Tail Recursion Checker for PicoML

by

Yuanjing Shi, Zhaoxing Li
ys26, zl50

Proposal for CS421 unit project
2018

Supervisor:

Assistant Professor Sasa Misailovic

TABLE OF CONTENTS

LIST OF FIGURES

Figure

ABSTRACT

Tail Recursion Checker for PicoML

by

Yuanjing Shi, Zhaoxing Li

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 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 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.

CHAPTER 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 inefficienct date organization and coordination between its database engine and the underlying file and storage system,(references), SQLite suffers from poor transactional performance.

Many efforts have been put to optimize SQLite performance. The optimization approaches mainly fall into two aspects: (1) Investigate SQLite IO characteristics of different database workloads and mitigate the journaling over journal problem *Shen et al.* (2014) *Jeong et al.* (2013); (2) Utilize emerging non-volatile memory technology, such as phase change memory, to eliminate small, random updates to device *Oh et al.* (2015) *Kim et al.* (2016). Though various mechanisms have been proposed, they all culminate with limited performance gain. In this work, we for the first time leverage the LSM-tree data structure to improve SQLite performance.

Key-value database engine, which offers higher efficiency, scalability, and availability, usually works with simple NoSQL schema. To utilize its advantages with SQL schema, Apache Phoenix Apa (2017) provides an SQL-like interface and translates SQL queries into a series of scans in a NoSQL database - HBase. Phoenix demon-

strates 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 data sets.

There exist key-value databases on mobile device, such as SnappyDB. However, there are not widely used in mobile devices for two major issues. First, lacking of the SQLite interface causes semantic mismatch between SQLite and key value databases; thus, they cannot be directly deployed in SQLite-based mobile applications. Second, current key value databases requires a large memory footprint with an in-memory meta-data management, in which all indexes from each data block will be scanned and put up into memory for upcoming queries. Such meta-data management approach can help reduce storage overhead, but it will result in notable memory occupation. In most of cloud computing environments, that is not an issue. However, limited memory space should be considered in mobile devices *Lee et al.* (2003).

In this paper, we propose a new SQLite-like NoSQL database engine, called SQLiteKV, which adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. To address the semantic mismatch and memory constraint issues, our SQLiteKV consists of two parts: (1) Front-end layer: A SQLite-to-KV compiler and a Slab-Allocation Caching. (2) Back-end layer: An LSM-tree-based key-value storage engine with an effective meta-data management scheme.

In the front-end, the compiler receives SQL queries and translate 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. Considering the memory constraints issue in mobile devices, we manage the caching with a slab-based approach to eliminate memory fragmentation *Ding et al.* (2013). Caching space are firstly segmented into slabs while each slab is further striped into an array of slots with equal size. Slot sizes in different slabs increase exponentially. Query results are buffered into a slab whose slot size is of the best fit with its own size.

As for the back-end, we deploy a LSM-tree-based key-value database engine which transform 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 KV engine, our strategy is to store the meta-data of those top levels in memory exclusively and leave others in disk.

We have implemented and deployed SQLiteKV on a Google Nexus Android platform. The experimental results with various workloads show that our SQLiteKV presents an improvement of 6 times in the operations per second for insert and query operations compared with SQLite, respectively. 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. Chapter 2 gives background. Chapter 3 describes the design and implementation. Experimental results are presented in Chapter 4. In Chapter 5, we conclude the paper.

CHAPTER II

Background

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

2.1 SQLite

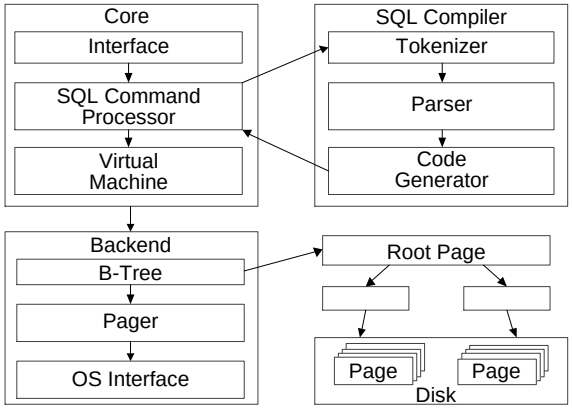


Figure 2.1: Architecture of SQLite.

SQLite is an in-process library, as well as an embedded SQL database widely used in mobile devices. Figure 2.1 gives the architecture of SQLite. SQLite exposes SQL interfaces to applications, and works by compiling SQL statement to bytecode, then running that bytecode using a virtual machine. When compiling one SQL statement, the SQL command processor first send it to the tokenizer. Then Tokenizer breaks

the SQL statement into tokens and hands those tokens one by one to the parser. The parser assigns meaning to tokens based on their context, and assembles tokens into a pase tree. After this, the code generator runs to analyze the parser tree and generate bytecode that performs the work of the SQL statement.

The data organization of SQLite database is based 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 fixed-size pages. The pages can be either table B-tree page, index B-tree 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 use a journal mechanism for crush recovery, which makes database file and journal file synchronized frequently with the disk and lead to a performance degradation consequently.

2.2 LSM-tree-based Key-Value Database

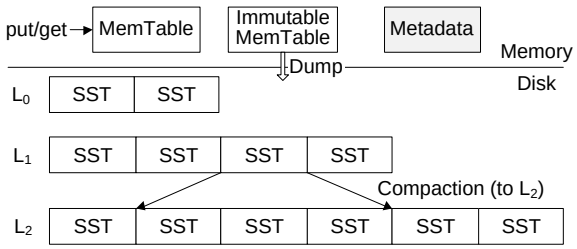


Figure 2.2: Architecture of LSM-tree-based Database.

A 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 2.2 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 string table (shown as SST in the figure) in the disk. To assist database query operations, meta-data, including indexes, bloom filters, key-value ranges and sizes of these in-disk SSTs

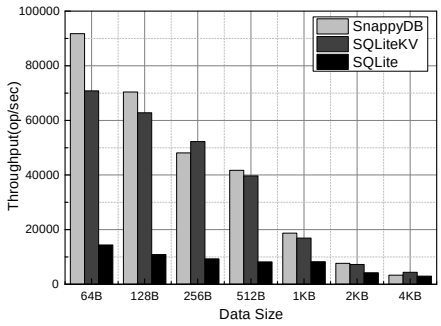


Figure 2.3: Throughput w. Insert operation.

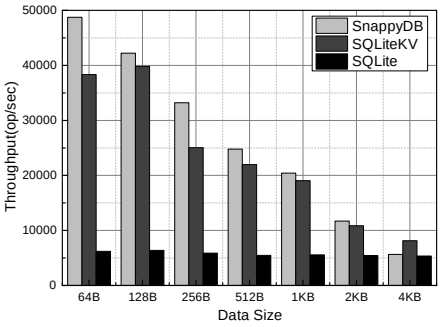


Figure 2.4: Throughput w. Query operation.

Figure 2.5: Performance comparison of SQLite vs SnappyDB

are maintained in memory *Sears et al.* (2012).

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 write. The database first use 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 SSTs. Key-value items in an SSTable are sorted according to their keys. Key range and a bloom filter of each SSTable that are maintained as metadata are cached in memory space to assist key-value search 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 2.2,

The youngest level, *Level 0*, is generated by writing the Immutable MemTable from memory to disk. Each level has a limit 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 limit.

2.3 Other SQL-Compatible Key-Value Databases

Apache Phoenix Apa (2017) is an open source SQL skin which receives SQL query by compiling it into a series of key-value operations of Apache HBase, a distributed, key-value, big data store. Phoenix provided a well-defined and industry standard APIs for OLTP and operational analytics for Hadoop. Nevertheless, without a deep integration with 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 data sets *Forman et al.* (1994), which means they can hardly fit in mobile environments as a matter of durability, battery life and portability *Sinha et al.* (2016).

In this paper, we propose an efficient LSM-tree-based lightweight Database engine, SQLiteKV, which retains SQLite interface for mobile devices, keeps a high performance compared with SQLite and adopt an efficient LSM-tree structure on its storage engine.

CHAPTER III

Project Methodology

In this section, we present SQLiteKV and its detailed design architecture. SQLiteKV is a variant of LSM tree-based key value storage engine, like Google's LevelDB or Facebook's Cassandra. It is augmented with two layers and four modules. The front-end layer includes a SQL-to-KV compiler, from which existing applications could migrate from SQLite to key-value storage without any modifications on their original SQLite codes, as well as a slab-allocation caching for memory efficiency and de-fragmentation. At the back-end layer of SQLiteKV, a new index management scheme is proposed to work along with the underlying LSM-tree-based storage engine. It can significantly improve the performance and scalability of meta-data management as well.

3.1 Design Overview

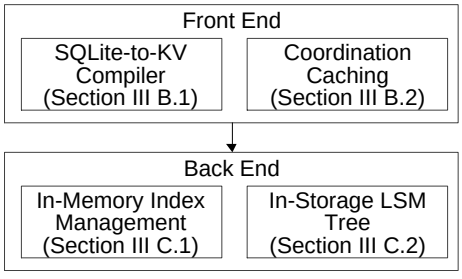


Figure 3.1: Architecture of SQLiteKV.

As shown in Fig 3.1, SQLiteKV is augmented with two layers, the front-end

layer and the back-end layer. Front-end layer consists of two modules: a SQLite to KV compiler and a slab-allocation 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 *Shao et al.* (2012), our back-end layer mainly focus on the memory and storage optimization. The two main modules of it are a re-designed index management and a LSM-tree based storage engine.

The overall architecture and these functional modules are illustrated in Figure 3.1. The two layers with four functional modules are described below in more details.

3.1.1 Front-End

In order to provide a SQLite-Compatible interface, two major components are designed and implemented on the front-side of SQLiteKV. Fig 3.2 shows the first one-a SQLite-like interface and the second one, a slab-allocation caching mechanism, is presented in Fig 3.4.

3.1.1.1 SQLite-Like Compiler

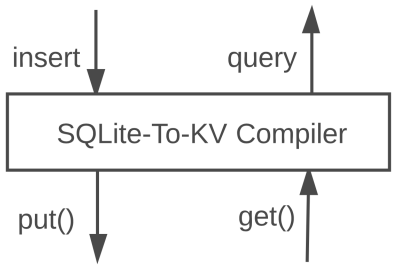


Figure 3.2: SQLite to KV Compiler work flow.

As for the interpretation of SQL statements, our SQLite-to-KV compiler firstly breaks down the statements into tokens. Then it would give each token meaning based on the context and assemble it into a parser tree. So far the compilation goes similarly as SQLite. The most important step is that it would generate key-value operations based on the result of parsing. Generally, there would be three kind of

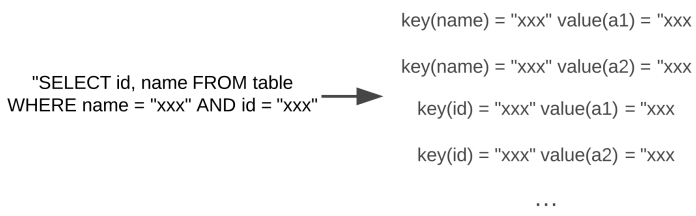
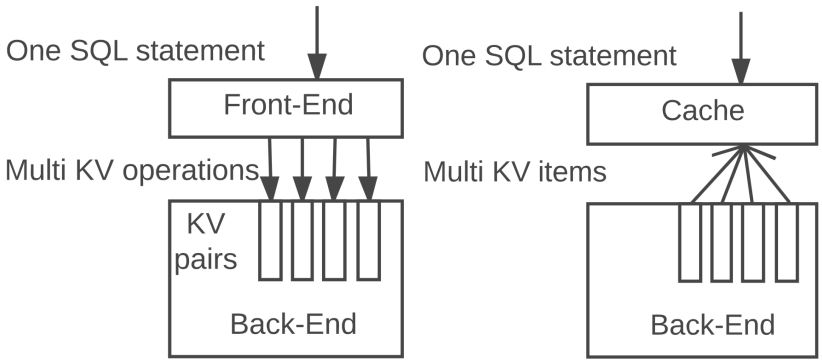


Figure 3.3: One Single SQL to Multi KV items.

operations including GET(), PUT() and DELETE(), which is also a common feature among NoSQL database. KV operations will be passed to and executed in the back-end storage engine.

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

3.1.1.2 Slab-Allocation Caching



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

Figure 3.4: SQLite Slab-Allocation Caching

SQLite uses a caching mechanism inside its back end engine to cache frequently-visited pages within the flash memory to avoid frequent visits to the disk. SnappyDB also implements a caching system for latest recently used KV items. However, one SQL statement usually corresponds to multi-KV items during the SQL compilation process of SQLiteKV. Neither of these above-mentioned caching could alleviate such

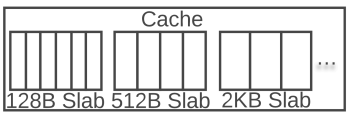


Figure 3.5: SQLiteKV Cache Structure.

data discrepancy between SQL statements and KV items. Therefore, we design a new caching mechanism as part of the front-end of SQLiteKV. It caches query results for SQL statements as a batch of KV items. Such mapping between SQL statement and KV items can significantly reduce the data discrepancy and improve the query performance. Moreover, a slab-allocation structure, as shown in Fig. 3.5 is deployed to solve the memory fragmentation issue caused by small KV items. The slab-based caching mechanism would separate memory space into different slabs with various entry sizes, like 128 bytes, 512 bytes or 4096 bytes. A specific KV item would be cached into a slab whose size fits the itself the most. Such design can effectively address the issue of memory fragmentation, especially on embedded memory constraints devices.

3.1.2 Back-End

3.1.2.1 Index Management On LSM Tree

LSM-based storage engines would commonly do a scan over the entire disk and collect all indexes back to the memory *Wu et al.* (2015). Usually index is put at the end of each data block. Hence when a query operation is to be executed, the in-memory meta-data would be binary-searched with the target key to locate the data block on disk. That data block would be visited upon then which means at most one disk seek is required for a single query on LSM-tree-based KV engine. However, this approach doesn't seem to be practical nor efficient since most mobile devices are memory constraints and not all the indexes could be accommodated in the memory. In accordance to this issue, we re-design the indexing management scheme, which exclusively stores indexes of data blocks in higher levels, like level 0 and 1, of

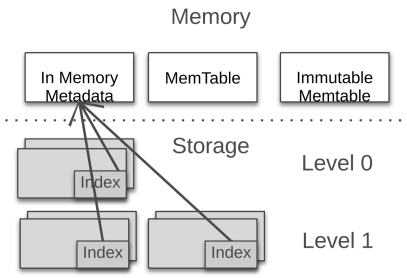


Figure 3.6: Back-End In-Memory Index Management.

the entire LSM tree. The reason is that with the level goes further down, data at lower level are less likely to be visited. In other words, the data on top levels are more likely to be newly-added or frequently-visited. Our approach reduce the huge overhead on SnappyDB's original in-memory meta-data management. At the same time, the worst disk seek time remains at same order of complexity. To sum up, our meta-data management strategy does leverage the LSM-tree structure and avoid possible memory constraints issue on the back-end side.

3.1.2.2 Storage Management

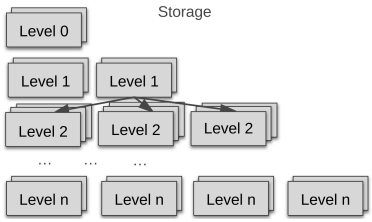


Figure 3.7: Back-End Storage Management.

The disk storage management of SQLiteKV relies mainly on its LSM-tree-based structure. As we mentioned before, once the MemTable, as shown in Fig 3.6, is converted to be an immutable table and dumped into disk. This SSTable would be placed at the first level, level 0. As long as one specific level of this LSM tree is full, those SSTables with overlapping indexes would be compacted together and dumped to the next level. During this compaction process, newly-generated SSTables on lower levels tend to have larger sizes and ranges of index consequently.

CHAPTER 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, then provide the experimental results with real-world benchmarks and synthetic workloads, with the overhead analysis provided at the end *Cooper et al.* (2010).

4.1 Experiment Setup

We have prototyped the proposed SQLiteKV on Google Nexus platform. Our implementation is based on SQLite and SnappyDB, and includes 2,344 lines of code.

All experiments have been conducted on Google's Nexus 6p smartphone that has a 2.0 GHz oct-core 64 bit Qualcomm Snapdragon 810 processor, 3 GB LPDDR4 RAM, Samsung-manufactured 32 GB eMMC 5.0 NAND flash and Android 7.1.2 with Linux Kernel 3.10.73.

SQLite 3.9 is utilized in the experiments as this is the current version in Google Nexus 6p. The page size of SQLite is set as 1024 bytes that is the default valud. SnappyDB 0.4.0 is adopted that is the latest version of a java implementation of Google's LevelDB. SQLiteKV is implemented based on the same version of SnappyDB.

To make a fair comparison, in our experiments, for each SQL query in SQLite,

Table 4.1: 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

it contains up to 999 variables that is the maximum value allowed. In this way, we avoid unnecessary inefficiency from one benchmark tool Sna (2017) from SnappyDB in which one SQLite query would only carry one variable at one time. Moreover, unnecessary calls like cursor.moveToFirst() after queries are not performed for the sake of efficiency.

4.2 Overall Performance

Based on the database benchmark tool *Cooper et al.* (2010), we have generated a set of real-world workloads to evaluate the overall performance. The ratio of insertions and queries of each workload is shown in Table 4.1.

To generate real-world requests to the database engines, 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, respectively. The object popularity, which determines the request sequence, follows a Zipfian distribution *Shen et al.* (2017), by which some records in the head will be extremely popular while some in the tail are not. In the Read Latest workload, it follows the Zipfian distribution except that most recently inserted records will be in the head of the distribution so they will be accessed more frequently. For all other workloads, record selections follow Zipfian distribution.

Figure 4.5 shows the experimental results by running SnappyDB, SQLiteKV, and SQLite with the four workloads in 4.1. For each workload, the key-value item sizes

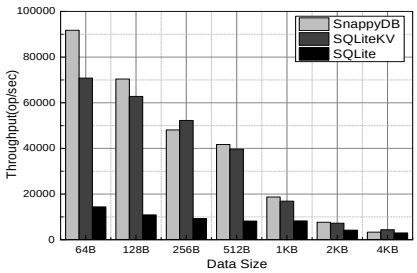


Figure 4.1: Throughput w. Update Heavy Workload.

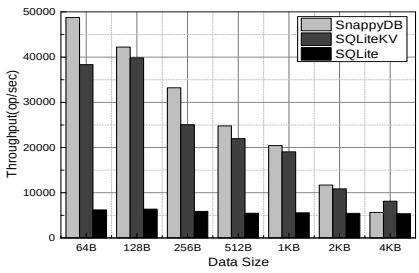


Figure 4.2: Throughput w. Read Most Workload.

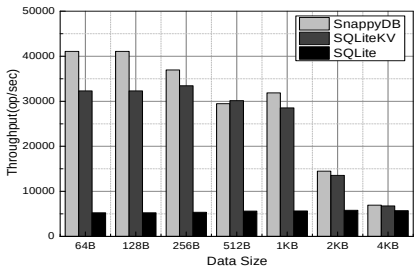


Figure 4.3: Throughput w. Read Heavy Workload.

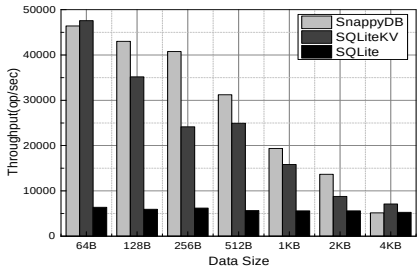


Figure 4.4: Throughput w. Read Latest Workload.

Figure 4.5: Overall Performance

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). 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), SQLite 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 *Atikoglu et al.* (2012)), we believe it is worthwhile for SQLiteKV to scarify this overhead while providing a SQLite interface.

When the key-value sizes are over 2048 bytes, SQLiteKV, as well as SnappyDB, does not significantly outperform SQLite. The reason is that 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. Re-

gardless 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 insertion/query ratios, we use the case with 128-byte KV items as an example. For the Update Heavy workload, for SQLiteKV, the throughput is 20% higher than that with the Read Heavy workload. The main reason is that in LSM-tree-based databases, for randomness-dominated workloads as these we use, write efficiency is better than read efficiency *Sears et al.* (2012). For SQLiteKV, the similar trends can be observed from other workloads.

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

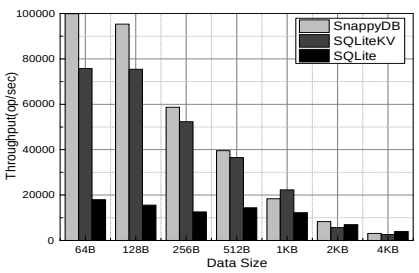
4.3 Performance with Different Sequential/Random Workloads

In the second set of experiments, we investigate the impact of random and sequential accesses on SQLiteKV, SQLite and SnappyDB. Here, the key size is 16 bytes and the value size varies from 64 bytes to 4096 bytes. Next, we present the results related to insertions and queries, respectively.

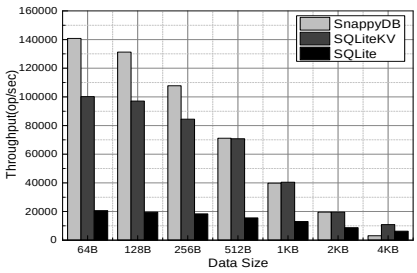
4.3.0.1 Insertion Performance

Figure 4.6b shows the results with random and sequential insertion operations, respectively, in which the key-value size varies from 64 bytes to 4096 bytes. For the sequential case, the key space is traversed in ascending order, while it is randomly traversed for the random case.

It can be observed that for SQLitKV its insertion performance with sequential

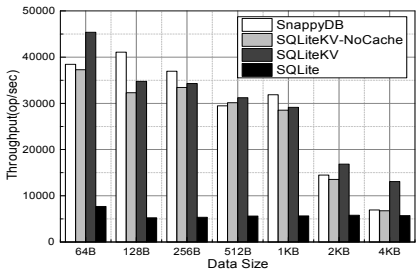


(a) Random Insertions.

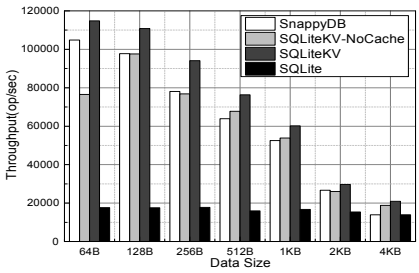


(b) Sequential Insertions.

Figure 4.6: Insertion Throughput vs. KV size



(a) Random Queries.



(b) Sequential Queries.

Figure 4.7: Query Throughput vs. KV size

records is better than that with random records (the average improvement is 40%). SnappyDB shows the simlar 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.

4.3.0.2 Query Performance

Similarly, Figure 4.7b shows the results with random and sequential query operations, respectively. 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).

In Figure 4.7b, we also present the results for SQLiteKV without cache. Compared with SQLiteKV, it can observed that our slab-allocation caching mechanism can help improve 10 - 20 % query performance. Furthermore, it can provide a noteworthy improvement, up to 2.x times, on large data sets.

CHAPTER V

Conclusion

In this paper, we proposed SQLiteKV that is a SQLite-like key-value database engine. SQLiteKV adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. It consists of two parts, a front end that contains a light-weight SQLite-to-key-value compiler and a cache, and a back end that contains a LSM-tree-based key-value database engine. We have implemented and deployed our SQLiteKV on a Google Nexus 6P Android platform. The experimental results with various workloads show that SQLiteKV can significantly outperform SQLite.

BIBLIOGRAPHY

BIBLIOGRAPHY

(2017), Apache phoenix, https://phoenix.apache.org/.

(2017), Snappydb, http://www.snappydb.com/.

Atikoglu, B., et al. (2012), Workload analysis of a large-scale key-value store, in *SIGMETRICS*.

Cooper, B., et al. (2010), Benchmarking cloud serving systems with ycsb, in *SoCC*.

Ding, H., et al. (2013), Integrated instruction cache analysis and locking in multi-tasking real-time systems, in *DAC*.

Forman, G., et al. (1994), The challenges of mobile computing, *Computer*.

Jeong, S., et al. (2013), I/o stack optimization for smartphones., in *ATC*.

Kim, W., et al. (2016), Nvwal: exploiting nvram in write-ahead logging.

Lee, H., et al. (2003), Energy-aware memory allocation in heterogeneous non-volatile memory systems, in *ISLPED*.

Oh, G., et al. (2015), Sqlite optimization with phase change memory for mobile applications, *VLDB*.

Sears, R., et al. (2012), blsm: a general purpose log structured merge tree, in *SIGMO/PODS*.

Shao, Z., et al. (2012), Utilizing pcm for energy optimization in embedded systems, in *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pp. 398–403, IEEE.

Shen, K., S. Park, et al. (2014), Journaling of journal is (almost) free., in *FAST*.

Shen, Z., et al. (2017), Didacache: A deep integration of device and application for flash based key-value caching., in *FAST*.

Sinha, S., et al. (2016), Low-power fpga design using memoization-based approximate computing, *VLSI*.

Wu, X., et al. (2015), Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items, in *USENIX ATC*.