2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems

# On the IO characteristics of the SQLite Transactions

Dam Quang Tuan, Seungyong Cheon and Youjip Won
Hanyang University, Korea
{damquangtuan | chunccc | yjwon}@hanyang.ac.kr

## ABSTRACT

This work is dedicated to study the IO characteristics of SQLite transaction in Android platform. We collect the block level IO trace from for six months. We develop an elaborate pattern matching algorithm. It allows us to identify the individual SQLite transactions from the raw IO trace, which is essentially an interleaved mixture of IO requests from concurrently running smartphone applications. Among the various observations obtained from the study, we can summarize the key findings as follows. We carefully believe that these deserve special attention. First, SQLite transaction is under extreme inefficiency. In an SQLite transaction, the IO's for SQLite journaling and EXT4 filesystem journaling account for over 75% of the entire IO volume in a transaction. Second, the suspend and the wakeup feature of the smartphone can leave the SQLite transaction to an extreme delay, a few minutes. Third, in fair number of occasions, the SQLite transactions are being used in inconsiderate manner. We find that a single SQLite transaction inserts 17.5 MByte of data to the database. It turns out to be the operation of initializing the map database. We hope that the findings obtained in this study are bifurcated to various efforts to makes the SQLite related IO more efficient and effective.

## Keywords

SQLite, Android, IO Stack, Real-time IO Characteristics, Measurement, EXT4

## 1. INTRODUCTION

With the steady growing in the development of smartphone in recent years, the need of understanding IO characteristic in Smartphone devices is becoming vitally important. SQLite is arguably the most widely used DBMS. The SQLite is the library based embedded DBMS which can easily be integrated with the existing source code. Mainly due to its versatility, it is widely adopted in various computing platform ranging from the smartphone, e.g. Android, iOS, Tizen, firefox [30] to the enterprise server, e.g. openstack [20]. Recent works reported that the SQLite is responsible for excessive writes in mobile platform [12, 17, 22]. A lot of effort have been put to optimize SQLite performance [12, 18, 23, 25, 28]. There have been a number of works to characterize the IO behaviors of the different database workload, e.g. OLTP [27], OLAP [4], NoSQL [6] and etc. Despite its popularity, on the other hand, we still do not have the comprehensive understanding on the SQLite's IO characteristics, e.g. the size of a transaction, the length of a transaction and etc. This work is dedicated to acquire comprehensive understanding on the SQLite's IO characteristics in smartphone platform. For this study, we collect the IO trace from four volunteers for six month. We develop an elaborate pattern matching algorithm which enables us to identify the individual SQLite transactions from raw block level IO trace. Combining the process name, the file name, the block address and the time, we obtain the key characteristics of an SQLite transaction: size, latency, journaling overhead and etc. The key findings can be summarized as follows.

- IO behavior of an SQLite transaction is subject to extreme inefficiency. In SQLite transaction, the IO's to the database file accounts for less than 25% of the entire IO volume. SQLite journaling and EXT4 filesystem journaling accounts for the rests. Fragmented SQLite journal structure and the synchronization overhead of the file-backed journaling mechanism are the main cause for this inefficiency.

- Suspend and wakeup feature of smartphones exposes the SQLite transaction under prohibitively long transaction latency. File-backed journaling mechanism of SQLite makes the SQLite transaction survive the suspension of the smartphone device without triggering crash recovery routine. A few SQLite transactions take several minutes to finish due to suspension of a smartphone. A simple abort and retry of a transaction should be good resort to avoid excessive delay.

- SQLite is often used in surprisingly inconsiderate manner. Some SQLite transactions are exceptionally large: We find that a single SQLite transaction is used to store 17.5 MByte of data. This is found to be the operation of initializing the subway route database. We carefully believe that the application developers should acquire thorough understanding on the IO implication of the SQLite transaction to avoid egregious mistake.


*MobileSoft'16, May 16-17, 2016, Austin, TX, USA*

DOI: http://dx.doi.org/10.1145/2897073.2897093

- Frequent `fdatasync()` call in SQLite transaction makes the EXT4 journaling extremely inefficient. In an SQLite transaction, average EXT4 journal size is 16 KByte. The individual EXT4 journal transaction harbors only two pages of the updated metadata. The filesystem journaling efficiency of EXT4, i.e. the ratio of log record size against the entire total log size is mere 50%.

There have been a number of works in examining the IO characteristics in Android IO[12, 14, 23]. These works use very simple workload, e.g. to insert single 100 Byte record to SQLite database [11] or to collect the trace for very short period of time, e.g. 5 min, executing a single smartphone application. These works successfully identify the essential characteristics on the inefficiency in Android IO stack, *Journaling of Journal* anomaly. However, their work do not carry any information on the characteristics behavior of SQLite IO in real settings, e.g. the size, latency, the login overhead of an SQLite transactions and possibly any type of anomalous behavior. As long as we are aware, this is the first study which characterizes the IO behavior of the SQLite transaction from real settings for an extensive period of time.

This work carries comprehensive understanding on IO behavior of the most widely used mobile database. This work is for posing a set of issues rather than for providing a solution. We believe that the issues raised in this study deserve the further attention from the community, whose effective resolution leads to more efficient computing platform.

The rest of the paper is organized as follows. Section 2 explains the basics of SQLite. Section 3 explains the algorithm to identify the SQLite transaction. Section 4 carries the result of the analysis. Section 5 introduces the preceding efforts on this subject and section 6 concludes the paper.

# 2. SYNOPSIS

## 2.1 SQLite Database Structure

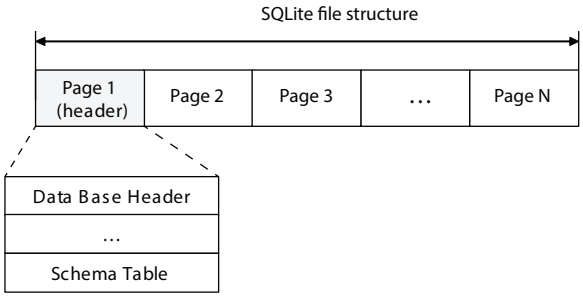



**Figure 1: SQLite Database File**

SQLite database file consists of the database pages [9] as illustrated in Fig. 1. The first page contains 100 Byte database header and the database schema. Other than the first page, a database page consists of a page header, cell pointer array and the cell contents (Fig. 2). The cell pointer array and the cells start from the beginning of a page and from the end of a page, respectively and grow in the opposite direction. One can reserve a space ($\leq$ 255 Byte) at the end of each page. The default size of the reserved area is 0. Cell pointer array is arranged in increasing order of the key

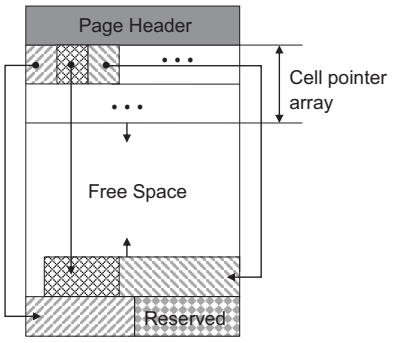



**Figure 2: SQLite Page Structure**

values. Each cell pointer is 2 Byte integer and contains the offset to the first byte of the cell content. Cell content region contains actual cell data. SQLite places cell content area at the end of a page next to the reversed place. SQLite adopts B+-tree [5] for its database table.

## 2.2 SQLite Journal

SQLite provides both rollback journal and rollforward journal mechanisms [23]. A rollback journal file consists of one or more log segments [9] as shown in Fig. 3. Each log segment contains segment header (journal header) and a number of old database pages (undo logs). Under normal circumstances, there exists only one log segment in a roll-back journal file since the SQLite removes the undo logs from the the roll-back journal when a transaction completes. A roll-back journal file may contain a number of log-segments when a transaction creates large number of dirty page cache entries. When a transaction creates large number of dirty page cache entries before the transaction completes, the database needs to reflect those changes and flushes the redo logs to the roll-back journal file so that it can accommodate the newly updated pages into buffer cache. Each flush yields a single log segment in the roll-back journal file. This technique is called *cache-spill-prior-to-commit* [29].

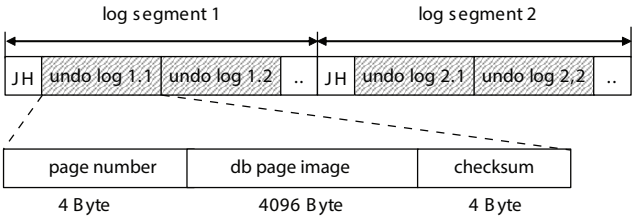



**Figure 3: SQLite Roll-back Journal File Structure**

A journal header stores the number of old pages in the log segment. This number is initialized to zero. Only after all undo logs in the journal file are synchronized to the disk, the actual number of old pages are synchronized to the disk. This field is used in the recovery phase to determine the number of undo logs to be recovered.

WAL file consists of the WAL header, the frame header and the WAL frame for each updated database page (redo page) [23]. Fig. 5 illustrates the structure of WAL file.

WAL header (32 Byte) contains the database page size and the checkpoint sequence number. The frame data is considered as valid if the values of `Salt-1` and `Salt-2` in WAL

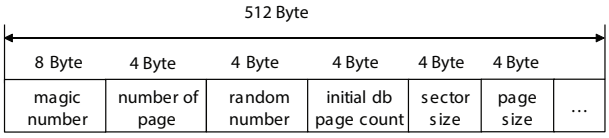



**Figure 4: SQLite Journal Header Structure**

header match the values of `Salt-1` and `Salt-2` in frame header and the checksum for each frame matches the checksum in the WAL header.

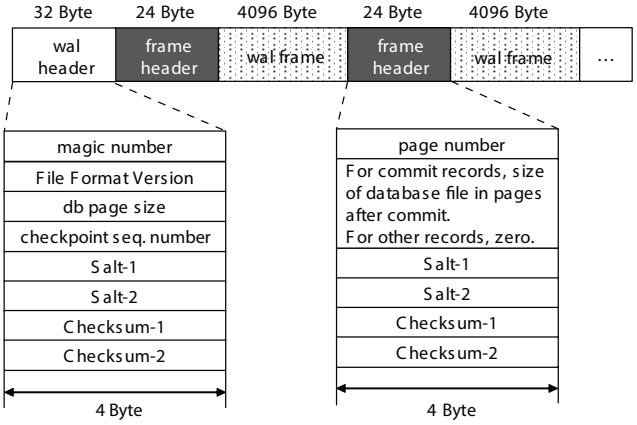



**Figure 5: SQLite Wal File Structure**

In roll-back journaling, SQLite first writes the undo logs to the roll-back journal file and then updates the database. After the database update completes, SQLite removes the undo logs from the roll-back journal file to denote that a transaction has completed successfully. There are three ways to remove the undo logs from the roll-back journal file subject to the three roll-back journal modes in SQLite. The details will be dealt with shortly. In WAL, SQLite writes the redo logs to the WAL file and later checkpoints the redo logs to the actual database.

There are six journal modes in SQLite. They are PERSIST, DELETE, TRUNCATE, WAL, MEMORY and OFF [31]. In MEMORY mode, journal file is maintained in memory. In MEMORY mode, the ACID property can be compromised in case of the unexpected system failure. There are three roll-back journaling modes: DELETE, TRUNCATE and PERSIST and one roll-forward journaling mode: WAL.

SQLite provides two options to synchronize the journal file: FULL vs. NORMAL. The behavior of each synchronization option governs the time when the journal file is synchronized to disk. In FULL synchronization option, SQLite synchronizes the roll-back journal file to disk immediately after it finishes writing the undo logs. In NORMAL synchronization option, SQLite does not synchronize the roll-back journal file. The roll-forward journaling yields similar behavior. In FULL synch, WAL file is synchronized to disk immediately after SQLite finishes logging. In NORMAL synch, the WAL file is synchronized not when the transaction is committed to WAL file but when the updated pages are checkpointed to the original database. For durability of a transaction, one should use FULL synch option.

SQLite provides two options on how to synchronize the file to the disk: `fsync()` vs. `fdatasync()`. It is compile time option. Both of these systems calls flush the dirty page cache entries for a given file to the disk. They differ in a way in which the filesystem handles the updated metadata. In `fdatasync()`, EXT4 filesystem journals the updated metadata only when a new block is allocated or an existing block is deallocated. In `fsync()`, any updates in the inode, e.g. change in `atime` field, triggers the EXT4 filesystem to journal the updated metadata.

## 2.3 SQLite Journaling Mode and EXT4 journal

We examine the interaction between SQLite and EXT4 journaling under different SQLite journaling modes. In roll-back journaling, SQLite reads the old pages from database file and logs them to the journal file (undo logging). Then, SQLite updates the database file (database update). As a final step, SQLite removes the undo records from the roll-back journal file (log reset). In undo-logging, SQLite first writes the undo logs and then updates the journal header to record the number of undo logs in the roll-back journal. One need to assure that the the number of undo logs should be written to disk *after* the undo logs reach the disk surface.

The three roll-back journal modes in SQLite, DELETE, TRUNCATE and PERSIST, differ in how they remove the undo logs when a transaction completes. In DELETE mode, SQLite removes the journal file. In TRUNCATE mode, SQLite truncates the journal file, i.e. deallocates all blocks. In PERSIST mode, SQLite puts a special pattern in the journal header to denote that a transaction has successfully completed. While the difference looks subtle, it bears grave implication on the IO behavior. In DELETE mode, SQLite needs to create a journal file everytime when a new transaction starts. It entails an update on inode bitmap, inode table, block bitmap, as well as the update on the directory block. TRUNCATE mode reuses the existing journal file. It requires only the allocation of new block. In PERSIST mode, SQLite reuses the existing file and does not require an allocation of new block.

In FULL synch option (default), SQLite synchronizes the journal file immediately after it finishes writing the undo logs. EXT4 filesystem is responsible for persistently storing the journal file to the disk. In the course of synchronizing the journal file to the disk, EXT4 filesystem journals the updated metadata for the respective journal file. The filesystem journal overhead varies widely subject to the SQLite journal mode. The synchronization option, `fsync()` vs. `fdatasync()`, is another important factor to govern the filesystem journaling overhead.

For concrete understanding, we run a simple experiment. We insert one record (100 Byte) to SQLite database table. We examine the amount of IO's incurred in inserting a record. We adopt three SQLite roll-back journal modes, DELETE, TRUNCATE and PERSIST and two synchronization option, (`fsync()` and `fdatasync()`).

Fig. 6 illustrates the amount of IO's in inserting a record. Each bar denotes the total amount of IO's incurred. Each bar consists of three components: IO's to SQLite database, SQLite journal file and the EXT4 journal. We can see the filesystem journal overhead varies widely subject to the SQLite journal mode. With proper selection of SQLite journal mode and synchronization option, IO volume reduces nearly to 1/2 from 80 KByte (DELETE with `fsync()`) to 36 KByte (PERSIST with `fdatasync()`).

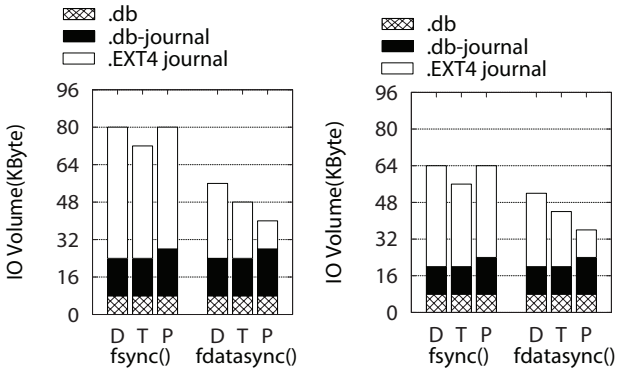



**Figure 6: Dissection of SQLite transaction: inserting 100 Byte. D: DELETE, T: TRUNCATE, P: PERSIST mode**

The default SQLite library packaged in the current stock Android (Android 6.0 Marshmallow) platform adopts PERSIST mode with `fdatasync()`.

# 3. IDENTIFYING AN SQLITE TRANSACTION

We lay two constraints in collecting the IO trace at the beginning of the study. The trace should reflect the real world user behavior and the trace should span sufficiently long period of time to capture any rarely observed anomalous behavior. Androtrace [24] can collect the IO trace from the Android smartphone without user intervention. This enables us to collect the IO trace for extensive period of time. However, Androtrace collects only the block level IO access information and does not contain any SQLite transaction related information. This leads us to develop a technique to identify the individual SQLite transactions from the block trace [10].

## 3.1 Overview

A low level block trace is an interleaved mixture of the IO requests issued from a number of applications as well as from the OS kernel. The *Androtrace* collects the block level IO trace from the smartphone device. While it collects comprehensive information on the block level IO request, the trace does not contain any information to denote the beginning and the end of a transaction. It is non-trivial task to identify an SQLite transaction from the block trace.

A transaction in SQLite is a sequence of file IO's for the SQLite database, the SQLite journal file and the EXT4 journaling. We develop an algorithm to identify the beginning and the end of the individual SQLite transactions from block level IO trace. An SQLite transaction is defined along the two axis: SQLite journal modes (DELETE, TRUNCATE vs. PERSIST) and synchronization mode (FULL vs. NORMAL). In WAL mode, SQLite stores the redo log to `*.wal` file and therefore we can determine the WAL mode via examining the extension of the journal file which is being accessed. However, three roll-back journal mode share the same extension, `*.db-journal` for the journal file.

While Android platform is packaged with stock SQLite library, an application can use its own SQLite library with its own SQLite journal mode and synchronization option. Proper identification of an SQLite transaction mandates that we should be able to determine the SQLite journal mode and the synchronization option solely based upon block access pattern. We develop a finite state machine which allows us to precisely identify the individual SQLite transaction without prior knowledge on the SQLite journal modes and the synchronization option.

The overall procedure can be sketched as follows. The low level block trace contains the block address, IO size, file name and the process name. First, from the raw block IO trace, we select the SQLite related block access, e.g. `*.db`, `*.db-journal`, `*.wal` and etc. The output from the first phase does not contain any filesystem journaling related IO's. Our algorithm only involves the SQLite related file accesses and does not require any other IO access information, e.g. filesystem journaling. Second, we sort the output from the first phase according to the process name and the time. Third, we apply a finite state machine to identify the beginning and the end of the individual SQLite transactions. Fourth, we augment the original IO trace with the SQLite transaction information. Via four steps, we reconstruct the SQLite transactions which contain SQLite related file IO's as well as the respective EXT4 journal activity. Fig. 7 schematically illustrates the procedure for identifying the SQLite transactions from a block trace.

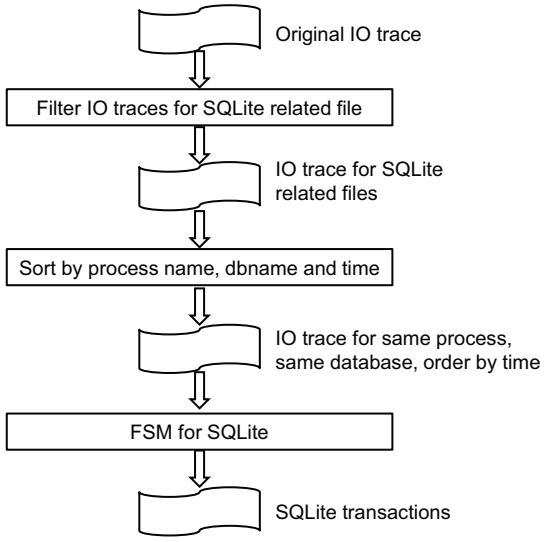
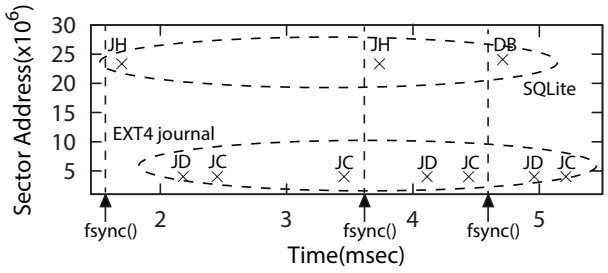



**Figure 7: Identify an SQLite transaction**

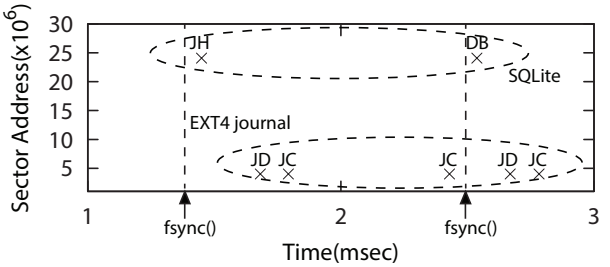
The key ingredient of the identification procedure is to develop a finite state machine for determining the set of IO's belonging to the individual SQLite transactions. We overhaul the IO behavior of different SQLite journal modes and of different synchronization options.

## 3.2 IO's in TRUNCATE mode

In TRUNCATE mode, SQLite logs the undo records to the roll-back journal file, updates the database and truncates the roll-back journal file. Fig. 8 illustrates the block access patterns in inserting a record to the database in TRUNCATE mode. These figures are drawn from the real IO trace. IO's in each figure are grouped into two and are circumscribed



(a) FULL Synch



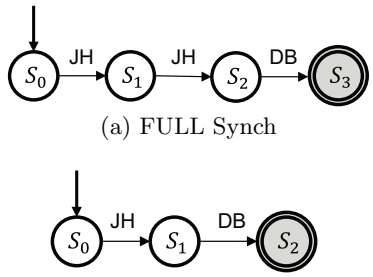
(b) NORMAL Synch

**Figure 8: DELETE/TRUNCATE mode**

with dashed line. They are IO's for SQLite related files and for EXT4 journal, respectively. In SQLite related IO's, "JH" and "DB" denote the accesses to the SQLite journal file and the access to the SQLite database file, respectively. In EXT4 journaling, "JD" and "JC" denote the activity of storing the journal descriptor ( the transaction header and updated metadata blocks in EXT4 journal) and the journal commit block in EXT4 filesystem, respectively. Each figure contains the `fsync()` mark which virtually denotes the time when the `fsync()` is called.

There are two figures: one for IO trace with FULL synch option (Fig. 8(a)) and the other for IO trace with NORMAL synch option (Fig. 8(b)). SQLite yields different IO pattern subject to the synch option. In FULL synch, SQLite synchronizes the roll-back journal file after it finishes logging. In NORMAL synch, it does not synchronize the roll-back journal file after logging. Let us take a detailed look. In Fig. 8(a), `fsync()` is called twice in the logging phase. The first `fsync()` is for flushing the undo log records to the journal file. The second `fsync()` is for flushing the updated journal header which contains the number of redo pages in the rollback journal. If we omit the first `fsync()`, the journal header with the number of redo logs may be written to disk surface before the redo logs reach the disk. Unexpected failure in the middle of logging may leave the roll-back journal with the updated number of redo logs without actual redo logs. Under this circumstances, SQLite may recover the database table with wrong pages and the database can get corrupt. The third `fsync()` is to synchronize the updated database pages. In NORMAL synch (Fig. 8(b)), SQLite does not call `fsync()` after updating the journal header. The durability can be compromised.

Based upon this characteristics, we can draw a simple state machine for TRUNCATE mode as in Fig. 9. For data blocks accesses, DELETE mode yields the similar if not identical block access patterns to TRUNCATE mode.

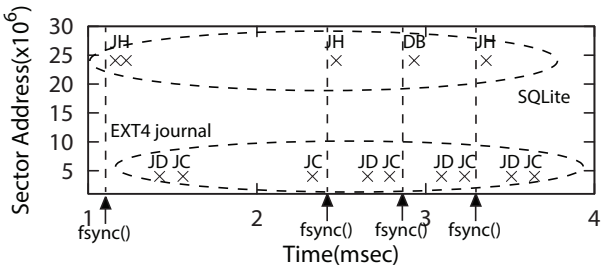


(a) FULL Synch

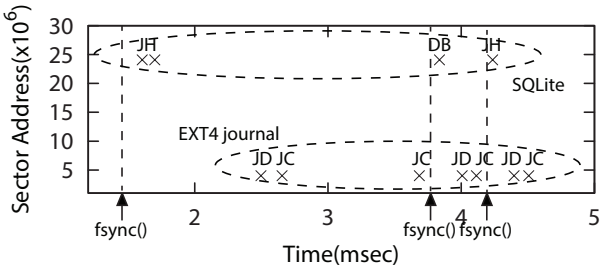
(b) NORMAL Synch

**Figure 9: State Diagram: DELETE/TRUNCATE mode**

## 3.3 IO's in PERSIST mode



(a) FULL Synch



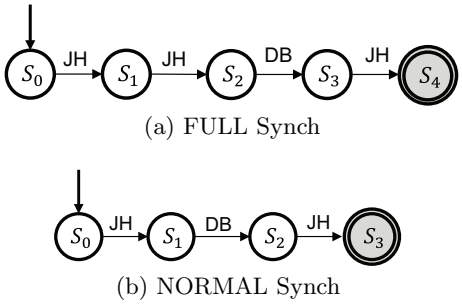
(b) NORMAL Synch

**Figure 10: PERSIST mode**

In logging phase, PERSIST mode yields the identical IO behavior to TRUNCATE mode. The difference arises in database update phase. In TRUNCATE mode, SQLite truncates the journal file at the end of a transaction, but it does not synchronize it. In PERSIST mode, SQLite zero-out the journal header and synchronize the journal file to denote that the transaction has successfully completed. Figures in Fig. 10 illustrate the IO access pattern for PERSIST mode for FULL sync option and NORMAL synch option, respectively. Figures in Fig. 11 illustrate the state transition diagram with FULL synch and NORMAL synch option in PERSIST mode, respectively.

## 3.4 Finite State Machine

We need to determine the beginning and the end of SQLite transactions based upon the state transition diagram in Fig. 9 and Fig. 11. There exists an ambiguity since the IO pattern for TRUNCATE mode is a substring of the IO pattern for PERSIST mode. Let us provide an example. Consider the block trace JH-JH-DB-JH. We cannot determine if the last



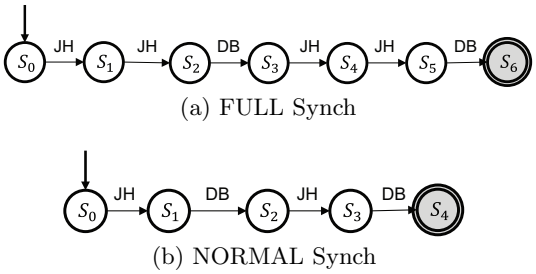

(a) FULL Synch

(b) NORMAL Synch

**Figure 11: State Transition Diagram: PERSIST mode**

JH in this sample string is the beginning of a new transaction in TRUNCATE mode with FULL synch, or the end of a transaction in PERSIST mode with FULL synch.

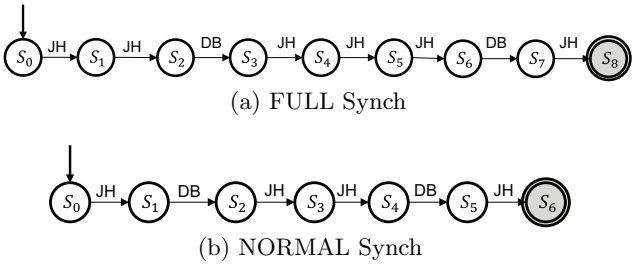
To develop a deterministic algorithm, we exploit the fact that the SQLite options are static, i.e. a process uses the same SQLite journal mode and synchronization option for the same SQLite journal file. By examining consecutive SQLite transactions, we can construct mutually disjoint state transition diagram. Fig. 12 and Fig. 13 illustrate the state machine to determine two consecutive transactions.




(a) FULL Synch

(b) NORMAL Synch

**Figure 12: State Diagram: DELETE/TRUNCATE mode**




(a) FULL Synch

(b) NORMAL Synch

**Figure 13: State Diagram: PERSIST mode**

Combining all together, we build a finite state machine for identifying an SQLite transaction as in Fig. 14.

Table 1 summarizes the final states and the respective SQLite transaction types. Reaching these final states, we conclude the two transactions for the respective journal mode exist.

Let us provide an example. Fig. 15) illustrates the original block trace. In the Figure, D stands for Data block IO, J stands for EXT4 journal IO and S stands for Synchronous

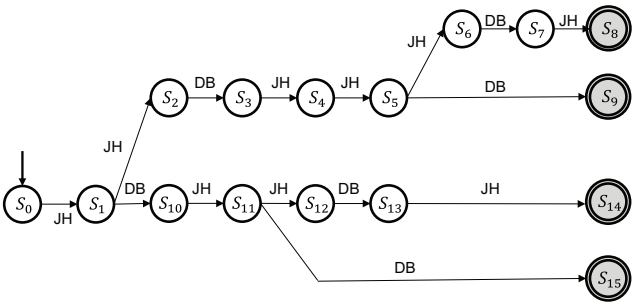



**Figure 14: SQLite State Diagram for Rollback Journaling Mode. JH is SQLite Journal File, DB is SQLite Database File.**

**Table 1: SQLite State Diagram Table for Rollback Journaling Mode**

| State | Journaling Mode | Synchronous Mode |
|---|---|---|
| $S_8$ | PERSIST | FULL |
| $S_9$ | DELETE/TRUNCATE | FULL |
| $S_{14}$ | PERSIST | NORMAL |
| $S_{15}$ | DELETE/TRUNCATE | NORMAL |

IO. From the block trace, we select the SQLite related file accesses and sort them with respect to the process name and the time. The SQLite related files correspond to roll-back journal file (`*.db-journal`), WAL file (`*.wal`) or database file (`*.db`). In this trace, there are two database files and the respective rollback journal files. The database files correspond to `authCache.db` and `client_settings.db`. The trace file includes the IO accesses to the respective roll-back journal files for these databases. From the first phase, we obtain ten trace records. They are for SQLite related file IO's: `authCache.db`, `authCache.db-journal`, `client_settings.db` and `client_settings.db-journal`.

| id | time(ms) | op | bt | pname | fname | db |
|---|---|---|---|---|---|---|
| 32322 | 08:36 631 | S | D | pool | authCache-journal | authCache |
| 32323 | 08:36 632 | S | D | pool | authCache | authCache |
| 32324 | 08:36 633 | S | D | pool | authCache | authCache |
| 32325 | 08:36 635 | S | J | jbd2 | | |
| 32326 | 08:36 637 | S | J | jbd2 | | |
| 32327 | 08:36 638 | S | D | pool | authCache-journal | authCache |
| 32328 | 08:36 655 | S | D | pool | clients_settings-journal | clients_settings |
| 32329 | 08:36 656 | S | J | jbd2 | | |
| 32330 | 08:36 658 | S | J | jbd2 | | |
| 32331 | 08:36 659 | S | J | jbd2 | | |
| 32332 | 08:36 660 | S | D | pool | clients_settings-journal | clients_settings |
| 32333 | 08:36 661 | S | D | pool | clients_settings | clients_settings |
| 32334 | 08:36 662 | S | J | jbd2 | | |
| 32335 | 08:36 666 | S | J | jbd2 | | |
| 32336 | 08:36 667 | S | D | pool | clients_settings-journal | clients_settings |
| 32337 | 08:36 669 | S | D | pool | clients_settings-journal | clients_settings |
| 32338 | 08:36 669 | S | D | pool | clients_settings-journal | clients_settings |
| 32339 | 08:36 687 | S | J | jbd2 | | |
| 32340 | 08:36 690 | S | J | jbd2 | | |

**Figure 15: Original IO Traces**

Fig. 16 illustrates the subset of IO traces. The trace id

is from 32322 to 32338. Let us closely examine the IO accesses to `client_settings` database. They consist of six trace records from 32328 to 32338 in Fig. 16. The pattern yields JH-JH-DB-JH-JH-JH. Applying this pattern to our finite state machine (Fig. 14), the fourth access which corresponds to "JH", leads us to $S_4$. The trace id of fourth JH is 32336. $S_4$ corresponds to the end of the first SQLite transaction in PERSIST mode with FULL Synch.

| id | time(ms) | op | bt | pname | fname | db |
|---|---|---|---|---|---|---|
| 32322 | 08:36 631 | S | D | pool | authCache-journal | authCache |
| 32323 | 08:36 632 | S | D | pool | authCache | authCache |
| 32324 | 08:36 633 | S | D | pool | authCache | authCache |
| 32327 | 08:36 638 | S | D | pool | authCache-journal | authCache |
| 32328 | 08:36 637 | S | D | pool | clients_settings-journal | clients_settings |
| 32332 | 08:36 655 | S | D | pool | clients_settings-journal | clients_settings |
| 32333 | 08:36 660 | S | D | pool | clients_settings | clients_settings |
| 32336 | 08:36 661 | S | D | pool | clients_settings-journal | clients_settings |
| 32337 | 08:36 667 | S | D | pool | clients_settings-journal | clients_settings |
| 32338 | 08:36 669 | S | D | pool | clients_settings-journal | clients_settings |

**Figure 16: IO Traces with SQLite related files**

Assuming that everything goes smooth, we identify an SQLite transaction which begins at 32328 and ends to 32336. Fig. 17 illustrates the final result. The SQLite transaction encapsulated by the dashed line corresponds to the one which we have just identified. It starts at 32328 and ends at 32336. The journal mode and synchronization option correspond to PERSIST and FULL, respectively. The total IO size for SQlite journal file and the SQLite database (`d_size`) corresponds to 48 KByte. The total IO size for EXT4 journal (`j_size`) corresponds to 40 KByte.

| start_id | end_id | s_mode | j_mode | process | db_name | d_size | j_size |
|---|---|---|---|---|---|---|---|
| 32309 | 32317 | FULL | D | pool | authCache | 56 | **0** |
| 32328 | 32336 | FULL | D | pool | clients_settings | 48 | 40 |
| 32337 | 32345 | FULL | D | pool | clients_settings | 56 | 0 |

**Figure 17: SQLite Transaction Result; start_id and e_id indicate beginning and end IO trace of a single SQLite transaction. It also shows what synchronous mode, journaling mode, process and db name. d_size and j_size are sum of IO volume for EXT4 data block and journal block respectively.**

# 4. ANALYSIS

## 4.1 Collecting IO traces

For this study, we collect the IO trace from four users for six month: August 2014 to Febrary 2015 We collect the IO trace using *Androtrace* [24]. Androtrace is a trace collection tool specifically tailored for Android platform. It consists client side trace collection module and the back end server. The trace collection module consists of the modified linux kernel and the transport module. The modified linux kernel flushes the collected trace to the disk. The modified linux kernel augments the block access request with the synchronization flag and the name of the process which issues the respective IO. The transport module transports the IO trace in the disk to the Androtrace server. Androtrace is designed to collect the trace without user intervention. It allows to collect the trace for extensive period of time in seamless manner.

## 4.2 Transaction Latency

We first examine the latency of the SQLite transactions. The median of the transaction latency corresponds to 11-12 msec in all four users. The average latency is 6x larger than the median in user 4. For 99.9% point, the transaction latency reaches over several seconds. For user 4, 99.9% of the latency is prohibitively long, yielding 13 sec. The extremely long transactions (>= 1 second) account for 0.16% of total transactions count. Fig. 18 illustrates the CDF of SQLite latency. Table 2 illustrates the quantile statistics.

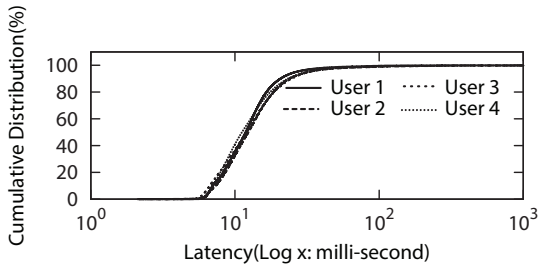



**Figure 18: SQLite Transaction latency**

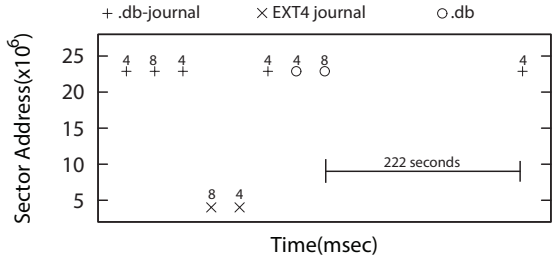
**Table 2: Quantiles of SQLite transaction latency (msec)**

| | Med. | Avg. | 75% | 99% | 99.9% | Max |
|---|---|---|---|---|---|---|
| User 1 | 11.9 | 27.9 | 15.4 | 62.3 | 266.6 | 110543.6 |
| User 2 | 12.4 | 42.8 | 17.0 | 109.4 | 4626.2 | 295934.1 |
| User 3 | 12.0 | 47.2 | 17.0 | 86.5 | 9261.6 | 222575.4 |
| User 4 | 11.2 | 68.8 | 16.0 | 84.4 | 12986.8 | 281887.2 |

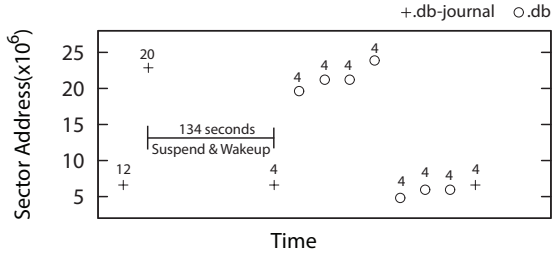
The most important characteristics which distinguish the smartphone from the rest of the computing system, e.g. PC or server is the suspend/wakeup behavior. Smartphone developers take special care to minimize the energy consumption. As a result, the smartphone frequently goes into suspend mode to avoid unnecessary energy consumption. We find that an SQLite transaction can span multiple suspend/wakeup cycle. Since the state of system is safely stored to the disk, the SQLite transaction safely resumes its execution after wakeup. Due to this characteristics, we find that the SQLite transaction can be as long as a few minutes. The file-backed journaling mechanism of SQLite makes the SQLite transaction suspend-resilient, but expose SQLite under excessive latency.

Figures in Fig. 19 illustrate the block trace of the SQLite transaction with extreme delay. In Fig. 19(a), we observe 222 sec interval between the database update and the log reset. It takes more than 4 min to complete this transaction. Similarly, we observe 134 sec interval between the first `fdatasync()` and the second `fdatasync()`(Fig. 19(b)). We run a simple experiment to verify if the intervals in these transactions are due to Suspend feature of the smartphone. We use the `Facebook` application and manually put the de-

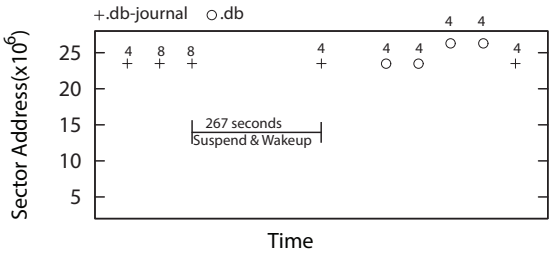
vice into the Suspend mode. Even though the device is suspended for a few second, there exists 267 sec interval in an SQLite transaction.




(a) Excessive Latency in User 3




(b) Excessive Latency in User 4




(c) Excessive Latency synthically created

**Figure 19: SQLite transactions with Excessive Latency**

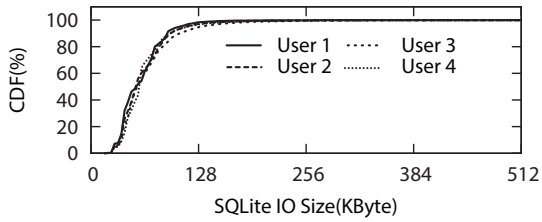



**Figure 20: IO Size per SQLite Transaction**

## 4.3 Transaction Volume

The total IO volume per transaction is an important measure on how the smartphone applications use SQLite. We measure the total IO size for each SQLite transaction. Fig. 20 and Table. 3 show the CDF and quantiles of IO size per SQLite transaction, respectively. The total IO size per SQLite transaction includes the IO's to SQLite journal file, SQLite database and the EXT4 journal.

The median size in a single SQLite transaction is 56 KByte in user 1, user 2, user 3 while it is 60 KByte in user 4. The average size in a single SQLite transaction are 62.2 KByte, 63.6 KByte, 68.5 KByte and 64.9 KByte in user 1, user 2, user 3 and user 4, respectively. However, we find that some transactions are prohibitively large size.

**Table 3: Quantiles of SQLite Transaction IO Size(KByte)**

| | Med. | Avg | 75% | 99% | 99.9% | Max |
|---|---|---|---|---|---|---|
| User 1 | 56 | 61.2 | 76 | 144 | 280 | 12928 |
| User 2 | 56 | 63.6 | 80 | 172 | 332 | 5356 |
| User 3 | 56 | 68.5 | 80 | 212 | 428 | 17880 |
| User 4 | 60 | 64.9 | 76 | 188 | 312 | 11364 |

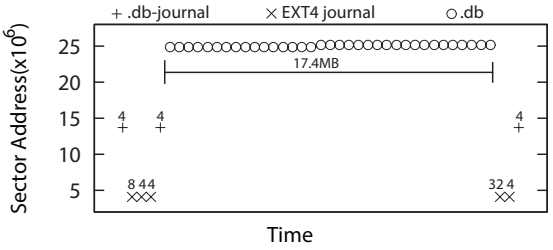



**Figure 21: SQLite maximum size in User 3**

In Table 3, we observe that a transaction inserts 17.5 MByte to the database file (user 3). We take a detailed look on this transaction of size 17.5 MByte. Fig. 21 illustrates the real IO trace for this transaction.

This transaction is issued by `Smarter Subway` application. This application shows the shortest subway route between two stations. It is found that the transaction initializes the map database. We also observe 10.9 MByte insert in user 4. Ironically, this transaction is also for initializing the database in `Subway` application.

The minimum size of an SQLite transaction is observed to be 16 KByte. In PERSIST mode with FULL synch, `fdatasync()` is called four times. If the size of the rollback journal file and the size of the database file remain the same, each `fdatasync()` call flushes as small as one page. The smallest SQLite transaction creates 16 KByte of IO.

## 4.4 Efficiency of a Transaction

Database transaction consists of the IO's not only to the database file but also to the roll-back journal file as well as the EXT4 journal. The file-backed journaling mechanism of SQLite involves relatively large number of file IO's. The overhead of synchronizing the file involves the journaling activity in the underlying filesystem. We examine the size of IO's to filesystem journal, SQLite journal and SQLite database in an SQLite transaction. This analysis allows us to understand the efficiency of an SQLite transaction.

According to our analysis, an SQLite transaction is under extreme inefficiency. Median IO's to the database file corresponds to 12 KByte. A transaction writes three 4 KByte pages to the database file. Given that the database header page is always updated whenever there is an update in the B+ tree, the two pages in the B+-tree are updated on the average. An SQLite transaction writes 24 KByte to rollback journal in median. The amount IO's to roll-back journal file is twice as much as the amount of IO's to the database file.

EXT4 journal IO accounts for 16 KByte in a transaction (median). To update 12 KByte in the SQLite database table, total 40 KByte of SQLite journal file and EXT4 journal is written to the storage. The overhead is more than 300%. From the total IO size in a transaction, less than 25% of the entire IO's are for the database file. Fig. 22, Table 4 and Fig. 23 summarize the CDF and the result of the analysis of size for each block type, respectively.

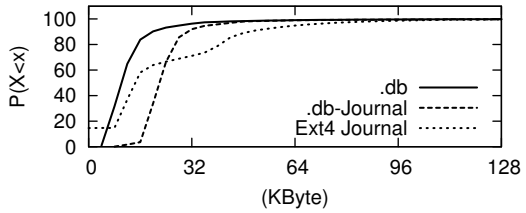



**Figure 22: SQLite filetype**

**Table 4: Quantiles of SQLite Transaction IO Type(KByte)**

| | Med. | 75% | 99% | 99.9% | Max |
|---|---|---|---|---|---|
| .db | 12 | 16 | 64 | 152 | 17816 |
| .db-journal | 24 | 28 | 68 | 172 | 5120 |
| EXT4 journal | 16 | 40 | 104 | 172 | 844 |

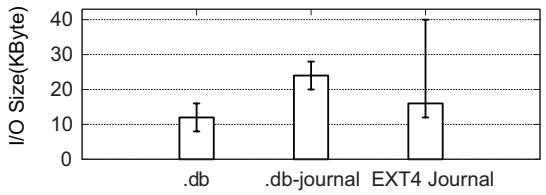



**Figure 23: Median size of SQLite IO file type. Lower bound is $1_{st}$ quantile, Upper bound is $3_{rd}$ quantile**

Android IO stack has gradually evolved in a direction to reduce the SQLite related IO's. Android switched its file synchronization option from `fsync()` in Gingerbread [1] to `fdatasync()` in Lollipop [2]. This is to reduce the amount of metadata journaled by the EXT4 filesystem. The default SQLite journaling mode has evolved: from DELETE mode(default before Android 4.0.4) to TRUNCATE mode (default since Android 4.0.4 [12]) and to PERSIST mode. Since the PERSIST mode recycles the existing journal file and does not require an allocation of a new file block, it can dispense with expensive filesystem journaling in the logging phase. The current stock SQLite adopts PERSIST mode with `fdatasync()` as default option.

PERSIST journaling mode combined with the `fdatasync()` synchronization option yields the most efficient IO behavior in Fig. 6. Despite these efforts, it is found that the SQLite transaction is still subject to severe inefficiency. Please refer to Lee et.al. [23] for further IO analysis under different SQLite journal modes.

One of the dominant causes for this inefficiency is due to the fragmented log organization. In logging for roll-back journal, SQLite writes the log header, a set of page number, the updated database pages and checksums. The 24 Byte log header, 4 Byte page number, 4 Kbyte database page and 4 Byte checksum for each undo log as shown in Fig. 3 are written with `write()` system call. Since an SQLite transaction updates only two or three database pages (12 KByte), the overhead of writing 24 Byte log header, 4 Byte page number and 4 Byte checksum for each undo log becomes rather significant. Header embedding [23] can effectively resolve this inefficiency. EXT4 filesystem journaling also accounts for significant amount of IO's in a transaction.

**Table 5: Total IO size of SQLite Transaction (GByte)**

| | .db | .db-journal | EXT4 Journal |
|---|---|---|---|
| User 1 | 3.3 | 6.0 | 5.6 |
| User 2 | 4.4 | 8.1 | 9.0 |
| User 3 | 13.4 | 22.7 | 20.9 |
| User 4 | 12.2 | 21.5 | 19.9 |

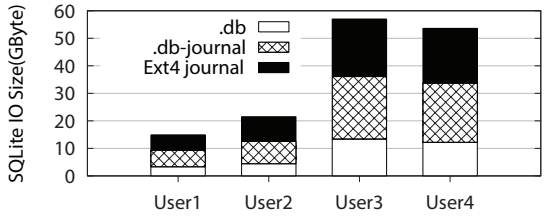



**Figure 24: Total IO size of IOs on SQLite Transaction (GByte)**

Table 5 and Fig. 24 illustrate the cumulative IO size for SQLite transactions in six month's trace. The cumulative IO sizes for SQLite transactions varies widely nearly by factor of four across the users (15 GByte in user 1 vs. 60 GByte in user 3). However, the ratio of database file IO against the total SQLite IO remains the same. The IO's for database file accounts for less then 1/4 of the entire IO's in the SQLite transaction.

## 4.5 Applications and SQLite

Smartphone applications name the SQLite database and SQLite journal file subject to the name of the application. By examining the file name in a SQLite transaction, we can identify the application which has issued the respective transactions. For each of four users, we identify top three applications which issues the SQLite transactions the most. Table 6 illustrates the result.

Messenger application, `KakaoTalk`[1], appears in top three applications in user 1 and user 3. The `prefs` database is one of the top three databases. `prefs` is an SQLite database used by the Facebook application to store the user preference data. It is synchronized with the server in the background. Other non-user applications such as `alarms` from Clock, `rmq` from Google Play services or `es0` from Google+ services also in the top frequently access in all of the four users.

The SQLite transaction counts are extremely skewed. `KakaoTalk` and `Facebook`, which account for dominant fraction of the SQLite transactions, are the two applications

[1]It is the most popular smartphone messenger appication in Korea

**Table 6: Top 3 database name SQLite Transaction in terms of count and volume. Trans: Transaction, Cnt: Count**

| | User1 | | | | User2 | | | |
|---|---|---|---|---|---|---|---|---|
| Rank | Trans Cnt (%) | .db (GB) | .journal (GB) | EXT4 (GB) | Trans Cnt (%) | .db (GB) | .journal (GB) | EXT4 (GB) |
| 1 | KakaoTalk (58%) | 2.0 | 3.7 | 3.2 | prefs (19%) | 0.9 | 1.7 | 1.6 |
| 2 | alarms (10%) | 0.2 | 0.5 | 0.4 | rmq (9%) | 0.2 | 0.6 | 0.6 |
| 3 | Cookies (6%) | 0.2 | 0.2 | 0.5 | google_analytics (6%) | 0.2 | 0.5 | 0.7 |
| | User3 | | | | User4 | | | |
| Rank | Trans Cnt (%) | .db (GB) | .journal (GB) | EXT4 (GB) | Trans Cnt (%) | .db (GB) | .journal (GB) | EXT4 (GB) |
| 1 | prefs (27%) | 3.4 | 6.2 | 5.2 | iu.upload (23%) | 3.1 | 5.4 | 3.1 |
| 2 | KakaoTalk (19%) | 2.2 | 4.1 | 4.1 | prefs (10%) | 1.3 | 2.3 | 2.3 |
| 3 | alarms (10%) | 0.6 | 1.6 | 1.3 | es0 (9%) | 1.3 | 2.2 | 1.4 |

account for major SQLite transactions. `KakaoTalk.db` from `KakaoTalk` application accounts for 58% of total transactions in user 1. In user 3 device, `Facebook` and `KakaoTalk` account for 46% total transactions as shown in Table 6.

# 5. RELATED WORKS

Journaling of journal anomaly is the main cause for excessive IO in Android IO stack [12]. Many researchers conduct their efforts to solve this problem and obtain some good results. Kim et al. [18] embeds journaling logs into database file by adopt Multi-version B+-tree [3] and exhibit 70% and 1220% performance gains against WAL mode and TRUNCATE mode, respectively. Lee et al. [23] modifies WAL log file structure by using header embedded technique, adopts DIRECT IO write with Group Synch and obtains 5x performance gain against WAL mode. Park et al. [26] remove duplicated writes at database checkpoint in WAL mode and gains 17% performance on average. Shen et al. [28] eliminate metadata journaling at system level and exhibit 7% performance gain.

Along with eliminating Journaling of journaling anomaly, other researchers adopt difference technique for faster SQLite transactions performance. Using three techniques such as eliminating unnecessary metadata journaling, external journaling and poll-based IO, Jeong et al. [12] yields 130% performance in SQLite transactions. Oh et al. [25] adapts the technique calls `per-page-logging` (PPL) for mobile data management using phase change memory and yields 8.25x and 16.54x performance gains against WAL and DELETE modes, respectively. Kang et al. [13] proposed a novel transactional FTL called `X-FTL` to offload the burden of guaranteeing the atomicity in SQLite transaction, take advantage of Copy-On-Write and implement on an SSD therefore improve SQLite transaction throughput. SQLite transactions can further be optimized by putting SQLite journaling information to NVRAM[15, 21].

There have been fair amount of characterization study for SQLite IO in Android platform. Kim et al. [17] declares that over 75% of `fsync()/fdatasync()` is from SQLite in Tizen based smartphone. Jeong et al. [12] claims that 90% of write requests are to SQLite database and journal in Android based platform while Lee et al. [22] reveal that 70% of write operations are 4KB which mostly from SQLite. There are a number of tools using to collect block trace such as [7, 8, 10, 16, 19, 32].

# 6. CONCLUSION

This work is dedicated to acquire in-depth understanding on the IO characteristics of the SQLite DBMS. SQLite is arguably the most widely used DBMS in modern computing platform, understanding on which lie far from being satisfactory. We collect the block level IO trace from four volunteers for six month.

Elaborate transaction mining algorithm allows us to identify the individual SQLite transactions from raw block IO traces, which is essentially an interleaved mixture of file IO's issued from concurrently running multiple smartphone applications. Among the various findings on the IO characteristics obtained in this study, we like to list the four key observations. First, SQLite performs a transaction in extremely inefficient manner where the SQLite journaling overhead and EXT4 filsystem journaling overhead accounts for over 75% of the entire IO volume. Second, suspend and wakeup feature of the smarphone may leave the SQLite transaction under extreme delay. Suspend feature of the smartphone stores the state of the machine to the device, which can be safely resumed in wakeup. As a result, SQLite transaction can stay alive across the suspend and wakeup, which leaves the transaction under up to few minutes latency for completion. Simple abort and retry may shorten the transaction latency. Third, the SQLite transaction is used to store extremely large amount of data, e.g. 17.5 MByte, which we carefully believe SQLite is not designed to handle. Special care needs to be taken to exploit the SQLite transaction in right way to avoid any egregious inefficiency. Fourth, the frequent `fsync()` and `fdatasync()` call in SQLite transaction leaves the EXT4 journaling under extreme inefficiency.

The result of this study is rather open-ended. This study presents the issues which deserve the special attention from the community. We hope that the results of this study is bifurcated to a number of efforts to make the smartphone IO stack more efficient.

# 7. ACKNOWLEDGEMENT

This work is supported by the ICT R&D program of MSIP/IITP.[R0601-15-1063, Software Platform for ICT Equipments], under the ITRC support program (IITP-2015- H8501-15-1006) by the IITP and by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device).

# 8. REFERENCES


[1] Android.com. Android 2.3 gingerbread. http://developer.android.com/about/versions/android-2.3-highlights.html.

[2] Android.com. Android 5.0 lollipop. http://developer.android.com/about/versions/lollipop.html.

[3] Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. An asymptotically optimal multiversion b-tree. *The VLDB Journal—The International Journal on Very Large Data Bases 5*, 4 (1996), 264–275.

[4] Bog, A., Sachs, K., and Plattner, H. Interactive performance monitoring of a composite oltp and olap workload. In *Proc. of ACM SIGMOD 2012 International Conference on Management of Data* (Scottsdale, Arizona, USA, 2012), pp. 645–648.

[5] Comer, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[6] Cruz, F., Maia, F., Matos, M., Oliveira, R., Paulo, J., Pereira, J., and Vilaça, R. Met: workload aware elasticity for nosql. In *Proc. of ACM EuroSys 2013 European Conference on Computer Systems* (Prague, Czech Republic, 2013), pp. 183–196.

[7] Godard, S. iostat. http://linux.die.net/man/1/iostat.

[8] Gregg, B. iotop. http://linux.die.net/man/1/iotop.

[9] Haldar, S. *SQLite Database System Design and Implementation*. Sibsankar Haldar, 2015.

[10] Jens Axboe, A. D. B., and Scott, N. blktrace. http://linux.die.net/man/8/blktrace.

[11] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. Androstep: Android storage performance analysis tool. In *Software Engineering (Workshops)* (2013), pp. 327–340.

[12] Jeong, S., Lee, K., Lee, S., Son, S., and Won, Y. I/O stack optimization for smartphones. In *Proc. of USENIX ATC 2013 Annual Technical Conference* (Berkeley, CA, USA, 2013).

[13] Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., and Min, C. X-ftl: transactional ftl for sqlite databases. In *Proc. of ACM SIGMOD 2013 International Conference on Management of Data* (New York, NY, USA, 2013), pp. 97–108.

[14] Kim, H., Agrawal, N., and Ungureanu, C. Revisiting storage for smartphones. *Trans. Storage 8*, 4 (Dec. 2012), 14:1–14:25.

[15] Kim, J., Min, C., and Eom, Y. Reducing excessive journaling overhead with small-sized nvram for mobile devices. *Consumer Electronics, IEEE Transactions on 60*, 2 (2014), 217–224.

[16] Kim, J.-M., and Kim, J.-S. Androbench: Benchmarking the storage performance of android-based mobile devices. In *Frontiers in Computer Education*. Springer, 2012, pp. 667–674.

[17] Kim, M., Lee, S., and Won, Y. Io workload characterization of tizen based consumer electronics. In *Proc. of IEEE ISCE 2014 International Symposium on Consumer Electronics* (2014), IEEE, pp. 1–4.

[18] Kim, W.-H., Nam, B., Park, D., and Won, Y. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *Proc. of USENIX FAST 2014 Conference on File and Storage Technologies* (Santa Clara, CA, USA, 2014).

[19] Kranenburg, P. strace. http://linux.die.net/man/1/strace.

[20] Kumar, R., Gupta, N., Charu, S., Jain, K., and Jangir, S. K. Open source solution for cloud computing platform using openstack. *International Journal of Computer Science and Mobile Computing 3*, 5 (2014), 89–98.

[21] Lee, E., Bahn, H., and Noh, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proc. of USENIX FAST 2013 Conference on File and Storage Technologies* (San Jose, CA, USA, 2013), pp. 73–80.

[22] Lee, K., and Won, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proc. of ACM EMSOFT 2012 International Conference on Embedded software* (Tempere, Finland, 2012), pp. 23–32.

[23] Lee, W., Lee, K., Son, H., Kim, W.-H., Nam, B., and Won, Y. Waldio: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proc. of USENIX ATC 2015 Annual Technical Conference* (Santa Clara, CA, USA, 2015).

[24] Lim, E., Lee, S., and Won, Y. Androtrace: framework for tracing and analyzing ios on android. In *Proc. of ACM INFLOW 2015 Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (Monterey, California, USA, 2015).

[25] Oh, G., Kim, S., Lee, S.-W., and Moon, B. Sqlite optimization with phase change memory for mobile applications. *In Proc. of the VLDB Endowment 8*, 12 (2015), 1454–1465.

[26] Park, D., and Shin, D. Removing duplicated writes at db checkpointing with file system-level block remapping. In *Proc. of ACM International Conference on Computing Frontiers 2015* (2015), ACM, p. 37.

[27] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of ACM SoCC 2012 Symposium on Cloud Computing* (San Jose, CA, USA, 2012), p. 7.

[28] Shen, K., Park, S., and Zhu, M. Jjournaling of journal is (almost) free. In *Proc. of USENIX FAST 2014 Conference on File and Storage Technologies* (Santa Clara, CA, USA, 2014).

[29] SQLite.org. Atomic commit in sqlite. https://www.sqlite.org/atomiccommit.html.

[30] SQLite.org. Well-known users of sqlite. https://www.sqlite.org/famous.html.

[31] SQLite.org. Pragma statements, 2012. http://www.sqlite.org/pragma.html#pragma_journal_mode.

[32] Ward, G. rtrace. http://radsite.lbl.gov/radiance/man_html/rtrace.1.html.