SQLiteKV: An Efficient LSM-tree-based SQLite-like Database Engine for Mobile Devices

Yuanjing Shi, Zhaoyan Shen, Zili Shao
The Hong Kong Polytechnic University
Hong Kong, China
Organization

• Introduction
• Background
• Our Design
• Evaluation
• Conclusion
Organization

- Introduction
- Background
- Our Design
- Evaluation
- Conclusion
Introduction

Android I/O Stack

SQLite is used in:
- Firefox, Chrome, Safari,
- Wordpress, Twitter, Facebook
- Instagram, Dropbox, Skype
and so on...

SQLite - server-less, transactional SQL database

•Current mobile devices rely heavily on SQLite
Problems

1. Complicated Data Management in B-Tree
   • Frequent split and merge operations for B-Tree -> small, random I/O operations

2. Journaling of Journal (JOJ)
   • Frequent synchronization between journal and database files
Previous Work

Optimization Strategies of SQLite

• Eliminating Journaling of Journal (JOJ) between SQLite and EXT4. [1]

• Utilizing Non-Volatile Memory (NVM) to eliminate small, random updates. [2]

Limited performance improvement with SQLite’s fundamental data structure - B-tree.


• Current Key-Value Store is much faster than SQLite on mobile devices. E.g. SnappyDB vs. SQLite

• Hence, we are looking for a SQL-compatible Key-Value database on mobile platforms.
Our Idea

• We hereby, for the first time, propose to leverage the LSM-tree-based key-value data structure to improve SQLite performance.

• LSM-tree provides:
  • High Efficiency, Scalability and Availability
  • NoSQL schema

• Existing Key-Value store cannot be directly adopted by mobile devices as:
  • It is designed for scalable and distributed computing environments with large datasets.
Organization

- Introduction
- Background
- Our Design
- Evaluation
- Conclusion
Background

- SQLite

![SQLite Diagram]

- Core Interface
- SQL Command Processor
- Virtual Machine
- Backend B-Tree
- Pager
- OS Interface

- SQL Compiler
  - Tokenizer
  - Parser
  - Code Generator

- Root Page
  - Page
  - Disk
  - Page

...
Background

- SQLite
Background

• LSM-Tree-based Key Value Store
Background

- LSM-Tree-based Key Value Store
Organization

- Introduction
- Background
- **Our Design**
- Evaluation
- Conclusion
**System Architecture**

- **SQLiteKV** is proposed to solve the above two issues when porting KV to mobile devices

- **LSM-tree-based Data Store**

- **Slab-based caching mechanism**  
  - Front End

- **Selective Index Management**  
  - Back End
Front End

• SQLite-to-KV Compiler

```
"SELECT id, name FROM table
WHERE name = "xxx" AND id = "xxx"
```

```
key(name) = "xxx" value(a1) = "xxx
key(name) = "xxx" value(a2) = "xxx
key(id) = "xxx" value(a1) = "xxx
key(id) = "xxx" value(a2) = "xxx
...
```
Front End

- SQL Compiler w. Cache
Front End

- Slab-based Caching

Diagram:
- Hash(SQL) to Bucket
- Index to Slab Slot
- Slab Slot to Memory Slabs
- Memory Slabs to Index
- Index to md sid offset

Memory Slabs:
- 32B
- 64B
- 128B
Back End

Diagram showing the structure of a back end system with key-value (KV) items stored in MemTable and ImmuteTable, with MetaData including Bloom Filters and Indexes. The system uses Log, Manifest, and Current layers on disk, with SSTables at different levels (Level 0, Level 1, and Level 2) indicating compaction processes, such as Major Compaction and Minor Compaction.
Organization

• Introduction
• Background
• Our Design
• **Evaluation**
• Conclusion
Experiment Setup

• Google Nexus 6p
  • 2.0GHz oct-core 64 bit Qualcomm Snapdragon 810
  • 3GB LPDDR4 RAM
  • 32GB Samsung eMMC NAND
• Android 7.1 with Linux Kernel 3.10
• SnappyDB 0.4.0, an Android implementation of Google’s LevelDB
• SQLite 3.9
### Workload Characteristics

<table>
<thead>
<tr>
<th>Workload</th>
<th>Query</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upload Heavy</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Read Most</td>
<td>0.95</td>
<td>0.05</td>
</tr>
<tr>
<td>Read Heavy</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Read Latest *</td>
<td>0.95</td>
<td>0.05</td>
</tr>
</tbody>
</table>
Experimental Results

- Overall Performance

(a) Update Heavy: Query vs. Insertion (0.5:0.5)  
(b) Sequential Insertions: (0.95:0.05)
Experimental Results

- Overall Performance

(a) Read Heavy: (1:0)                                           (b) Read Latest: (0.95:0.05)
Experimental Results

- **Micro Performance**

![Bar Chart](image1)
- **Random Insertions**
- **Sequential Insertions**

![Bar Chart](image2)
Experimental Results

- Micro Performance

(a) Random Queries

(b) Sequential Queries
Organization

- Introduction
- Background
- Our Design
- Evaluation
- Conclusion
Conclusion

• SQLite is not efficient with low transactions per second

• We proposed SQLiteKV:
  
  • Front End: SQL interface & Slab-based Caching
  
  • Back End: Selective Index Management & Key-Value Data Store

• We conducted experiments with real devices

  • Outperforms SQLite in various workloads by around 6 times
Thanks!

Q&A