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 inefficient date organization and coordination between its database engine and the underlying file and storage system, 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 [1] [2]; (2) Utilize emerging non-volatile memory technology, such as phase change memory, to eliminate small, random updates to device [3] [4]. 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 [5] provides an SQL-like interface and translates SQL queries into a series of 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 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 [6].

II. BACKGROUND

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

A. SQLite

SQLite is an in-process library, as well as an embedded SQL database widely used in mobile devices. 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 parse tree. After this, the code generator runs to analyze the parse tree and generate bytecode that performs the work of the SQL statement.

B. LSM-tree-based Key-Value 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 1 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 are maintained in memory [7].

III. SQLITEKV

In this section, we present SQLiteKV and its detailed design architecture. SQLiteKV is a variant of LSM tree-based key-value storage engine, like Googles LevelDB or Facebooks
Cassandra. It is augmented with two layers and four modules. The front-end layer includes a SQL-to-KV compiler. 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.

A. Design Overview

![Fig. 2: Architecture of SQLiteKV.](image)

As shown in Fig 2, 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 [8], 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 2. The two layers with four functional modules are described below in more details.

B. 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 shows the first one—a SQLite-like interface and the second one.

![Fig. 3: SQLite to KV Compiler work flow.](image)

1) SQLite-Like Compiler: 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 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 possible that existing applications could run smoothly with original SQL statements and leverage the potentials of key value storage.

C. Back-End

1) Index Management On LSM Tree: 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 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.

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 4, 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.

REFERENCES