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

Zhaoyan Shen, Yuanjing Shi, Zili Shao, Yong Guan

Abstract—SQLite has been deployed in millions of mobile devices from web to smartphone applications on various mobile operating systems. However, due to the uncoordinated IO interactions with the underlying file system (e.g., ext4), 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.

We separate SQLiteKV into the front-end and back-end. In the front-end, we develop a light-weight SQLite-to-KV compiler to solve the semantic mismatch, so SQL statements can be efficiently translated into KV operations. We also design a novel coordination-caching mechanism with memory defragmentation so query results can be effectively cached inside SQLiteKV by alleviating the discrepancy of data management between front-end SQLite statements and back-end data organization. In the back-end, we adopt an LSM-tree-based key-value database engine, and propose a lightweight metadata management scheme to mitigate the memory requirement. We have implemented and deployed SQLiteKV on a Google Nexus 6P smartphone. Experiments results with various workloads show that SQLiteKV outperforms SQLite up to 6 times.

Index Terms—Mobile device, SQLite, LSM-tree, Key-value, Database engine.

I. INTRODUCTION

Smart mobile devices, e.g., smartphones, phablets, tablets, and smartTVs, are becoming prevalent. SQLite [1], which is a server-less, transactional SQL database engine, is of vital importance and has been widely deployed in these mobile devices [2–4]. 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, SQLite suffers from poor transactional performance [5–10].

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 [5–7, 10, 11]. Lee et al. [7] points out that the excessive IO activities caused by uncoordinated interactions between EXT4 journaling and SQLite journaling are one of the main sources that incur inefficiency to mobile devices. Jeong et al. [5] integrate external journaling to eliminate unnecessary file system metadata journaling in the SQLite environment. Shen et al. [6] optimize SQLite transactions by adaptively allowing database files to have their custom file system journaling mode. (2) Utilizing emerging non-volatile memory technology, such as phase change memory (PCM), to eliminate small, random updates to storage devices [12–15]. Oh et al. [12] optimize SQLite by persisting small insertions or updates in SQLite with the non-volatile, byte-addressable PCM. Kim et al. [13] develop NVWAL (NVRAM Write-Ahead Logging) for SQLite to exploit byte addressable NVRAM to maintain the write-ahead log and to guarantee the failure atomicity and the durability of a database transaction. 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 [16–19]. Key-value databases have simple interfaces (such as Put() and Get()) and are more efficient than the traditional relational SQL databases in cloud environments [20–22]. To utilize the advantages of key-value database under SQL environments, Apache Phoenix [23] provides an SQL-like interface which translates SQL statements into a series of key-value operations in a NoSQL database HBase. Phoenix demonstrates outstanding performance in data cluster environment. However, the approach cannot be directly adopted by resource limited mobile devices as targeting at scalable and distributed computing environments with large datasets [24, 25].

There exist key-value databases on mobile device, such as SnappyDB [26], UnQlite [27], and Leveledown-Mobile [28]. 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 database engine. Thus, mobile applications need to be redesigned to adopt key-value databases, which incurs too much overhead. Secondly, current key-value databases require large memory footprints to maintain in-memory metadata [29–31]. Such an in-memory metadata management approach introduces the overhead of notable memory occupation. In most
of cloud computing environments, this is not a critical issue. However, for mobile devices with constrained memory space, this is nontrivial [32].

To make mobile applications benefit from the efficient key-value database engine, 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 layers: (1) A front-end layer that includes an SQLite-to-KV compiler and a novel coordination caching mechanism. (2) A back-end layer that includes an LSM-tree-based key-value database engine with an effective metadata management scheme.

In the front-end, the SQLite-to-KV compiler receives SQL statements and translates them into the corresponding key-value operations. A read cache mechanism is designed to alleviate the discrepancy of data organization between SQLite and the key-value database. Considering the memory constraints issue in mobile devices, we manage the caching with a slab-based approach to eliminate memory fragmentation. Cache space is firstly segmented into slabs while each slab is further striped into an array of slots with equal size. One query result is buffered into the slab whose slot size is of the best fit with its own size [33–35].

For the back-end, we deploy an LSM-tree-based key-value database engine. The LSM-tree-based key-value database engine transforms random writes to sequential writes by aggregating multiple updates in memory and dumping them to storage in a “batch” manner, which makes it I/O efficient. Besides, it stores all key-value items in a sorted order, thus improving its query performance. To deal with the limited memory and energy resources for mobile devices [36], we further propose to store exclusively the metadata for the top levels of the LSM tree in memory and leave others on storage to mitigate the memory requirement.

We have implemented and deployed the proposed SQLiteKV on a Google Nexus Android platform based on a key-value database-SnappyDB [26]. The experimental results with various workloads show that, our SQLiteKV presents up to 6 times performance improvement compared with SQLite. Our contributions are concluded as follows:

- We for the first time propose to improve the performance of SQLite by adopting the LSM-tree-based key-value database engine while remaining the SQLite interfaces for mobile devices.
- We design a slab-based coordination caching scheme to solve the semantic mismatch between the SQL interfaces and the key-value database engine, which also effectively improves the system performance.
- To mitigate the memory requirement for mobile devices, we have re-designed the index management policy for the LSM-tree-based key-value database engine.
- We have implemented and deployed a prototype of SQLiteKV with a real Google Android platform, and the evaluation results show the effective of our proposed design.

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

II. BACKGROUND

This section introduces some background information about SQLite, the LSM-tree-based key-value database, and other SQL-compatible key-value databases.

A. SQLite

SQLite is an in-process library, as well as an embedded SQL database widely used in mobile devices [1, 37]. Figure 1 gives the architecture of SQLite. SQLite exposes SQL interfaces to applications, and works by compiling SQL statements to bytecode, which then will be executed by a virtual machine. During the compiling of one SQL statement, the SQL command processor first sends it to the tokenizer. The tokenizer breaks the SQL statement into tokens and passes those tokens to the parser. The parser assigns meaning to each token based on its context, and assembles the tokens into a parse tree. Thereafter, the code generator analyzes the parser tree and generates virtual machine code that performs the work of the SQL statement. The virtual machine will run the generated virtual machine code with different operations files.

The data organization of SQLite is based on B-tree. One separate B-tree is used for each table in the database. The B-tree indexes 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. Moreover, SQLite uses a journal mechanism for crush recovery, which makes the database file and journal file [38, 39] synchronized frequently with the disk and leads to a performance degradation consequently.

B. LSM-tree-based Key-Value Database

An LSM-tree-based key-value database maps a set of keys to associated values [16, 17, 40]. Applications access their
data through simple `Put()` and `Get()` interfaces, that are the most generally used in NoSQL database [41–43]. Figure 2 presents the architecture of an LSM-tree-based key-value storage engine, 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 on-disk SSTables, are maintained in memory [21, 35].

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 first 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 ranges 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 localization. A multilevel tree-like structure is built to progressively sort key-value items in this architecture as shown in Figure 2.

The youngest level, Level0, 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 exceeds its limitation.

**C. Other SQL-Compatible Key-Value Databases**

Apache Phoenix [23] is an open source relational database, in which an SQL statement is compiled into a series of key-value operations for HBase [44], a distributed, key-value database. Phoenix provides well-defined and industry standard APIs for OLTP and operational analytics for Hadoop [45, 46]. 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 [47], which means they can hardly fit in mobile environments with limited resources [48].

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. Motivation**

To compare the performance of SQLite and the key-value based database engine, we choose one lightweight LSM-tree-based key-value database, called SnappyDB [26], and measure the throughput (operation per second, ops/sec) by running them with a Google Nexus smartphone. We use the Zipfian distribution [49] to generate the request popularity and request sizes are varied from 64 bytes to 4096 bytes. Figure 3 presents the throughput for both insert and query operations of these two databases, respectively.

Figure 3(a) shows the throughput for insert operations with SQLite and SnappyDB over vary-sized requests. It is obvious that SnappyDB outperforms SQLite significantly across the board. For instance, with the request size of 64 bytes, SnappyDB outperforms SQLite 7.3 times. The reason is mainly two folds. First, SQLite is a relational database which has strict data organization schema. All insert requests have to strictly follow the data organization schema and the slow transaction process of SQLite. The second reason is the journaling of journal problem. In SQLite, an insert transaction firstly logs the insertion at an individual SQLite journal, and then inserts the record to the actual database table. At the end of each phase, SQLite calls fsync() to persist the results. Each fsync() call makes the underlying file system (e.g., ext4) update the database file and write the new metadata to the file system journal. Hence, a single insert operation may result in up to 9 I/Os to the storage device, and each I/O is done with the 4KB unit. This uncoordinated I/O interaction brings much write amplification and sacrifices the performance of SQLite a lot. On the other hand, SnappyDB maintains a shared log, and an insert operation is firstly logged in the log file, and then served by its memory table as shown in Figure 2. This process is much simple and incurs less I/Os to the storage device compared with SQLite.

It can also be observed from Figure 3(a) that with the request size increasing, the performance improvement of SnappyDB over SQLite decreases. This is because when the function fsync() is called, the underlying file system will do write operation with the 4KB unit. When the request size increasing close to 4KB, the write amplification overhead decreases. When the request size is of 4KB, the throughput of SQLite and SnappyDB are almost the same. When the record size is larger than 4KB, the performances of SQLite and SnappyDB show the same trend and are basically the same. So, we omit the results for records with the sizes larger than 4KB in Figure 3. Figure 3(b) gives the throughput for query operations of SQLite and SnappyDB, and it shows the same trend with insert operations.

We can conclude that the efficient LSM-tree-based key-value database engine outperforms the traditional relational database SQLite significantly, and directly deploying applications based on the LSM-tree-based key-value database definitely improves applications’ performance. However, most applications running on mobile devices depend on the SQL
interface to access databases. Since the key-value database engine does not support SQL statements, most mobile applications cannot directly benefit from its high performance. Redesigning these mobile applications to support the key-value interface will bring too much development overhead. To address this issue, in this paper, we propose a new database engine, called SQLiteKV, which retains the SQLite interface for mobile devices and adopts an efficient LSM-tree-based key-value data structure on its storage engine.

IV. SQLiteKV: An SQLite-like Key Value Database

To make mobile applications benefit from the efficient key-value database engine, we propose SQLiteKV, which not only inherits the high performance of key-value database engines, but also provides the application compatible SQLite interfaces. In this section, we first present an overview of the SQLiteKV design, and then give the detailed descriptions for each of its modules.

A. Design Overview

Figure 4 presents the architecture of SQLiteKV. Similar to SQLite, SQLiteKV is composed of a front-end statement parser layer and a back-end data storage management layer. SQLiteKV’s front-end layer mainly consists of two function modules: an SQLite-to-KV processor and a coordination read cache. Instead of translating SQL statements into virtual machine code in SQLite, the front-end of SQLiteKV parses the SQL statements into the corresponding key-value operations (e.g., \texttt{Put}, \texttt{Get}). The coordination read cache is used to buffer and quickly serve hot query requests. To reduce memory fragmentation, we adopt a slab-based way to manage the cache space. The SQLiteKV back-end layer is used to maintain the key-value pairs on disk with the LSM-tree-based data structure, and serve the parsed key-value requests. It also includes two function modules: a redesigned in-memory index management module which is used to save memory space for mobile devices, and an LSM-tree-based storage engine.

With SQLiteKV, when an SQL query statement comes, it will first search the coordination read cache, if the request data states in cache, the query will be directly returned. Otherwise, the SQL query statement will be translated into its corresponding key-value \texttt{Get} operations through the SQLite-to-KV compiler. At last, the \texttt{Get} operation will be served by the back-end key-value database engine. In the following sections, we will introduce these four function modules in detail.

B. Front-End Layer

The SQLiteKV front-end layer includes two major components: an SQLite-to-KV compiler (Figure 5) which provides the compatible SQLite interface, and a coordination read cache (Figure 6) which accelerates the query process.
1) SQLite-to-KV Compiler: As shown in Figure 5, the function of the SQLite-to-KV compiler is to translate an SQL statement to the corresponding key-value operations. It provides users SQLite-compatible interface while storing/retrieving key-value pairs with put and set operations in the back-end database. When an SQL statement comes, the SQLite-to-KV compiler firstly breaks down the statement into several tokens. Then it will assign each token a meaning based on the context and assemble it into a parser tree. The parsing process of the compiler is similar to SQLite. The noteworthy difference is that the SQLite-to-KV compiler generates key-value operations based on the result of the parse tree instead of SQL bytecode that must be run with a virtual machine. The reason why SQLite is utilizing such a virtual-machine layer between SQL statements and execution is that, query processing is becoming increasingly CPU-bound. With a virtual-machine layer, SQLite can provide portability with a relatively reasonable performance. However, the circumstances have changed on mobile devices, since it is obvious that a typical mobile platform, like Android, is still resource-limited and the utilization of such a VM layer does not prove to be as efficient as it is on mainstream big-data platforms [50]. In our SQLiteKV, we use the SQLite-to-KV compiler to transform SQL operations to key-value operations, and the back-end LSM-tree-based key-value database provides direct execution of key-value operations, which make it more efficient. Basically, the SQLite-to-KV compiler translates the most commonly used SQLite interfaces Insert() and Select() to the corresponding key-value Put() and Set() operations. Since we adopt an LSM-tree-based key-value database engine that does not support in-place updates, the SQLite interface delete() is transferred into one key-value Put() operations with an invalid value (e.g., “NULL” in Figure 5). To perform these interface translation, we build a mapping between records in the relational database and key-value items in the key-value database. In our experiments, we found that for mobile applications, the data schema of SQLite database is very simple. Many database tables only include two columns. Thus, when transforming a relational record to one key-value item, we make the primary key of this record as the key for the key-value item and make the other columns as the value of this item. This mapping is done by the SQLite-to-KV compiler, and there is no need for any additional mapping data structure.

Algorithm IV.1 Insert operation in SQLiteKV.
Input: 
1: insert values(Primary_key, column1)
Output: Perform key-value put operation
2: SQLiteKV.getWriteableDatabase();
3: //open the database for write
4: SQLiteKV.beginTransaction();
5: Container.put(Primary_key);
6: //construct the key for KV pair
7: container.put(column1);
8: //construct the value for KV pair
9: SQLiteKV.put(container.key, container.value);
10: //put the KV item in the container to the KV database
11: SQLiteKV.endTransaction();
12: return;

Algorithm IV.2 Query operation in SQLiteKV.
Input: 
1: select from test where column = values
Output: Perform key-value get operation
2: SQLiteKV.getWriteableDatabase();
3: select_token = column+"=?";
4: arg_token = values;
5: hash_sql = hash(select_token, arg_token);
6: result = SQLiteKV.Cache.get(hash_sql)
7: // first access cache
8: if result == NULL then // not in cache
9: result = SQLiteKV.get(arg_token);
10: // get KV pairs from KV database
11: SQLiteKV.Cache.put(hash_sql, result);
12: // buffer the result in cache
13: end if
14: return result;

value part (NULL) in SQLiteKV as shown in Algorithm IV.3.

Algorithm IV.2 describes the working process for a query operation in SQLiteKV. Similar to insert operation, SQLiteKV firstly parse the SQL statement into tokens (lines 3-4). Then it calculates a hash value based on these tokens, and search the cache with the hashed value. With cache hits, the data is directly returned from the cache. Otherwise, a corresponding key-value get() operation is issued to the key-value database engine to retrieve the value to users and store it in the cache.

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

2) Coordination Caching: Caching mechanism is of vital importance for improving the query efficiency of databases. Through buffering part of hot data in memory, query operations can be served fast without accessing the low-latency disk. In SQLiteKV, there are two cache configuration choices. As shown in Figure 6, the first choice is to maintain the cache in the back-end key-value database engine, and the second one is to put the cache in the front-end and before the SQLite-to-KV compiler module.

In SQLiteKV, we propose to adopt the second configuration
Algorithm IV.3 Delete operation in SQLiteKV.

Input:
1: delete from test where column = values

Output: Perform key-value delete operation
2: SQLiteKV.getWriteableDatabase();
3: delete_token = column+"="+values;
4: arg_token = values;
5: hash_sql = hash(delete_token, arg_token);
6: result = SQLiteKV.Cache.get(hash_sql)
7: // first access cache
8: if result != NULL then // clean cache
9: SQLiteKV.Cache.delete(hash_sql);
10: end if
11: SQLiteKV.put(arg_token, NULL);
12: return;

An SQL statement

Fig. 6: SQLiteKV Coordination Caching Mechanism.

as shown in Figure 6(b). The reason is that in Figure 6(a), the KV cache module stays in the back-end key-value database engine. Hot data buffered by this cache are maintained in the format of key-value pairs. When an SQL statement comes, SQLiteKV firstly analyzes its tokens, gets the hash token, and then return to the users. Besides, in this configuration, whether cache hits or not, an incoming SQL statement always needs to go through the SQLite-to-KV compiler, which incurs reasonable overhead. In the second configuration, the cache stays in the front-end layer, and the hot data are maintained in an SQL statement-oriented approach. When an SQL statement comes, if cache hits, the results can be directly returned without performing the SQLite-to-KV translation.

To further utilize the memory space efficiently, we adopt a slab-based memory allocation scheme as shown in Figure 7.

Fig. 7: Slab-based cache management.

array of equal-sized slots. Each slot stores one request data. Slabs are logically organized into different slab classes based on the slot sizes (e.g., 32B, 64B, 128B, ..., 4KB, ...). The data for one SQL query is stored into a class whose slot size is the best fit for its size. For example, if a key-value pair with size 6KB needs to be buffered in the cache, one slab will be allocated to the class with the slot size that is equal to 8KB (the best fit for the 6KB key-value item), and a slot will be used to buffer this key-value pair. A hash mapping table is used to record the position of each SQL statement. The key for the hash table is calculated by hashing tokens of each SQL query request as shown in Algorithm IV.2. When one query comes, SQLiteKV firstly analyzes its tokens, gets the hash value, and then read the data by combining the slab “sid” with offset “offset”. Such a design can effectively address the issue of memory fragmentation, and utilizes the limited embedded memory resource more properly.

C. Back-End Layer

In SQLiteKV, we adopt an LSM-tree-based key-value database engine, like Google’s LevelDB [16], and Facebook’s Cassandra [17]. In this section, we will introduce our proposed new metadata management scheme, and the data storage in this database engine.

1) Data Storage Management: Figure 8 shows the data storage engine for the back-end LSM-tree-based key-value store. As described in Section II, LSM-tree-based data store aggregates key-value items into equal-sized tables. There are three kinds of tables in the key-value database engine: memory table (MemTable), immutable table (ImmuteTable), and on-disk SSTable (SST). Memtable and ImmuteTable are maintained in memory, and SSTables are stored on disk. The SSTables are maintained in several levels (e.g., Level0, Level1, Level2). Each level contains different numbers of SSTables, and its capacity grows exponentially. Log, Manifest, and Current, are three configuration files used to assist the working process of the database engine.

Fig. 8: Back-End in-memory index management.
A SQL statement can be promised. In this work, we do not optimize the write operation to the disk. So, all the key-value items stored in MemTable also have consistency, and then buffered into the ImmutaTable, and the data durability can be promised. In this work, we do not optimize the Log file design, which can be a good research direction for future work. Once the MemTable becomes full, the key-value items in this table will be sorted and stored in the ImmutaTable. A minor compaction process will flush the key-value items in ImmutaTable to one disk SSTable in Level0. Key-value pairs stored in SSTables are sorted with their keys as shown in Figure 9. Each SSTable consists of several data blocks and index blocks. The index blocks maintain the mapping of the key range to the data blocks. With more SSTables flushed, if one level runs out of its space, a major compaction will be triggered to select one of its SSTable and do the merge sort operation with several SSTables in the next level (As shown in Figure 8, one SSTable in Level1 is compacted with 4 SSTables in Level2).

2) Index Management: As described, each SSTable maintains some indexes to assist the quick search of key-value items. Usually, the indexes are stored at the end of each SSTable. To accelerate the query process, LSM-tree-based storage engines commonly scan over the entire disk and maintain a copy of all indexes in memory [51]. Hence when a query operation is to be executed, the in-memory meta data is accessed 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 engines.

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 redesign the indexing management approach, which exclusively stores indexes of data blocks from higher levels, like Level0 and Level1, of the entire LSM tree. The reason is that as the level goes further down, data at lower levels 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 Level0 and Level1 [20]. This approach helps reduce the memory requirement in our key-value database with minimum overhead.

V. Evaluation

We have prototyped the proposed efficient LSM-tree-based SQLite-like database engine - SQLiteKV, on a Google mobile platform. Our implementation of the database engine is based on SnappyDB [26], which is a representative key-value database for mobile devices. Our SQLiteKV totally includes 2,506 lines in Java. In this section, we will first introduce the basic experimental setup, and then provide the experimental results with real-world benchmarks [49] and synthetic workloads.

A. Experiment Setup

The prototype of our proposed SQLiteKV is implemented 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 8.0 operating system with Linux Kernel 3.10. In the evaluation, SQLite 3.18 is utilized in the experiments as it is the current version in Android 8.0 Oreo. The page size of SQLite is set as 1024 bytes, which is the default value. SnappyDB 0.5.2, which is the latest version of a Java implementation of Google’s LevelDB, is adopted. The performance of the LSM-tree-based database engine is evaluated with the indexes of all the levels of the entire LSM tree buffered in memory.

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

B. Basic Performance

We first evaluate the basic performance of SQLiteKV and SQLite, mainly in terms of throughput (operations per second, ops/sec). In this experiment, we set the data size vary from 64 bytes to 4096 bytes, and investigate the throughputs of both random and sequential accesses. We test and compare the throughput of SQLiteKV and SQLite with the commonly used insert, query, and delete operations.

1) Insertion Performance: Figure 10a and Figure 10b show the performances of SQLiteKV and SQLite with random and sequential insertion operations, respectively. The request sizes vary from 64 bytes to 4096 bytes. For the sequential access workload, the requests are in ascending order, while the requests are randomly traversed for the random case.

It can be observed that the performances of SQLiteKV outperform SQLite significantly for request of all sizes. For sequential operations, with the request size of 64 bytes, the throughput of SQLiteKV is 1.41 × 10^5, which is 6.1 times higher than that of SQLite. For random operations, the SQLiteKV outperforms SQLite 3.8 times in maximum with request size of 64 bytes. As the request size increases, the throughputs decrease across the board for both SQLite and SQLiteKV. The performance improvement of SQLiteKV over SQLite also decreases with the request size increasing. This is because the write amplification effect caused by SQLite’s journal of journal problem declines as the request size increases. Besides, it can be observed that for SQLiteKV, the insertion performance with sequential access workloads are better than that with random access ones (the average
improvement is 40%). On the contrary, for SQLite, there are basically no differences between the random and sequential cases.

2) Query Performance: Figure 11b shows the performance of SQLiteKV and SQLite with random and sequential query operations, respectively. Basically, query operations show the same trend with insert operations. For sequential queries, with the request size of 64 bytes, the throughput of SQLiteKV is $1.01 	imes 10^5$, which is 4.91 times higher than that of SQLite. For random queries, the performance improvement of SQLiteKV over SQLite is up to 5.4 times with the request size of 128 bytes. Furthermore, the sequential query throughput of SQLiteKV is much higher than the random query one, which is about 1.3 times.

3) Delete Performance: Figure 12 presents the throughput of delete operations for SQLiteKV and SQLite, respectively. It is obvious that the throughput for delete operations of SQLiteKV is much higher than that of SQLite. For instance, with the request size of 4KB, it even takes more than several minutes to delete a record for SQLite. However, in SQLiteKV, the delete operation is implemented by the key-value Put operation with invalid data area. Thus, the delete operations
of SQLiteKV perform the same as insert operations.

We further test the random insert, random query, sequential insert, and sequential query performance of SQLiteKV and SQLite with request sizes that follow the Zipfian [33, 49, 52] distribution. The results are given in Figure 13. We can conclude that SQLiteKV outperforms SQLite for all cases. Especially, for sequential write operations, the improvement is up to 5.3 times.

C. Overall Performance

<table>
<thead>
<tr>
<th>Workload(s)</th>
<th>Query</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update 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>

To evaluate the overall performance, we further test the proposed SQLiteKV with a set of YCSB [49, 50] core workloads that define a basic benchmark as shown in Table I. Here, the update heavy workload has a mix of 50/50 reads and writes, the read most workload has a 95/5 reads/write mix, the read heavy workload is 100% read, and in the read latest workload, new records are inserted, and the most recently inserted records are the most popular.

To conduct the experiments, we first generate 100 thousand key value pairs to populate SQLiteKV and SQLite, and then use the object popularity model to generate 100 thousand requests sequence. The object popularity, which determines the requests sequence, follows the Zipfian distribution [33, 49, 52], by which records in the head will be extremely popular while those in the tail are not. For the read latest workload, we make most recently inserted records in the head that will be accessed more frequently. For the request size, similarly, we use both the fixed request sizes from 64 bytes to 4KB and the request sizes which follows the Zipfian distribution.

Figure 14 shows the experimental results by running SQLiteKV and SQLite with the four workloads in Table I. For each workload, the request size varies from 64 bytes to 4096 bytes. It can be observed that, compared with SQLite, SQLiteKV significantly increases the throughput across the board with varied request sizes. For the update heavy workload, the throughput of SQLiteKV is 3.9 times higher than that of SQLite on average. With the request size of 256 bytes, the performance improvement achieves the highest point, which is about 5.9 times. On average, the performance improvement of SQLiteKV over SQLite for the other three workloads: read most, read heavy and read latest, are 2 times, 2.4 times and 1.9 times, respectively.

We also notice that when the key-value sizes are over 2048 bytes, SQLiteKV only outperform SQLite slightly. The reason is that bigger request size can reduce the write amplification effect in SQLite. 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, as most data sets in mobile applications only contain very few large requests, SQLiteKV can still significantly outperform SQLite in practice.

We also test the database performance with request sizes following the Zipfian distribution [33, 49, 52]. Figure 15 presents the results with the four workloads. It can be observed that with the update heavy workload, SQLiteKV achieves the highest performance improvement over SQLite, which is 2.7 times. On average, the SQLiteKV outperforms SQLite 1.7 times for all these four workloads.

We further evaluate the efficiency of our proposed coordination read cache with different workloads. Figure 16 presents the performance comparisons of SQLiteKV with and without cache. Similarly, the request sizes vary from 64 bytes to 4KB, and for all these experiments, we configured the cache size fixed with 1MB. Through the figures, we can clearly see that the coordination cache can effectively improve the database performance. The average performance improvement are 12.7% for the update heavy workload, 28.9% for the read most workload, 14.7% for the read heavy workload, and 43% for the read latest workload. The highest performance improvement with the coordination cache achieves 57.9% for the read latest workload with the request size of 256 bytes. We also test the cache effect with the request sizes that follow the Zipfian distribution. As shown in Figure 17, similarly the coordination cache effectively improves the throughput across all the workloads.

D. CPU and Memory Consumption

In this section, we will investigate the efficiency of our re-designed index management policy, and then compare the memory and CPU consumption of SQLite, SnappyDB, and SQLiteKV. For SQLiteKV, we have enabled the coordination cache. In this experiment, we also generated 100 thousand request data to pollute the database, and then issue insert and query requests to investigate their effects on the CPU and memory.

<table>
<thead>
<tr>
<th>Databases</th>
<th>CPU(%)</th>
<th>Memory/(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert</td>
<td>Query</td>
</tr>
<tr>
<td>SQLite</td>
<td>41</td>
<td>38</td>
</tr>
<tr>
<td>SnappyDB</td>
<td>26</td>
<td>50</td>
</tr>
<tr>
<td>SQLiteKV</td>
<td>32</td>
<td>61</td>
</tr>
</tbody>
</table>

Memory Footprint. Table II presents the memory utilization status for SQLite, SnappyDB, and SQLiteKV. SnappyDB consumes the most memory space for both insert and query operations. The reason is that in SnappyDB, the LSM-tree-based data structure needs to maintain the indexes of all the SSTables from all the levels in memory, which is non-trivial. As we have described in Section IV-C, nearly 90 percent query requests goes to Level0 and Level1 in real key-value store applications [20]. Thus, in SQLiteKV, we only maintain the indexes of Level0 and Level1 in memory, and leaves the
others on disk. The experimental results in Table II show that our index management policy significantly reduces the memory requirement. Comparing with SnappyDB, SQLiteKV saves 56.2% and 57.4% memory space for insert and query operations, respectively.

We have conducted experiments to compare the performances of SQLiteKV with the index of all levels (SQLiteKV in Figure 18) are cached and with the index of Level 0 and Level 1 are cached (SQLiteKV L01 in Figure 18). Figure 18 shows the result. We can conclude that our re-designed index management policy does bring some overhead to SQLiteKV, but the overhead is acceptable. With workload composed of all read requests (Read_heavy), the performance overhead is the highest, which is 10.8%. On average, our re-designed index management policy decreased the SQLiteKV throughput by 6.5%.

**CPU Utilization.** We can observe that for insert operations, SQLite requires the most CPU resource. This is because SQLite needs to maintain its in-memory B-tree index structure, which may include many split and compaction processes. On the contrary, the indexes management for SnappyDB and SQLiteKV maintain the bloom filters and key ranges information, whose operation are relatively simple. However, we also notice that our SQLiteKV consumes nearly 20% more CPU resource compared with SnappyDB. The reason is that in SQLiteKV, the SQLite-to-KV compiler requires the participation of CPU to keep translating the incoming SQL statements into the corresponding key-value operations. For query operations, SnappyDB and SQLiteKV require more CPU resources compared with SQLite, and SQLiteKV consumes the most. Since in SnappyDB and SQLiteKV, to locate a request, they may need to check several bloom filters and the indexes of SSTables in more than one levels, which requires extra CPU resource. For SQLiteKV, except the SQLite-to-KV compiler consumption, it may need to do more search to located one key-value item compared with SnappyDB, since it only stores the metadata of Level0 and Level1 in memory. Thus, SQLiteKV requires the most CPU resource.
VI. Conclusion and Future Work

In this paper, we propose a new database engine for mobile devices, called SQLiteKV, which is an 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.

As the first exploration of SQLite-like key-value database engines, the translation of SQL operations to key-value operations in SQLiteKV is straightforward. In the future, we will extend the compiler design to make its translation more efficient and complete.

ACKNOWLEDGEMENTS

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152223/15E, GRF 152736/16E, and GRF 152066/17E) and Shenzhen Science and Technology Foundation (JCYJ20170816093943197).
REFERENCES


R. Nambiar and M. Poess, Performance evaluation and benchmarking, traditional to big data to Internet of things, 2017.


Yuanjing Shi received the BS degree in computing from the Hong Kong Polytechnic University, Hong Kong, China in 2018. He is currently an upcoming graduate student at the University of Illinois at Urbana-Champaign. His research interests include Databases, Data-Intensive Systems, Distributed Systems and Cyber-Physical Systems.

Zhaoyan Shen received the BE and ME degrees in Computer Science and Technology, Shandong University, China, in 2012 and 2015, respectively, and the Ph.D. degree in Department of Computing, Hong Kong Polytechnic University, in 2018. Currently, he is an assistant professor of Shandong University. His research interests include big data systems, storage systems, embedded systems, system architecture, and hardware/software codesign.

Zili Shao is currently an Associate Professor with the Department of Computer Science and Engineering at the Chinese University of Hong Kong (CUHK). Before joining CUHK in 2018, he was with the Department of Computing, the Hong Kong Polytechnic University, where he served since 2005. He received the B.E. degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the M.S. and the Ph.D. degrees from the Department of Computer Science, University of Texas at Dallas, Dallas, TX, USA, in 2003 and 2005, respectively. His current research interests include big data systems, storage systems, embedded software and systems and related industrial applications. He is an associate Editor for IEEE Transactions on Computers, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ACM Transactions on Cyber-Physical Systems, and ACM Transactions on Design Automation of Electronic Systems.

Yong Guan received the Ph.D. degree in computer science from China University of Mining and Technology, China, in 2004. Currently, he is a professor of Capital Normal University. His research interests include formal verification, PHM for power and embedded system design. Dr. Guan is a member of Chinese Institute of Electronics Embedded Expert Committee. He is also a member of Beijing Institute of Electronics Professional Education Committee, and Standing Council Member of Beijing Society for Information Technology in Agriculture.