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

Yuanjing Shi
Department of Computing
The Hong Kong Polytechnic University
Email: 13104584d@connect.polyu.hk

Zhaoyan Shen
Department of Computing
The Hong Kong Polytechnic University
Email: cszyshen@comp.polyu.edu.hk

Zili Shao
Department of Computing
The Hong Kong Polytechnic University
Email: cszlshao@comp.polyu.edu.hk

***Abstract*—SQLite has been deployed in millions of mobile devices from web to smartphone applications on various mobile operating systems. However, SQLite is not efficient with low transactions per second. In this paper, we for the first time propose a new SQLite-like database engine, called SQLiteKV, which adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. With its SQLite interface, SQLiteKV can be utilized by existing applications without any modification, while providing high performance with its LSM-tree-based data structure.**

In SQLiteKV, we develop a light-weight SQLite to key-value compiler to solve the semantic mismatch, so SQL statements can be efficiently translated into KV operations. We also design a novel coordination caching mechanism so query results can be effectively cached inside SQLiteKV by alleviating the discrepancy of data management between front-end SQLite statements and back-end key-value data organization. We have implemented and deployed SQLiteKV on a Google Nexus 6P smartphone. Experiments results show that SQLiteKV outperforms SQLite up to 6 times.

I. Introduction

SQLite is a server-less, transactional SQL database engine which has been widely deployed in mobile devices. Popular mobile applications such as messenger, email and social network services rely on SQLite for data management. However, due to the inefficient date organization and the poor coordination between its database engine and the underlying storage system [1, 2], SQLite suffers from poor transactional performance.

Many efforts have been done to optimize the performance of SQLite. These optimization approaches mainly fall into two categories: (1) Investigating I/O characteristics of different workloads of SQLite and mitigating its journaling over journal problem [1, 2]; (2) Utilizing emerging non-volatile memory technology, such as phase change memory, to eliminate small, random updates to storage devices [3, 4]. Though various mechanisms have been proposed, they all culminate with limited performance improvement. In this work, we for the first time propose to leverage the LSM-tree-based key-value data structure to improve SQLite performance.

Key-value database engine, which offers higher efficiency, scalability, availability, and usually works with simple NoSQL schema, is becoming more and more popular. To utilize the advantages of key-value database under SQL environments, Apache Phoenix [5] provides a SQL-like interface and translates SQL queries into a series of key-value scans in a NoSQL database HBase. Phoenix demonstrates outstanding performance in data cluster environment. However, it cannot be directly adopted by mobile devices as it is designed for scalable and distributed computing environments with large datasets.

There exist key-value databases on mobile device, such as SnappyDB [6]. However, they are not widely used in mobile devices for two major reasons. Firstly, nowadays, most mobile applications are built with SQL statements. Lacking of the SQL interface causes semantic mismatch between applications and key-value databases. Thus, mobile applications need to be redesigned to adopt key-value database, which incurs too much overhead. Secondly, current key-value databases require large memory footprints to maintain in-memory meta-data. Indexes for on disk data blocks need to be maintained in memory to serve key-value queries. Such meta-data management approach improves query performance with notable memory occupation. In most of cloud computing environments, this is not a critical issue. However, mobile devices are usually constrained with limited memory space [7].

To make mobile applications benefit from the efficient key-value database engine with SQL interfaces, in this paper, we propose a novel SQLite-like database engine, called SQLiteKV, which adopts the LSM-tree-based key-value data structure but retains the SQLite interfaces. SQLiteKV consists of two parts: (1) Front-end layer: A SQLite-to-KV compiler and a novel coordination caching mechanism. (2) Back-end layer: An LSM-tree-based key-value database engine with an effective meta-data management scheme.

In the front-end, the compiler receives SQL queries and translates them into the corresponding key-value operations. A caching mechanism is designed to alleviate the discrepancy of data organization between SQLite and key-value database. As for the back-end, we deploy an LSM-tree-based key-value database engine which transforms random writes to sequential writes by aggregating multiple updates in memory and dumping them to storage in a "batch" manner. To mitigate the memory requirement by our key-value engine, we propose to store exclusively the metadata for top levels of the LSM tree in memory and leave others on disk.

We have implemented and deployed SQLiteKV on a Google Nexus Android platform based on a key-value database-

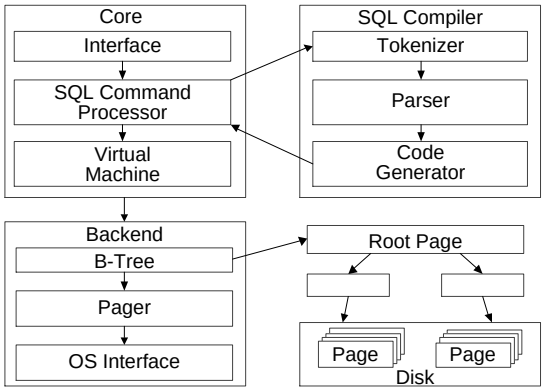


Fig. 1: Architecture of SQLite.

SnappyDB [6]. The experimental results with various workloads show that, on average, our SQLiteKV presents abut 6 times performance improvement compared with SQLite. To the best of our knowledge, this is the first work that optimizes SQLite by adopting the LSM-tree-based key-value database engine with SQLite interfaces for mobile devices.

The rest of paper is organized as follows. Section II presents some background information. Section III describes the design and implementation. Experimental results are presented in Section IV. Section V concludes the paper.

II. Background

This section introduces some background information about SQLite and the LSM-tree-based key-value database.

A. SQLite

SQLite is an in-process library, as well as an embedded SQL database widely used in mobile devices. Figure 1 gives the architecture of SQLite. SQLite exposes SQL interfaces to applications, and works by compiling SQL statement to bytecode, which then will be executed by a virtual machine. During the compiling of one SQL statement, the SQL command processor first send it to the tokenizer. The tokenizer breaks the SQL statement into tokens and passes those tokens to the parser. The parser assigns meaning to tokens based on their context, and assembles tokens into a parse tree. Thereafter, the code generator analyzes the parser tree and generates bytecode that performs the work of the SQL statement.

The data organization of SQLite is based on B-tree. A separate B-tree is used for each table and index in the database. The B-tree module requests data from the disk in fix-sized pages. The pages can be either data page, index page, free page or overflow page. All pages are of the same size and are comprised of multi-byte fields. The pager is responsible for reading, writing, and caching these pages. SQLite communicates with the underlying file system by system calls like `open`, `write` and `fsync`. Also, SQLite uses a journal mechanism for crush recovery, which makes database file and journal file synchronized frequently with the disk and leads to a performance degradation consequently.

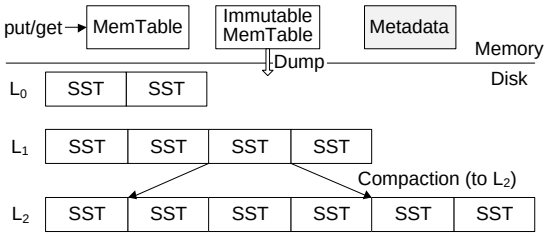


Fig. 2: Architecture of LSM-tree-based Database.

B. LSM-tree-based Key-Value Database

An LSM-tree-based key-value database maps a set of keys to the associated values. Applications access their data through simple `SET` and `GET` interfaces. Figure 2 describes the architecture of an LSM-tree-based key-value database implementation, which consists of two MemTables in main memory and a set of sorted SSTables (shown as SST) in the disk. To assist database query operations, metadata, including indexes, bloom filters, key-value ranges and sizes of these in-disk SSTs, are maintained in memory [8].

The LSM-tree-based key-value design is based on two optimizations: (1) New data must be quickly admitted into the store to support high-throughput writes. The database first uses an in-memory buffer, called MemTable, to receive incoming key-value items. Once a MemTable is full, it is transferred into a sorted immutable MemTable, and dumped to disk as an SSTable. Key-value items in one SSTable are sorted according to their keys. Key range and a bloom filter of each SSTable are maintained as metadata cached in memory space to assist key-value query operations. (2) Key-value items in the store are sorted to support fast data location. A multilevel tree-like structure is build to progressively sort key-value items in this architecture as shown in Figure 2.

The youngest level, *Level 0*, is generated by writing the immutable MemTable from memory to disk. Each level has a limitation on the maximum number of SSTables. In order to keep the stored data in an optimized layout, a compaction process will be conducted to merge overlapping key-value items to the next level when the total size of *Level l* exceeds its limitation.

C. Other SQL-Compatible Key-Value Databases

Apache Phoenix [5] is an open source relational database, in which a SQL query is compiled into a series of key-value operations for HBase, a distributed, key-value database. Phoenix provides well-defined and industry standard APIs for OLTP and operational analytics for Hadoop. Nevertheless, without a deep integration with the Hadoop framework, it is difficult for mobile devices to adopt either HBase as its storage engine or Phoenix for SQL-to-KV transitions. Besides, Phoenix, along with other Hadoop related modules, is designed for scalable and distributed computing environments with large datasets [9], which means they can hardly fit in mobile environments with limited resources [10].

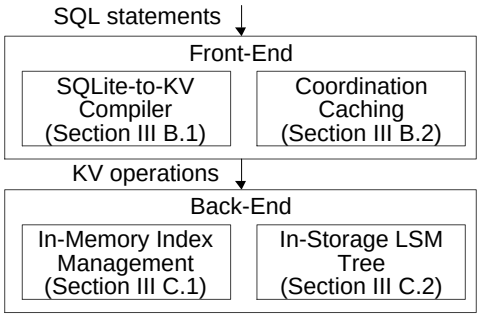


Fig. 3: Architecture of SQLiteKV.

In this paper, we propose an efficient LSM-tree-based lightweight database engine, SQLiteKV, which retains the SQLite interface for mobile devices, provides better performance compared with SQLite and adopts an efficient LSM-tree structure on its storage engine.

III. SQLiteKV: A SQLite-like Key Value Database

To make mobile applications benefit from the efficient key-value database engine, we propose SQLiteKV. In this section, we first present an overview of the SQLiteKV design, and then give the detailed descriptions for each of its module.

A. Design Overview

As shown in Figure 3, SQLiteKV is composed of two layers: the front-end layer and the back-end layer. The front-end layer deals with SQL statements. It is in charge of compiling the SQL statements and translating them to the corresponding key-value operations. The back-end layer serves key-value operations. Its responsibility is to maintain the management of key-value pairs on disk with the LSM-tree data structure, and serve the incoming key-value requests. The front-end layer mainly consists of two function modules: a SQLite-to-KV compiler and a coordination caching. As energy optimization is of vital importance in mobile devices and memory contributes to a large portion of total energy consumption of embedded devices [11], our back-end layer mainly focuses on the memory and storage optimization. The back-end layer includes a re-designed meta data management and an LSM-tree based storage engine.

The overall architecture and these functional modules are illustrated in Figure 3. In the following sections, we will introduce these function modules in detail.

B. Front-End

In order to provide a SQLite-compatible interface, two major components are designed and implemented on the front-end side of SQLiteKV: the SQLite-to-KV compiler (Figure 4), and the coordination caching (Figure 5).

1) SQLite-to-KV Compiler: The SQLite-to-KV compiler is used to translate a SQL statement to the corresponding key-value operations. When one SQL statement comes, our SQLite-to-KV compiler firstly breaks down the statements into several tokens. Then it will give each token meaning based on the context and assemble it into a parser tree. The working flow of the compiler is similar as SQLite. The noteworthy difference is that the SQLite-to-KV compiler would generate key-value operations based on the result of parsing instead of SQL bytecode. Generally, there are three kind of key-value operations including GET(), PUT() and DELETE(), which are the general interfaces used among NoSQL databases. Key-value operations will be passed to and executed in the back-end storage engine.

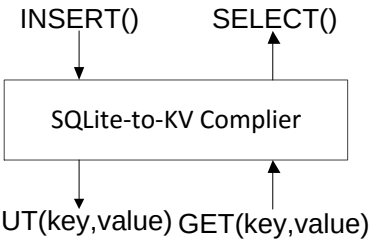


Fig. 4: SQLite to KV Compiler Translator.

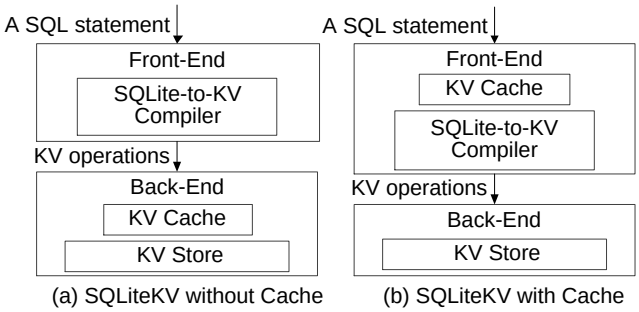


Fig. 5: SQLiteKV Coordination Caching Mechanism.

This SQLite-to-KV compiler makes it possible that existing applications could run smoothly with original SQL statements and leverage the potentials of key value storage.

2) Coordination Caching: SQLite uses a caching mechanism inside its back end engine to cache frequently-visited pages within the flash memory in order to avoid frequent visits to the disk. Key-value based databases also implement caching system for latest recently used key-value items. However, the cache data management of these two databases are different. Data cached in SQLite are organized to fit SQL statements while data cached in key-value databases are buffered as key-value pairs. In our proposed SQLiteKV, we support SQL interfaces while the back end database engine is key-value store. If SQLiteKV directly use the caching mechanism of the key-value store as shown in Figure 5(a), when an incoming SQL request comes, if the result is buffered in cache, we still need to go through the SQLite-to-KV compiler and re-organize the data format. This translation process could result in some overhead. To address this issue, we propose to move the back-end key-value cache to front-end, and maintain the data with SQL-friendly format.

As shown in Figure 5(b), in SQLiteKV, the coordination cache is maintained in the front-end. When an incoming SQL

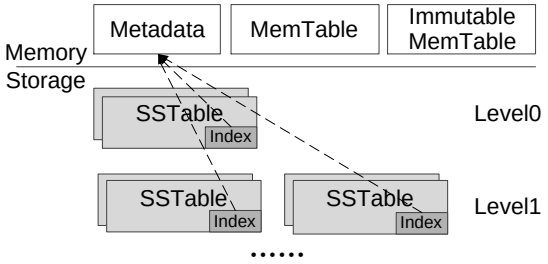


Fig. 6: Back-End In-Memory Index Management.

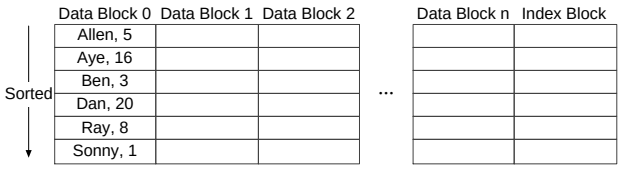


Fig. 7: Back-End Data Management (SSTable).

request comes, if the result is buffered in the cache, it can be directly returned.

C. Back-End

In SQLiteKV, we adopt an LSM-tree based key-value database engine, like Google's LevelDB, and Facebook's Cassandra. In this section, we will introduce the data and metadata management in this database storage system.

1) Index Management On LSM Tree: LSM-tree based key-value database store key-value items in separate SSTables on disk. Each SSTable maintain some indexes to assist the quick search of one key-value item. Usually, the indexes are stored at the end of each SSTable. To accelerate the query process, LSM-based storage engines would commonly scan over the entire disk and maintain a copy all indexes in memory [12]. Hence when a query operation is to be executed, the in-memory meta data is accesses quickly with the target key to locate the data block on disk. Thus, the data block is visited to get the key-value item. Generally, one disk seek is required for a single query on LSM-tree-based key-value database engine.

However, this approach is not practical nor efficient for mobile devices. Since most mobile devices are memory constraint and cannot accommodate all the indexes in memory. Considering this limitation, we re-design the indexing management approach, which exclusively stores indexes of data blocks from higher levels, like level 0 and 1, of the entire LSM tree. The reason is that as the level goes further down, data at lower level are less likely to be visited. In other words, the data on top levels are fresher and more likely to be visited, since nearly 90 percent request are served by level 0 and level 1 [13]. This approach helps reduce the memory requirement in our key-value database with minimum overhead.

2) Storage Management: Key-value pairs are firstly inserted into memory tables. Once a memory table becomes full, the key-value pairs in this memory table needs to be reorganized and flushed into disks as SSTable, as shown in Figure 7. Each SSTable consists of several data blocks and index blocks. All the key-value pairs in these data blocks are sorted. SSTable is organized in a layered approach. As shown in Figure 2, the number of SSTables increases exponentially with the layer increases. Once one layer is full, a compaction process is triggered to merge these key-value pairs to the next layer.

TABLE I: Workload Characteristics.

Workload(s)	Query	Insert
Update Heavy	0.5	0.5
Read Most	0.95	0.05
Read Heavy	1	0
Read Latest	0.05	0.95

IV. Evaluation

This section presents the evaluation results by comparing SQLiteKV with SQLite and SnappyDB, which are representative SQL and Key-value databases, respectively. In this section, we first introduce the experimental setup, and then provide the experimental results with real-world benchmarks and synthetic workloads [14].

A. Experiment Setup

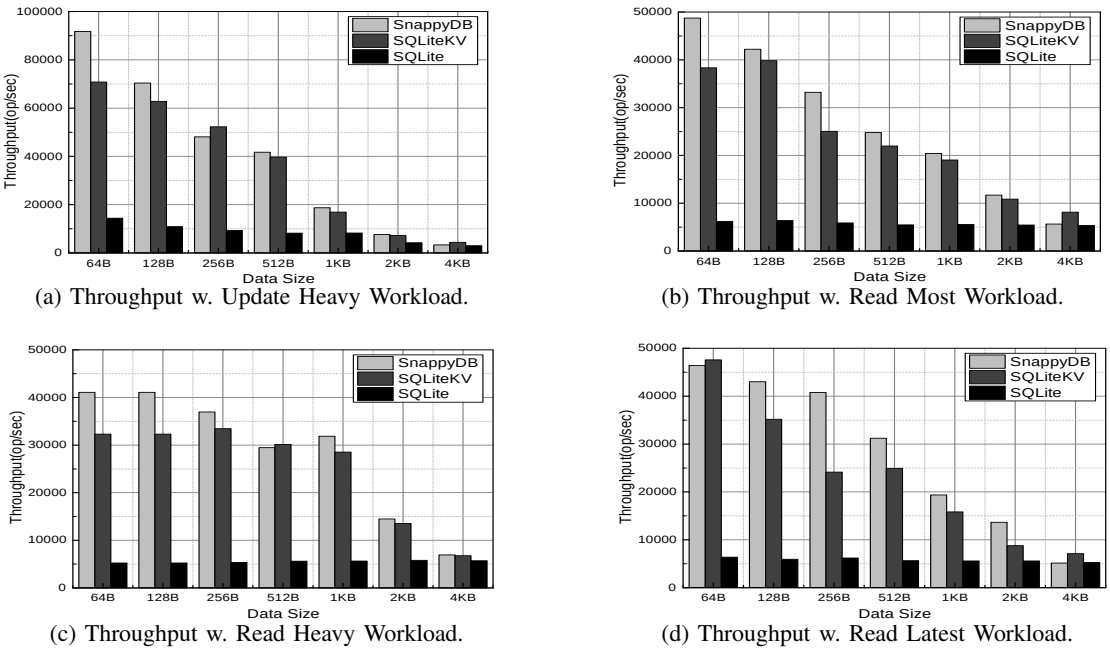
We have implemented a prototype of our proposed SQLiteKV on a Google mobile platform-Google Nexus 6p, which is equipped with a 2.0GHz oct-core 64 bit Qualcomm Snapdragon 810 processor, 3GB LPDDR4 RAM, and 32GB Samsung eMMC NAND Flash device. We use the Android 7.1 operating system with Linux Kernel 3.10.

Our prototype is developed based on SQLite and SnappyDB, and includes 2,344 lines of code. In the evaluation, SQLite 3.9 is utilized in the experiments as it is the current version in Google Nexus 6p. The page size of SQLite is set as 1024 bytes, which is the default value. SnappyDB 0.4.0, which is the latest version of a Java implementation of Google's LevelDB, is adopted.

Since in most of the real-world workloads, one SQLite query always carry more than one keyword. In our experiments, each SQL query in SQLite contains up to 999 keywords, which is the maximum value allowed. With this performance tuning method, we can make a fair comparison between SQLiteKV, SnappyDB and SQLite. Moreover, trivial calls, like moving cursors after queries in SQLite, are omitted for the sake of efficiency.

B. Overall Performance

To evaluate the overall performance of the databases, we first generate 100 thousand key value pairs to populate our databases, and then use the object popularity model to generate 100 thousand SQLite and KV requests. The object popularity, which determines the request sequence, follows a Zipfian distribution [15], by which some records in the head will be extremely popular while some in the tail are not. For the testing workloads, the Read Latest workload follows the Zipfian



(a) Throughput w. Update Heavy Workload.

(b) Throughput w. Read Most Workload.

(c) Throughput w. Read Heavy Workload.

(d) Throughput w. Read Latest Workload.

Fig. 8: Overall Performance

distribution except that most recently inserted records will be in the head and will be accessed more frequently. For all other workloads, record selections follow Zipfian distribution.

Figure 8 shows the experimental results by running SnappyDB, SQLiteKV, and SQLite with the four workloads in Table I. For each workload, the key-value item sizes vary from 64 bytes to 4096 bytes. It can be observed that when the key-value item size is less than 2048 bytes, compared with SQLite, SQLiteKV significantly increases the throughput (operations per second, ops/sec). For example, the throughput with the 64-byte key-value items is improved by over 6.1 times. At the same time, with the same configuration (64-byte key-value), SQLiteKV only introduces from 18.2% to 40.76% throughput degradation against SnappyDB. As most data sets in SQLite with mobile applications are dominated with small requests (i.e. less than 100 bytes [16]), we believe it is worthwhile for SQLiteKV to scarify this overhead by providing a SQLite interface.

When the key-value sizes are over 2048 bytes, SQLiteKV, as well as SnappyDB, only outperform SQLite slightly. The reason is that bigger data size can reduce the write amplification in SQLite significantly. Besides, for LSM-tree-based databases, keys and values are written at least twice: the first time for the transactional log and the second time for storing data to storage devices. Thus, the per-operation performance of SQLiteKV is degraded by extra write operations. Regardless of this degradation, since most data sets in mobile applications only contain very few large requests, SQLiteKV can still significantly outperform SQLite in practice.

To check the influence with the different insert/query ratios, we use the key-value items of 128-byte as the example. With the Update Heavy workload, for SQLiteKV, the throughput is 20% higher than that with the Read Heavy workload. The main reason is the random write operation of small data in SQLite will incur many amplification, thus reducing the system performance. However, SQLiteKV use the LSM-tree-based data structure, which transfers random write operations into sequential writes. Therefore, SQLiteKV benefits more with write-dominated workloads. The similar trends can be observed from other workloads.

In summary, SQLiteKV can significantly outperform SQLite by over 6 times in throughput for different workloads with various insert/query ratios when the sizes of requests are small (no more than 128 bytes).

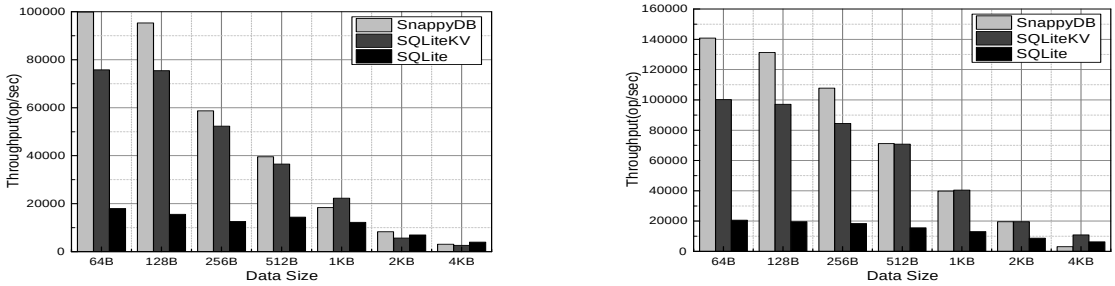
C. Performance with Sequential/Random Workloads

In the second set of experiments, we investigate the impact of random and sequential accesses on SQLiteKV, SQLite and SnappyDB. In this experiment, we set the key size as 16 bytes and the value sizes vary from 64 bytes to 4096 bytes. We test the performance for both insert and query operations.

1) Insertion Performance: Figure 9b shows the performances with random and sequential insertion operations, respectively. The key-value sizes vary from 64 bytes to 4096 bytes. For the sequential access workload, the key space is in ascending order, while it is randomly traversed for the random case.

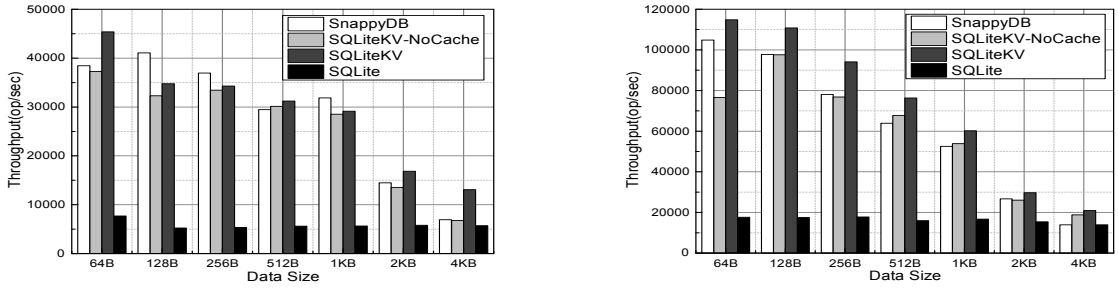
It can be observed that for SQLitKV, the insertion performance with sequential access workload is better than that with random access (the average improvement is 40%). SnappyDB shows the similar trend to SQLiteKV. On the contrary, for SQLite, there are no differences between the random and sequential cases, as its records are organized via B-Tree indexes.

2) Query Performance: Figure 10b shows the performances with random and sequential query operations, respec-



(a) Random Insertions. (b) Sequential Insertions.

Fig. 9: Insertion Throughput vs. KV size



(a) Random Queries. (b) Sequential Queries.

Fig. 10: Query Throughput vs. KV size

tively. The similar trends can be observed as with insertion operations. Specially, for SQLitKV, its query performance with sequential records is much better than that with random records (the average improvement is around 2 times).

Figure 10b shows the effect of our read cache in SQLiteKV. Comparing the performance of SQLiteKV with and without caching, we can conclude that our coordination caching mechanism can help improve 10 - 20 % query performance for small data sizes. Furthermore, it can provide a noteworthy improvement, up to 2.x times, on large data sizes.

V. Conclusion

In this paper, we propose a new database engine for mobile devices, called SQLiteKV, which is a SQLite-like key-value database engine. SQLiteKV adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. SQLiteKV consists of two parts: a front end that contains a light-weight SQLite-to-key-value compiler and a coordination caching mechanism; a back end that adopts a LSM-tree-based key-value database engine. We have implemented and deployed our SQLiteKV on a Google Nexus 6P Android platform based on a key-value database SnappyDB. Experimental results with various workloads show that the proposed SQLiteKV outperforms SQLite significantly.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This work is partially supported by National Natural Science Foundation of China (Project 61373049), Research Grants Council of Hong Kong (GRF 152736/16E and GRF 15222315/15E), Hong Kong Polytechnic University (4-BCBB).

References

[1] K. Shen, S. Park *et al.*, "Journaling of journal is (almost) free." in *FAST*, 2014.
[2] S. Jeong *et al.*, "I/O stack optimization for smartphones." in *ATC*, 2013.
[3] G. Oh *et al.*, "SQLite optimization with phase change memory for mobile applications," *VLDB*, 2015.
[4] W. Kim *et al.*, "NVWAL: exploiting NVRAM in write-ahead logging," 2016.
[5] "Apache phoenix," https://phoenix.apache.org/.
[6] "SnappyDB: a key-value database for Android," http://www.snappydb.com/.
[7] H. Lee *et al.*, "Energy-aware memory allocation in heterogeneous non-volatile memory systems," in *ISLPED*, 2003.
[8] R. Sears *et al.*, "bLSM: a general purpose log structured merge tree," in *SIGMO/PODS*, 2012.
[9] G. Forman *et al.*, "The challenges of mobile computing," *Computer*, 1994.
[10] S. Sinha *et al.*, "Low-power FPGA design using memoization-based approximate computing," *VLSI*, 2016.
[11] 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.
[12] X. Wu *et al.*, "LSM-trie: an LSM-tree-based ultra-large key-value store for small data items," in *USENIX ATC*, 2015.
[13] D. Wei and S. Ryan, "The missing manual for leveled compaction strategy," goo.gl/En73gW, 2016.
[14] B. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.
[15] Z. Shen *et al.*, "DIDACache: a deep integration of device and application for flash based key-value caching." in *FAST*, 2017.
[16] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.