Exploiting Multi-Block Atomic Write in SQLite Transaction

Seungyong Cheon and Youjip Won
Department of Computer and Software
Hanyang University, Seoul, Korea
{chunccc|yjwon}@hanyang.ac.kr

***Abstract*—This work is dedicated to resolve the journaling overhead of widely used DBMS, SQLite. Database journaling and EXT4 filesystem journaling cause enormous write operations because of the frequent fdatasync() call and Journaling of Journal anomaly between SQLite and EXT4 filesystem. While Write-Ahead-Logging reduces the number of fdatasync() call and write volume, JOJ and database journaling overhead is still remained. In this work, we develop a new SQLite journal mode, MAW mode. We exploit the F2FS filesystem and its Multi-block Atomic Write feature to write updated database pages atomically. MAW mode completely eliminated database journaling and keeps database consistency from unexpected system failure. MAW mode only writes the actually updated database pages, thereby generates smaller write volume. We implement MAW mode at SQLite with most recent version of F2FS. With MAW mode, database insert transaction performance is increased by 487% and write volume is decreased by 67% than stock SQLite PERSIST mode in EXT4 filesystem.**

SQLite; F2FS; Android; DBMS; Multi-block Atomic Write;

I. INTRODUCTION

With the tremendous development trend and massively growing market of smartphone in recent years, the interaction time between user and smartphone device became essential feature. Unlike huge improvement of core hardware, e.g. processor and DRAM, the storage device is still remained as bottleneck because of its physical limitation. Android IO stack is a low-level software layer which is composed of SQLite, filesystem, etc. Android applications manage their data through IO stack, therefore, to optimize Android IO stack is crucial factor of smartphone performance.

One of the well-known problem of Android IO stack is the excessive amount of write operation, and the main reason is identified as SQLite and Journaling of Journal anomaly [1, 2, 3]. Notwithstanding a number of research to optimize its behavior [1, 4, 5, 6, 7], SQLite still generates supernumerary IO because of the basic concept of database journaling. Furthermore, SQLite uses explicit synchronization system call, `fdatasync()`, to flush updated journal files and database files to disk, which causes heavy IO workload and latency.

Most of the recent Android devices use EXT4 filesystem and `PERSIST` journal mode of SQLite in default. One transaction inserting 100 B record to database needs to update 3 database pages (12 KB), however, in practice, 9 pages (36 KB) are written to disk on account of roll-back journal file record and EXT4 filesystem journaling. Moreover, `fdatasync()` is called four times to synchronize files in each roll-back journaling phase. SQLite introduced new journal mode, `WAL` [8], which is the fastest journal mode in SQLite, yet it still has the Journaling of Journal anomaly [7] and checkpoint overhead.

F2FS [9] is introduced as an optimized filesystem for flash-memory based storage and shows better SQLite performance compare with EXT4 [1]. In 2014, F2FS introduced new feature, Multi-block Atomic Write [10] which writes multiple blocks in atomic manner and shows all-or-nothing. We exploit this feature and develop a new journal mode of SQLite, `MAW` mode. `MAW` mode completely eliminates the journaling procedure by atomically writes updated data to database file. Without journaling, `MAW` mode greatly decreases the amount of write IO, thereby improves Android IO stack performance. We implement `MAW` mode at SQLite version 3.8.10.2. With `MAW` mode, compared with stock `PERSIST` mode in EXT4 filesystem, the SQLite insert transaction performance is increased by 389% and 487% in SATA AHCI SSD and PCIe NVMe SSD, respectively. In the view of the write volume, `MAW` mode writes only 33% of stock `PERSIST` mode does.

II. BACKGROUND

A. SQLite

SQLite is the server-less embedded DBMS for small-scale database management that is widely used in various platforms, e.g. Android, iOS, Tizen and desktop applications, e.g. Chrome web browser, Firefox, Adobe Acrobat Reader, Skype [11]. In particular, SQLite has a huge influence on performance of smart devices because it maintains the application records and data.

Unlike most common-use DBMS [12, 13, 14], SQLite does not have its own storage management module. SQLite uses *file* to store the database and maintain the log, thus heavily relies on the operating system and filesystem. After the database transaction has been completed, SQLite uses explicit synchronization, i.e. `fdatasync()`, for updating the database file and committing the log file to protect them against unexpected system failure.

SQLite database file consists of the database pages [15] as illustrated in Fig. 1. The size of a page is 4 KB in default. The first page consists of 100 B database header page including file change counter which should be increased after

the database file has been modified. The other pages consist of a page header, cell pointers and the cell contents. SQLite adopts B+tree [16] to manage its database table, and the B+tree node size is 4 KB in default, equal to database page size. B+tree node information is stored in the page header. The cell pointer is 2 B integer and indicates the cell content which stores the actual database record.

B. SQLite Journaling

SQLite adopts six journal modes for database consistency and durability: `DELETE`, `TRUNCATE`, `PERSIST`, `WAL`, `MEMORY` and `OFF`. The default journal mode of SQLite was `DELETE` and `TRUNCATE` mode in Android 2.3 (Ginger Bread) and Android 4.0 (IceCream Sandwich), respectively. `PERSIST` mode have been used for default journal mode since Android 4.4 (KitKat). `MEMORY` and `OFF` mode keeps the journal information in volatile memory and does not maintain journal information, respectively, thus both modes do not support database recovery. The remaining four journal modes are classified in two categories: roll-back journaling and roll-forward journaling. `DELETE`, `TRUNCATE` and `PERSIST` modes are roll-back journaling, and `WAL` mode is roll-forward journaling.

SQLite roll-back journaling has three phases: (i) logging, (ii) database update and (iii) log-reset. In logging phase, SQLite logs the unmodified database pages into journal file. The roll-back journal file consists of journal header and a number of journal records, as illustrated in Fig. 2. The journal header is 512 B and stores journal file information including magic number, record count, database page size, etc. To prevent the database recovery with invalid journal records, the record count value of journal header is filled with zero and written with journal records first, and then, the actual journal header is written. The journal record consists of original database page, page number (P#) and checksum (Ck). In database update phase, the updated database pages by database transaction are written to the database file.

In the log-reset phase, the existing journal file is invalidated to make sure that database transaction is successfully completed. The log-reset phase of three journal modes are slightly different. DELETE mode simply deletes the journal file. `TRUNCATE` mode truncates the journal file to 0 size. `PERSIST` mode keeps the journal file and just marks at the journal header that the transaction has been completed. Though these difference seems trivial, the influence on filesystem journaling and metadata update is substantial because of the Journaling of Journal anomaly [7].

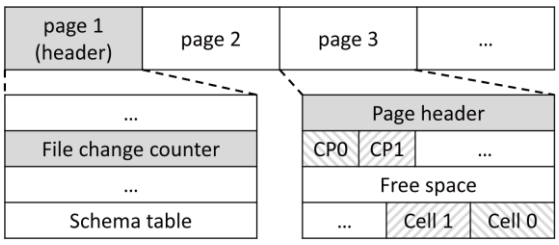


Figure 1. SQLite Database File Structure

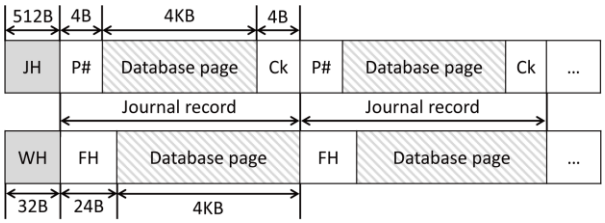
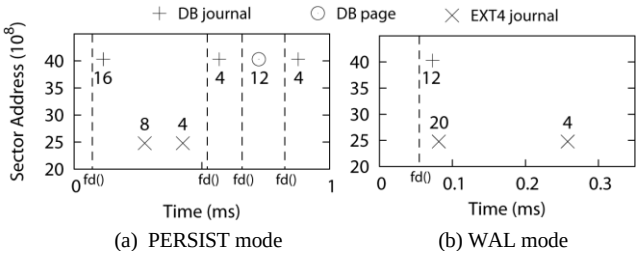


Figure 2. SQLite Roll-back Journal File and WAL File Structure, JH: Journal Header, P#: Page Number, Ck: Checksum, WH: WAL Header, FH: WAL Frame Header.

Journaling of Journal (JOJ) is well-known problem between SQLite journal and EXT4 filesystem journal, and causes a number of additional write IO. Fig. 3(a) illustrates the block IO pattern of `PERSIST` mode in EXT4 filesystem. After logging the journal file (16KB), metadata of journal file is modified and EXT4 filesystem journals the metadata (12KB, journal descriptor + original metadata + journal commit). The JOJ overhead of `DELETE`, `TRUNCATE` and `PERSIST` mode is 32KB, 44KB and 12KB, respectively (Table 1). These differences correlate with log-reset phase. `PERSIST` mode reuses the inode and the file blocks of journal file. In logging phase, only time related metadata is updated and consequently the amount of EXT4 journaling is less than `DELETE` or `TRUNCATE` mode [5].

SQLite adopts Write-Ahead Logging (`WAL`) [8] for roll-forward journaling. In `WAL` mode, SQLite writes the updated database pages solely to the WAL file. The structure of WAL file is illustrated in Fig. 2. WAL header (WH) includes the information about WAL file, e.g. database page size, checkpoint sequence number, etc. The journal record of WAL file consists of 24 B WAL frame header (FH) and updated database page. WAL frame header indicates the information of updated database page, and includes error detection code (checksum). SQLite appends the journal records to the WAL file when the database transaction has committed. `WAL` mode reduces the amount of file write and uses only single `fdatasync()`, but EXT4 journaling overhead is greater than `PERSIST` mode (Fig. 3(b)). When the database table is closed or the number of log record has been reached to the maximum value (1,000 pages in default), SQLite performs checkpoint. In checkpoint procedure, SQLite reads WAL file from the beginning and writes updated database pages to the database file.



(a) PERSIST mode (b) WAL mode

Figure 3. SQLite Block IO Pattern on EXT4 Filesystem: Insert 100 B record to SQLite (Unit: KB), `fd()`: `fdatasync()`.

Table 1 summarizes the write IO volume of four SQLite journal modes and Table 2 shows the `PERSIST` mode IO trace from blktrace[1]. One record insert entails 3 database page update (12 KB), but SQLite journal and filesystem journal generate excessive writes. 12 KB of database update incurs 4.3×, 5.3×, 3× more write in `DELETE`, `TRUNCATE`, `PERSIST` mode, respectively. While `WAL` mode yields the smallest journaling overhead, it still writes 2× more IO because of the filesystem journaling.

III. SQLITE ON F2FS

Flash-Friendly File System (F2FS) is a log-structured file system designed for flash memory based storage device [9, 17] and merged into Linux kernel version 3.8. F2FS shows better performance than stock filesystem, i.e. EXT4, in flash memory based storage device, e.g. eMMC, SSD, etc. [1]. Fig. 4 illustrates the on-disk layout of F2FS filesystem.

The main area is divided into a smaller unit: segment. A segment consist of multiple blocks, and default size of segment and block is 2 MB and 4 KB, respectively. Each block in the segment is categorized in two types: *node* and *data*. A node block contains inode or direct pointer to data blocks. A data block contains directory or file data. All blocks and segments are wrote in append-only manner. When a block is updated, old block is just invalidated and new block is appended.

With these characteristics, SQLite performs quite different in F2FS filesystem. Fig. 5 illustrates the block IO pattern of `PERSIST` and `WAL` mode on F2FS. F2FS does not have JOJ problem, but all database file and journal file updates are followed by 4 KB node block update to point the updated data blocks. In `PERSIST` mode, journal file and database file write sequence is same with EXT4 (Fig. 3(a)), but 4KB node page is written after each `fdatasync()`. As a result, the amount of total write is 52 KB, which is larger

TABLE I. IO VOLUME IN INSERTING 100 B ON EXT4 FILESYSTEM

Journal Mode	IO Type (Unit: KB)		
	Data	Journal	Total
DELETE	32	32	64
TRUNCATE	32	44	76
PERSIST	36	12	48
WAL	12	24	36

Data: EXT4 data region, Journal: EXT4 journal region.

TABLE II. IO TRACE OF PERSIST MODE IN INSERTING 100 B ON EXT4 FILESYSTEM

Time (ms)	IO	Sector	Size
33.073	WS	403073000	24
33.074	WS	403199000	8
33.236	WS	247796880	16
33.384	WS	247796896	8
33.534	WS	403199000	8
33.674	WS	402935984	8
33.676	WS	402936000	16
33.825	WS	403199000	8

WS: Write Synchronous, Size Unit: Sector Size (512 B).

[1] blktrace. https://linux.die.net/man/8/blktrace

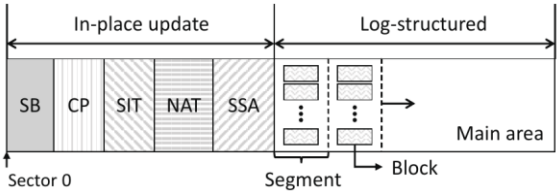


Figure 4. F2FS Filesystem On-Disk Layout, SB: Superblock, CP: Checkpoint, SIT: Segment Information Table, NAT: Node Address Table, SSA: Segment Summary Area.

than EXT4 filesystem (48 KB). In contrast, due to the elimination of JOJ, `WAL` mode only writes 16 KB with single `fdatasync()` which is much less than EXT4 filesystem (36 KB). The performance of `WAL` mode is highly improved against EXT4 filesystem, but it still has checkpoint overhead.

IV. ELIMINATE SQLITE JOURNALING WITH F2FS MAW

A. Multi-Block Atomic Write

In 2014, F2FS supports new feature called Multi-block Atomic Write [10]. This feature makes it possible that updating multiple blocks to the storage atomically in filesystem layer with simple commands. F2FS exploits `ioctl()` system call with *fd* (Fig. 7) to mark the beginning (`ioctl(F2FS_IOC_START_ATOMIC_WRITE)`) and the end (`ioctl(F2FS_IOC_COMMIT_ATOMIC_WRITE)`) of the atomic region. All write operations occurred inside the atomic region are marked as atomic page, and flushed at the end of the atomic region. The atomic pages written in an atomic region will be shown in all or nothing by F2FS recovery procedure.

In detail, when the atomic region is started, F2FS marks `FI_ATOMIC_FILE` at the file inode to indicate as atomic file. All write commands to the atomic file are just registered to the in-memory page in inode, instead of marking the page as dirty or submit block IO. Afterward, when the atomic region is ended, all in-memory pages in inode are marked as dirty and then synchronized to the storage. After writing the data pages, direct node page with atomic mark is also synchronized which will be used for F2FS roll-forward recovery procedure.

B. Providing ACID Property with MAW

SQLite adopts *Pager* module to communicate with in-

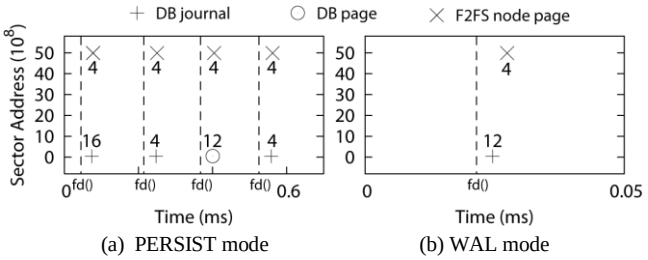


Figure 5. SQLite Block IO Pattern on F2FS Filesystem: Insert 100 B record to SQLite (Unit: KB), `fd()`: `fdatasync()`.

memory B+tree database table and on-disk, file-based database. SQLite writes database file and journal file via Pager module after database transaction has been committed. Fig. 6 illustrates the function call procedure of Pager module.

When the write transaction begins, Pager module is started and opens the journal file. During the write transaction, updated database pages are marked as dirty and inserted into dirty page list. After write transaction commit, Pager module writes the journal contents to journal file (*jfd*, journal file descriptor) and synchronizes it. After the journal contents write is completed, Pager module updates journal header record count, then call `fdatasync()` (logging phase). When the journal file is successfully synchronized, Pager module writes the dirty page lists to the database file (*fd*, database file descriptor) and synchronize it with another `fdatasync()` (database update phase). Finally, SQLite finalize the journal file according to the journal mode (log-reset phase).

Note that all function call procedures related with *jfd* in Fig. 6 are solely database journal-related operations. The journaling overhead of SQLite is excessive as described in section II.B. A severalfold amount of write volume beside updated database pages are written to disk, and more than two `fdatasync()` is called to maintain database journal. The actual database file update is writing dirty page lists in Pager commit procedure, and other procedures are just for database journaling. We exploit F2FS Multi-block Atomic Write feature to write dirty page lists in SQLite Pager module, and call it as `MAW` mode. Fig. 7 illustrates the function call procedure of `MAW` mode.

In `MAW` mode, we completely eliminate the journaling procedure with atomically write the dirty page lists. The database consistency is maintained by atomic update of database header, internal schema table and database pages. F2FS recovery procedure guarantees that atomically committed pages are shown all or nothing, it means, the database can be safely recovered from system failure without journaling, and database does not need any additional recovery procedure.

```
1 function pager_procedure(jfd, fd)
2 |   write journal contents to jfd ;
3 |   fdatasync(jfd) ;
4 |   write journal header to jfd ;
5 |   fdatasync(jfd) ;
6 |   write dirty page lists to fd ;
7 |   fdatasync(fd) ;
8 |   finalize journal file ;
9 end
```

Figure 6. SQLite Pager Module Function Call Procedure.

```
1 function MAW_procedure(fd)
2 |   ioctl(START_ATOMIC_WRITE, fd) ;
3 |   write dirty page lists to fd ;
4 |   ioctl(COMMIT_ATOMIC_WRITE, fd) ;
5 end
```

Figure 7. `MAW` Mode Function Call Procedure.

V. EXPERIMENT

A. Setup

We implement `MAW` mode in SQLite version 3.8.10.2[2]. `MAW` mode needs the recent version (at least v4.0) of F2FS which supports Multi-block Atomic Write feature, for that reason, we progress the test in Ubuntu 16.04.1 LTS (Linux kernel 4.8.0). To observe the performance of SQLite in flash-based storage device, we examine the performance on two types of Solid-State Drive (SSD). We use Samsung 840 pro (256GB, AHCI, SATA) for mainstream SSD and Samsung 950 pro (256GB, NVMe, PCIe) for high-end SSD. We examine the SQLite performance with `PERSIST`, `WAL` and `MAW` mode in EXT4 and F2FS filesystem. We use MobiBench[3] to generate SQLite workload, and measure the database transactions per second (TPS) and IO volume.

B. Transactions per Second

We examine the SQLite transaction processing speed (transaction per second). Each transaction has exactly one operation (insert, update, delete) for 100 B of string record and autoincrement primary key. We generate 10,000 transactions for each operation and measure average transaction per second. Fig. 8 demonstrates the result.

With `MAW` mode, compared with EXT4 filesystem, the performance of database insert transaction is increased by 389% and 75% than EXT4 `PERSIST`, `WAL` mode, respectively, in 840 pro. The performance improvement becomes remarkable in faster device. In 950 pro, MAW mode is faster than EXT4 PERSIST and WAL mode by 487% and 105%, respectively. `MAW` mode is even the fastest mode among the journal modes in F2FS filesystem. Compared with F2FS `PERSIST` and `WAL` mode, the insert transaction performance is increased by 275% and 3% in 840 pro, 283% and 1% in 950 pro, respectively.

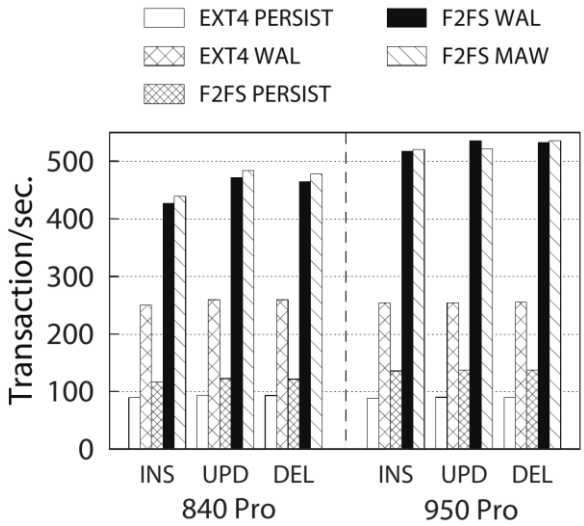


Figure 8. Transactions per second of SQLite in two SSDs.

[2] `MAW` source code. https://github.com/chunccc/MAW
[3] MobiBench. http://www.mobibench.co.kr

TABLE III. IO VOLUME IN INSERTING 100 B RECORD

Journal Mode	IO Type (Unit: KB)		
	SQLite	Filesystem	Total
EXT4 PERSIST	36	12	48
EXT4 WAL	12	24	36
F2FS PERSIST	36	16	52
F2FS WAL	12	4	16
F2FS MAW	12	4	16

SQLite: SQLite database and journal file IO, Filesystem: filesystem metadata or journaling IO.

C. IO Volume

We examine the write volume that each insert/update/delete transaction issues. We use blktrace to trace the issued IO to storage and analyze the amount of write that each transaction generates. Table 3 summarizes the amount and type of IO.

The necessary amount of write for 100 B record insert is 12 KB (database header, database page and internal schema table). However, `PERSIST` mode writes 20 KB (journal header, original database page and journal commit) to journal file in logging phase, 4 KB (reset journal header) to journal file in log-reset phase, and EXT4 filesystem writes 12 KB of JOJ. `WAL` mode reduces the amount of file write and uses only single `fdatasync()`, but EXT4 journaling overhead is greater than `PERSIST` mode. F2FS does not have JOJ problem, however, when `fdatasync()` is called, the node page should be updated with data page for filesystem consistency. `PERSIST` mode in F2FS, the filesystem overhead is larger than EXT4 because of writing 16 KB of node page (4 `fdatasync()`). `WAL` and `MAW` modes write 12 KB data page and 4KB node page.

Though `WAL` mode shows great performance and the smallest IO volume, it still has some disadvantages which are non-negligible. `WAL` mode writes updated database pages to WAL file up to 1,000 pages, then executes checkpoint which accompanies extra read and write operations. Moreover, each database connection creates its own WAL file to record roll-forward logs, and the WAL file would be grown to the maximum size (4 MB), thereby cause storage space overhead. `WAL` mode also has the consistency problem with multi-database transaction [8].

VI. CONCLUSION

Optimizing Android IO stack is vital to improve the performance of smartphone. In this work, we completely eliminate the journaling overhead of SQLite by exploiting Multi-block Atomic Write feature of F2FS filesystem. We implement `MAW` mode in SQLite. `MAW` mode accomplishes 487% of performance gain against stock `PERSIST` mode in EXT4 filesystem. The IO volume of `MAW` mode is reduced by 67% than EXT4 `PERSIST` mode. As a result, `MAW` mode relieves the excessive write request of Android IO stack and improves the performance of SQLite transaction. We also expect the extension of flash memory life span with decreased amount of write. As a future work, we try to implement Multi-File Atomic Write to guarantee the atomicity of multi-database transaction. Also, the hardware supporting Multi-block Atomic Write would draw additional performance gain.

ACKNOWLEDGMENT

This work is supported by IT R&D program MKE/KEIT [No. 10041608, Embedded System Software for New-memory based Smart Device], and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) [R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services].

REFERENCES

[1] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," Proc. USENIX Annual Technical Conference (ATC '13), USENIX, June 2013, pp. 309-320.

[2] M. Kim, S. Lee, and Y. Won, " IO workload characterization of Tizen based consumer electronics," Proc. IEEE International Symposium on Consumer Electronics (ISCE 2014), IEEE, June 2014, pp. 1-4, doi: 10.1109/ISCE.2014.6884550.

[3] K. Lee, and Y. Won, "Smart layers and dumb result: IO characterization of an Android-based smartphone," Proc. ACM International Conference on Embedded Software (EMSOFT '12), ACM, Oct. 2012, pp. 23-32, doi:10.1145/2380356.2380367.

[4] W. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in Android I/O: Multi-version B-tree with lazy split," Proc. USENIX Conference on File and Storage Technologies (FAST '14), USENIX, Feb. 2014, pp. 273-285.

[5] W. Lee, K. Lee, H. Son, W. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," Proc. USENIX Annual Technical Conference (ATC '15), USENIX, July 2015, pp. 235-247.

[6] G. Oh, S. Kim, S. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," VLDB Endowment, vol. 8, pp. 1454-1465, Aug. 2015.

[7] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," Proc. USENIX Conference on File and Storage Technologies (FAST '14), USENIX, Feb. 2014, pp. 287-293.

[8] SQLite.org, Write-Ahead Logging, https://www.sqlite.org/wal.html.

[9] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," Proc. USENIX Conference on File and Storage Technologies (FAST '15), USENIX, Feb. 2015, pp. 273-286.

[10] J. Kim, F2FS: support atomic_write feature for database, https://lkml.org/lkml/2014/9/26/19.

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

[12] W. Effelsberg, and T. Haerder, "Principles of database buffer management," ACM Transactions on Database Systems (TODS), vol. 9, pp. 560-595, Dec. 1984.

[13] T. Lang, C. Wood, and E. B. Fernández, "Database buffer paging in virtual storage systems," ACM Transactions on Database Systems (TODS), vol. 2, pp. 339-351, Dec. 1977.

[14] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database system concepts. 4th ed., McGraw-Hill, 1997.

[15] S. Haldar, "SQLite Database System Design and Implementation," Sibsankar Haldar, 2015.

[16] D. Comer, "Ubiquitous B-tree," ACM Computing Surveys (CSUR), vol. 11, pp.121-137, June 1979.

[17] M. Rosenblum, and J. K. Ousterhout, "The design and implementation of a log-structured file system," ACM Transactions on Computer Systems (TOCS), vol. 10, pp. 26-52, Feb. 1992.