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

23rd Asia and South Pacific Design Automation Conference (ASP-DAC 2018)

Organization

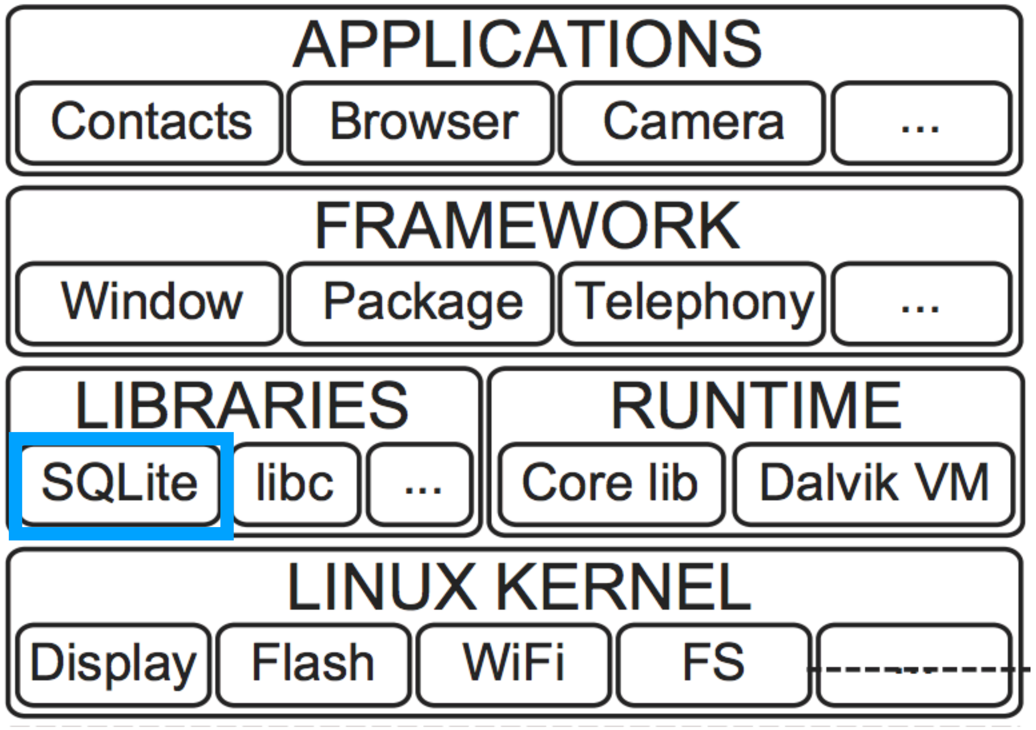
- Introduction
- Background
- Our Design
- Evaluation
- Conclusion

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]

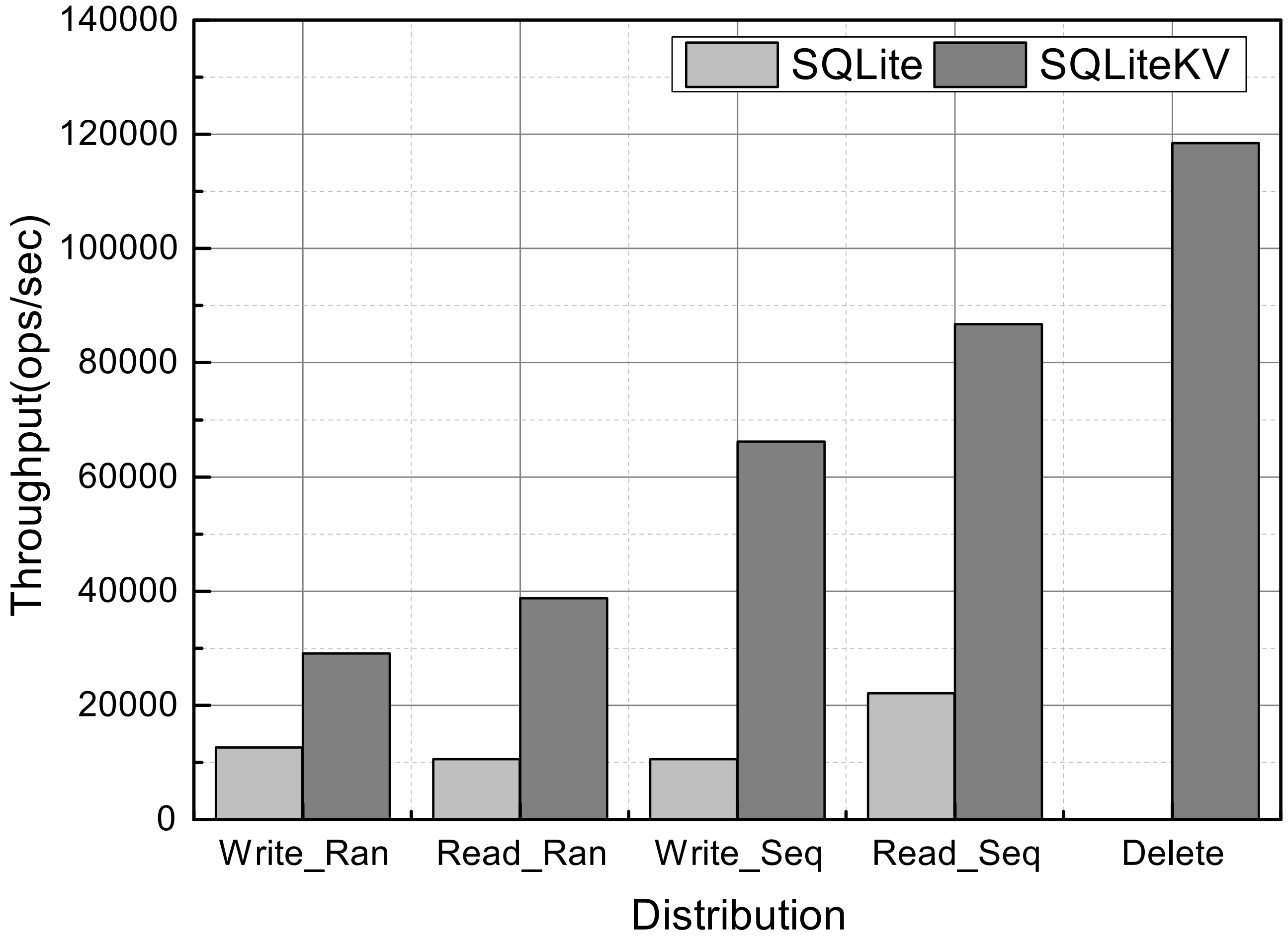
Limited performance improvement with SQLite's fundamental data structure - B-tree.

[1]: Kim, Wook-Hee, et al. "Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split." *FAST*. 2014.

[2]: Z. Shao et al., "Utilizing PCM for energy optimization in embedded systems," in VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on. IEEE, 2012, pp. 398–403.

KV store

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



- Hence, we are looking for a <u>SQL-compatible</u> Key-Value database on mobile platforms.

Our Idea

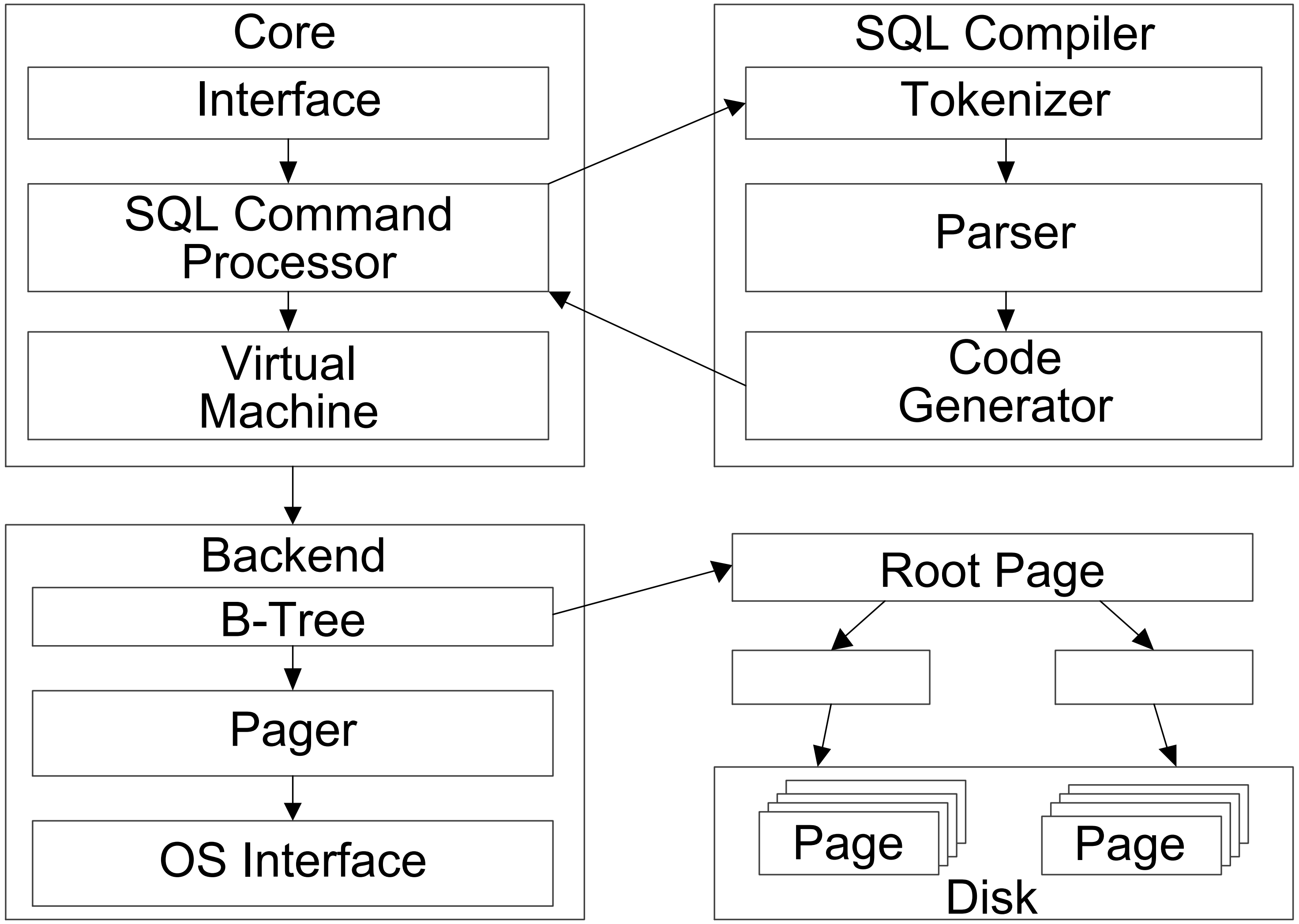
- We hereby, for the first time, propose to leverage the LSM-tree-based key-value data structure to improve SQLite performance.
- LSM-tree provides:
  - High **E**fficiency, **S**calability and **A**vailability
  - **NoSQL** schema
  - Existing Key-Value store cannot be directly adopted by mobile devices as:
    - It is designed for scalable and distributed computing environments with large datasets.

Organization

- Introduction
- **<u>Background</u>**
- Our Design
- Evaluation
- Conclusion

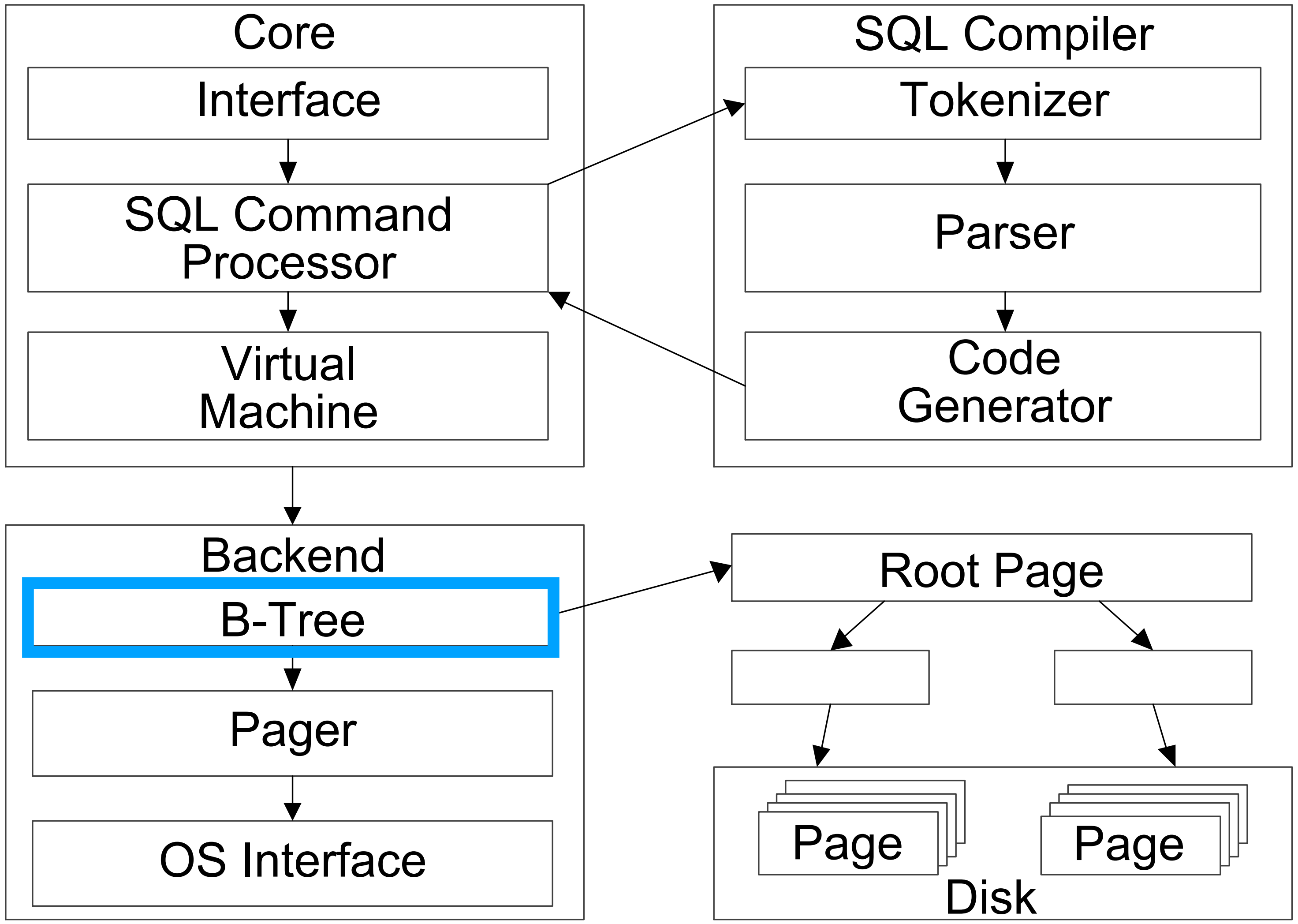
Background

- SQLite



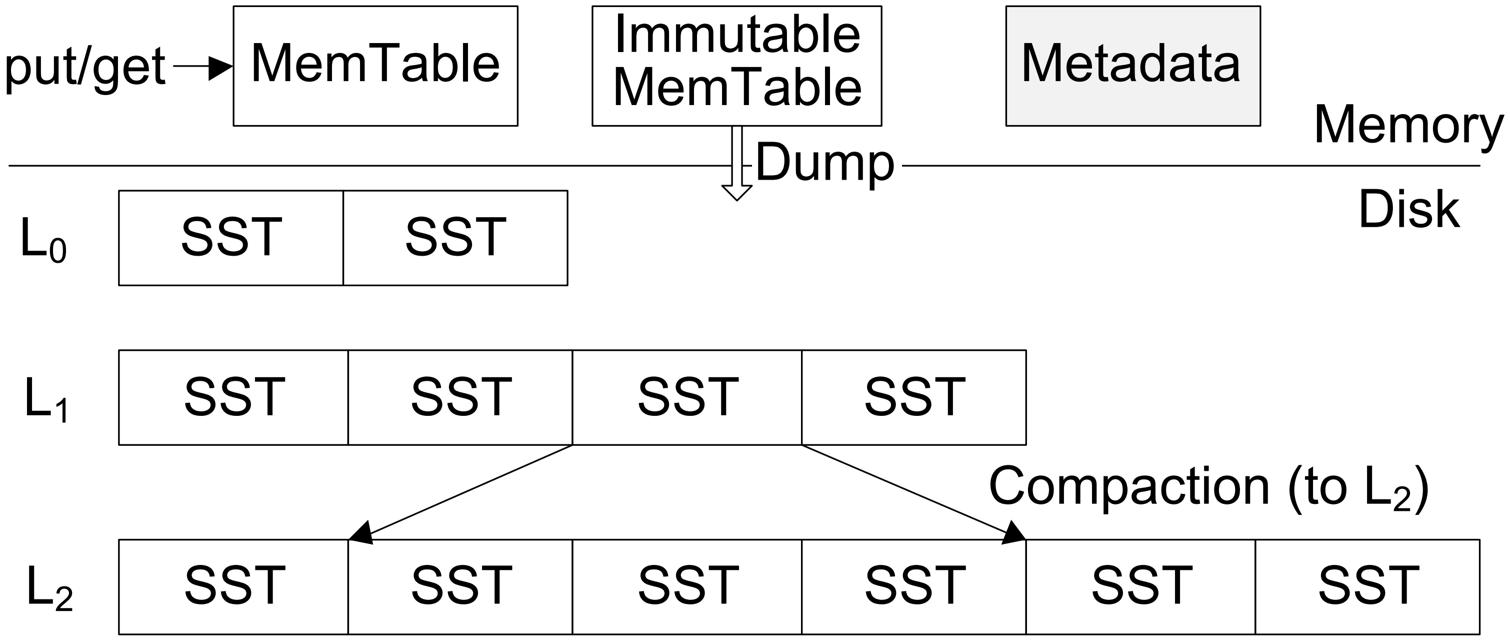
Background

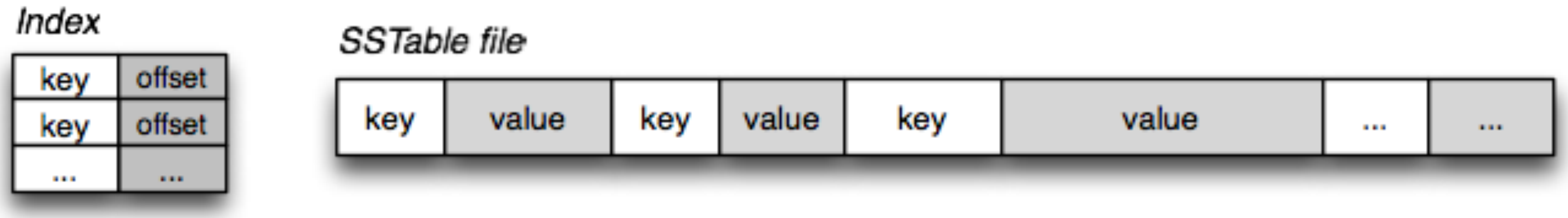
- SQLite



Background

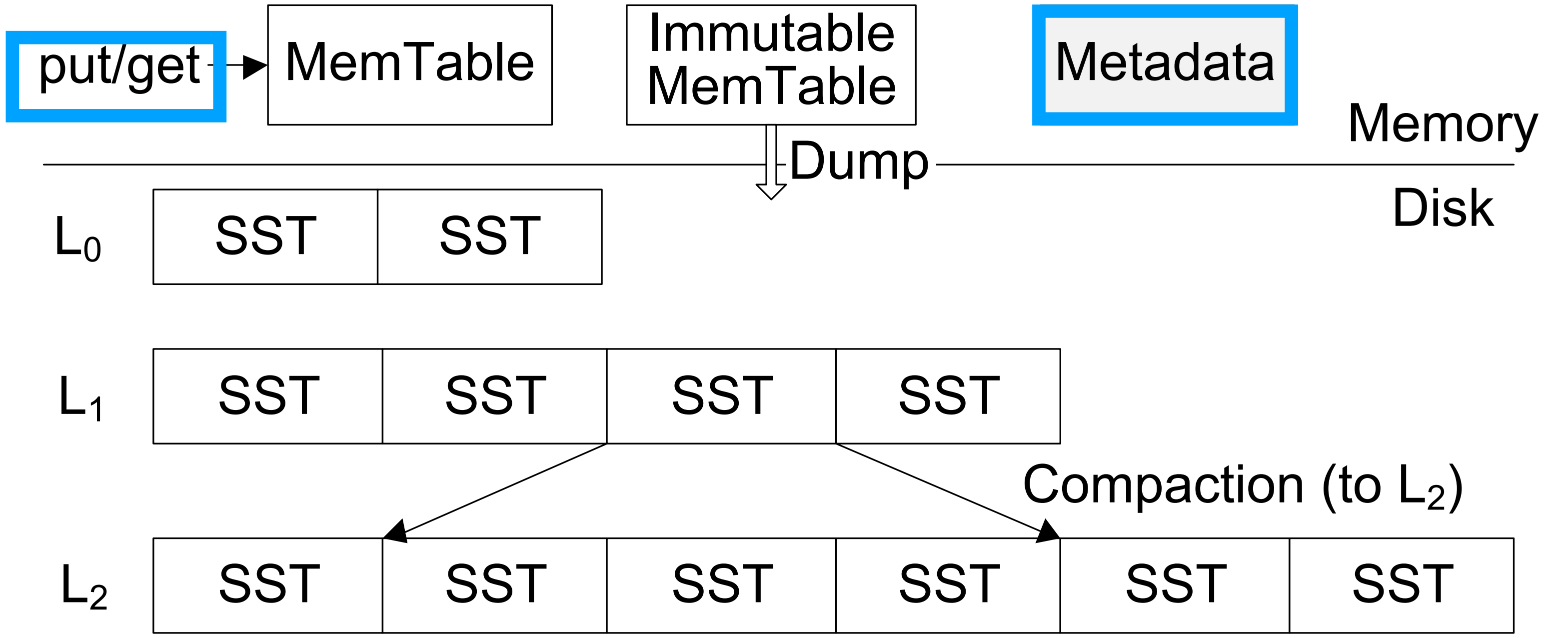
- LSM-Tree-based Key Value Store

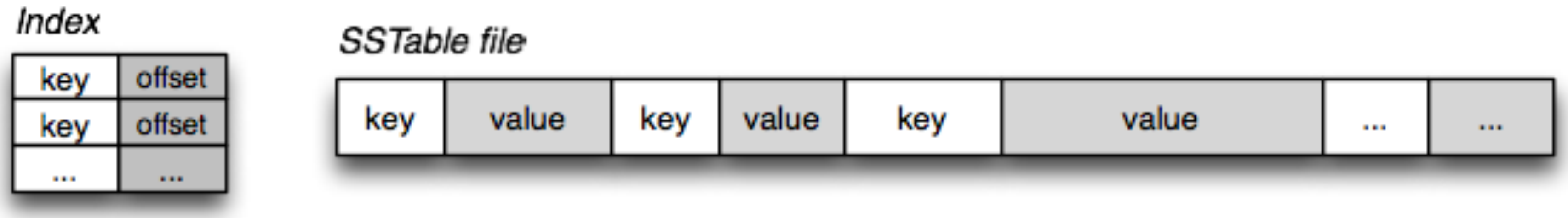




Background

- LSM-Tree-based Key Value Store



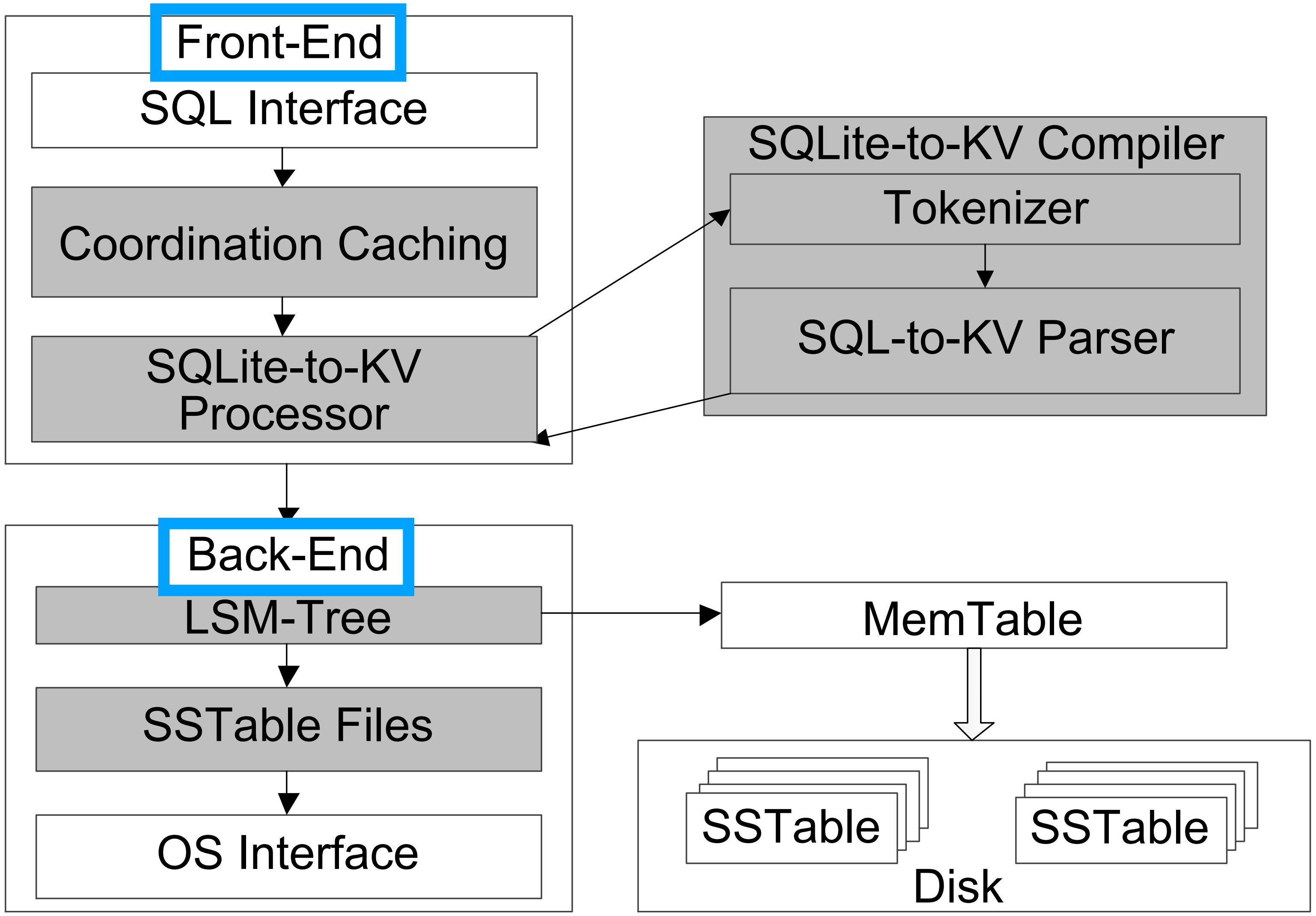


Organization

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

System Architecture

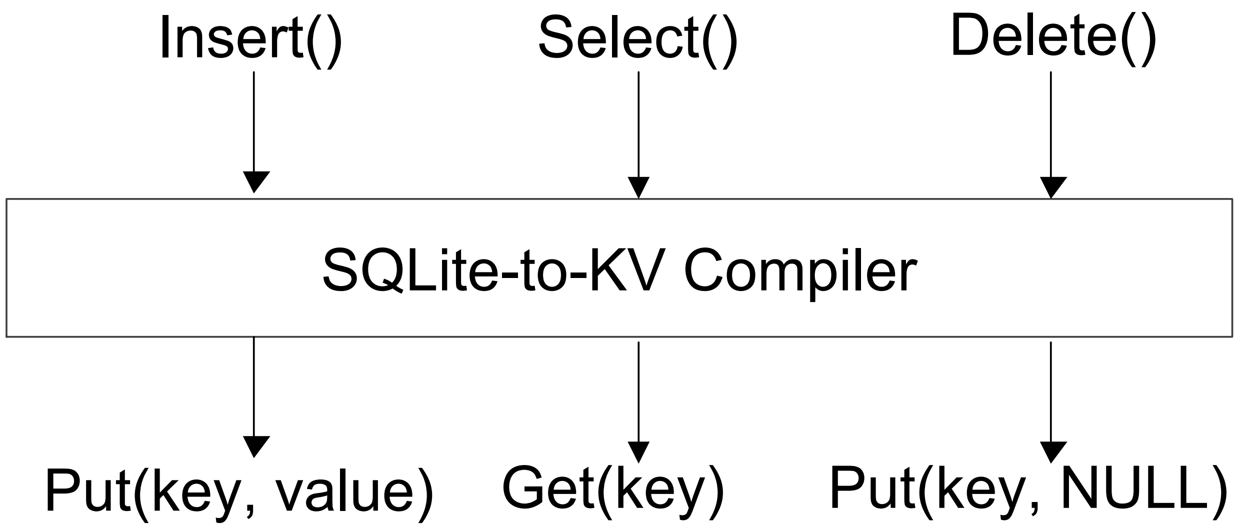
- **SQLiteKV** is proposed to solve the above two issues when porting KV to mobile devices

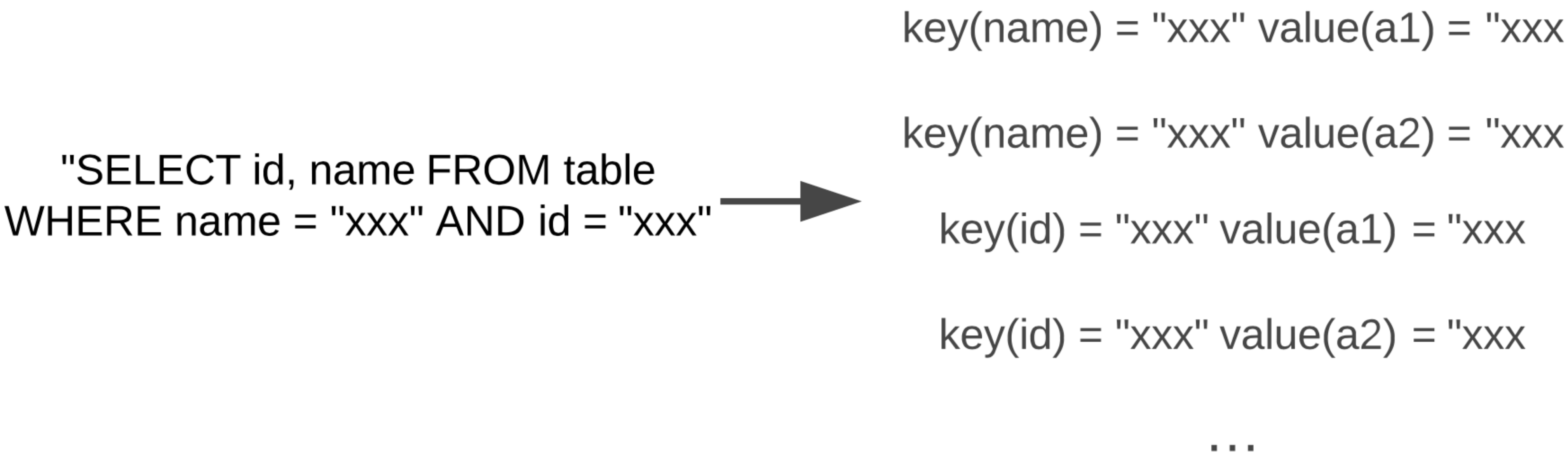


- **LSM-tree-based** Data Store
- **Slab-based** caching mechanism - Front End
- **Selective Index** Management - Back End

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

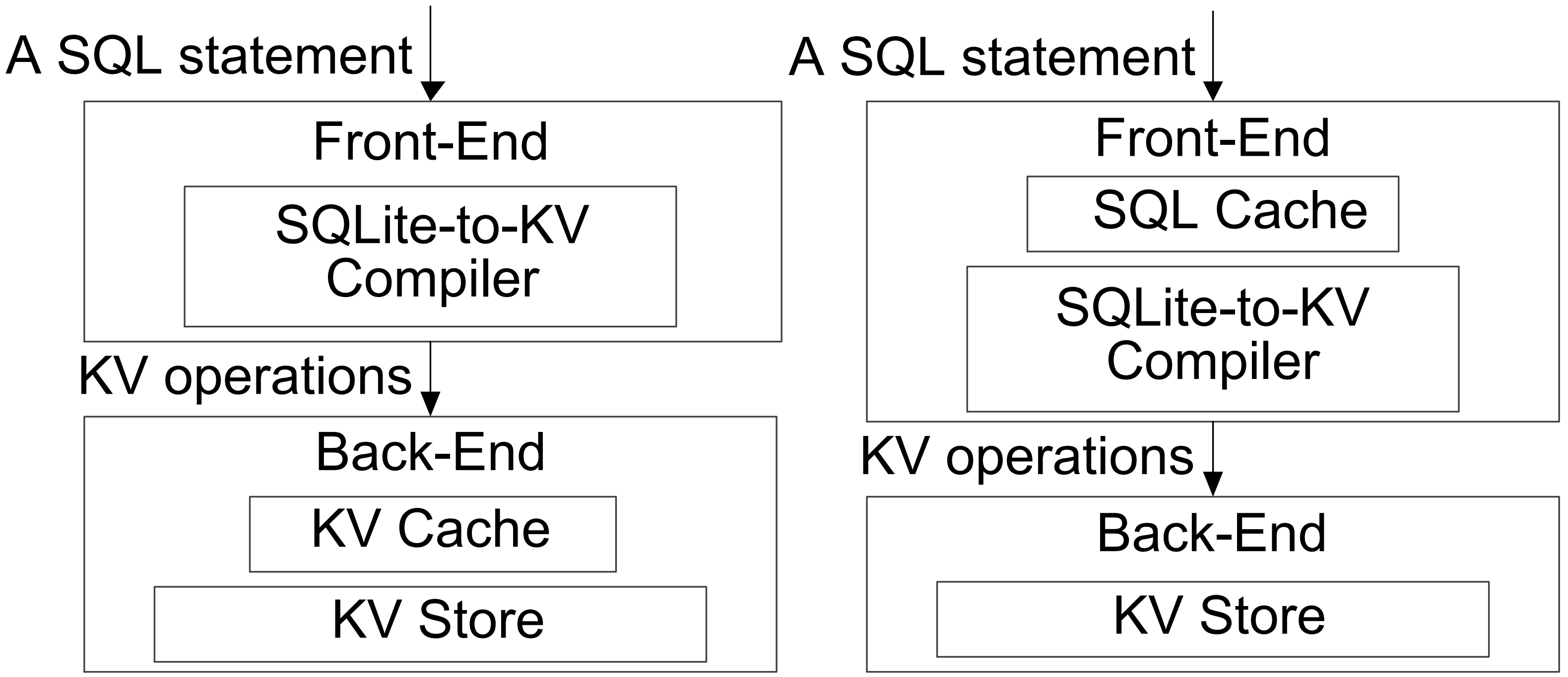
key(id) = "xxx" value(a1) = "xxx

key(id) = "xxx" value(a2) = "xxx

…

Front End

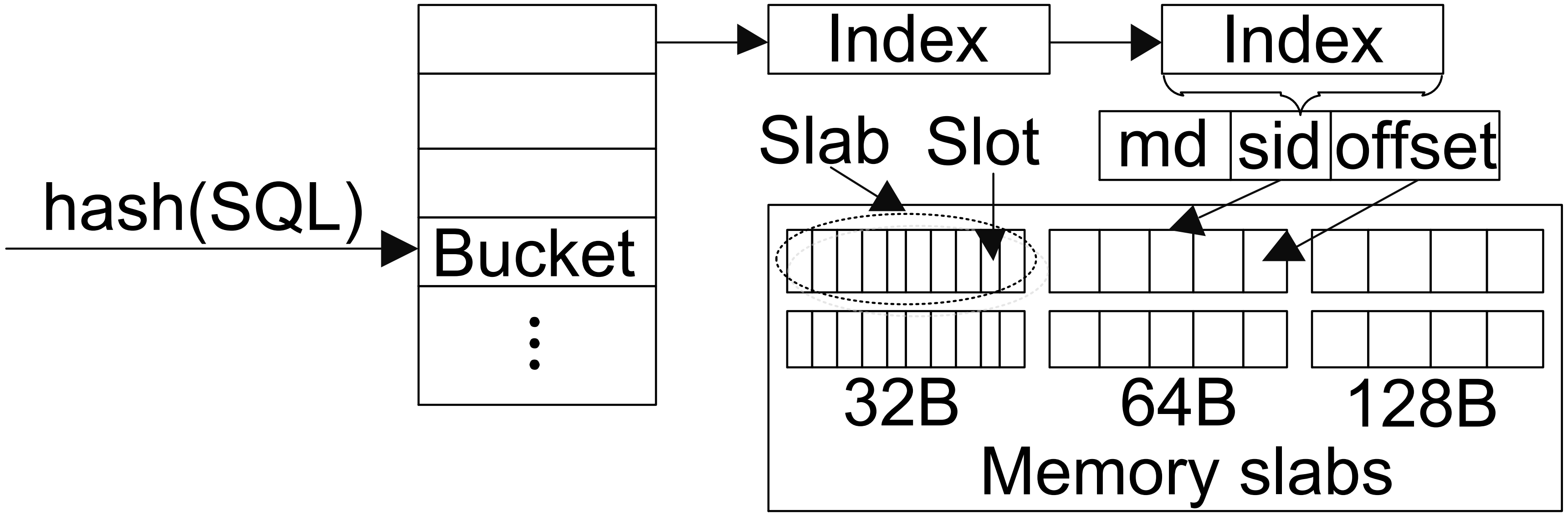
- SQL Compiler w. Cache



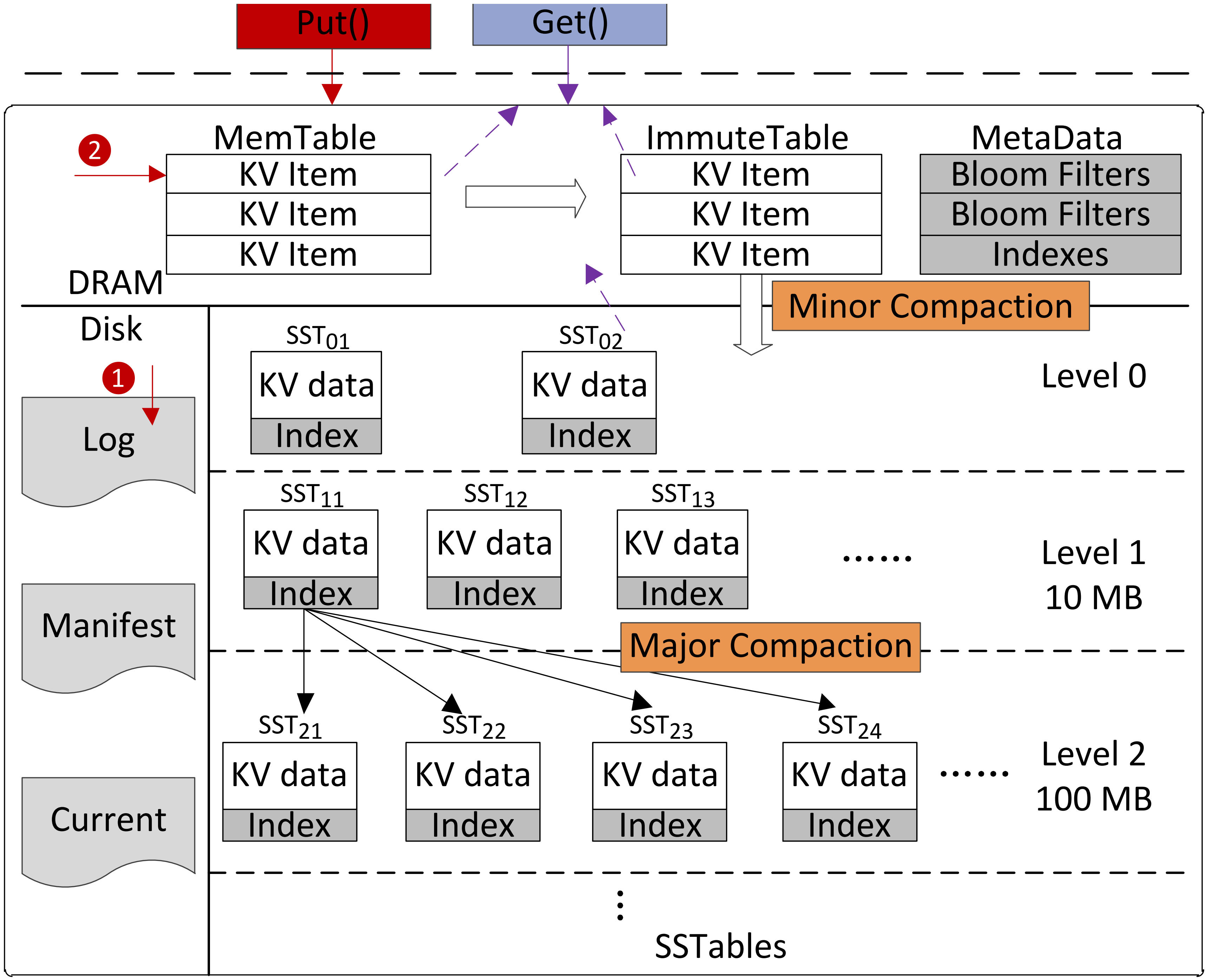
(a) SQLiteKV without Cache (b) SQLiteKV with Cache

Front End

- Slab-based Caching



Back End

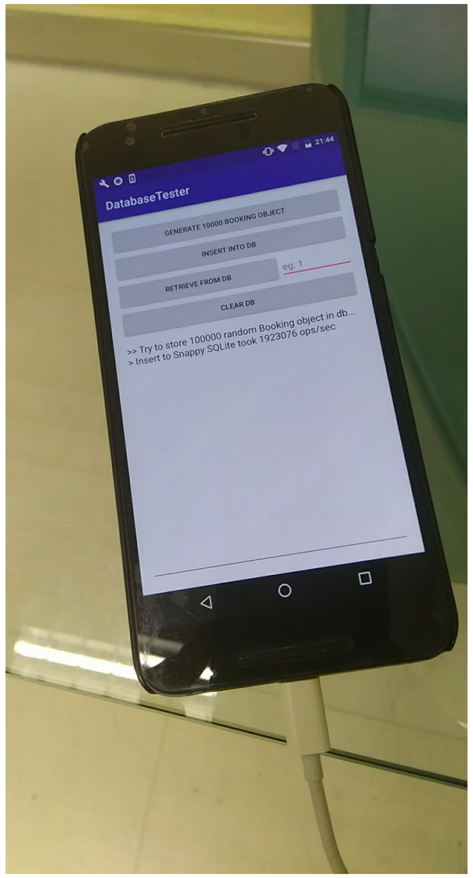

Put()
Get()
②
MemTable
KV Item
KV Item
KV Item
DRAM
ImmuteTable
KV Item
KV Item
KV Item
MetaData
Bloom Filters
Bloom Filters
Indexes
Minor Compaction
Disk
①
Log
Manifest
Current
SST01
KV data
Index
SST02
KV data
Index
Level 0
SST11
KV data
Index
SST12
KV data
Index
SST13
KV data
Index
……
Level 1
10 MB
Major Compaction
SST21
KV data
Index
SST22
KV data
Index
SST23
KV data
Index
SST24
KV data
Index
……
Level 2
100 MB
⋮
SSTables

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



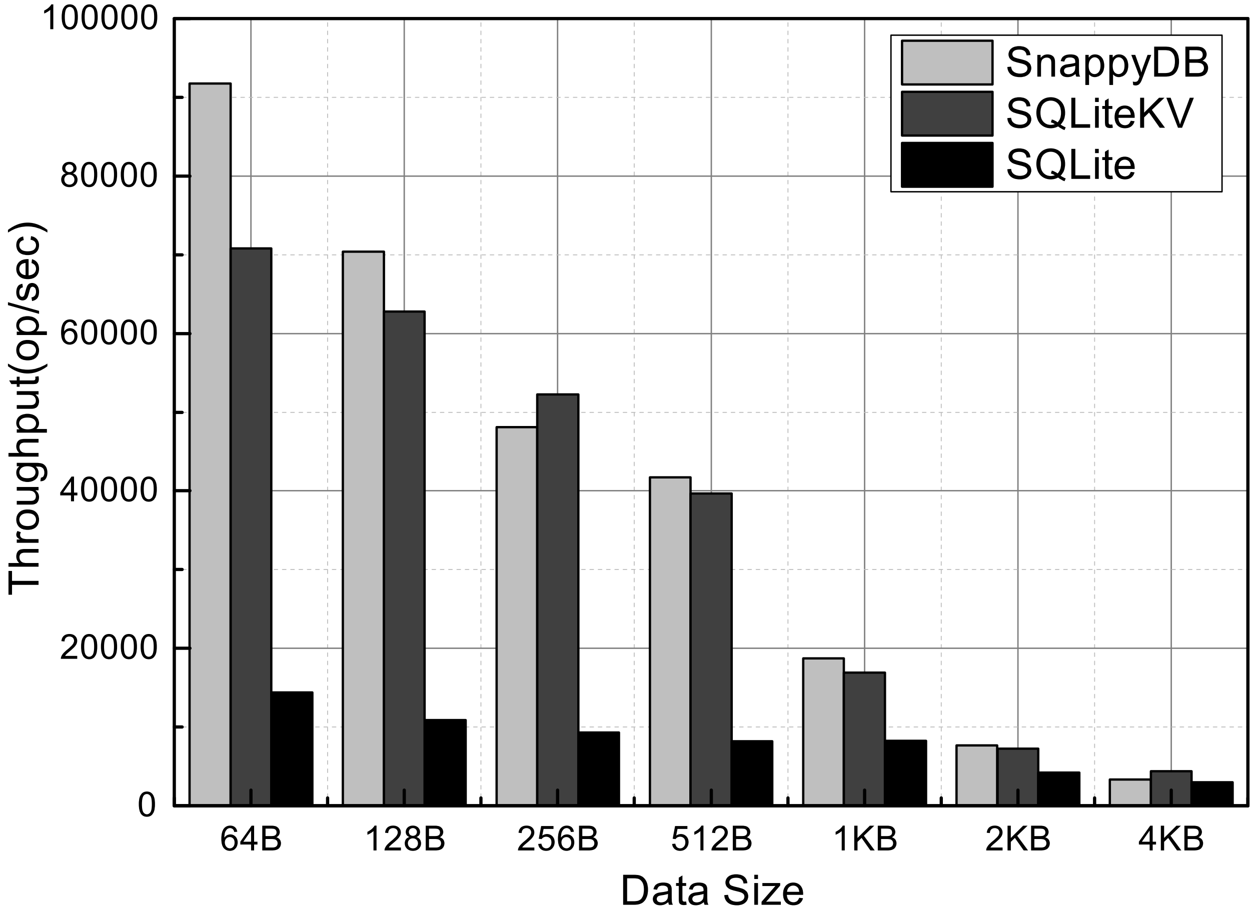
```
>> Try to store 100000 random Booking object in db...
> Insert to Snappy SQLite took 1923076 ops/sec
```

Workload Characteristics

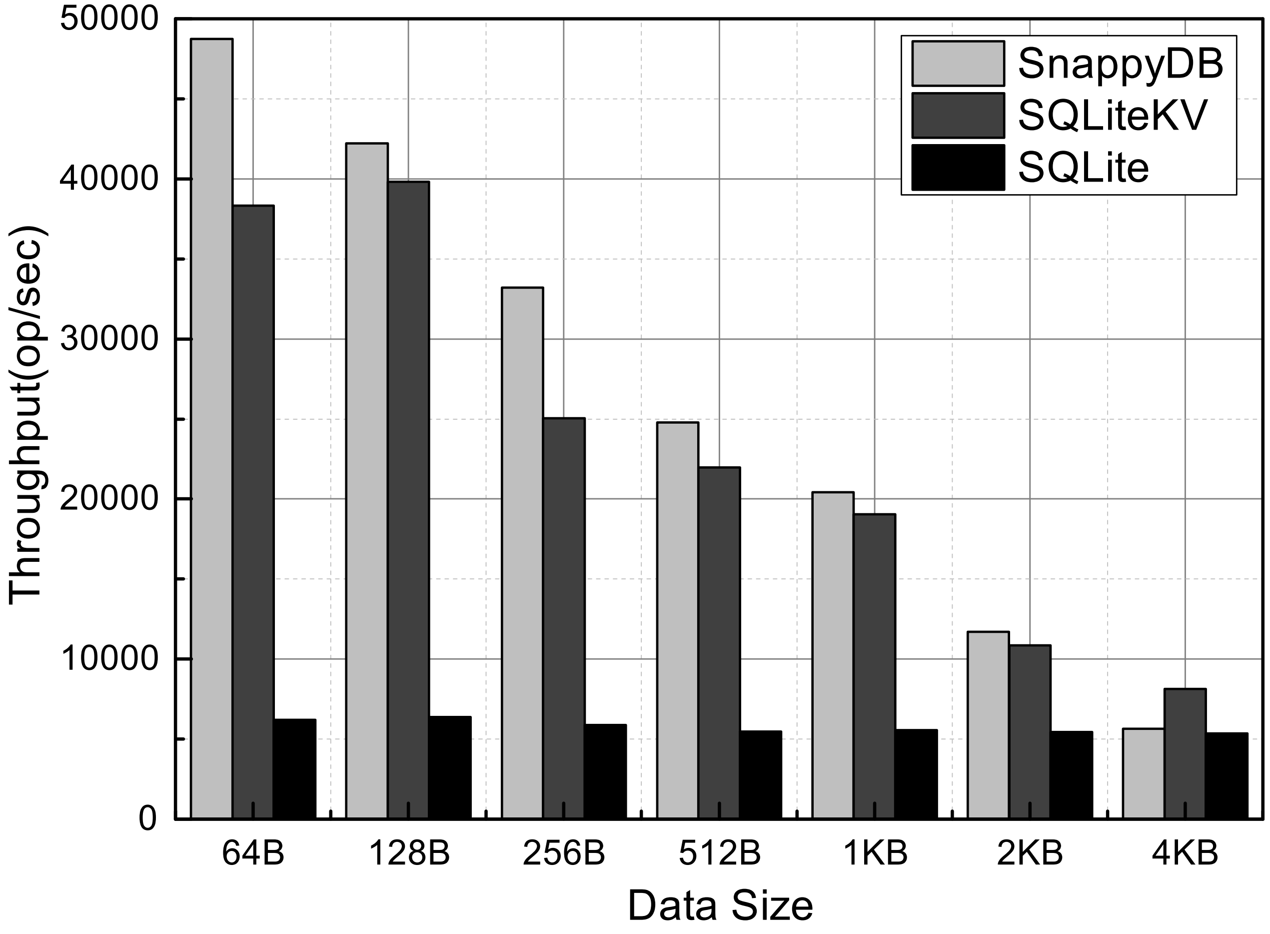
Workload	Query	Insert
Upload Heavy	0.5	0.5
Read Most	0.95	0.05
Read Heavy	1	0
Read Latest *	0.95	0.05

Experimental Results

- Overall Performance



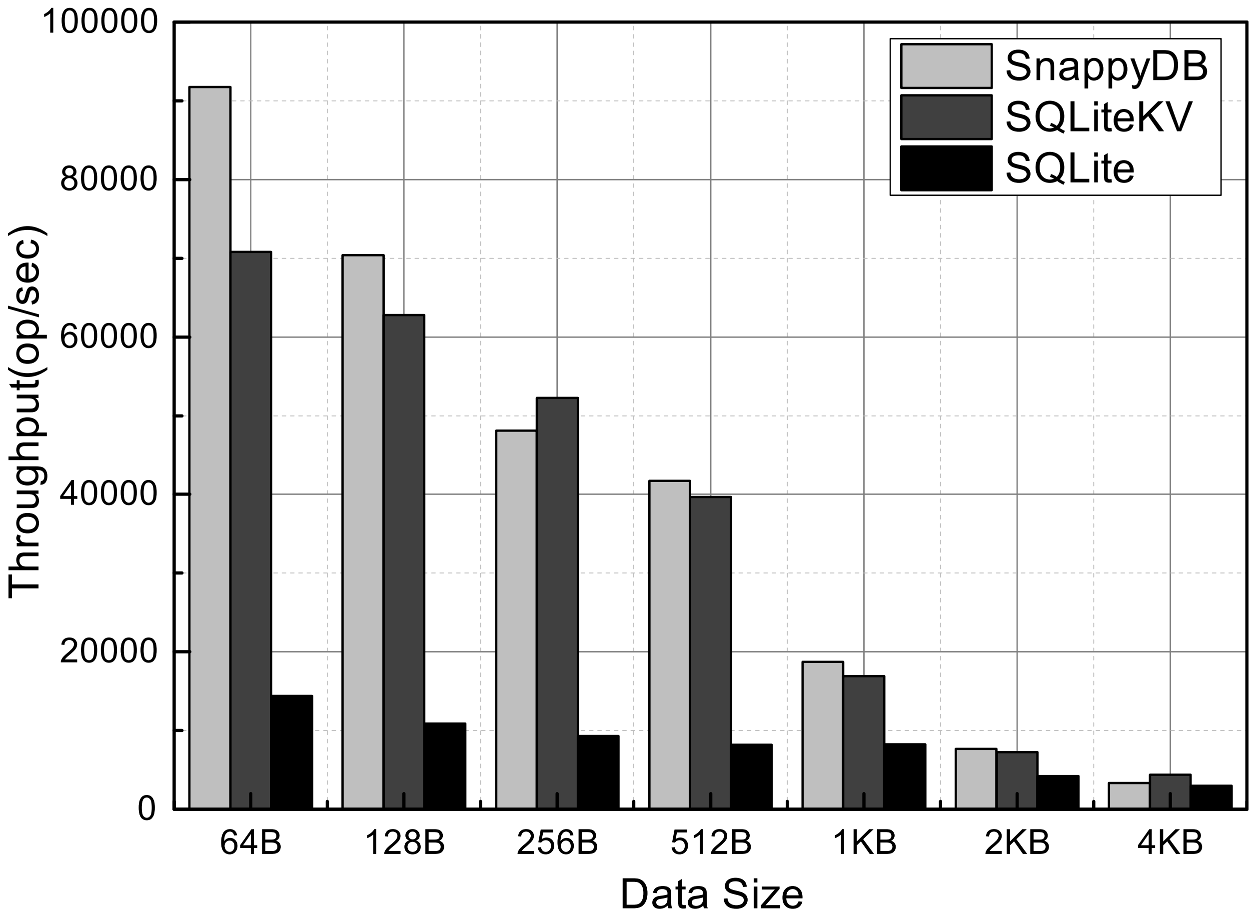
(a) Update Heavy: Query vs. Insertion (0.5:0.5)



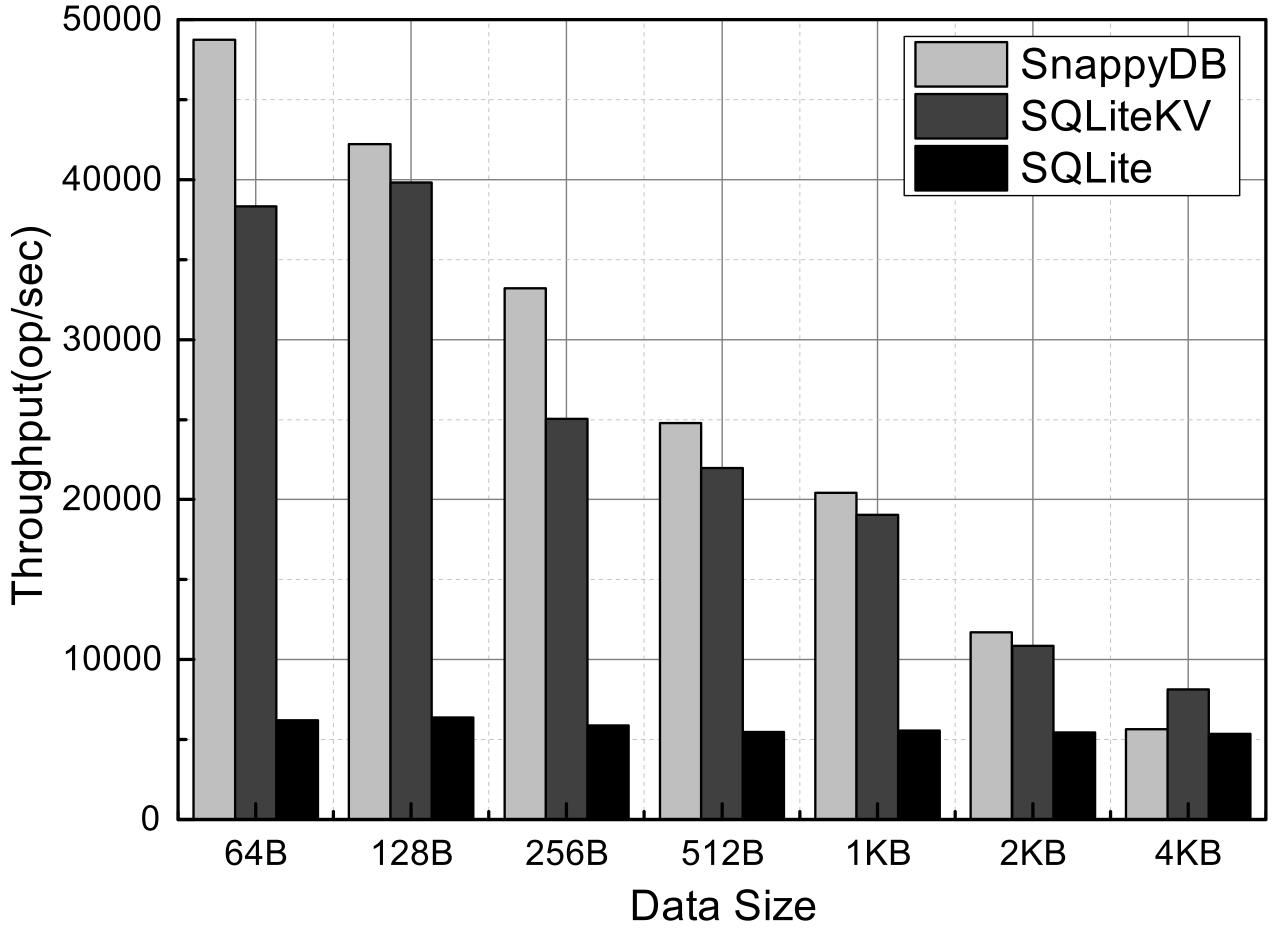
(b) Sequential Insertions: (0.95:0.05)

Experimental Results

- Overall Performance



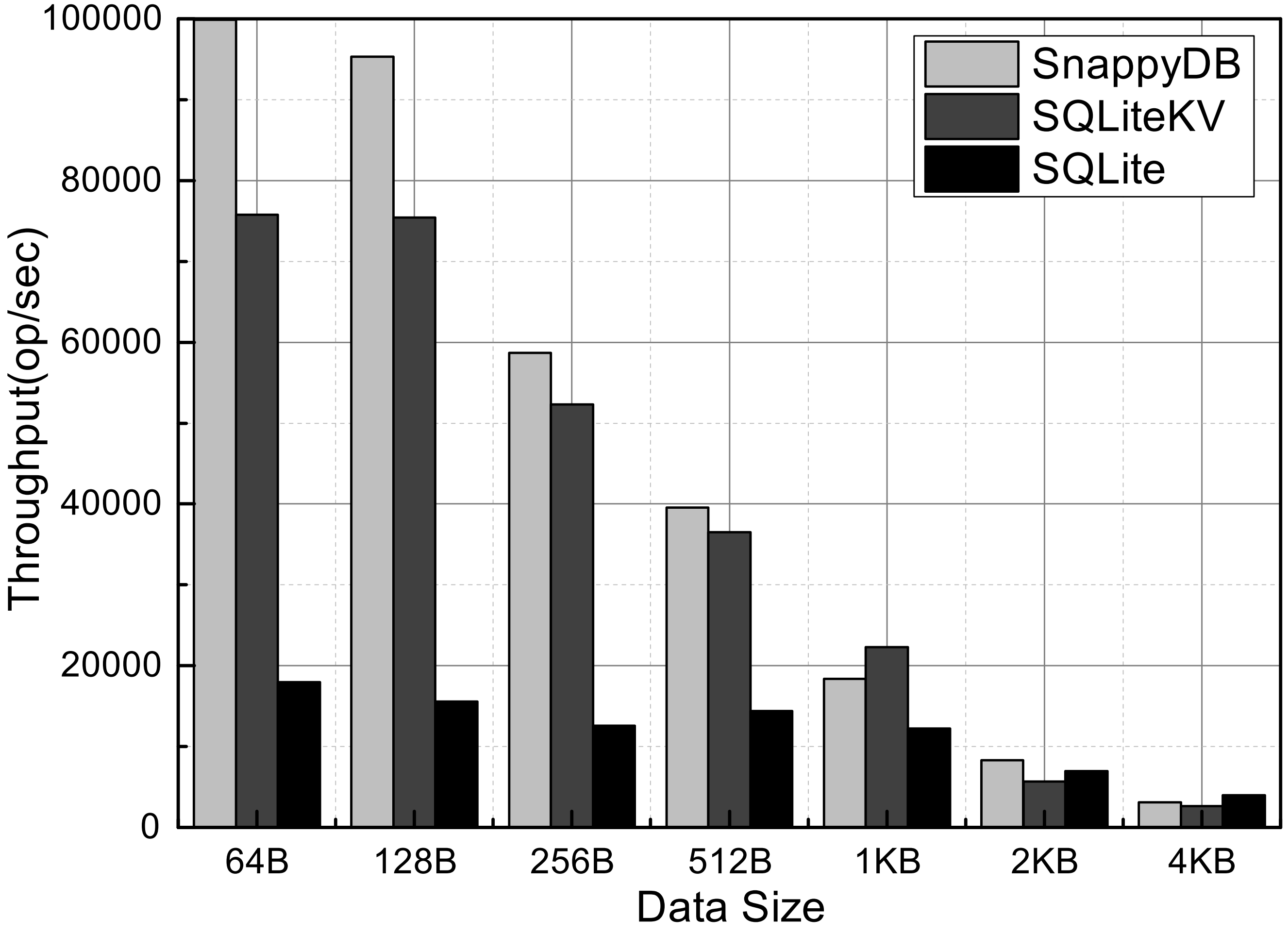
(a) Read Heavy: (1:0)



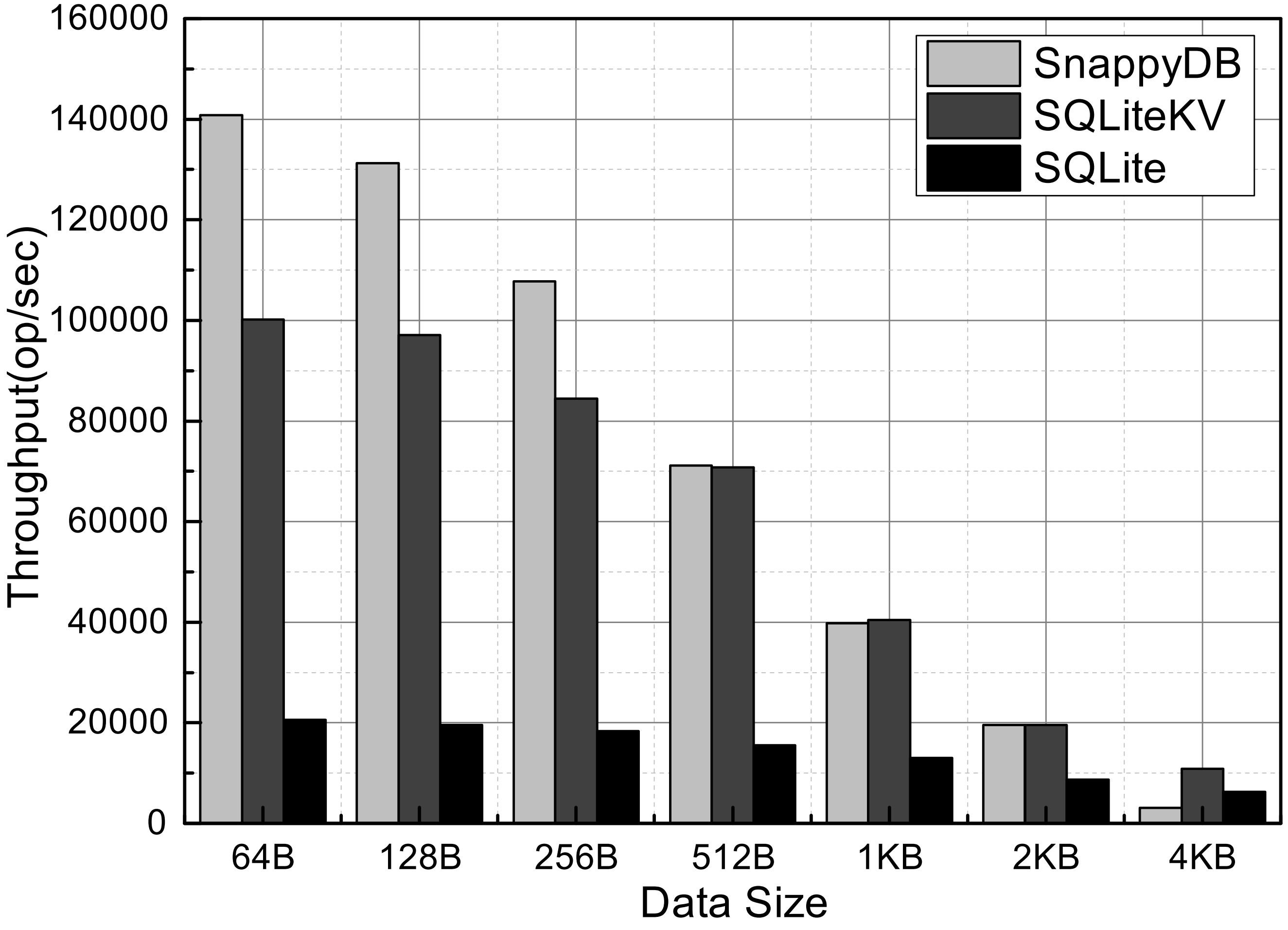
(b) Read Latest: (0.95:0.05)

Experimental Results

- Micro Performance



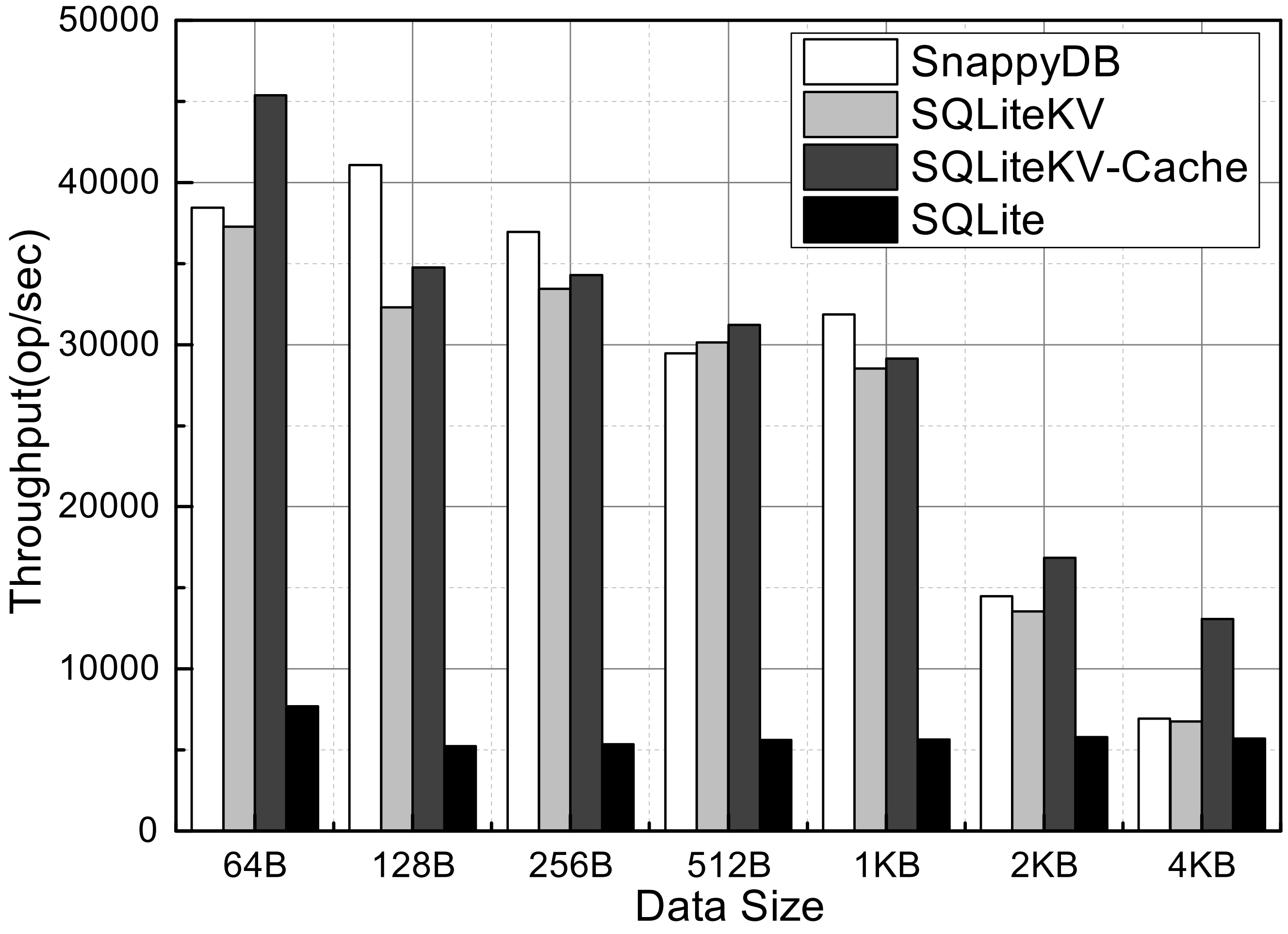
(a) Random Insertions



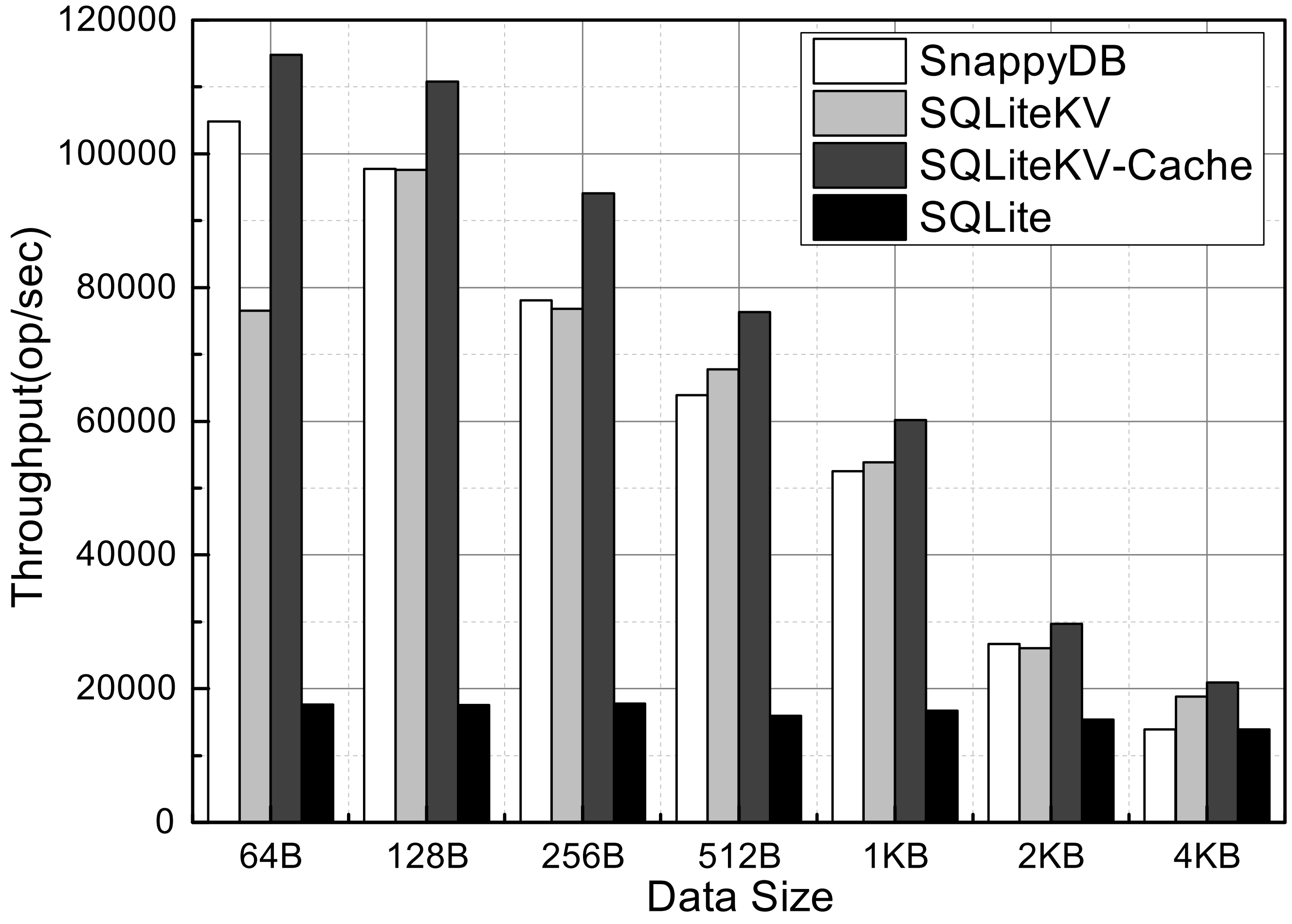
(b) Sequential Insertions

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:
    - **F**ront End: SQL interface & Slab-based Caching
    - **B**ack 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