WALDIO: ELIMINATING THE FILESYSTEM JOURNALING IN RESOLVING THE JOURNALING OF JOURNAL ANOMALY

Wongun Lee¹, Keonwoo Lee¹, Hankeun Son¹, Wook-Hee Kim², Beomseok Nam², and Youjip Won¹

¹ Hanyang University
² Ulsan National Institute of Science and Technology
OUTLINE

• Motivation
• Background
• Eliminating the JOJ Anomaly
• Problem Assessment
• Proposal
  ▪ Preallocation in with Explicit Journaling
  ▪ Header Embedding
  ▪ Group Sync
• Durability
• Experiment
• Summary
000. Motivation

INTRO
SQLite

filesystem

Block Device Driver

Insert/update/delete

Read/write
In Smartphone, SQLite maintains everything!

Web browser

Messenger

SNS

Temp. files

cookies

URLs

SQLite is responsible for 70% of the I/O [Lee, EMSOFT 2012]
No | comments
---|---------
0  | 'Hello'
1  | 'What..'
Eliminate EXT4 Journaling from SQLite Journal IO!
Background

001. Journaling of Journal Anomaly
INSERT (PERSIST MODE)

App → INSERT a data (2 byte) → SQLite → 

- fdatasync() 
- fdatasync() 
- fdatasync() 

Time

Put journal header → commit undo log 
Write magic number → Update database. 
Write SQLite journal file → Write database file → fdatasync() 

reset log

Wongun Lee, et.al. Hanyang University
EXT4 JOURNALING (ORDERED MODE)

Minimum 12 KByte!!

Journal area

Data area

Metadata area

write() fsync()

Time

data

TxB

journal record

TxE
JOURNALING OF JOURNAL ANOMALY (PERSIST MODE)

Inserting ‘Hi’ (2 Byte) → 40 KByte to storage !!!

- App
- SQLite
- EXT4

Time

- Write journal file
- Write database file
- fdatasync ()

Wongun Lee, et.al. Hanyang University
USENIX ATC, Santa Clara, USA, 2015, JULY 9
IO ACCESSES IN SQLITE FOR AN INSERT

DELETE mode: INSERT

WAL mode: INSERT

TRUNCATE mode: INSERT

PERSIST mode: INSERT
002: Resolving the Journaled Anomaly Approach
ALLOCATING A BLOCK IN WRITE

Write block 3: Non-Allocating write, No metadata update

Write block 9: Allocating write, metadata updated
WRITING A BLOCK TO THE STORAGE:
VIA FSYNC()
WRITING A BLOCK TO THE STORAGE: VIA \texttt{FDATASYNC()} : ALLOCATING WRITE
WRITING A BLOCK TO THE STORAGE:
VIA `fdatasync()` : NON ALLOCATING WRITE

- User buffer
- Page cache
- Storage
  - EXT4 journal region
  - EXT4 data region
DIRECT IO

- Bypass the page cache.
- Write directly to the storage.
- Write to writeback cache at the storage.
- IO should be sector aligned.
WRITING A BLOCK TO THE STORAGE VIA DIRECT IO: NON-ALLOCATING WRITE

User buffer

Page cache

Storage

EXT4 journal region

EXT4 data region

DIO write
For Non-Allocating write,
DIO write == buffered write() + fdatasync()
We use Direct IO for log commit in SQLite!!

INSERT

**SQLite**

Buffered `write()` to journal

**EXT4**

`fsync()`

**Journal area**

LBA

Data area

Time

**VS**

Direct `write()` to journal

Direct IO

**LBA**

Time

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9
003. Problem Assessment
ISSUES IN USING DIO WRITE FOR SQLITE JOURNALING

• Issue 1
  Avoid metadata update in SQLite log commit.

• Issue 2
  Make the SQLite IO size direct IO compatible and block device friendly.

• Issue 3
  Make the DIO write efficient and flash memory friendly.
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

Allocating Buffered write  ➔  Non-Allocating DIO write

User buffer

Page cache

Storage

write()

metadata

DIO write + fdatasync()

fdatasync()

EXT4 journal region

EXT4 data region

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9
ISSUE 3. MAKE THE DIO WRITE MORE EFFICIENT

User buffer

Page cache

Storage

write()

fsync()

DIO write

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9
004. Preallocation with Explicit Journaling
 ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

PREALLOCATE WITH EXPLICIT JOURNALING

SQLite

OS

Storage

fallocate() uninitialized block
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

PREALLOCATION WITH EXPLICIT JOURNALING

SQLite

OS

Storage

Initialized Block
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

PREallocation WITH EXPLICIT JOURNALING

fdatasync()
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

INITIALIZING THE AlLOCATED BLOCKS

SQLite

OS

Storage

fallocate(

uninitialized block
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

INITIALIZING THE ALLOCATED BLOCKS

SQLite

OS

Storage

falallocate() uninitialized block

Crash!

010 010 010 010
010 010 010 010
010 010 010 010
010 010 010 010

Block Initialize flag

Loss

010 010 010 010
010 010 010 010
010 010 010 010
010 010 010 010

Crash!
ISSUE 1. AVOID METADATA UPDATE IN SQLite LOG COMMIT

INITIALIZING THE_ALLOCATED BLOCKS

Preallocation with Explicit Journaling should return the initialized blocks without exposing the stale content.

SQLite

Block Initialize flag

OS

Read

Storage

uninitialized block
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

INITIALIZING THE ALLOCATED BLOCKS

• Zero-Fill the allocated blocks.

• Allocating the blocks with “initialized” flag set.

• We use discard/trim to avoid the exposure of the stale content
  – unmap the Flash pages when files are deleted (DISCARD mount option).
  – unmap the Flash pages when the blocks are allocated (discard pages within
    fallocate()).
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

DISCARD / TRIM COMMAND

OS

Flash Storage

Stale contents

Mapping Table

0101
1010
1001

0101
1010
1001

0101
1010
1001

0101
1010
1001

0101
1010
1001
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

DISCARD / TRIM COMMAND

OS

Flash Storage

Discard/Trim (1,3,5,8)

Mapping Table

Stale contents

0101 1010 1001
0101 1010 1001
0101 1001
0101 1001
0101 1001
 ISSUE1. AVOID METADATA UPDATE IN SQLite LOG COMMIT

DISCARD / TRIM COMMAND

OS

Flash Storage

Stale contents

Mapping Table

read()
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

INITIALIZING THE ALLOCATED BLOCKS

• Zero-Fill the allocated blocks.

• Allocating the blocks with “initialized” flag set.

• We use discard/trim to avoid the exposure of the stale content
  – unmap the Flash pages when files are deleted (DISCARD mount option).
  – unmap the Flash pages when the blocks are allocated (discard pages within fallocate()).
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 2. ALLOCATING INITIALIZED BLOCK WITH DISCARD MOUNT

Mount with discard option
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 2. ALLOCATING INITIALIZED BLOCK WITH DISCARD MOUNT

Remove

discard / trim

Process

OS

Storage

file

01010 10101 01010 01010 01010 01010 01010 01010
01100 01010 01010 01010 01010 01010 01010 01010
01010 01111 01111 01111 01111 01111 01111 01111

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 2. ALLOCATING INITIALIZED BLOCK WITH DISCARD MOUNT

---

discard / trim

---

file

Process

OS

Storage
ISSUE1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

INITIALIZING THE AlLOCATED BLOCKS

• Zero-Fill the allocated blocks.

• Allocating the blocks with “initialized” flag set.

• We use discard/trim to avoid the exposure of the stale content
  – unmap the Flash pages when files are deleted (DISCARD mount option).
  – unmap the Flash pages when the blocks are allocated (discard pages within
    fallocate()).
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 3. ALLOCATING INITIALIZED BLOCK WITH DISCARD COMMAND
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 3. ALLOCATING INITIALIZED BLOCK WITH DISCARD COMMAND

```c
fallocate(file A)
```
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 3. ALLOCATING INITIALIZED BLOCK WITH DISCARD COMMAND

File A

OS

Discard/Trim (1,3,5,8)

Flash Storage

Mapping Table

010100
101010
100101
010100
010101
010010
010100
010101
010010

010100
010101
010010
010100
010101
010010
010100
010101
010010

010100
010101
010010
010100
010101
010010
010100
010101
010010

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9
ISSUE 1. AVOID METADATA UPDATE IN SQLITE LOG COMMIT

APPROACH 3. ALLOCATING INITIALIZED BLOCK WITH DISCARD COMMAND

OS

File A

Flash Storage

Mapping Table

read()
005. Header Embedding
ISSUE 2. MAKE THE SQLITE IO SIZE DIRECT IO COMPATIBLE

IO SIZE VS. IO PERFORMANCE

Direct IO Performance

Aligned the IO with multiples of 4KB !!!
ISSUE2. MAKE THE SQLITE IO SIZE DIRECT IO COMPATIBLE

SQLITE JOURNAL FILE STRUCTURE

- Undo log file

- Redo log file

NOT compatible with direct IO
NOT aligned sector size !!
ISSUE 2. MAKE THE SQLite IO SIZE DIRECT IO COMPATIBLE

HEADER EMBEDDING

\[ \text{WAL Frame} \]

\[ \text{FH} \]

\[ \text{4096B} \] \[ \text{24B} \]

\[ \text{HEADER EMBEDDING} \]

\[ \text{WAL Frame} \]

\[ \text{FH} \]

\[ \text{4096B} \] \[ \text{24B} \]
006. Group Synchronization
ISSUE3. MAKE THE DIO WRITE EFFICIENT

DIO WRITE IS INEFFICIENT

OS merges the IO in buffered IO.

Direct IO write cannot be merged.
ISSUE 3. MAKE THE DIO WRITE EFFICIENT

GROUP SYNCHRONIZATION

SQLite
Frame Buffer
16 KByte

Flash Storage

WAL file

LC  LC  LC  LC

LC: log commit
ISSUE 3. MAKE THE DIO WRITE EFFICIENT

GROUP SYNCHRONIZATION

SQLite

Frame Buffer
16 KByte

Group Sync
2 msec group interval

Flash Storage

WAL file

LC  LC  LC  LC  LC: log commit
007. Durability
DURABILITY GUARANTEE OF BUFFERED IO
DURABILITY OF DIRECT IO

HOST

Page cache

DIO write()

Flash Storage

Writeback cache

NAND Flash

Volatile

Durable
TYPE 1. USE OF FULL SYNC

HOST

Page cache

buffered write()

DIO write()

Flash Storage

Writeback cache

fdatasync()

NAND Flash

fdatasync()

Volatile

Durable
TYPE 2. PERSISTENT DIRECT IO

- We develop “Persistent Direct IO” using “reliable write” command in eMMC5.1
- Eliminating the writeback cache flush overhead.
TYPE 3. SMARTPHONE WITH NON-REMOVABLE BATTERY

The writes to the writeback cache can become durable.
We do not need explicit cache flush!!
We can use normal sync option

HOST

Flash Storage

Page cache

DIO write()

Writeback cache

NAND Flash

Volatile

Durable
Evaluation

008. Experiment
IMPLEMENTATION

Galaxy S5 (KitKat 4.4.2, Linux 3.4)

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Snapdragon 801 MSM8974AC</td>
</tr>
<tr>
<td>CPU</td>
<td>Krait 2.457 GHz Quad-core</td>
</tr>
<tr>
<td>RAM</td>
<td>2 GB / LPDDR3 SDRAM</td>
</tr>
<tr>
<td>Internal Storage</td>
<td>32 GB eMMC</td>
</tr>
</tbody>
</table>
IO TRACE: WAL VS. WALDIO IO TRACE

Insert 100Byte, 10 times

Resolve Journaling of Journal anomaly!!
PERFORMANCE COMPARISON

Insert 100Byte, 10000 times

Transaction/sec

DELETE TRUNCATE PERSIST WAL

LS-MVBT

WALDIO-PS+NoGS WALDIO-FS+GS4 WALDIO-PD+NoGS WALDIO-PD+GS4 WALDIO-NS+NoGS WALDIO-NS+GS4

4332 insert/sec

2967 insert/sec

1290 insert/sec

2.7x

2.2x

5.0x

4.0x

7.4x

WALDIO-NS

WALDIO-PD

WALDIO-FS

WALDIO-PS

Stock SQLite LSMVBT

Insert Update Delete Insert Update Delete Insert Update Delete Insert Update Delete Insert Update Delete

Wongun Lee, et.al. Hanyang University

USENIX ATC, Santa Clara, USA, 2015, JULY 9

63
IO VOLUME AT SQLITE OPERATION

Insert 100Byte, 10000 times

Filesystem journal
File Data/File Metadata

Volume(MB)

I U D
DEL
I U D
TRU
I U D
PER
I U D
WAL
I U D
LSM
I U D
DIO

1/6
SUMMARY

- We develop Preallocation with Explicit Journaling, Header Embedding and Group Synch.
- We use direct IO in SQLite log commit avoiding expensive EXT4 log commit.
- We develop new SQLite journal mode “WALDIO”.
- As much as 7.4x performance against WAL,
- The total IO volume decreases to 1/6 against WAL.
- We successfully resolve Journaling of Journal anomaly.
Thank you!

Questions?

- contact: inamind@hanyang.ac.kr