Atomic Multi-database Transaction of WAL Journaling Mode in SQLite

Sungmin Kim, Minseok Kim, Dam Quang Tuan, Youjip Won

Dept. of Computer Software, Hanyang University, Seoul, Korea

patch913@hanyang.ac.kr, orc1226@hanyang.ac.kr, damquangtuan@hanyang.ac.kr, yjwon@hanyang.ac.kr

Abstract— **This works is to propose a solution for multi-database atomicity problem of WAL journaling mode in SQLite. SQLite is the most widely deployed and used DBMS in mobile system [1]. SQLite has several journaling modes. WAL (Write-Ahead Logging) is the one of those journaling modes included from version 3.7.0. WAL is significantly faster in most scenarios and provides more concurrency as reading and writing can proceed concurrently. However, transactions that involve changes with multiple attached databases do not guarantee atomicity across all databases as a set [2]. We modify transaction and recovery procedure of WAL to solve this problem. This work consists of three parts: (i) Enabling Use of Master Journal for WAL. (ii) Creation of 'mj-stored' File in Multi-Database Transaction. (iii) Rollback with 'mj-stored' File. In multi-database transaction, we create 'mj-stored' file for each WAL file to save the state before the transaction begins and the master journal file name. If crash occurs during the transaction, database with WAL journaling mode try to find the 'mj-stored' file and corresponding master journal file to roll back to the state before the transaction began in recovery time. With this solution, multi-database transaction with WAL journaling mode can guarantee atomicity.**

Keywords— **Atomicity, Database, Multi-database, SQLite, Write-Ahead Logging**

I. INTRODUCTION

SQLite is server-less, zero-configuration and transactional SQL database engine. SQLite is the most widely deployed and used relational database management system in the world. It is embedded in millions of mobile devices, web browsers, operating systems and applications [1].

SQLite uses the rollback journal to implement atomic commit and rollback. SQLite also supports for atomic commit for multi-database transaction. Atomic commit for multi-database transaction means that either changes with all databases occur or none of them occur [3]. This property is kept with the other journaling modes such as DELETE, TRUNCATE or PERSIST modes. However, a new 'Write-Ahead Log' referred as WAL introduced in SQLite version 3.7.0 does not guarantee atomicity in multi-database transaction. WAL has several advantages like faster operations and more concurrency, but not atomicity in multi-database transaction [2].

In journaling mode, SQLite uses the master journal file to guarantee atomicity with multi-database transaction. Each of the database participated in the transaction tries to find the rollback journal file and the master journal file to decide whether to recover the database or not. However, WAL is not possible to use this strategy in multi-database transaction in the latest release. This work proposes a solution to resolve this problem with three key ingredients.

- **Enabling Use of Master Journal for WAL**: WAL mode is not possible to create master journal file in multi-database transaction. We unlock this limit and create it with WAL mode. We write 'mj-stored' file name in master journal instead of rollback journal that WAL mode does not have.
- **Creation of 'mj-stored' File in Multi-Database Transaction:** We create a file for each database with WAL journaling mode to save the state of WAL file before the transaction begins and the master journal file name. If crash occurs during the multi-database transaction, the database uses this file to roll back from the crash to guarantee atomicity with the other databases.
- **Rollback with 'mj-stored' File:** In recovery time, WAL mode tries to find mj-stored file in the same directory as the database file. If the file is found, it checks validity of the file and find the corresponding master journal file. if the master journal file is found, it means that the previous transaction was not finished completely. WAL mode decides to roll back to the state before the transaction using 'mxFrame' written in mj-stored file.

WAL journaling mode is a better choice for many applications. However, the problem that atomicity is not guaranteed in multi-database transaction makes it not interesting to use for some developers. With this solution, we can guarantee atomicity in multi-database transaction of WAL journaling mode. Applications that data integrity is critical can use WAL journaling mode to enhance performance of the database with this solution.

This work is based on SQLite version 3.13.0.

II. WAL JOURNALING MODE

Rollback journal is used to maintain atomicity of database in unexpected situations such as power failure or network connection loss. SQLite provides six journaling modes: OFF, DELETE, TRUNCATE, PERSIST, MEMORY and WAL

modes. Write-Ahead Logging (referred as WAL) is the newest journal mode of SQLite available from version 3.7.0 [2].

A. How It Works

Unlike rollback journal mode preserves the original content in the journal file, the rollforward WAL mode preserves the original content in the database file. The WAL file is named as '-wal' attached at the end of the database file name. For example, the database named 'database.db' uses 'database.db-wal' for WAL file. Every change to the database is appended to the separate WAL file when each commit occurs. A Single WAL file can be used by multiple transactions by append each transaction's dirty pages to the end of WAL file sequentially.

As transactions commit continuously, WAL file size gets larger. The larger the size of the WAL file, the time to scan the WAL file takes longer. SQLite transfers the pages append on the WAL file to the database file. It is called 'checkpoint'. Checkpoint occurs automatically when the WAL file's page number reaches to 1000 by default [2]. Users can set when the automatic checkpoint occurs by changing the page size limit, change this default number of pages or calling the checkpoint command directly. If checkpoint threshold is small, WAL file size limit becomes smaller and SQLite can read data from WAL faster. In this case, the write transaction becomes slower as expensive checkpoint operation is executed more frequently.

When the read transaction begins, reader obtains the index number of the last valid frame in the WAL file to prevent concurrency problem. This index number is called 'mxFrame' [4]. The reader only checks the WAL frame that has a smaller index number than obtained mxFrame value that reader obtained. This technique makes every reader have its own snapshot of WAL file by ignoring WAL frames attached after the read transaction began and make the write transactions not to block read transactions. With this property, database with WAL journaling mode has substantial performance improvement compared to the rollback journaling modes.

When a reader tries to find a frame in WAL file, it scan the entire WAL file to know where the frame is located because it does not know where the target frame is located. It becomes costly if WAL file is big enough. To prevent that scanning all frames happens every time a read transaction is executed, SQLite uses a separate file called WAL-index. Because WAL-index is conceptually shared memory, it is named '-shm' attached on the database file name. When a reader wants to read a page, WAL-index finds it in the snapshot by using the page number and the mxFrame value obtained from the reader. It returns the index of the frame found or NULL in case the target frame does not exist. After the transaction aborts in case of crash, the database tries to recover when it is opened and read. WAL-index file is reconstructed using the WAL file. Each frame in WAL file is decoded and checked if the second slot of WAL frame header is zero or not. Table 1 illustrates the structure. This slot means the database size after the commit if it is commit frame. For the other frames, it is zero. This slot is used to ignore the frames that are not to be used to guarantee atomicity of transaction.

TABLE 1. WAL HEADER FORMAT

Offset	Size	Description
0	4	Magic number. 0x377f0682 or 0x377f0683
4	4	File format version. Currently 3007000
8	4	Database page size. Example: 1024
12	4	Checkpoint sequence number
16	4	Salt-1: random integer incremented with each checkpoint
20	4	Salt-2: a different random number for each checkpoint
24	4	Checksum-1: First part of a checksum on the first 24 bytes of header
28	4	Checksum-2: Second part of the check-sum on the first 24 bytes of header

B. Advantages and Disadvantages

WAL mode has significant advantages compared to the rollback journaling modes. Write transactions do not block read transactions and vice versa. This property provides more concurrency and improves performance in multi-user or multi-threaded environment. Because all write transactions are append to the end of the WAL file, disk I/O operations tend to be sequential.

However, it has some disadvantages and restrictions. In a case that most of the transactions are read transaction and there are few write transactions, WAL is slightly slower than traditional journaling modes. Moreover, WAL-index restricts the running environment to system that shared memory is possible to use. Due to the limit, WAL cannot be used in a network file system.

In WAL journaling mode, it is possible to execute write transaction in multiple attached databases. In this case, however, WAL maintains atomicity for each single database but not each other as a single set. This multi-database problem with WAL journaling mode is the main topic of this work. Details such as the case which atomicity can be broken is treated in section 3 [2].

III. MULTI-DATABASE TRANSACTION

SQLite supports multi-database transaction by using ATTACH command. Multi-database transaction in SQLite guarantees atomicity except WAL journaling mode. This section discusses how the traditional journaling mode guarantees atomicity and why WAL journaling mode does not support for the atomic multi-database transaction.

A. Traditional Journaling Mode

1) Master Journal File: The traditional journaling modes such as TRUNCATE, PERSIST, DELETE modes satisfy the atomicity for multi-database transaction by using an additional file called 'Master Journal'. The master journal file contains the file names of each journal files used for each databases that participated in the multi-database transaction. To maintain atomicity, all of write operations are executed or none of them should be executed. If the transaction is aborted, all databases that already have been modified must roll back. SQLite

determines which database belongs to aborted transaction with the master journal. The atomic commit of the multi-database transaction in traditional journaling modes follows the sequence: (i) Create the journal file for each database and store original data in the journal file. (ii) Create a master journal file and store each journal file name in it. (iii) Write the master journal file name in each journal file. (iv) Synchronize the database file. (v) Delete the master journal file. (vi) Handle each journal file according to the journal mode of each database. For example, PERSIST mode sets journal header to zero [3].

2) Recovery: If a journal file exists at the beginning of read transaction, SQLite checks the hotness of the journal file. In case the journal file does not store a master journal file name, SQLite determines rollback is necessary or not by journal header. If the journal file has a master journal file name in it but the master journal file does not exist, SQLite handles journal file without rollback. In this case, the last transaction crashed after synchronizing all the database files and the master journal file is deleted. Conversely, if the master journal file exists, SQLite checks whether the journal header is corrupted or not. The database that has a journal with valid header rolls back. SQLite cleans up the journal file and deletes the master journal file only if all the child journals contained do not exist or not valid.

B. Problem in WAL Mode

Situation that atomicity is broken in multi-database transaction with WAL is illustrated in Figure 1.

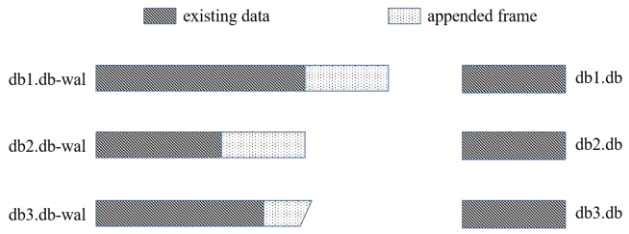

Figure 1. Sample of the multi-database transaction in WAL journaling mode that brakes atomicity

In the initial state, there are three databases with WAL journaling mode that have some frames in WAL file. In this case, we suppose that db1 has 5 frames, db2 has 3 frames and db3 has 4 frames. The write transaction inserts two frames to each database. We assume that appending the new frames to db1 and db2 completes successfully and the crash occurs during writes for db3. When each database is read again after the crash, It decides whether the database needs a rollback or not. Since db1 and db2 have already finished write operation completely, these two databases do not roll back. However, since the insertion process of db3 is not completed, it rolls back to the state before the last transaction began. Finally, db1 has 7 frames, db2 has 5 frames and db3 only has 4 frames. The write transaction that tried to inserts new frames in three databases is partially executed even after the rollback. Therefore, the atomicity between the databases are broken.

IV. ATOMIC MULTI-DATABASE TRANSACTION WITH 'MJ-STORED' FILE

A. Enabling Use of Master Journal for WAL

Master journal is the key part of multi-database transaction recovery as described in section 3. This file contains each child rollback journal file paths. However, database with WAL journaling mode does not make master journal in multi-database transaction. We unlock this limit and make master journal for it.

One different thing is that WAL mode writes 'mj-stored' file name in master journal while the rollback journaling modes write rollback journal file name. Additionally, we design a function to read master journal name in mj-stored file and let the other journaling modes use it on recovery. With this way, master journal handling strategy does not break the other journaling modes.

B. Creation of 'mj-stored' File

We design a file named 'mj-stored' to record the master journal file name and the mxFrame of WAL file before the transaction begins. Figure 2 illustrates the structure of the file. The mj-stored file consists of 4 bytes of mxFrame of the WAL file, master journal file name with various length, 4 bytes of length of the master journal file name, 4 bytes of checksum and 8 bytes of magic number. Checksum value is the summation of all bytes of the master journal file name as unsigned 32-bit integer. Checksum and magic number are used to test validity of the file.

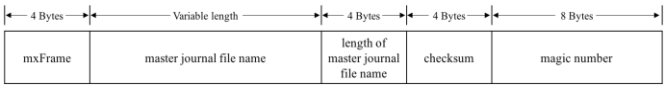

Figure 2. Structure of mj-stored file

This file is created before the new frames are appended to the WAL file, after the master journal file is created. At this point, WAL is ready to safely roll back in case of crash during the multi-database transaction. After the transaction is completely finished, mj-stored file is invalidated by zeroing the checksum and the magic number. This file is closed and deleted only at the time WAL is closed. Invalidated mj-stored file is handled as the same way as the file does not even exists. This is not to confuse the database if crash occurs during the other operations than multi-database transaction. Recovery using the invalidated mj-stored file will not be conducted even the file exists. With this way, we are able to save costs opening and deleting the file every time the multi-database transaction is executed.

C. Rollback with 'mj-stored' File

Recovery is conducted when WAL index is considered corrupted when read transaction starts. We add function to check that mj-stored file exists. If the file is found, its validation is tested with magic number and checksum. We find master journal file from the master journal file name written in the mj-stored file. If it exists, that means WAL file should roll back because the transaction did not finish

completely. If the master journal file is not found, it means that the transaction finished but crashed during the cleanup process. In this case, it does not need to roll back. It just invalidates the mj-stored file. If it decides to roll back, recovery proceeds even the WAL index is not corrupted. SQLite reads the WAL file from the beginning and checks for the valid header and frame. In this step, we force it to ignore the frames with frame number bigger than mxFrame written in the mj-stored file. After reading the WAL frames is done, WAL file is synced and mj-stored file is invalidated. The master journal file is deleted if all the child journal files or mj-stored file contained in it are invalidated or do not exist. This is the end of the recovery procedure.

V. VERIFICATION TEST

A. *Tcl Test Scripts*

SQLite provides test scripts written in Tcl. This test set includes 37,706 distinct test cases to achieve the reliability of the program [2]. We modify a few test scripts to suit for the solution. For example, the pager test regards that the database in WAL journaling mode does not use master journal while the traditional journaling mode does. We add crash recovery test for multiple databases with WAL. This test simulates crash during transaction with multiple databases. After the crash, it checks atomicity and integrity of the databases. We modify test scripts and confirm that full test passes on various platforms such as Linux or macOS. The test result shows that no error or memory leak occurs.

VI. BENCHMARK

This section discusses the performance of traditional journaling modes and WAL journaling mode with this solution. We use Mobibench on Samsung Galaxy S3(SHV-E210L). Mobibench is a benchmark tool designed to measure I/O performance of Android using SQLite [6]. We compare performance of 1,000 INSERT, UPDATE, DELETE transaction on four journaling modes each: (i) DELETE, (ii) PERSIST, (iii) original WAL, and (iv) WAL that this solution is applied.

Figure 3. The benchmark of single-database transaction for each journaling mode

Figure 3 shows that the performance of single-database transaction by how many transactions are executed per second. Because the solution does not actually use mj-stored file, the performance of WAL journaling mode with the solution does not show significant difference with the original WAL journaling mode. It still shows about 3.8X faster performance than DELETE journaling mode.

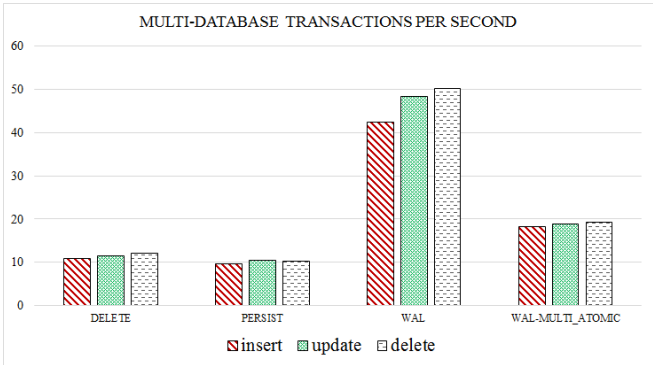

Figure 4. The benchmark of single-database transaction for each journaling mode

Figure 4 shows the performance of multi-database transaction by the same method of Figure 3. This process attaches two additional databases with the same journaling mode and does the same operation to each of them. In this case, WAL mode with the solution performs about the half of transactions per second than the original WAL mode does. The performance decline is caused by additional file I/O by the solution. However, it still shows about 2X faster performance than DELETE journaling mode.

VII. RELATED WORKS

Journaling of journal anomaly is the well-known issue for excessive IO in Android IO stack [7]. Many scholars conduct their efforts to solve this problem and obtain some good results. Kim et al. [10] obtains 70% and 1220% performance gains against WAL mode and TRUNCATE mode, respectively using the idea of embedding journaling logs into database file by adopting Multi-version B+-tree [5]. Lee et al. [12] 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. [14] remove duplicated writes at database checkpoint in WAL mode and gains 17% performance on average. Shen et al. [15] 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. [7] yields 130% performance in SQLite transactions. Kang et al. [8] 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. Oh et al. [13] 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. SQLite transactions can further be optimized by putting SQLite journaling information to NVRAM [9], [11].

VIII. CONCLUSIONS

With this solution, atomicity can be guaranteed in multi-database transaction in WAL journaling mode. Since the solution must create master journal file and additional mj-stored file, multi-database transaction is clearly slower than the original WAL. However, it is still faster than the traditional journaling modes. Moreover, atomicity is invaluable to some applications that reliability of database is critical. This solution will be considered fascinating to those applications in spite of the decline of performance.

Suitable conditions of database to use our solution are as follows: (i) Write transactions occur in high rate. (ii) Multi-database transactions occur in relatively low rate. (iii) Better performance than traditional journaling modes is required. (iv) Atomicity after the recovery in crash of multi-database transaction is critical.

ACKNOWLEDGMENT

This research was supported by the MISP(Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW supervised by the IITP(Institute for Information & communications Technology Promotion). (R7719-16-1003)

REFERENCES

[1] Most widely deployed and used database engine. https://www.sqlite.org/mostdeployed.html.
[2] Write-ahead logging. https://www.sqlite.org/wal.html.
[3] Atomic commit in SQLite. https://www.sqlite.org/atomiccommit.html.
[4] wal.c. SQLite source.
[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. The VLDB Journal-The International Journal on Very Large Data Bases, 5(4):264–275, 1996
[6] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won. Androstep: Android storage performance analysis tool. In Software Engineering (Workshops), volume 13, pages 327–340, 2013.
[7] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In Proc. of USENIX ATC 2013 Annual Technical Conference, Berkeley, CA, USA, 2013.
[8] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min. X-FTL: transactional FTL for SQLite databases. In Proc. of ACM SIGMOD 2013 International Conference on Management of Data, pages 97–108, New York, NY, USA, 2013.
[9] J. Kim, C. Min, and Y. Eom. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. Consumer Electronics, IEEE Transactions on, 60(2):217–224, 2014.
[10] W.-H. Kim, B. Nam, D. Park, and Y. Won. 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.
[11] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In Proc. of USENIX FAST 2013 Conference on File and Storage Technologies, pages 73–80, San Jose, CA, USA, 2013.
[12] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. 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.
[13] G. Oh, S. Kim, S.-W. Lee, and B. Moon. SQLite optimization with phase change memory for mobile applications. In Proc. of the VLDB Endowment, 8(12):1454–1465, 2015.
[14] D. Park and D. Shin. Removing duplicated writes at DB checkpointing with file system-level block remapping. In Proc. of ACM International Conference on Computing Frontiers 2015, page 37. ACM, 2015.
[15] K. Shen, S. Park, and M. Zhu. Journaling of Journal Is (Almost) Free. In Proc. of USENIX FAST 2014 Conference on File and Storage Technologies, Santa Clara, CA, USA, 2014.

Sungmin Kim is an undergraduate student at Division of Computer Science & Engineering, Hanyang University. From September 2014, He has been working in TEAM42, Seoul, Korea as an iOS Developer. He is interested in mobile app and database.

Minseok Kim is an undergraduate student at Division of Computer Science & Engineering, Hanyang University. From September 2016, he has been working in NEMOUX, Seoul, Korea as a Developer. He is interested in utilizing the web cloud service.

Dam Quang Tuan received MS in electronics and computer engineering from Hanyang University, Korea in August 2016. He had worked as MS student/researcher at ESOS lab, Hanyang University from 2014-2016. He is now a researcher/collaborator at Human Machine Interaction lab, University of Engineering and Technology, Vietnam National University, Ha Noi, Vietnam.

Youjip Won is currently Professor at Division of Electrical and Computer Engineering, Hanyang University, Seoul Korea. He is leading Embedded Software System Lab. He did his BS and MS in Dept. of Computer Science, Seoul National University, Seoul, Korea in 1990 and 1992, respectively. He received his Ph. D in Computer Science from University of Minnesota in 1997. Before joining Hanyang University in 1999, he worked at Intel Corp. as Server Performance Analyst. His research interests include Network Traffic Modeling, Analysis and Characterization, Multimedia system and networking, File and Storage subsystem , Lower power Storage System. In 2006, Multimedia File System project funded by Samsung Electronics was awarded "Best Academy-Industry Collaboration Practice in Samsung Electronics". In 2007, he was awarded "National Research Lab" grant which is highly selective and prestigeous governmental grant.