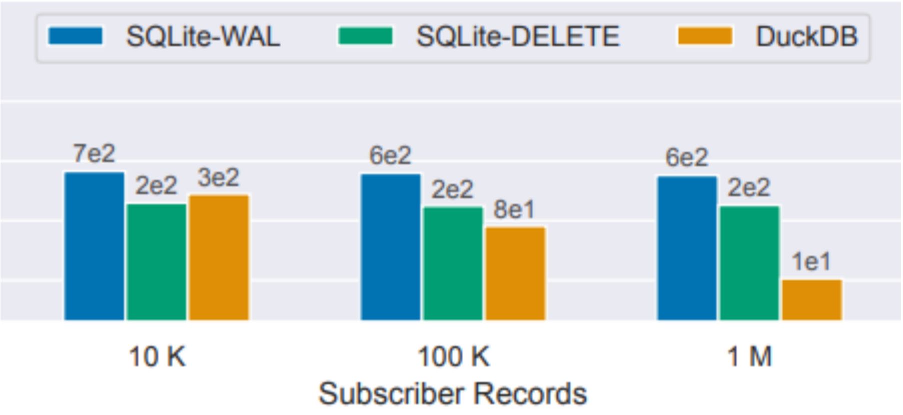
- SQL Compiler – Tokenizer, parser and code generator
    - Output: bytecode
- Core – Execution engine structured as a virtual machine
- Backend – B tree module. DB file is a collection of B trees
- Accessories – suite for tests and memory allocation, string utils and random number generators



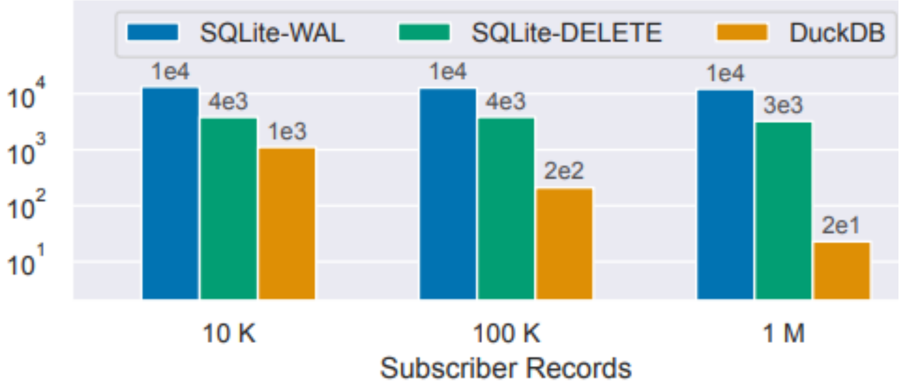
Evaluation of Workloads and Optimizations

SQLite for OLTP workloads

- TATP benchmark
  - 80% read only and 20% Updates, Inserts and Deletes
- Evaluate on DELETE, TRUNCATE and WAL journal mode.
- TATP is not an ideal workload for DuckDB.
- DuckDB is optimized for bulk updates, like adding a column to a table, appending a large batch of rows, rather than fine-grained operations present in OLTP workloads.
- Observation
  - SQLite-WAL reaches a throughput of 10 thousand TPS, which is 10X faster than DuckDB .
  - On the Raspberry Pi, the performance gap is smaller yet still significant



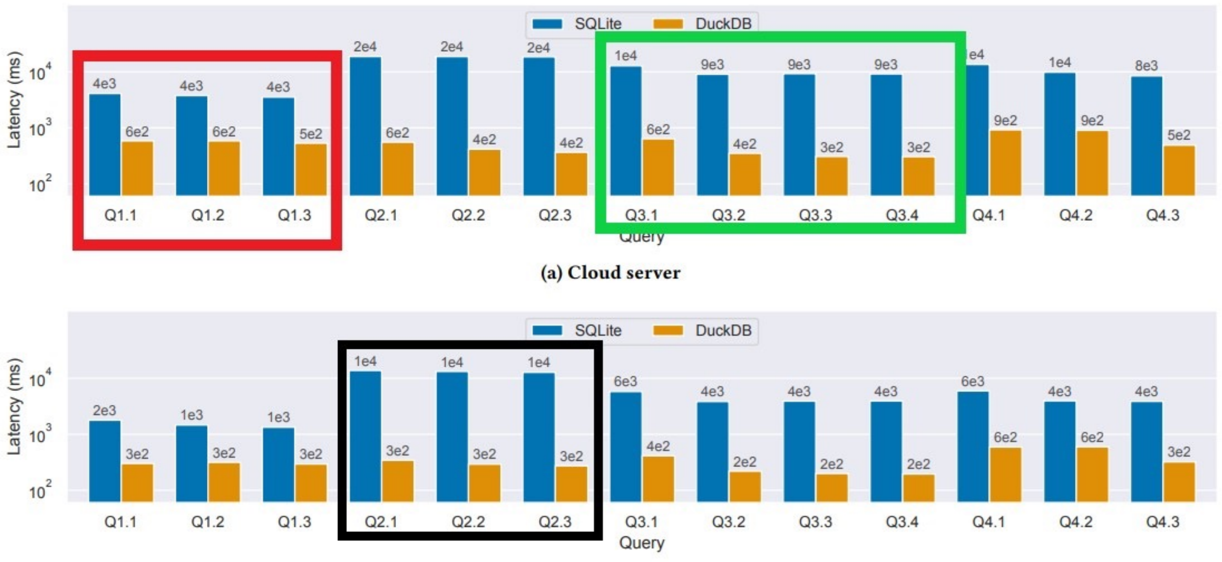
(b) Raspberry Pi



(a) Cloud server

SQLite with OLAP workloads

- SSB benchmark
  - Has a large fact table and smaller dimension tables.
- SSB queries involve joins between the fact table and the dimension tables with filters on dimension table attribute
- Observation:
  - Widest performance margin is on query flight 2, for which DuckDB is 30-50X faster.
  - SQLite’s fastest queries are in flight 1, whereas DuckDB’s fastest queries are in flight 3.



(a) Cloud server

(b) Raspberry Pi

Figure 3: SSB latency (logarithmic scale, lower is better).

Optimization

Profiling to reason on observations

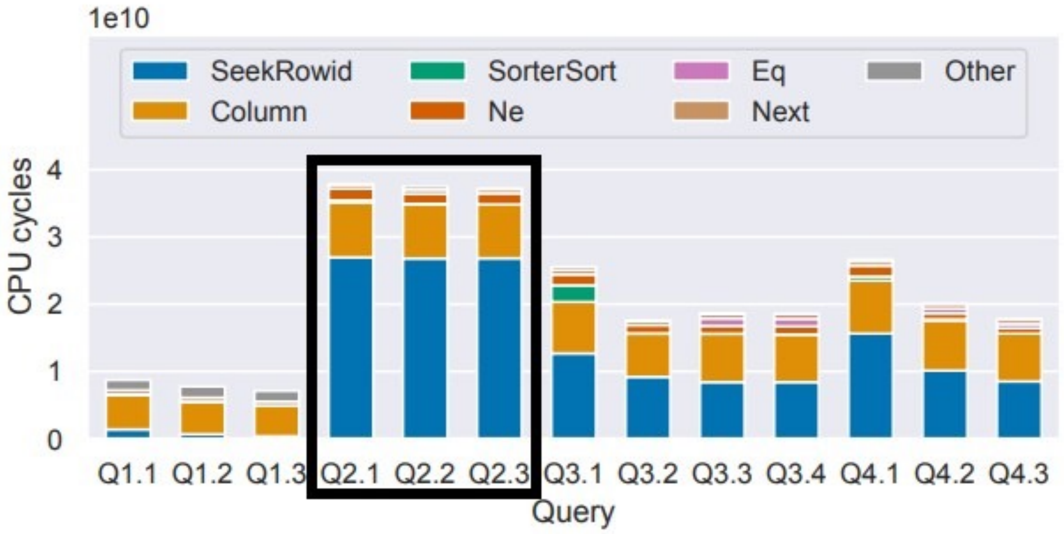
- `SeekRowid` instruction searches a B-tree index for a row with a given row ID.
- The `Column` instruction extracts a column from a given record

Key optimization targets

- Avoiding unnecessary B-tree probes – *Detailed*
- Streamlining value extraction - *Discarded*
  - sacrifice the stability and portability of the database file format for the added performance.

Observation

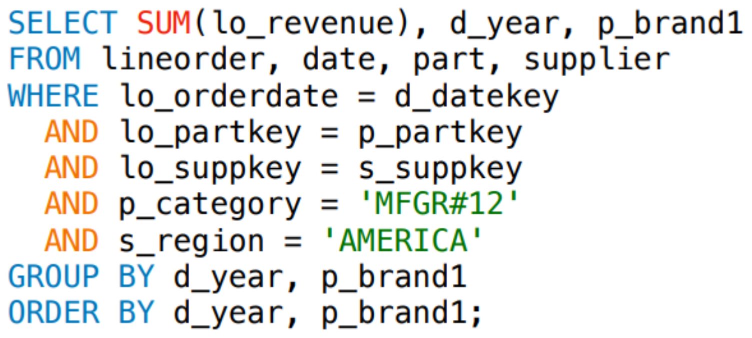
- Large CPU cycles for flight 2 by `SeekRowid`



SSB Performance Profile

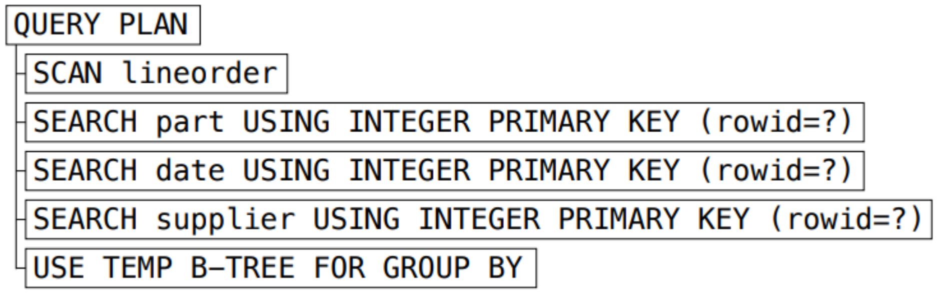
Avoiding unnecessary B-tree probes

- An example:
  - Inner Tables: date, part, supplier
  - Outer Table: Line order
- Costly to probe the primary key of Part table as it is the largest
- Only 0.8% of the lineorder tuples satisfy the restrictions on p_category and s_region
- A large portion of B-tree probes are excluded from the result.
- Solution: **Bloom filters**



```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
  AND p_category = 'MFGR#12'
  AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

(a) SQL

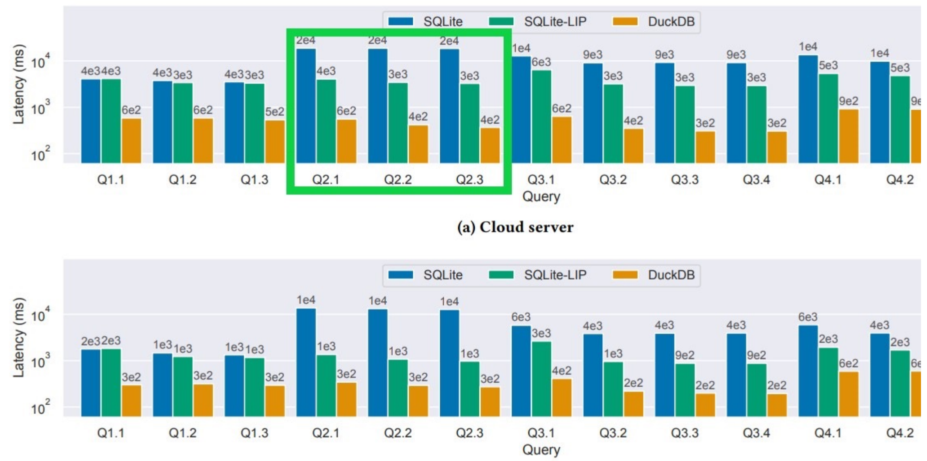


```
QUERY PLAN
 |-SCAN lineorder
 |-SEARCH part USING INTEGER PRIMARY KEY (rowid=?)
 |-SEARCH date USING INTEGER PRIMARY KEY (rowid=?)
 |-SEARCH supplier USING INTEGER PRIMARY KEY (rowid=?)
 `-USE TEMP B-TREE FOR GROUP BY
```

(b) Query plan pre-optimization

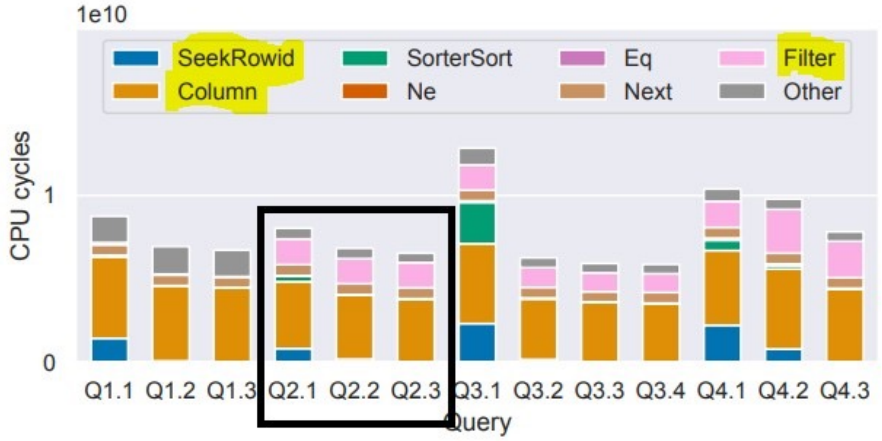
Avoiding unnecessary B-tree probes

- Bloom Filters
  - Implement Lookahead Information Passing(LIP)
  - Create Bloom filters on all the inner (dimension) tables before the join processing starts.
  - Pass the Bloom filters to the first join operation.
  - Probe the Bloom filters before carrying out the rest of the join.
- Result
  - SQLite is now 4.2X faster on SSB.



(a) Cloud server

(b) Raspberry Pi



(b) Post-optimization

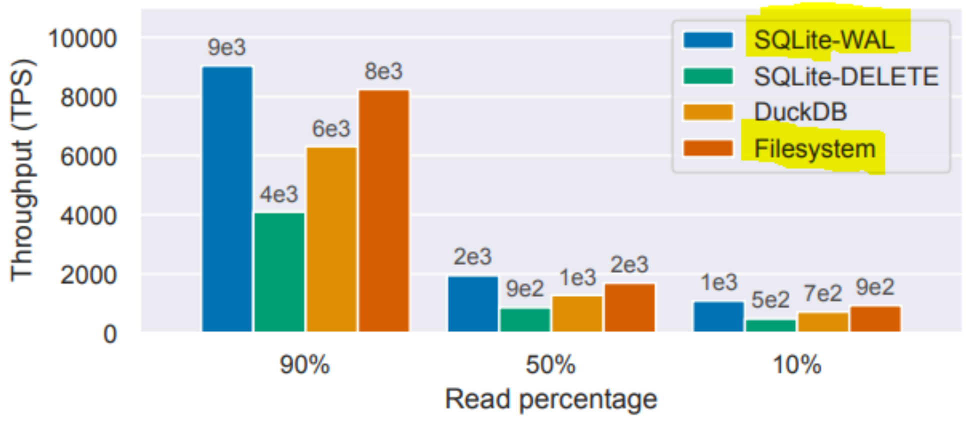
SQLite and Blob I/O

Blob Benchmark

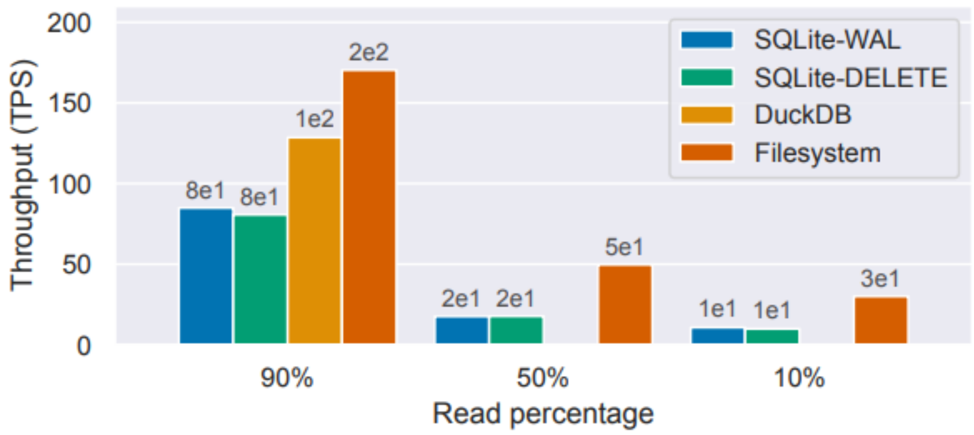
- simulates an application that uses a database engine to manage raw blob data
- A table is created in the database with a single row and a single column of blob data with a given size.
- repeatedly either read or write the entire blob, based on specified probabilities.

Observation

- 100 KB blobs - SQLite-WAL produces the highest throughput of the transactional methods.
- SQLite-WAL even has a slight edge over the filesystem for small blobs.
  - Due to SQLite's ability to serve read requests from its cache, whereas the filesystem serves read requests with calls to fread.
- 10 MB blobs, DuckDB produces the highest throughput of the transactional methods.



(a) Cloud server, 100 KB blob



(b) Cloud server, 10 MB blob

Conclusion

- The widespread deployment of SQLite is likely a result of its cross platform code and file format, compact and self-contained library, extensive testing, and low overhead.
- Is primarily designed for efficient OLTP.
- Bloom Filters have been integrated into SQLite and resulted in up to 4.2X speedup on SSB.

Thoughts and Questions