Lightweight Application-Level Crash Consistency on Transactional Flash Storage

Changwoo Min, Woon-Hak Kang[†], Taesoo Kim, Sang-Won Lee[†], Young Ik Eom[†]

Georgia Institute of Technology
[†]Sungkyunkwan University

Application's data is not consistent after random failures

Mobile Phone

Hang
while booting

Bank Account

Financial loss

Application's data is not consistent after random failures

Mobile Phone

Bank Account

Power Outage

Hardware Errors

Software Panics (OS, Device Driver)

Hang while booting

Financial loss

Example: inserting records in two databases

Application

```
write(/db1, “new”);
write(/db2, “new”);
```

Example: inserting records in two databases

Application

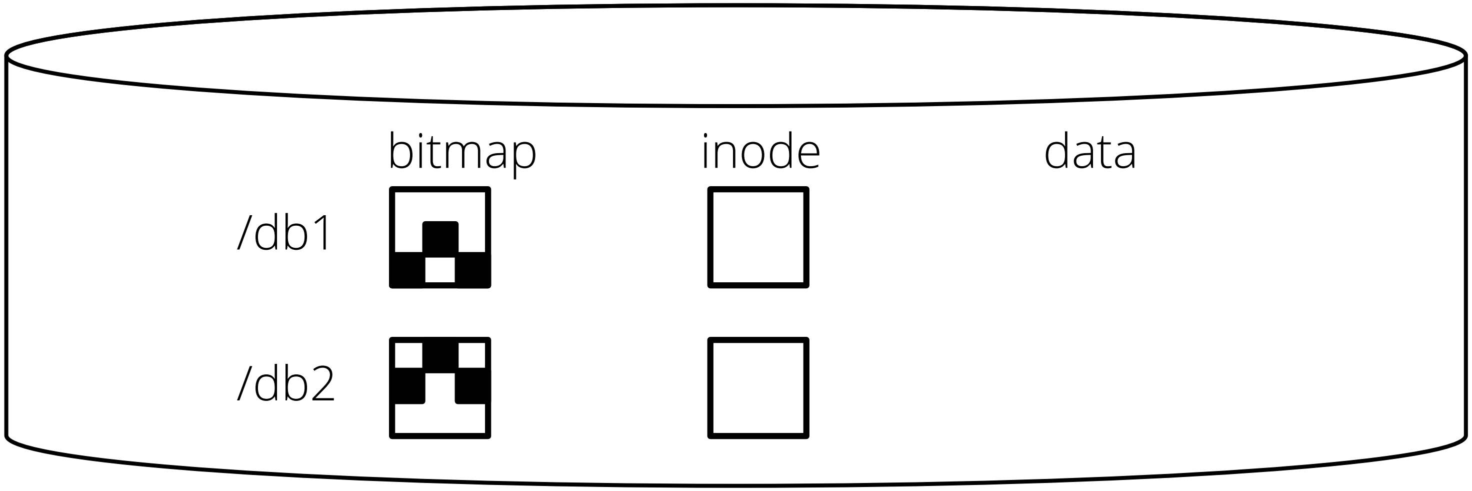
write(/db1, “new”);
write(/db2, “new”);

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Storage Device



Example: inserting records in two databases

Application

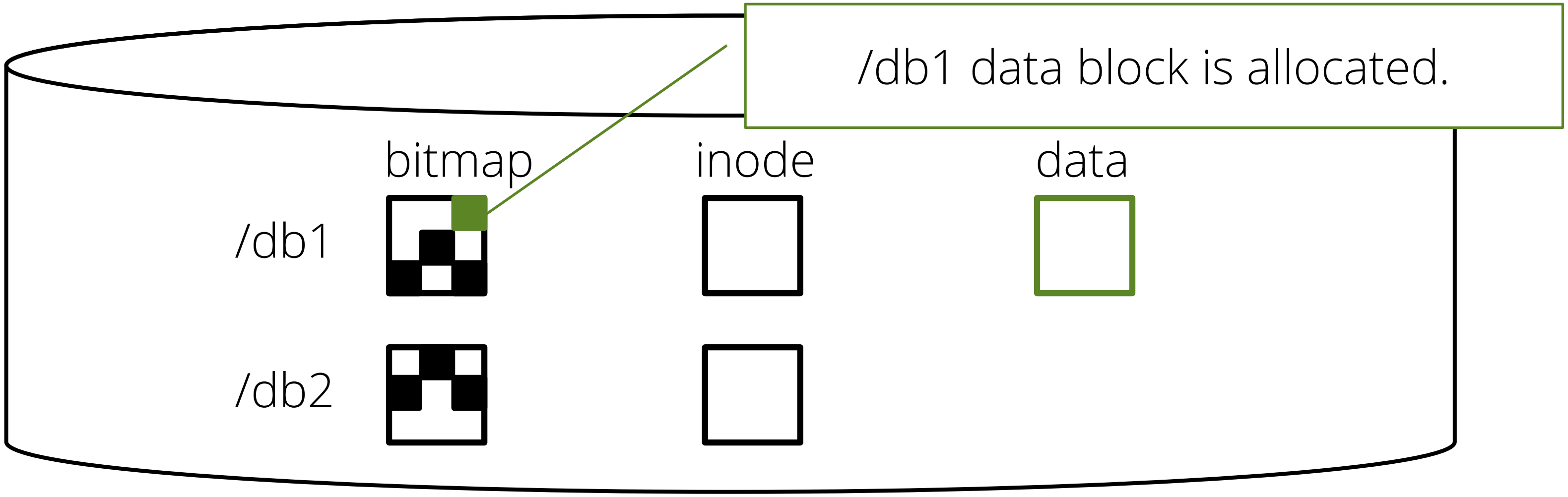
```
write(/db1, “new”);
write(/db2, “new”);
```

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Storage Device



/db1 data block is allocated.

Example: inserting records in two databases

Application

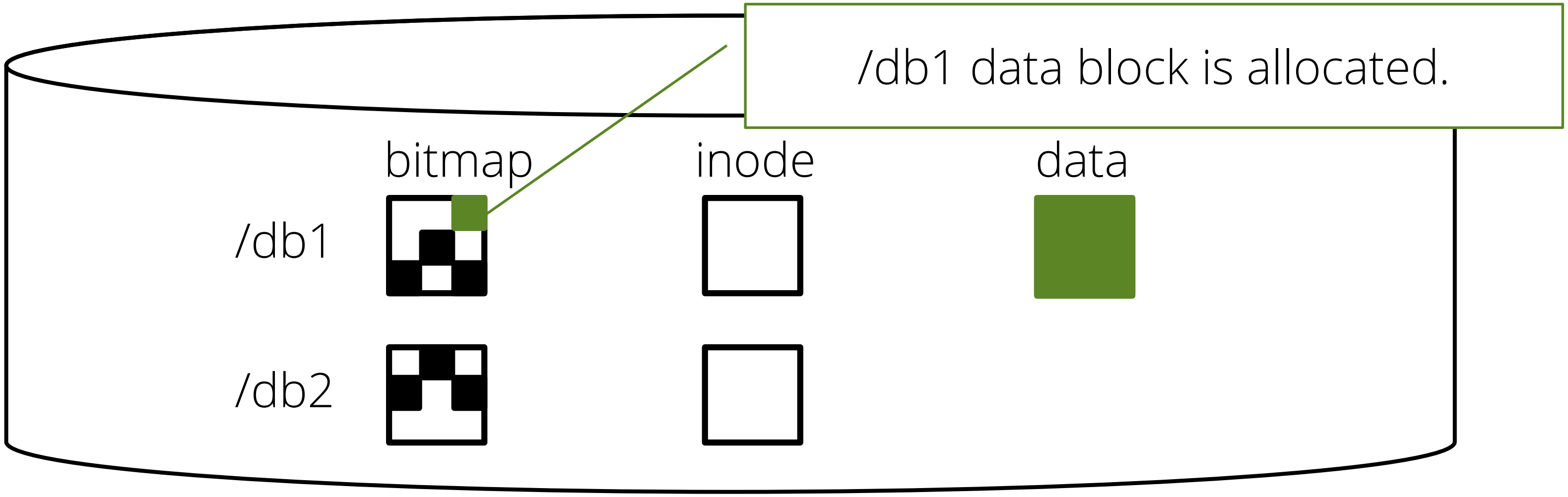
```
write(/db1, “new”);
write(/db2, “new”);
```

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Storage Device



/db1 data block is allocated.

Example: inserting records in two databases

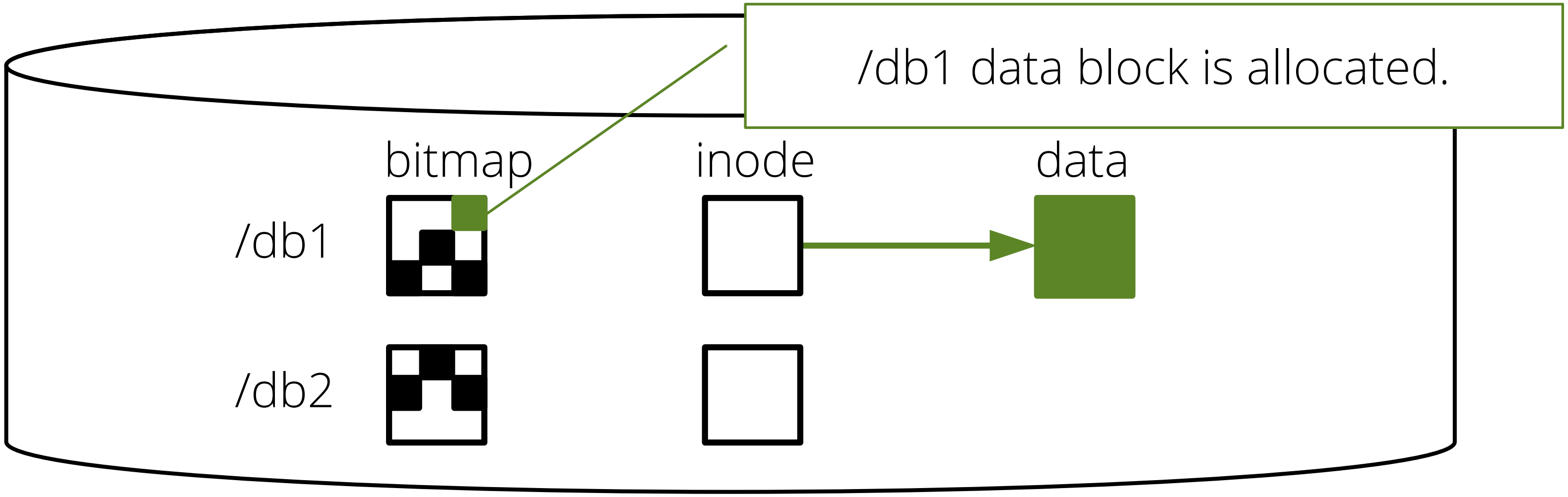
Application

write(/db1, "new");
write(/db2, "new");

File System

For each database
- Allocates new data block (bitmap)
- Fills data block with user data (data)
- ▶ Sets location of data block (inode)

Storage Device



/db1 data block is allocated.

Example: inserting records in two databases

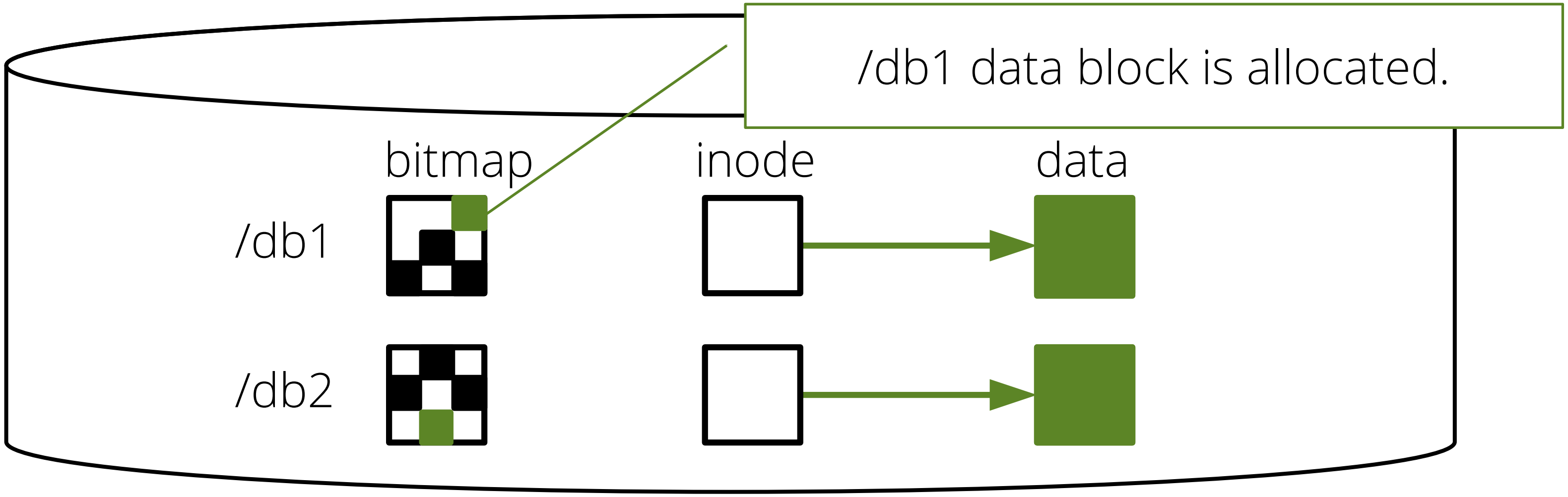
Application

write(/db1, "new");
write(/db2, "new");

File System

For each database
- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Storage Device



Example: inserting records in two databases

Application

write(/db1, “new”);
write(/db2, “new”);

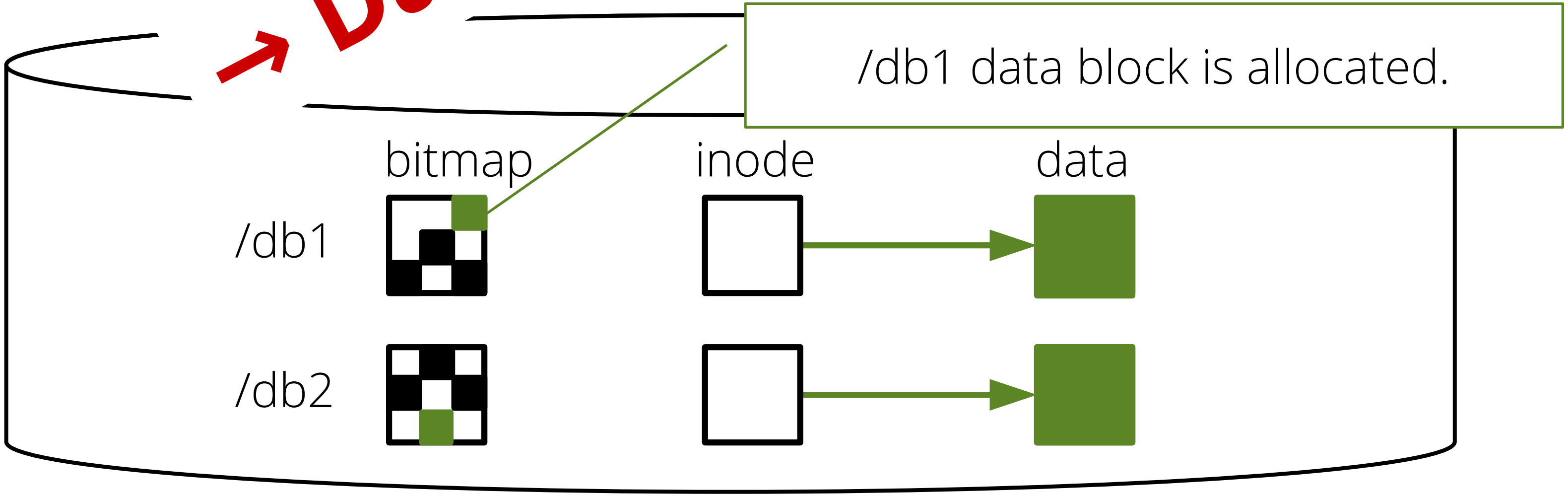
File System

For each databas

- Allocates n
- Fills d ata (data)
- block (inode)

Random failures?
→ Data corruption!

Storage Device



/db1 data block is allocated.

How to Achieve Crash Consistency
: The Case of SQLite

```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```

→

Logging (i.e., journaling) & Crash Recovery

"Write logs first before writing data in place"

How to Achieve Crash Consistency
: The Case of SQLite

```
atomic_update {
  write(/db1, “new”);
  write(/db2, “new”);
}
```

Logging (i.e., journaling) & Crash Recovery

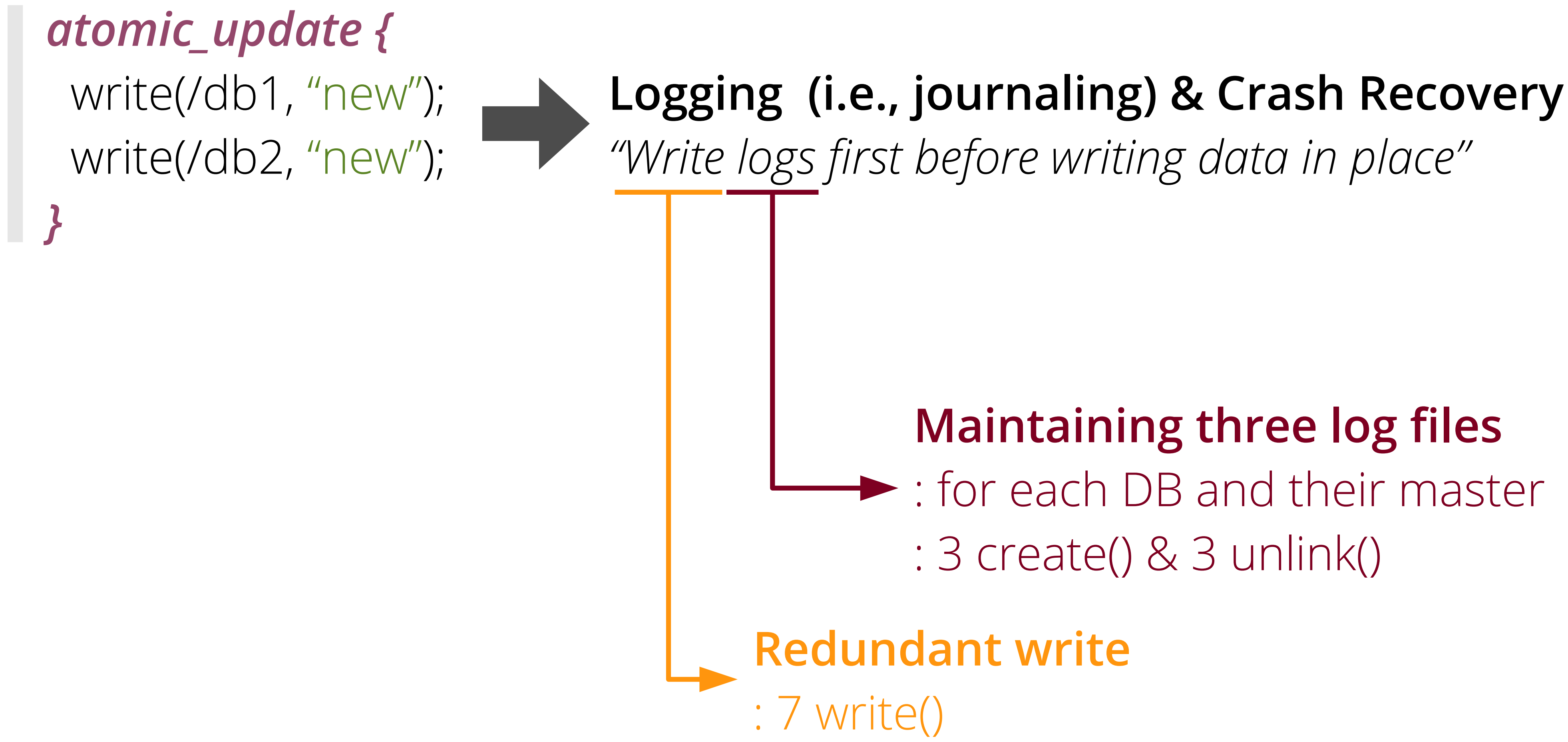
“Write logs first before writing data in place”

Maintaining three log files

: for each DB and their master

: 3 create() & 3 unlink()

How to Achieve Crash Consistency : The Case of SQLite



```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```

Logging (i.e., journaling) & Crash Recovery

"Write logs first before writing data in place"

Maintaining three log files

: for each DB and their master

: 3 create() & 3 unlink()

Redundant write

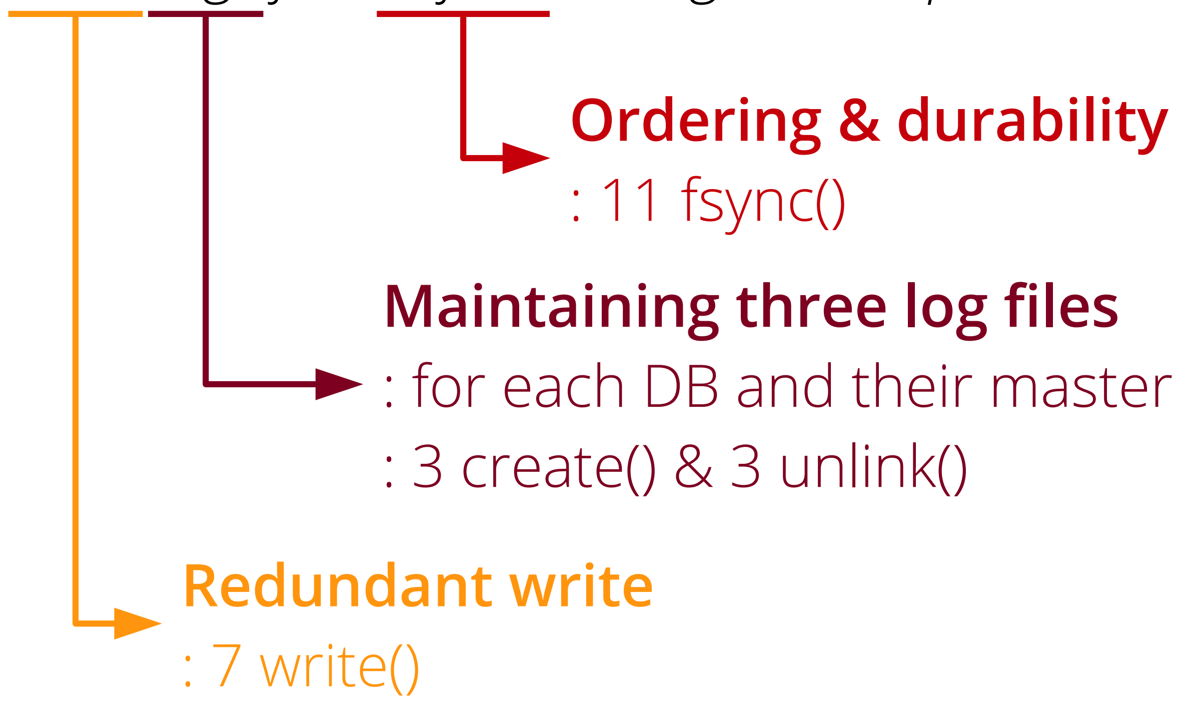
: 7 write()

How to Achieve Crash Consistency : The Case of SQLite

```
atomic_update {
  write(/db1, “new”);
  write(/db2, “new”);
}
```

Logging (i.e., journaling) & Crash Recovery

“Write logs first before writing data in place”



Ordering & durability

: 11 fsync()

Maintaining three log files

: for each DB and their master

: 3 create() & 3 unlink()

Redundant write

: 7 write()

How to Achieve Crash Consistency
: The Case of SQLite

```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```

```
// create master journal
open(/master.jnl);
write(/master.jnl, "/db1,/db2");
fsync(/master.jnl);
fsync(/);
// update db1
open(/db1.jnl);
write(/db1.jnl, "old");
fsync(/db1.jnl);
fsync(/);
write(/db1.jnl, "master.jnl");
fsync(/db1.jnl);
write(/db1, "new");
fsync(/db1);
```

```
// update db2
open(/db2.jnl);
write(/db2.jnl, "old");
fsync(/db2.jnl)
fsync(/);
write(/db2.jnl, "master.jnl");
fsync(/db2.jnl);
write(/db2, "new");
fsync(/db2);
// clean up journals
unlink(/master.jnl);
fsync(/);
unlink(/db1.jnl);
unlink(/db1.jnl);
```

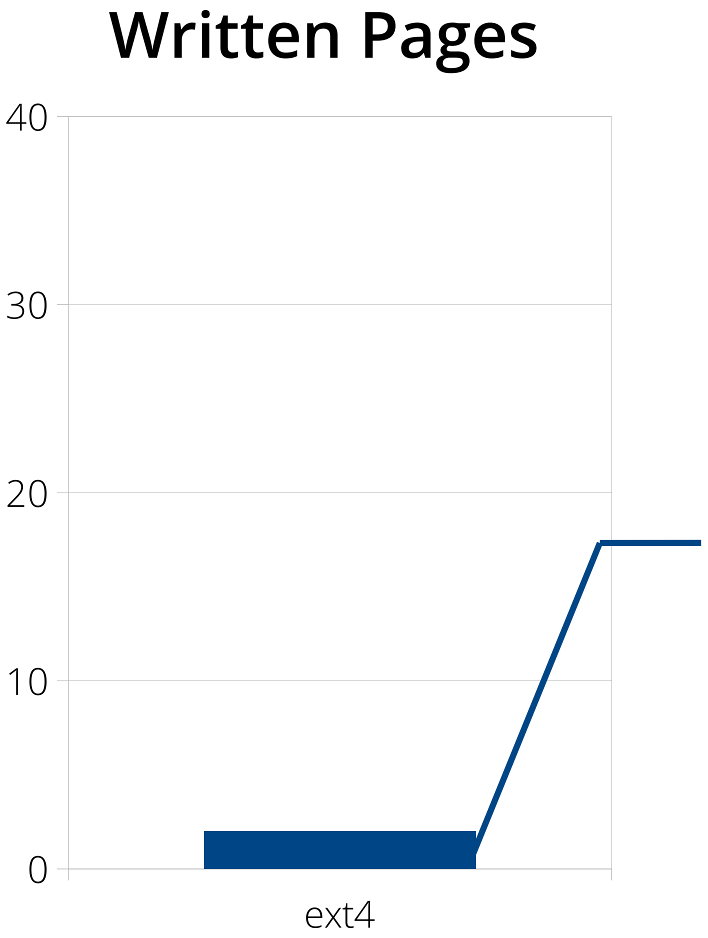
How to Achieve Crash Consistency
: The Case of SQLite

```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```

```
// create master journal
open(/master.jnl);
write(/master.jnl, "/db1,/db2");
fsync(/master.jnl);
fsync(/);
// update db1
open(/db1.jnl);
write(/db1.jnl, "old");
fsync(/db1.jnl);
fsync(/);
write(/db1.jnl, "master.jnl");
fsync(/db1.jnl);
write(/db1, "new");
fsync(/db1);
```

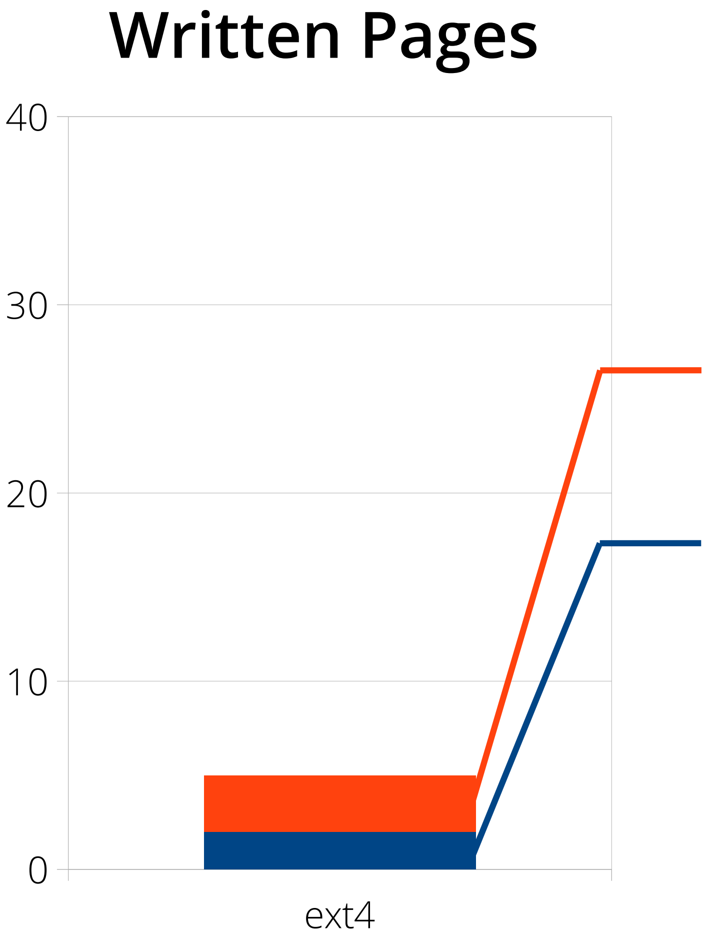
```
// update db2
open(/db2.jnl);
write(/db2.jnl, "old");
fsync(/db2.jnl)
fsync(/);
write(/db2.jnl, "master.jnl");
fsync(/db2.jnl);
write(/db2, "new");
fsync(/db2);
// clean up journals
unlink(/master.jnl);
fsync(/);
unlink(/db1.jnl);
unlink(/db1.jnl);
```

Cost of Crash Consistency
: The Case of SQLite



Database update: 2 pages

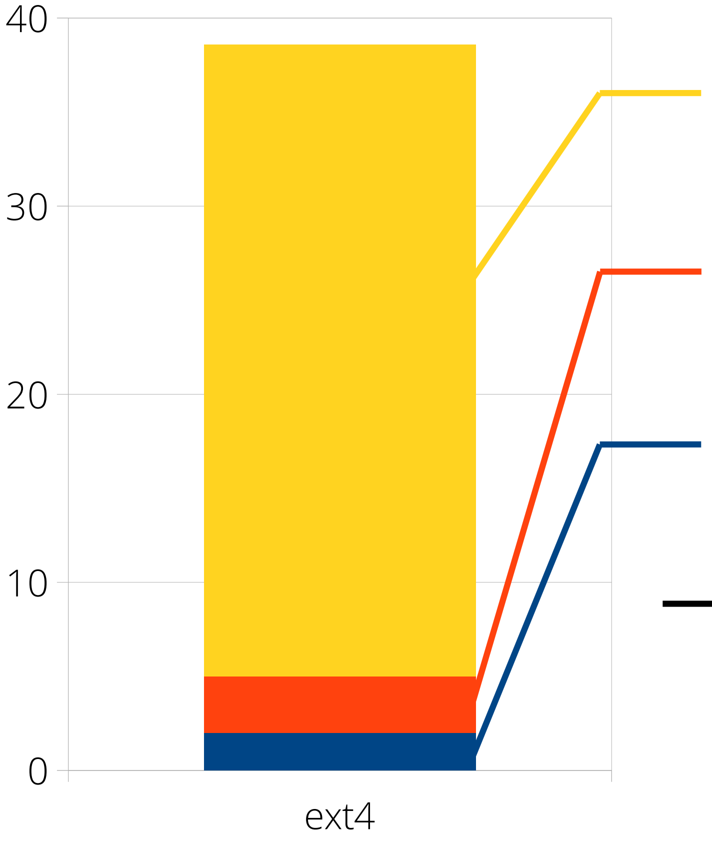
Cost of Crash Consistency
: The Case of SQLite



SQLite journaling: 3 pages

Database update: 2 pages

Cost of Crash Consistency
: The Case of SQLite


Written Pages

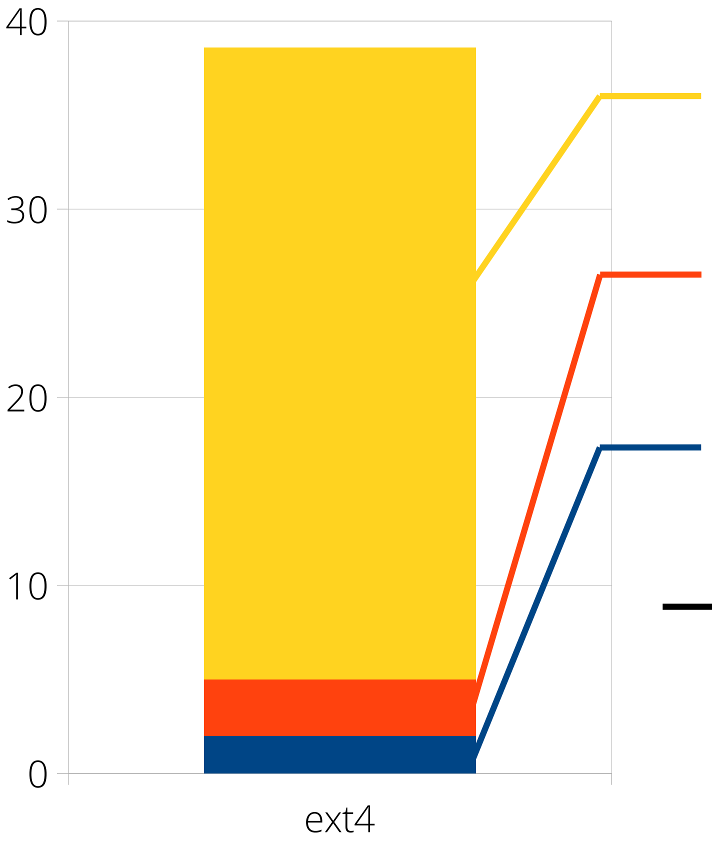
Ext4 journaling+metadata: 33.6 pages

SQLite journaling: 3 pages

Database update: 2 pages

Write Amplification = 38.6/2 = 19.3

Cost of Crash Consistency
: The Case of SQLite



Written Pages

Ext4 journaling+metadata: 33.6 pages

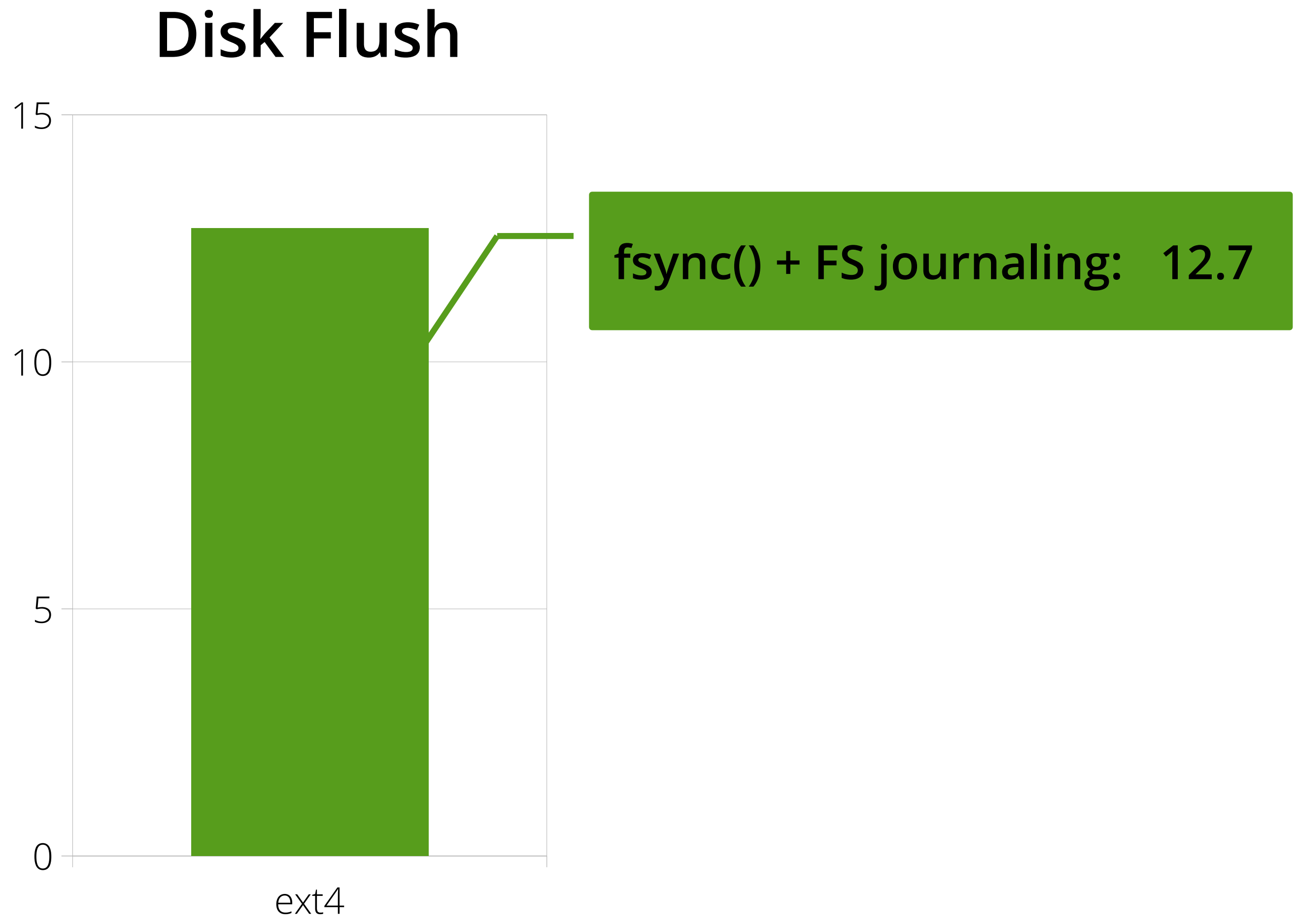
SQLite journaling: 3 pages

Database upd

Redundant disk writes per single data page write

Write Amplification = 38.6/2 = 19.3

Cost of Crash Consistency
: The Case of SQLite

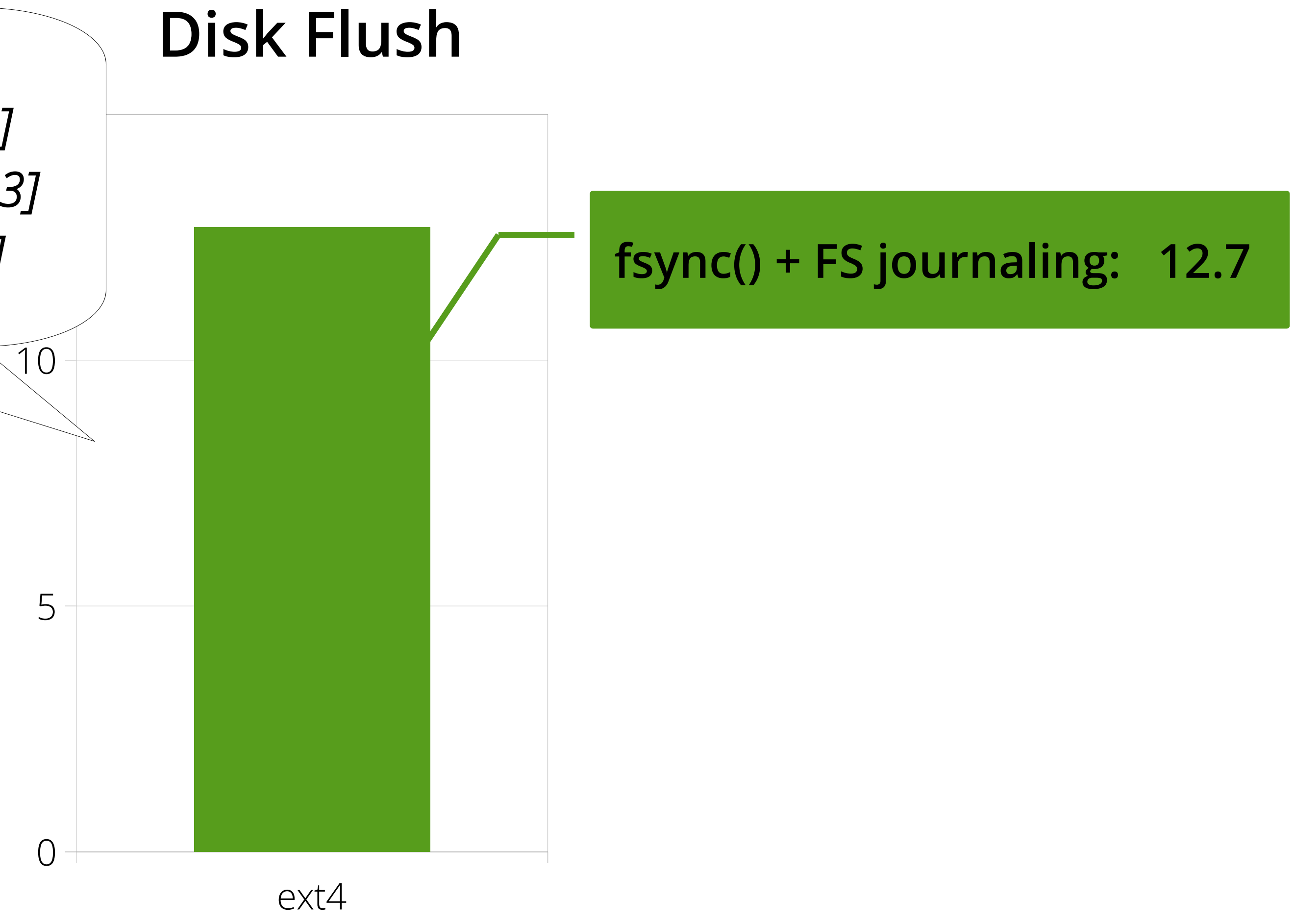


fsync() + FS journaling: 12.7

Cost of Crash Consistency
: The Case of SQLite

Notoriously slow!

xsyncfs [Nightingale:OSDI06]
OptFS [Chidambaram:SOSP13]
DuraSSD [Kang:SIGMOD14]
...



Problem: complex, redundant software stack for crash consistency

How can we **simplify** mechanisms for **application crash consistency**?

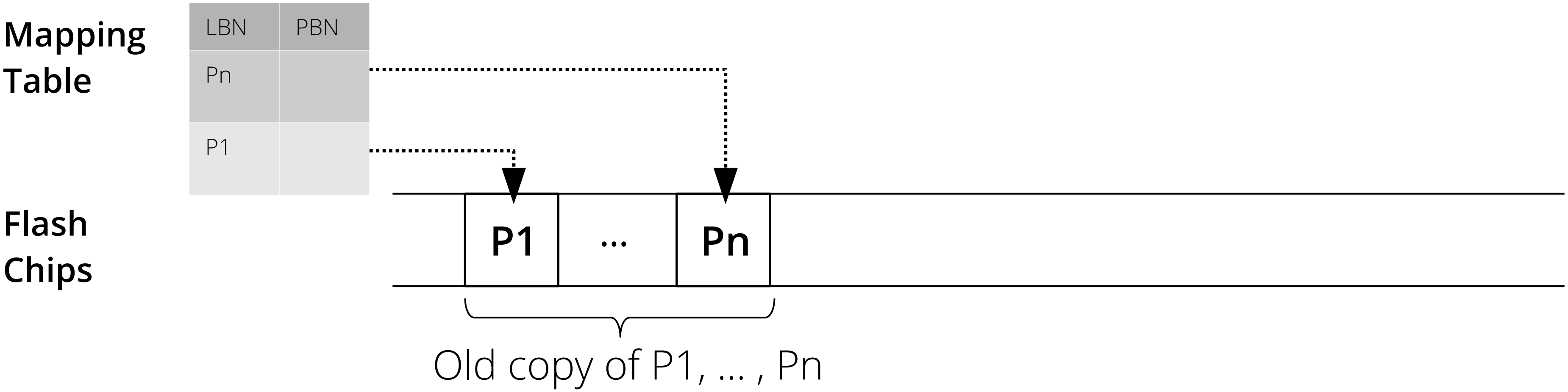
Problem: complex, redundant software stack for crash consistency

How can we simplify mechanisms for application crash consistency?

*Can we use **atomic updates of multi pages** provided by **transactional flash**?*

Transactional Flash 101

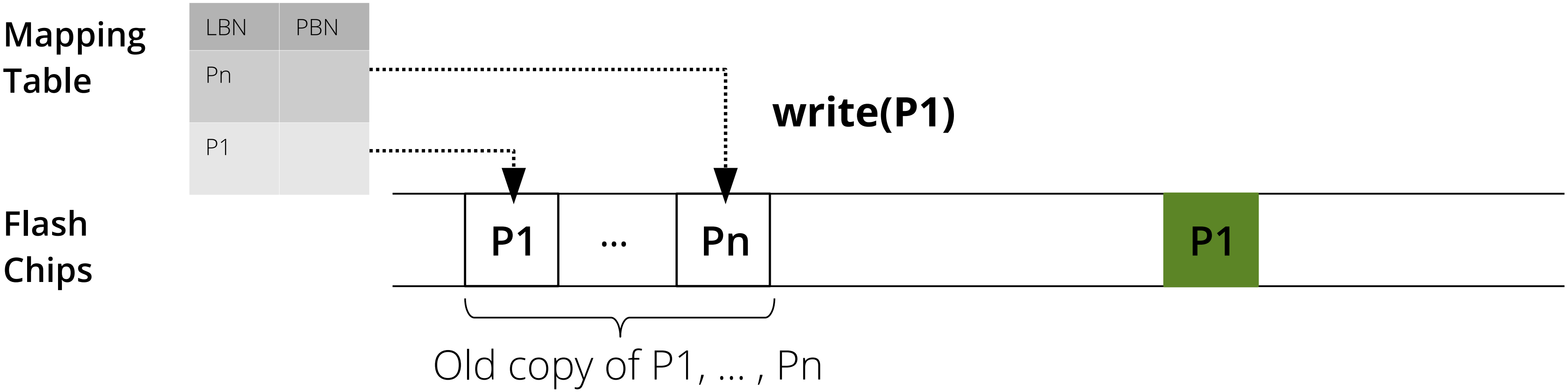
- NAND Flash SSD
  - No in-place update
  - Log-structured write
  - Mapping table: logical address → physical address



LBN	PBN
Pn	
P1	

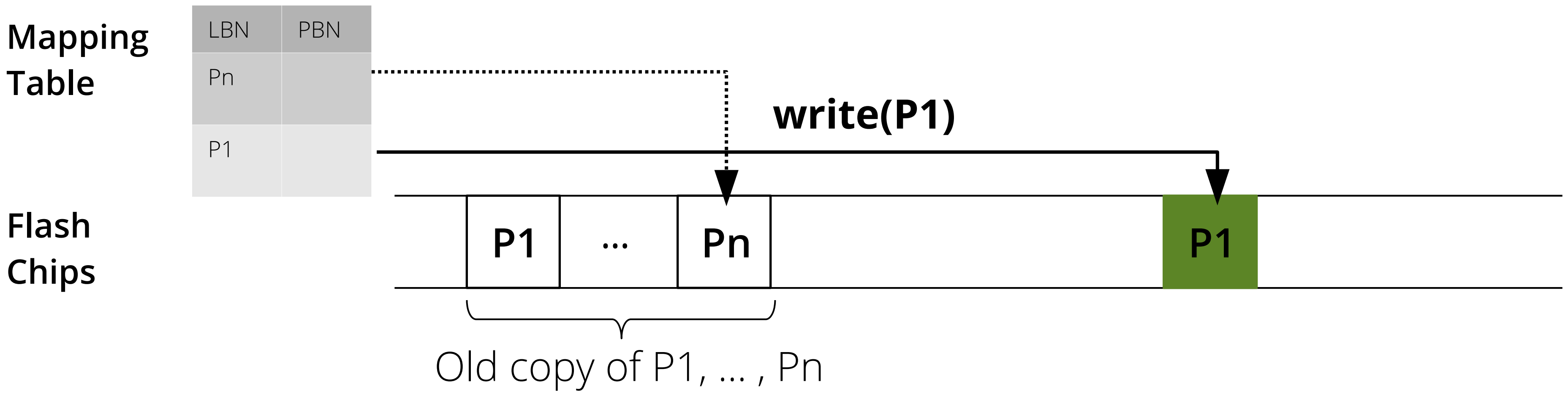
Transactional Flash 101

- NAND Flash SSD
  - No in-place update
  - Log-structured write
  - Mapping table: logical address → physical address



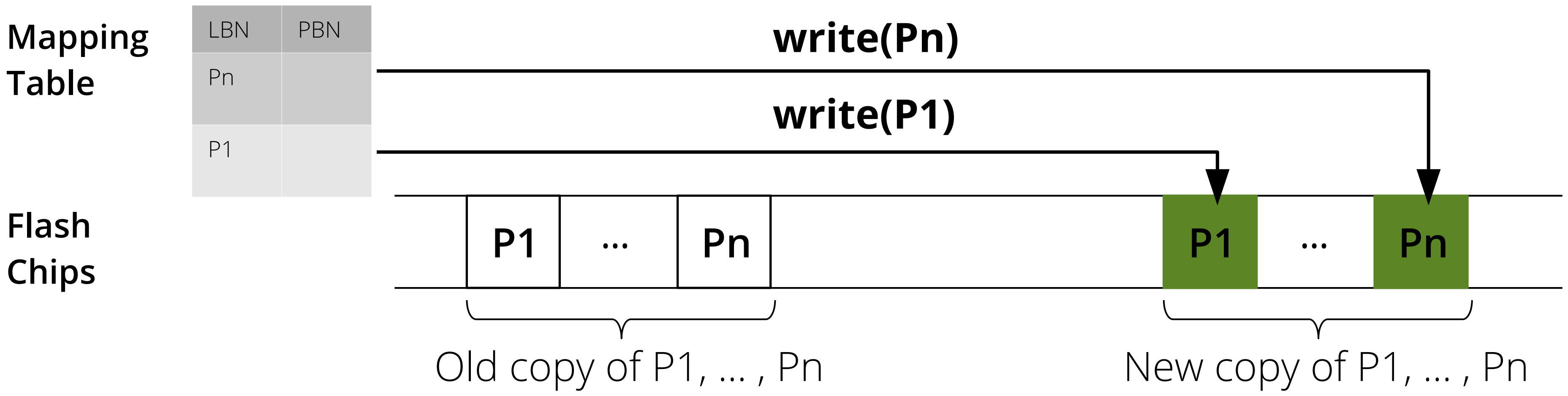
Transactional Flash 101

- NAND Flash SSD
  - No in-place update
  - Log-structured write
  - Mapping table: logical address → physical address



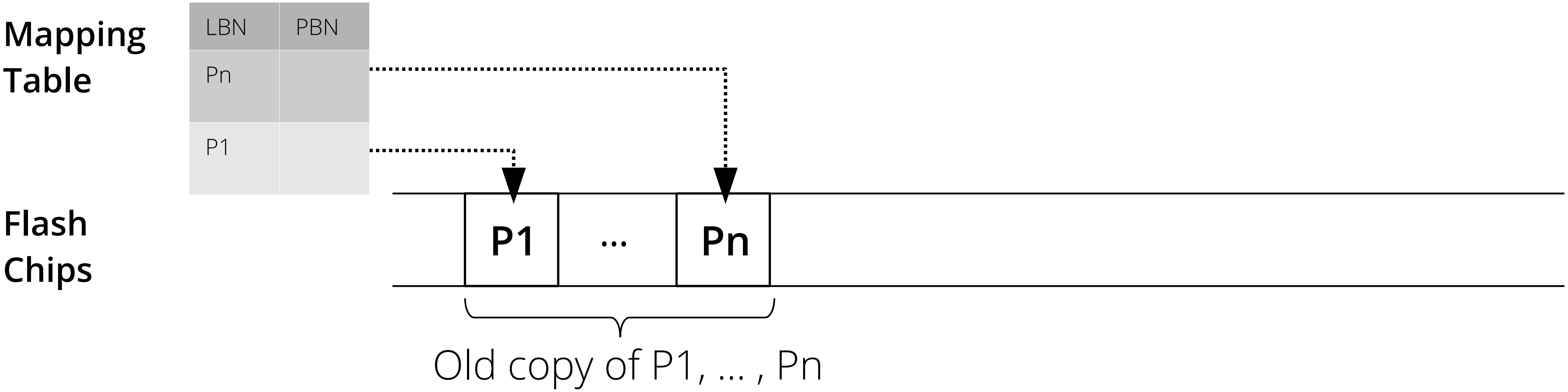
Transactional Flash 101

- NAND Flash SSD
  - No in-place update
  - Log-structured write
  - Mapping table: logical address → physical address



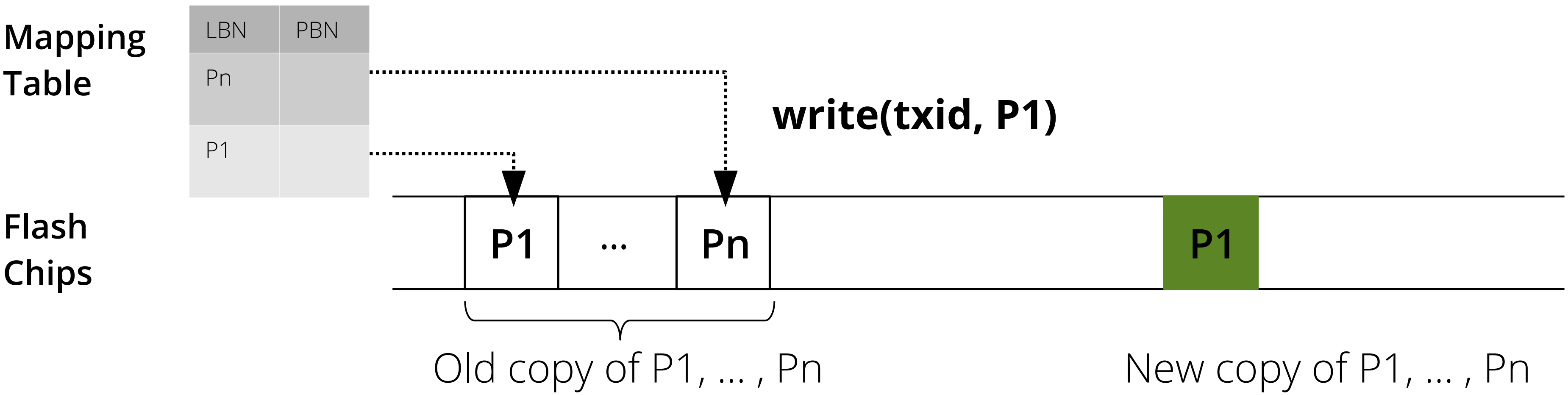
Transactional Flash 101

- Transactional Flash SSD
  - Atomic multi-page write by atomically updating the mapping table at commit request
    - write(txid, page), commit(txid), abort(txid)
  - H/W implementation or S/W emulation



Transactional Flash 101

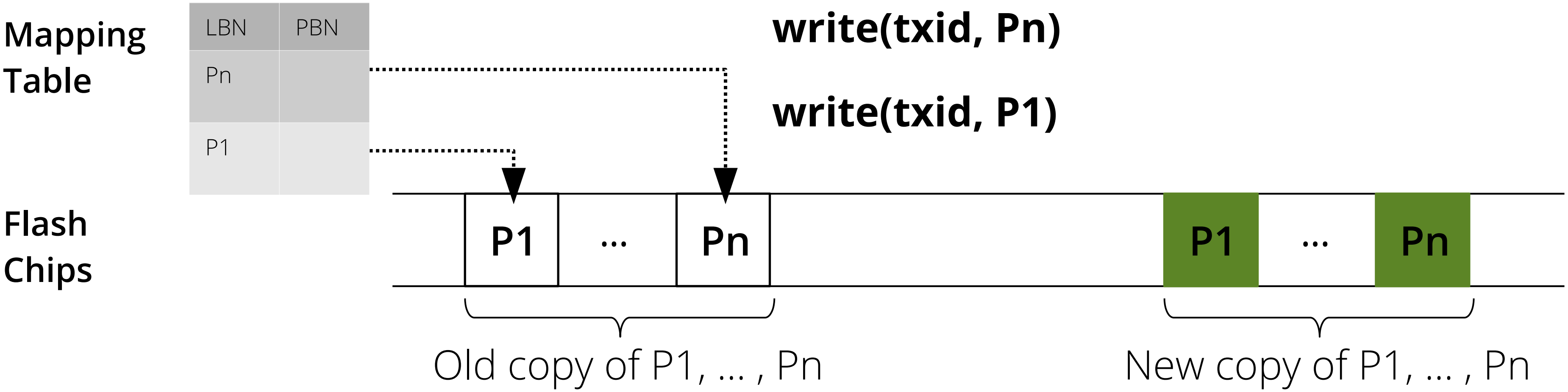
- Transactional Flash SSD
  - Atomic multi-page write by atomically updating the mapping table at commit request
    - write(txid, page), commit(txid), abort(txid)
  - H/W implementation or S/W emulation



LBN	PBN
Pn	
P1	

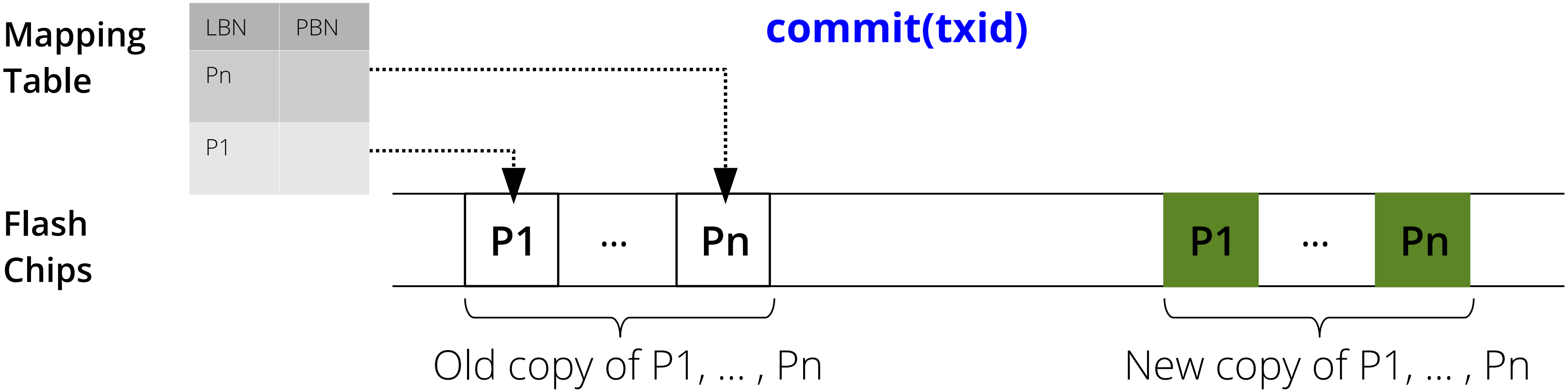
Transactional Flash 101

- Transactional Flash SSD
  - Atomic multi-page write by atomically updating the mapping table at commit request
    - write(txid, page), commit(txid), abort(txid)
  - H/W implementation or S/W emulation



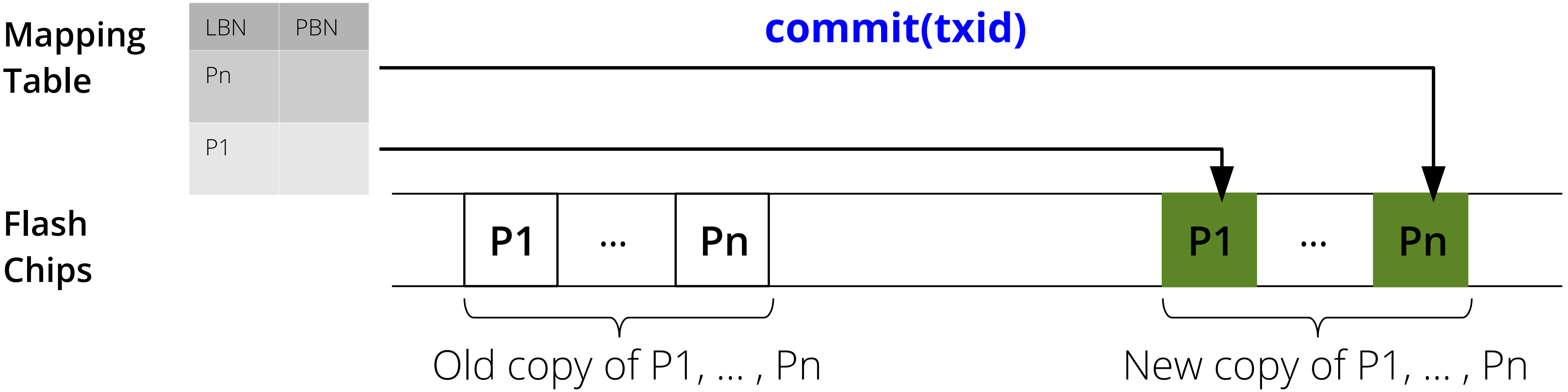
Transactional Flash 101

- Transactional Flash SSD
  - Atomic multi-page write by atomically updating the mapping table at commit request
    - write(txid, page), commit(txid), abort(txid)
  - H/W implementation or S/W emulation



Transactional Flash 101

- Transactional Flash SSD
  - Atomic multi-page write by atomically updating the mapping table at commit request
    - write(txid, page), commit(txid), abort(txid)
  - H/W implementation or S/W emulation



Our Solution: CFS, a new file system using transactional flash

Simplifying

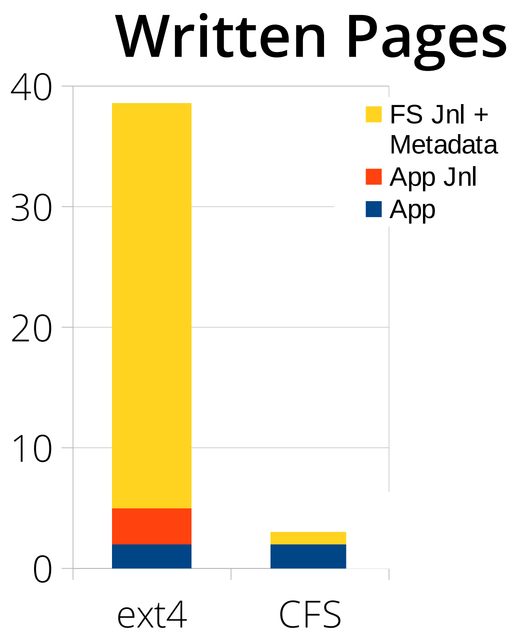
applications' crash consistency

using atomic multi-page write

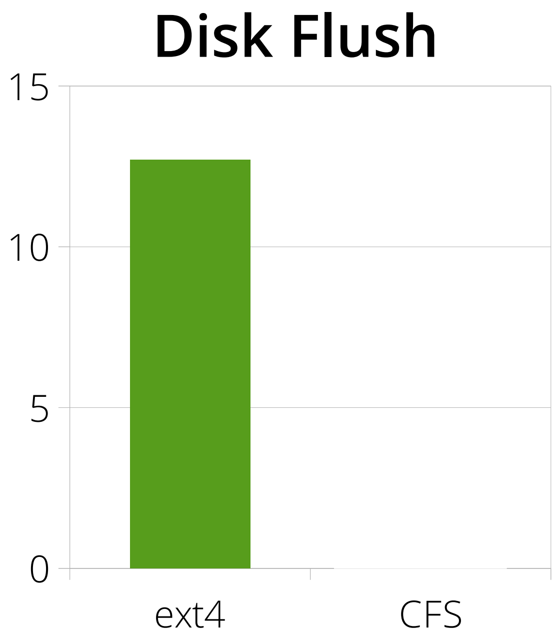
of transactional flash

Our Solution: CFS

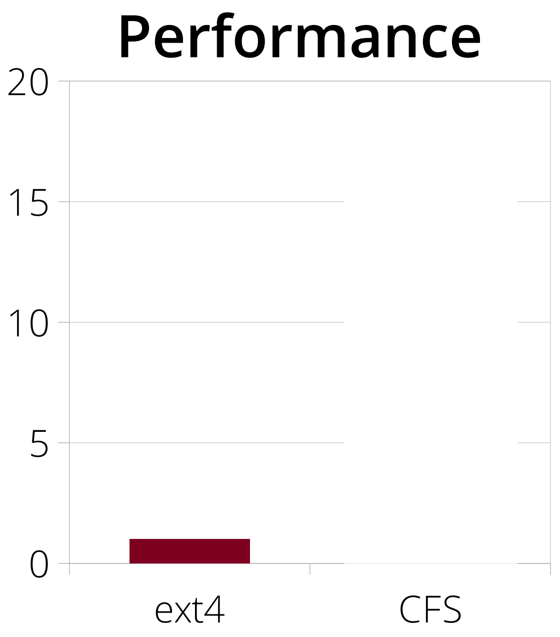
```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```



Written Pages



Disk Flush



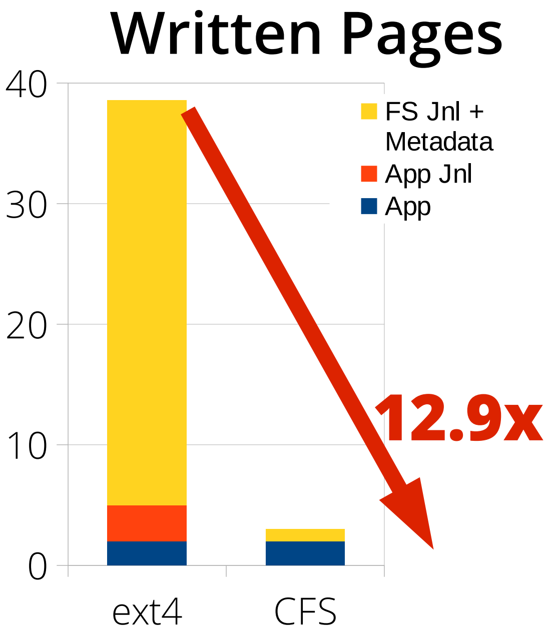
Performance

Our Solution: CFS

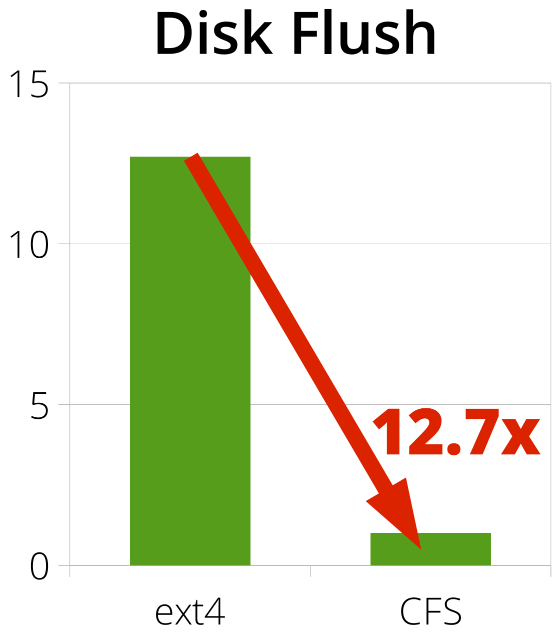
```
atomic_update {
  write(/db1, "new");
  write(/db2, "new");
}
```

→

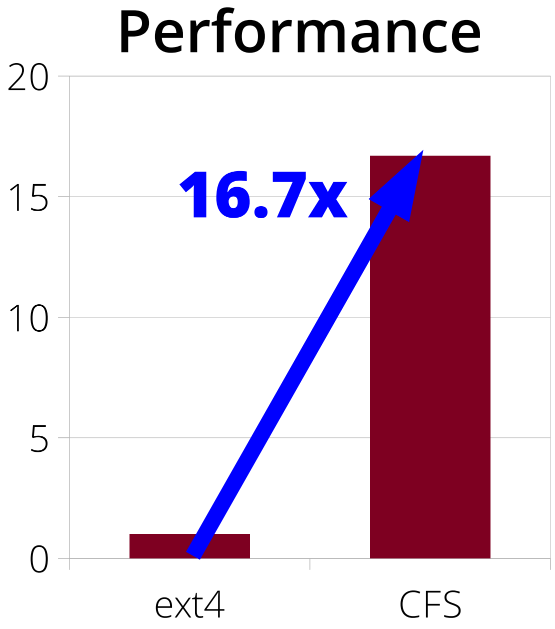
```
+ cfs_begin();
    write(/db1, "new");
    write(/db2, "new");
+ cfs_commit();
```



Written Pages



Disk Flush

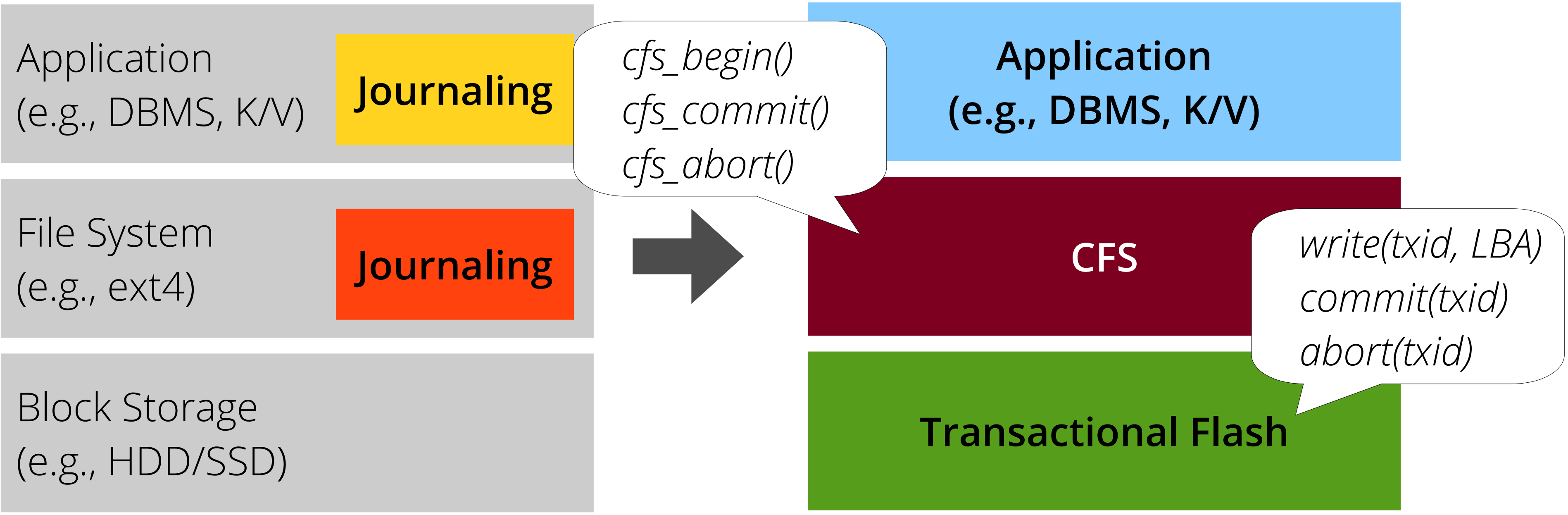


Performance

Outline

- Introduction
- **CFS Design**
- Evaluation
- Conclusion

CFS Architecture



Removing redundant journaling with new primitives provided by transactional flash

Four Challenges

1. Finding a set of pages for atomic updates
2. File system metadata consistency in every case
3. Concurrency control among atomic updates
4. Legacy application support without any modification

#1. Unit of Atomic Write
: Atomic Propagation Group

Application

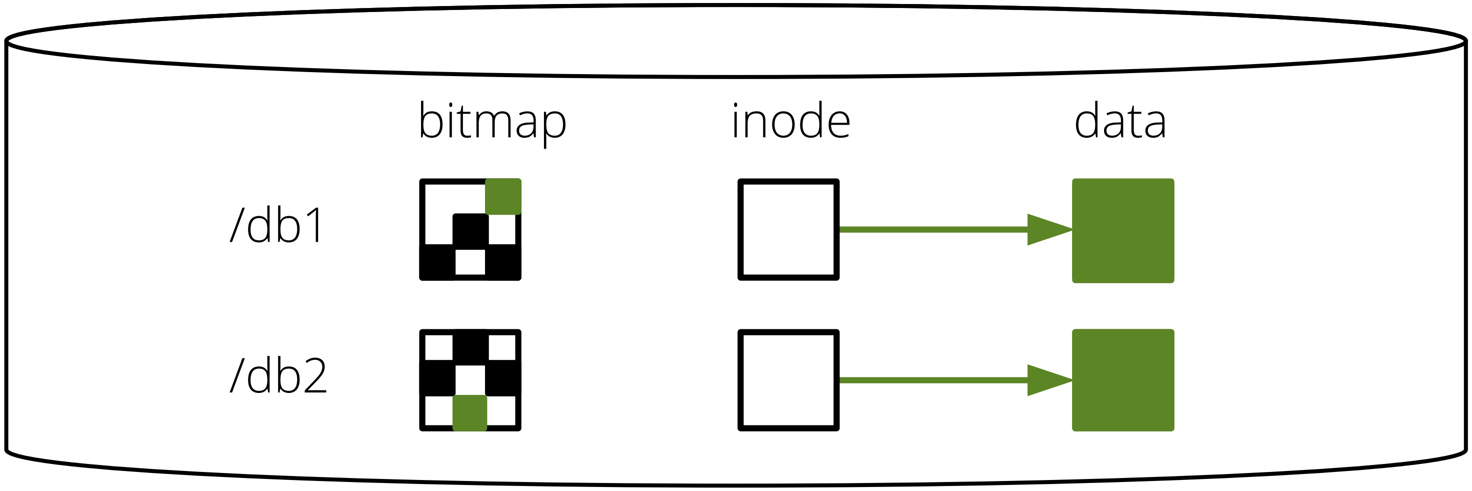
```
+ cfs_begin();
  write(/db1, “new”);
  write(/db2, “new”);
+ cfs_commit();
```

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Transactional Flash



#1. Unit of Atomic Write : Atomic Propagation Group

Application

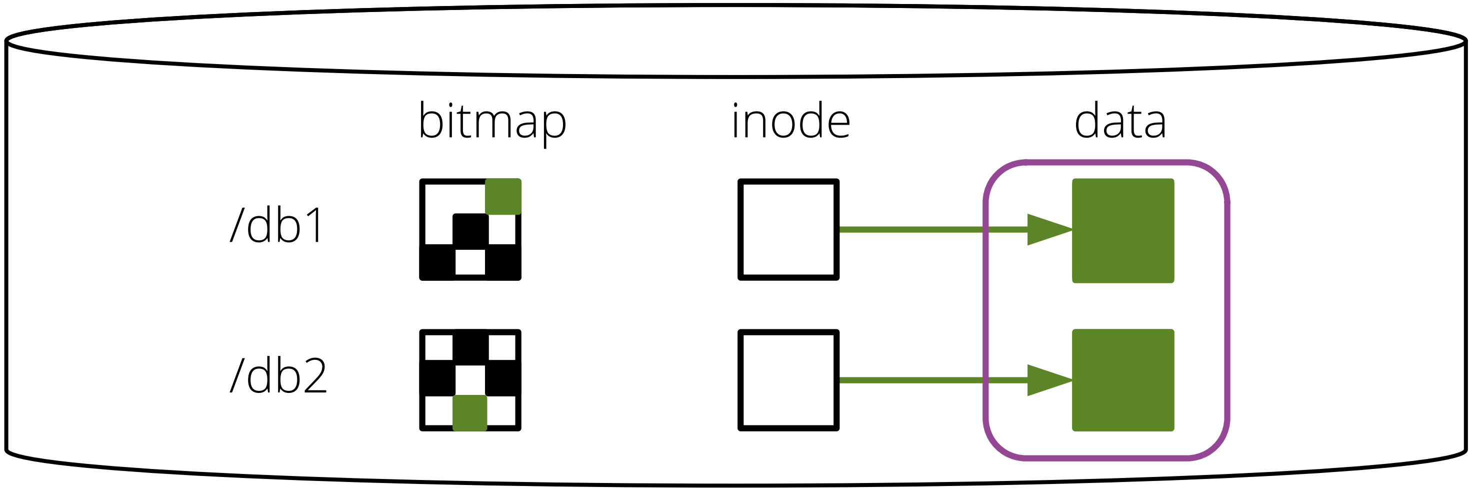
```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Transactional Flash



#1. Unit of Atomic Write
: Atomic Propagation Group

Application

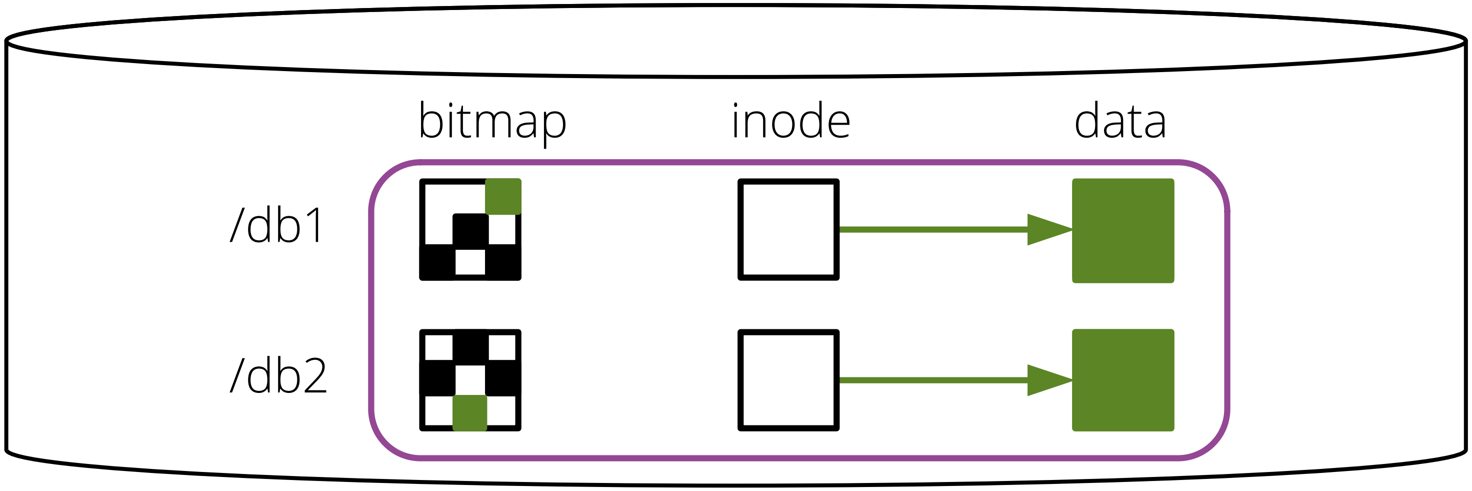
```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

File System

For each database

- Allocates new data block (bitmap)
- Fills data block with user data (data)
- Sets location of data block (inode)

Transactional Flash



#1. Unit of Atomic Write
: Atomic Propagation Group

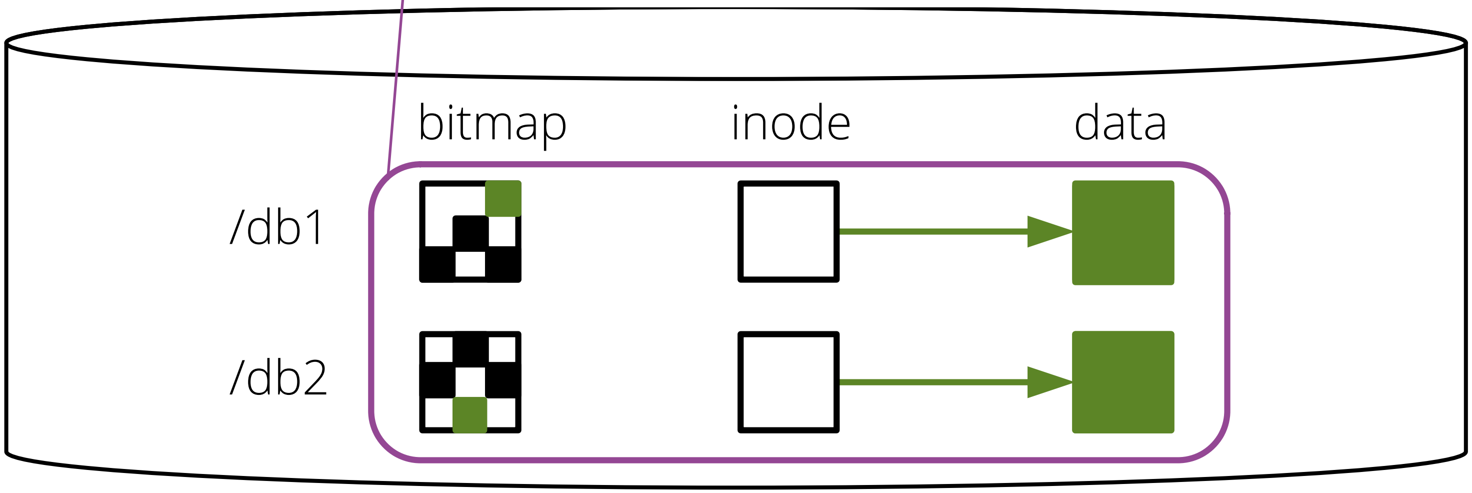
Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

- Updated data and file system metadata pages need to be atomically updated.
- Use atomic multi-page write operations

→ Atomic Propagation Group

Transactional Flash

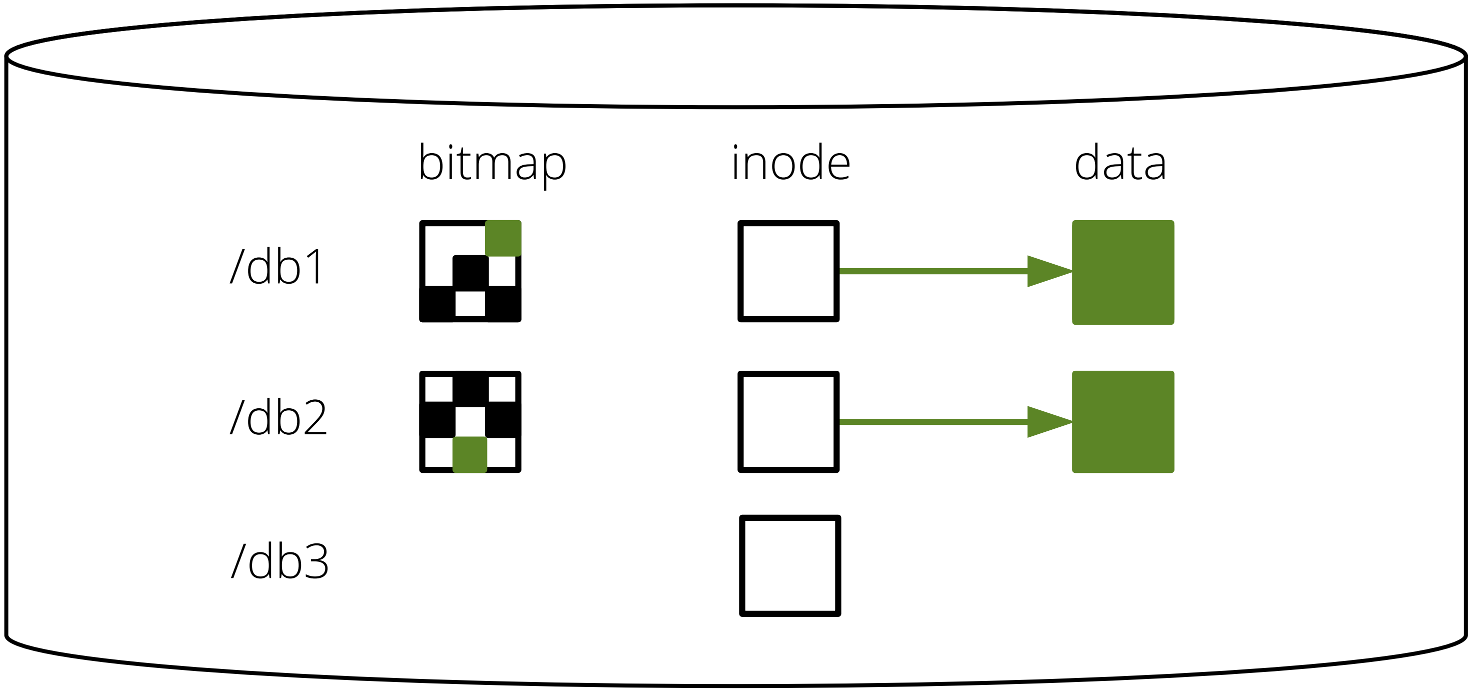


Example: Two apps. are running

Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

Transactional Flash



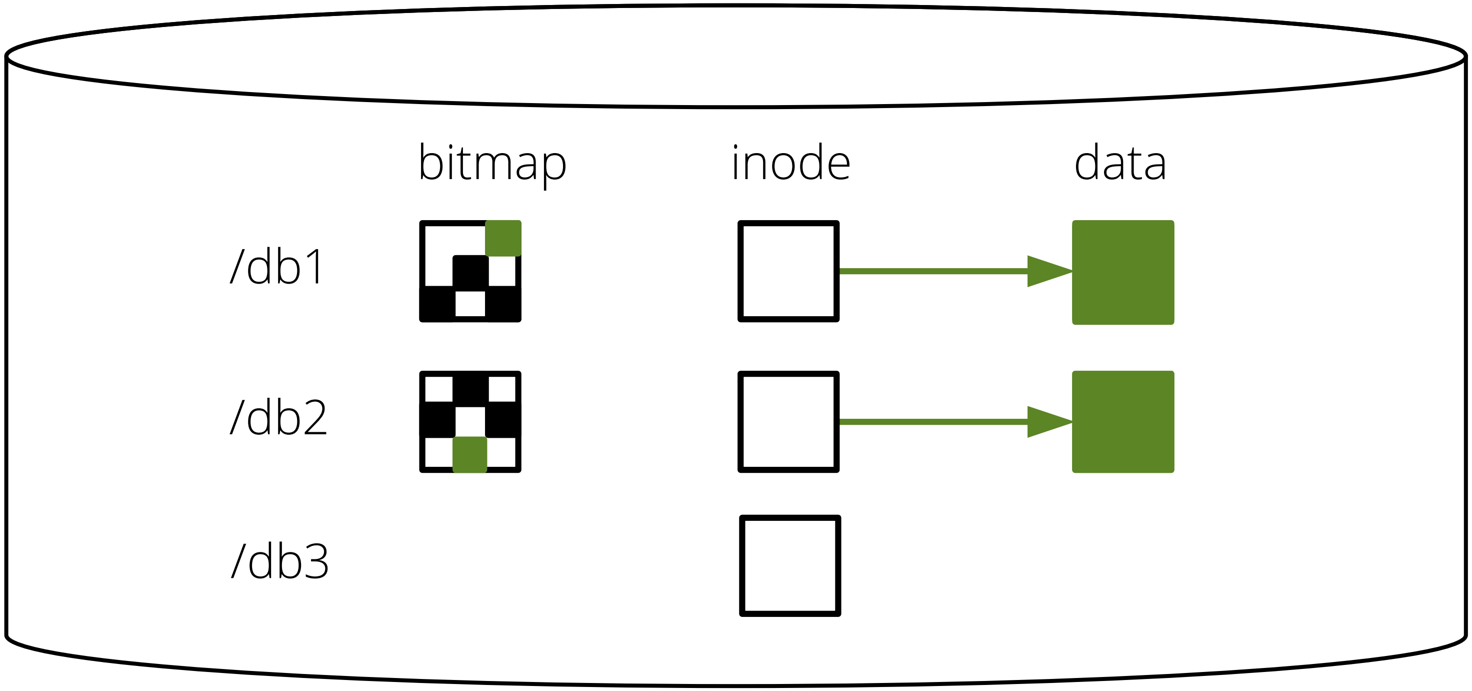
Example: Two apps. are running

Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

```
+ cfs_begin();
  write(/db3, "new");
  ...
```

Transactional Flash



Example: Two apps. are running

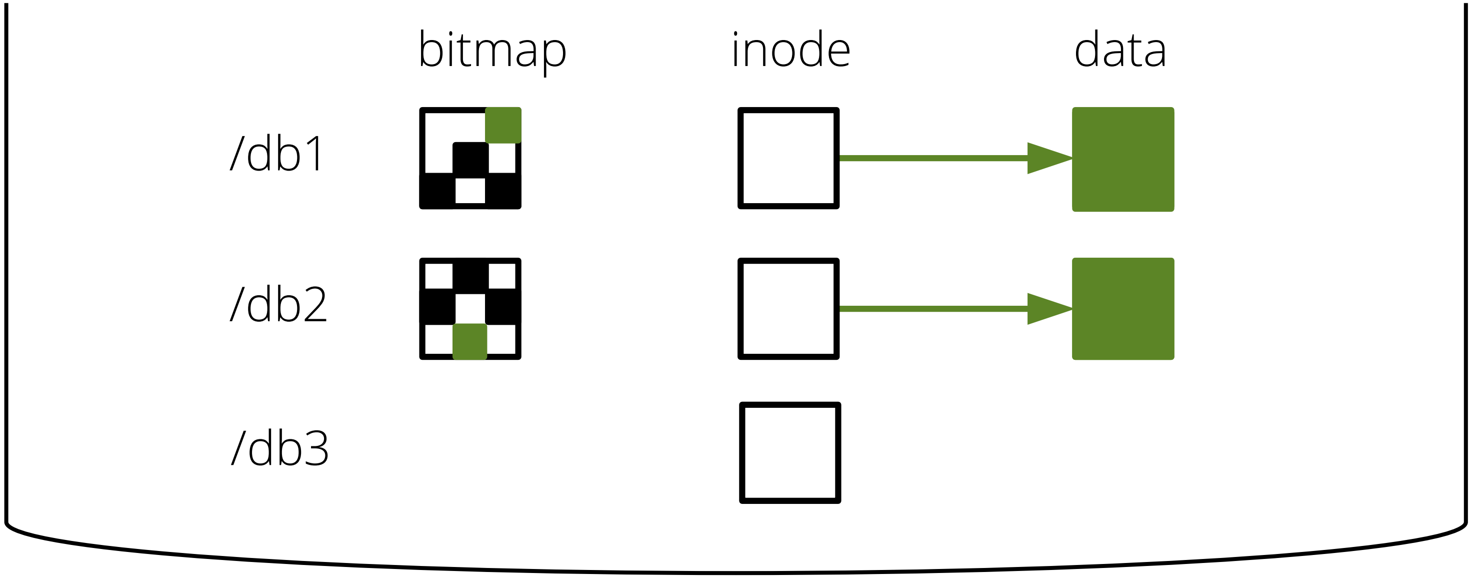
Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

```
+ cfs_begin();
  write(/db3, "new");
  ...
```

What if /db3 bitmap locates in the /db2 bitmap?

Transactional Flash



Example: Two apps. are running

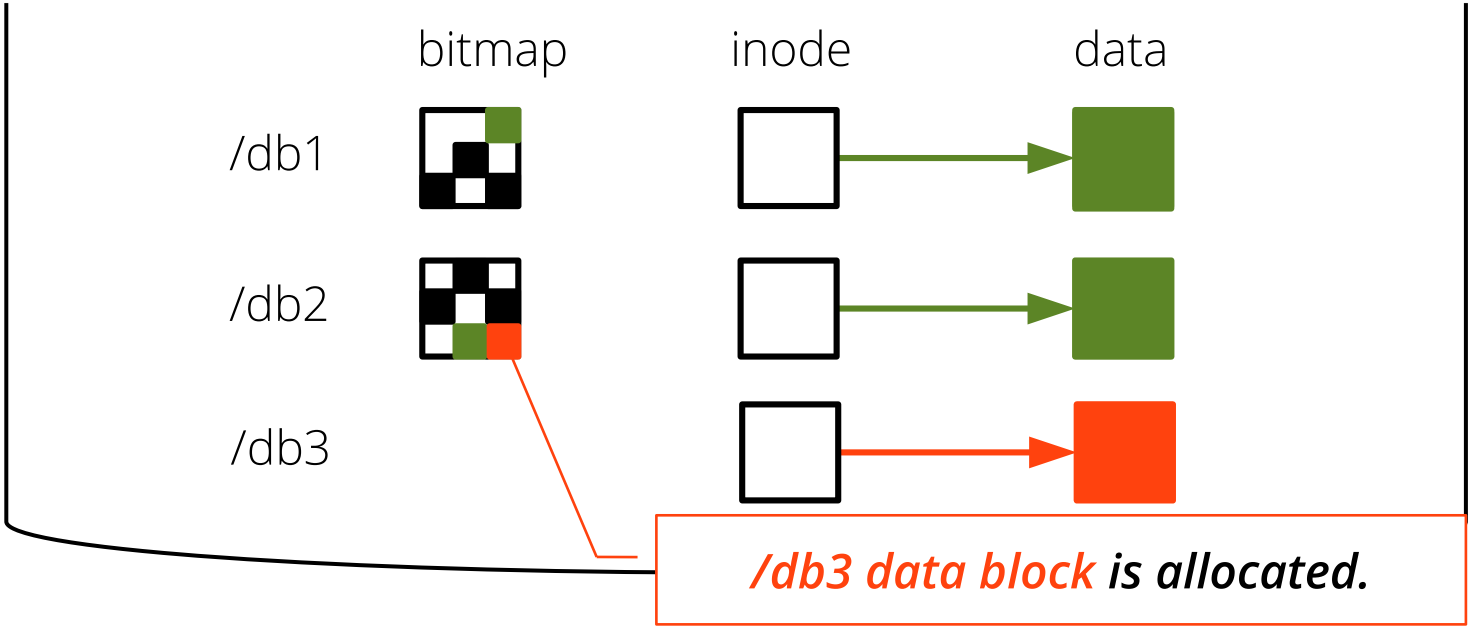
Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

```
+ cfs_begin();
  write(/db3, "new");
  ...
```

What if /db3 bitmap locates in the /db2 bitmap?

Transactional Flash



/db3 data block is allocated.

Example: Two apps. are running

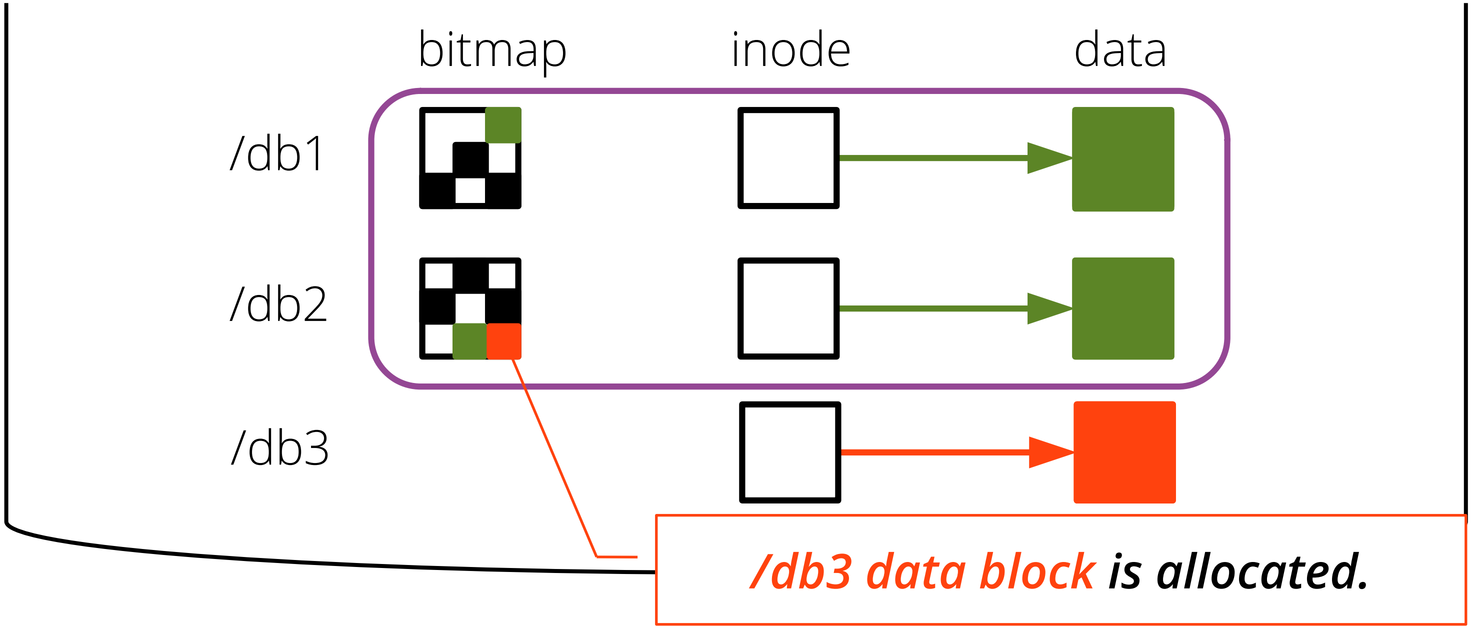
Application

```
+ cfs_begin();
  write(/db1, "new");
  write(/db2, "new");
+ cfs_commit();
```

```
+ cfs_begin();
  write(/db3, "new");
  ...
```

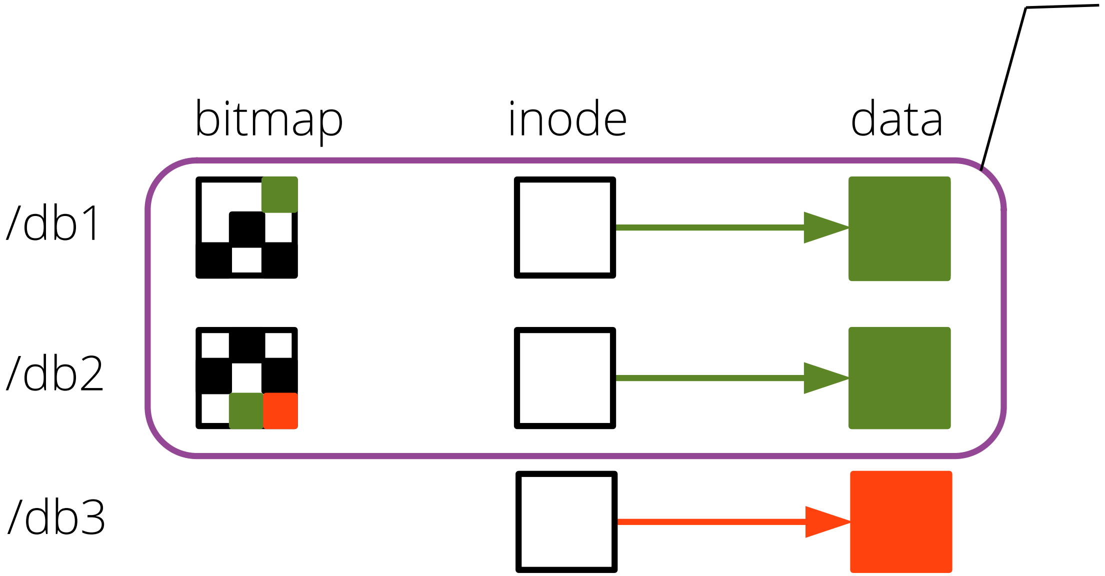
What if /db3 bitmap locates in the /db2 bitmap?

Transactional Flash



/db3 data block is allocated.

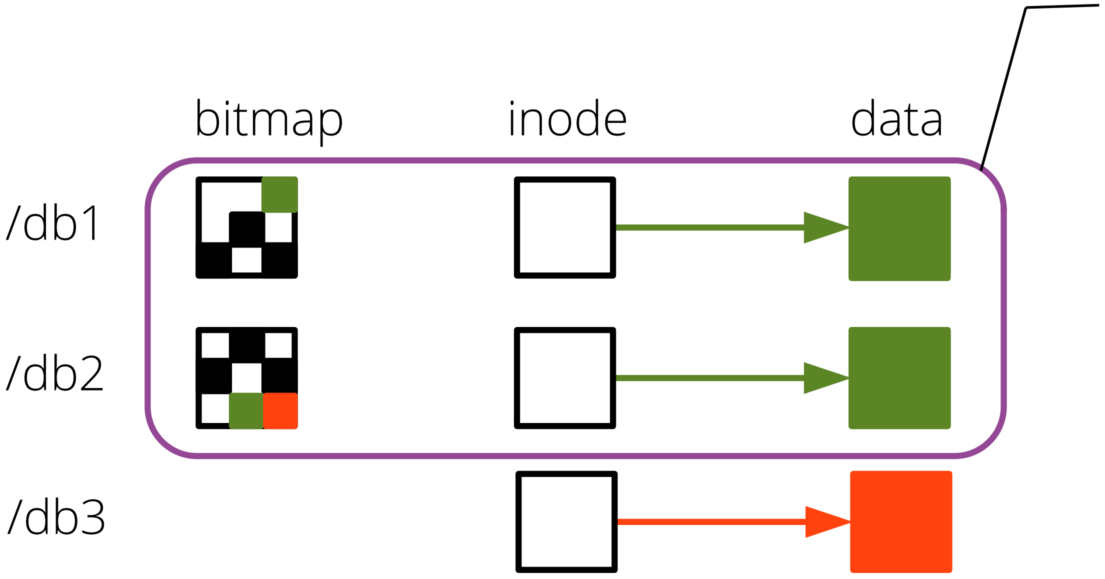
Problem: entangled disk pages



1. cfs_commit()

- /db1 and /db2 are persistent

Problem: entangled disk pages



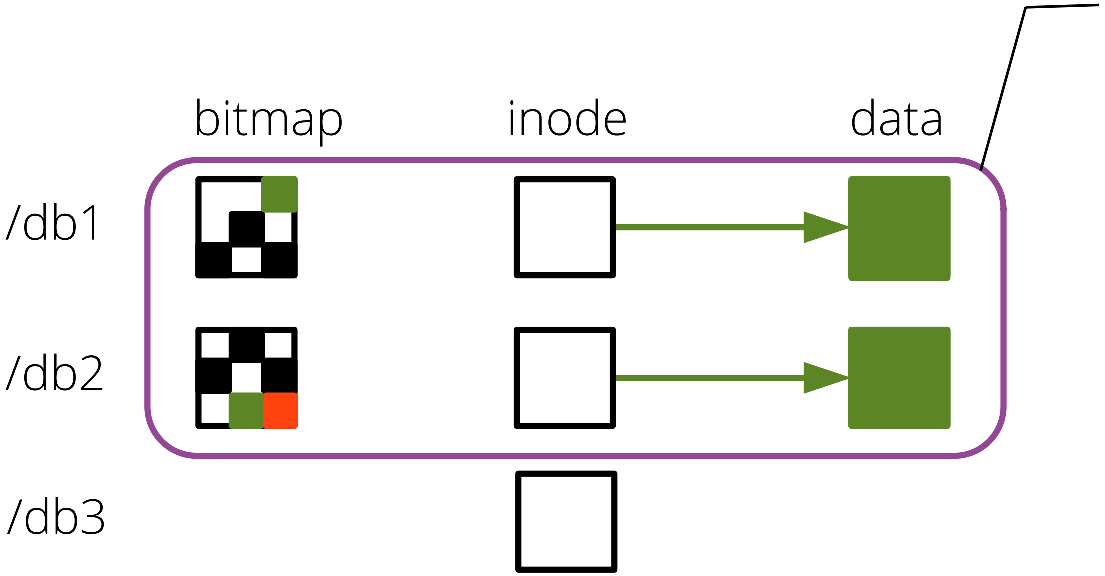
1. cfs_commit()

- /db1 and /db2 are persistent

2. Sudden Power Outage

- /db3 changes are lost.

Problem: entangled disk pages



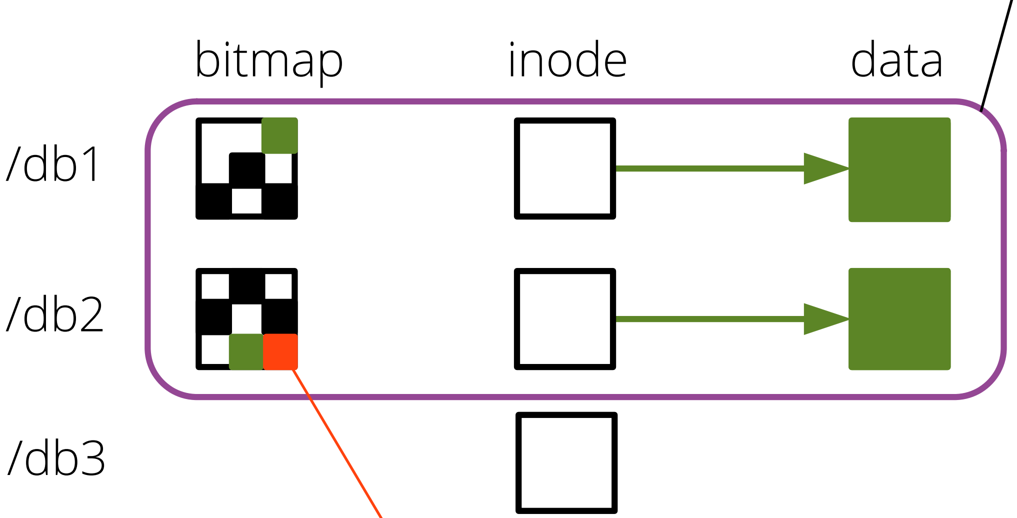
1. cfs_commit()

- /db1 and /db2 are persistent

2. Sudden Power Outage

- /db3 changes are lost.

Problem: entangled disk pages



1. cfs_commit()

- /db1 and /db2 are persistent

2. Sudden Power Outage

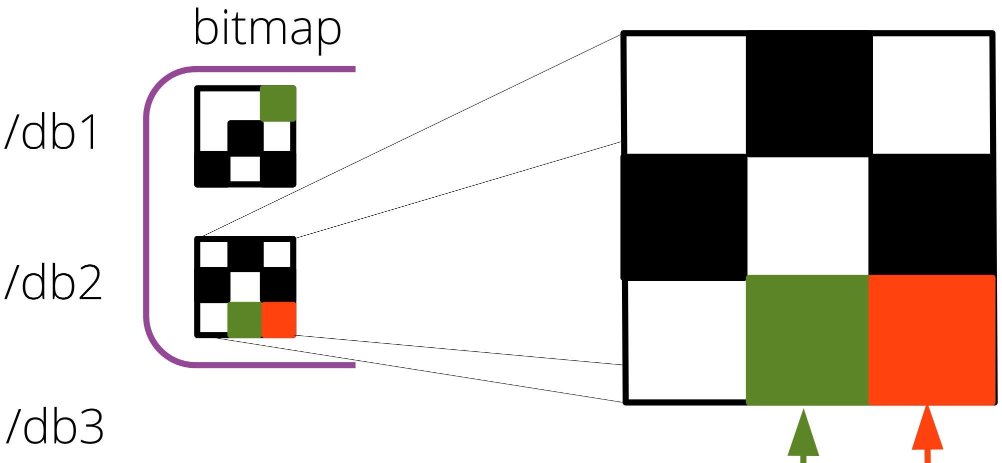
- /db3 changes are lost.

~~***/db3 data block is allocated.***~~

Incorrect bitmap

- Free block is marked as allocated.

Problem: entangled disk pages



False Sharing of Metadata Pages

```
+ cfs_begin();
  write(/db1, “new”);
  write(/db2, “new”);
+ cfs_commit();
```

```
+ cfs_begin();
  write(/db3, “new”);
  …
-  sudden power outage
```

#2. Metadata False Sharing
: In-Memory Metadata Logging

- Operational Logging for in-memory metadata change
  - *toggle_bit(free_block_bitmap, LBA)*
  - *sub(free_block_count, 1)*
- Maintain two versions of in-memory metadata to selectively propagate only relevant changes to storage
  - Memory version: on-going modification
  - Storage version: committed version, used for storage IO

REDO or UNDO operational logs

```
cfs_commit() {
  storage version += REDO(logs);
  write(txid, storage version);
  commit(txid);
}
```

```
cfs_abort() {
  memory version -= UNDO(logs);
  abort(txid);
}
```

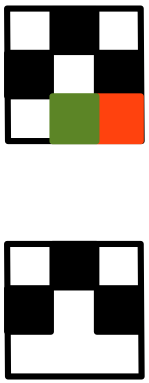
Example: Two apps. are running

In-Memory Metadata

bitmap

Memory Version

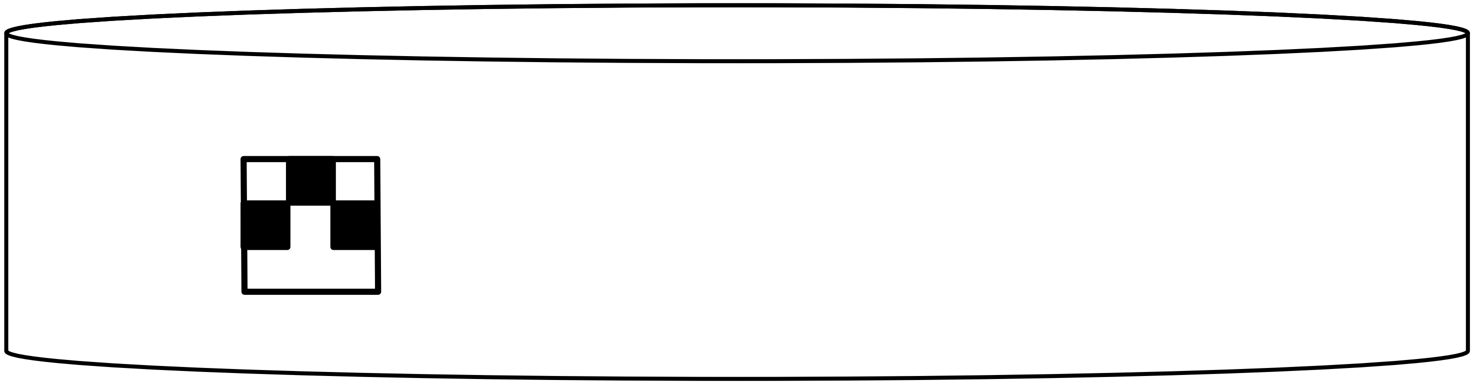
Storage Version



Operational Log for Atomic Propagation Group

App 1	App 2
Turn on a bit 🟩	Turn on a bit 🟥

Transactional Flash

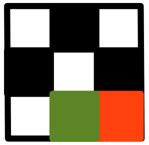


Example: Two apps. are running

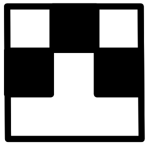
In-Memory Metadata

bitmap

Memory Version



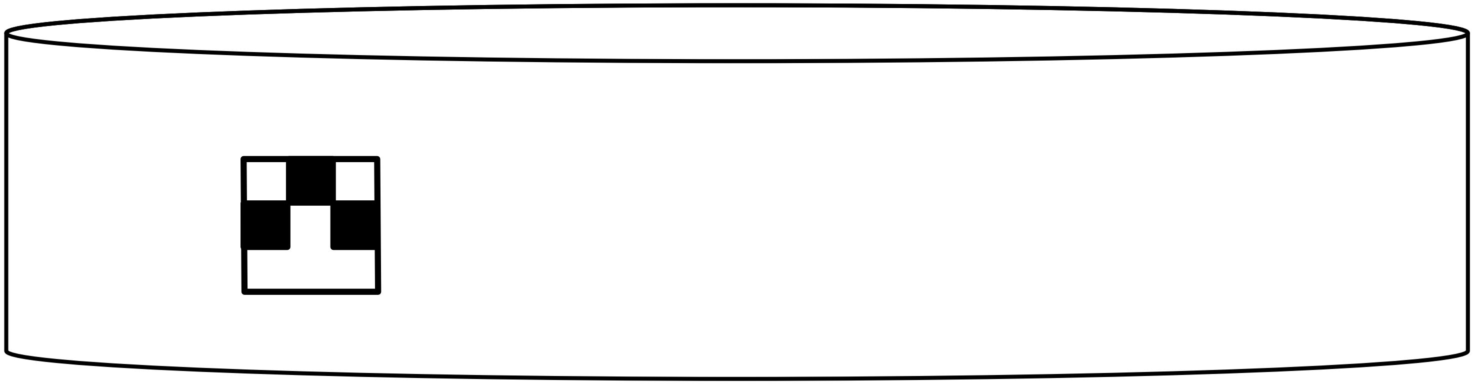
Storage Version



Operational Log
for Atomic Propagation Group

App 1	App 2
Turn on a bit 🟩	Turn on a bit 🟥
cfs_commit()	

Transactional Flash

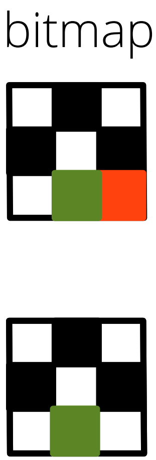


Example: Two apps. are running

In-Memory Metadata

Memory Version

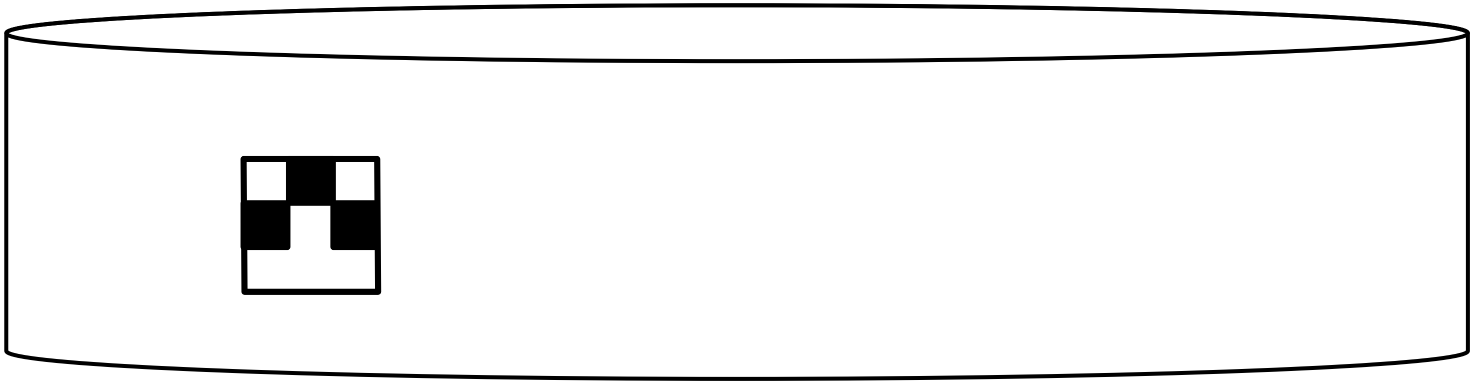
Storage Version



Operational Log
for Atomic Propagation Group

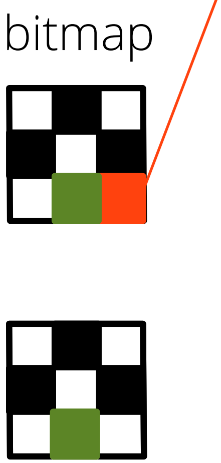
App 1	App 2
✓ Turn on a bit 🟩	Turn on a bit 🟥
cfs_commit()	

Transactional Flash



Example: Two apps. are running

In-Memory Metadata

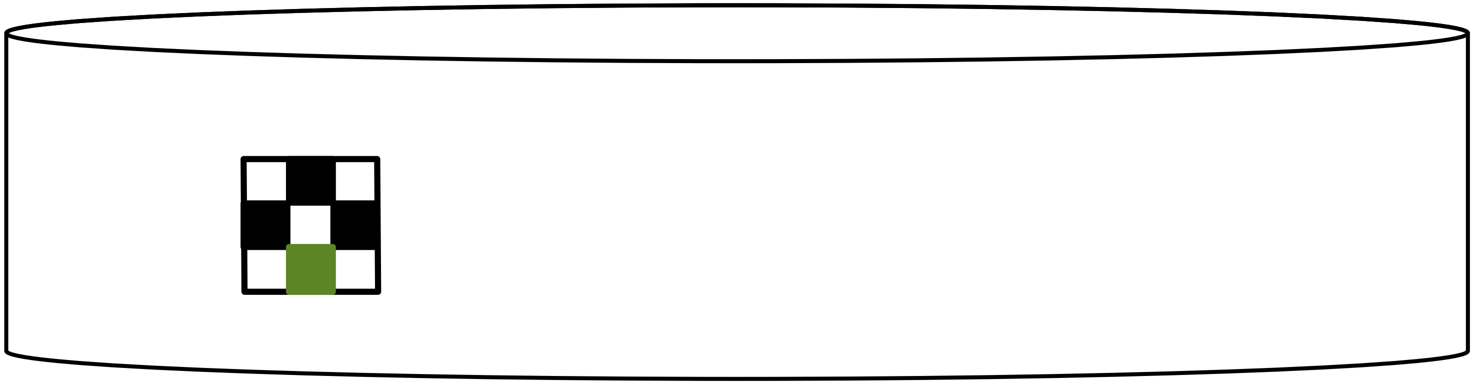


Bitmap for db3 ***is not written.***

Operational Log
for Atomic Propagation Group

App 1	App 2
✅ Turn on a bit 🟩	Turn on a bit 🟥
✅ cfs_commit()	

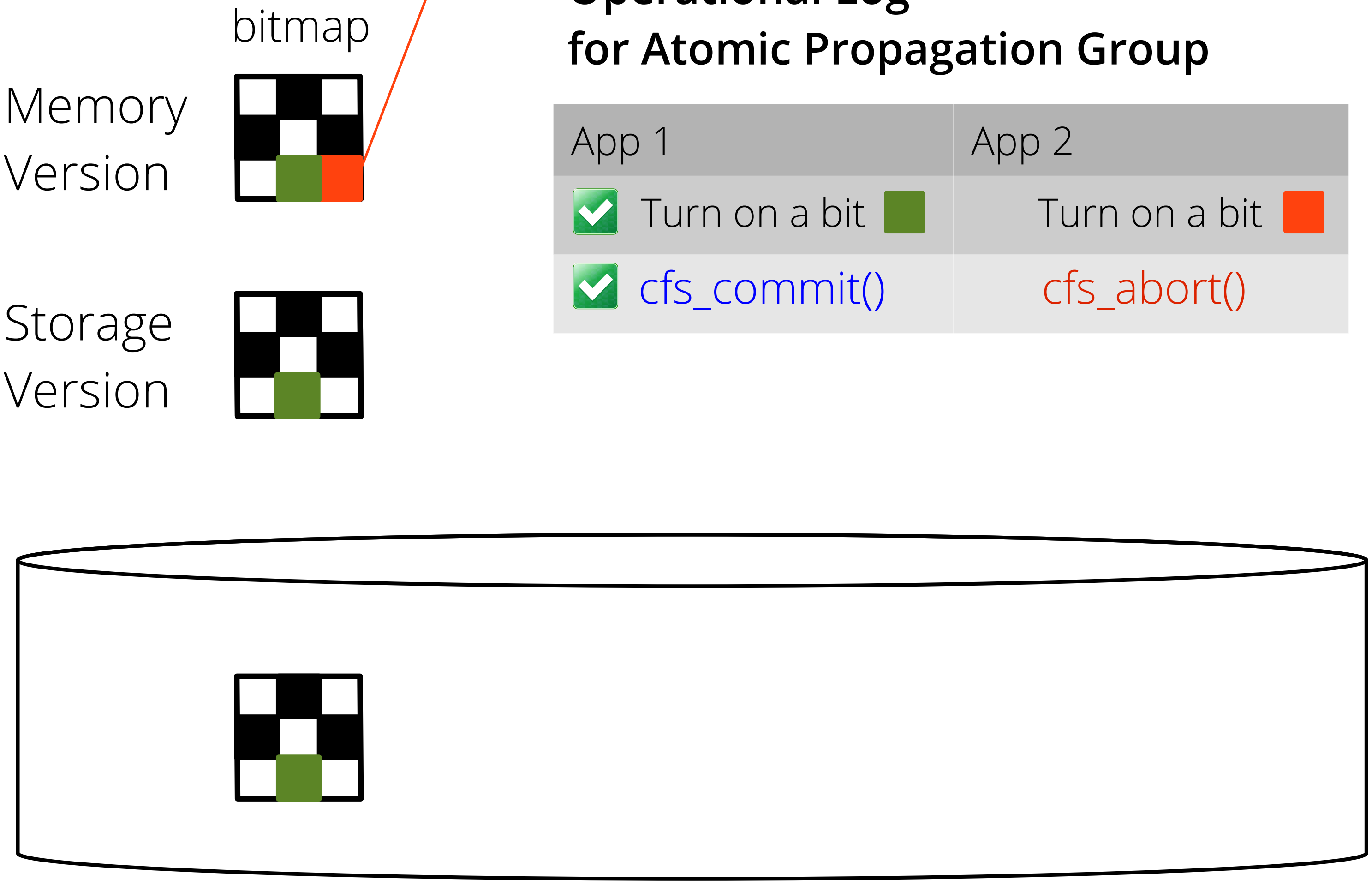
Transactional Flash



Example: Two apps. are running

Bitmap for db3 is not written.

In-Memory Metadata



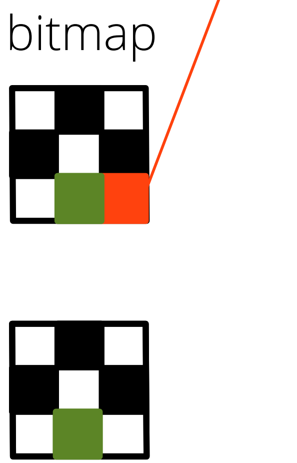
Operational Log
for Atomic Propagation Group

App 1	App 2
✓ Turn on a bit 🟩	Turn on a bit 🟥
✓ cfs_commit()	cfs_abort()

Transactional Flash

Example: Two apps. are running

In-Memory Metadata

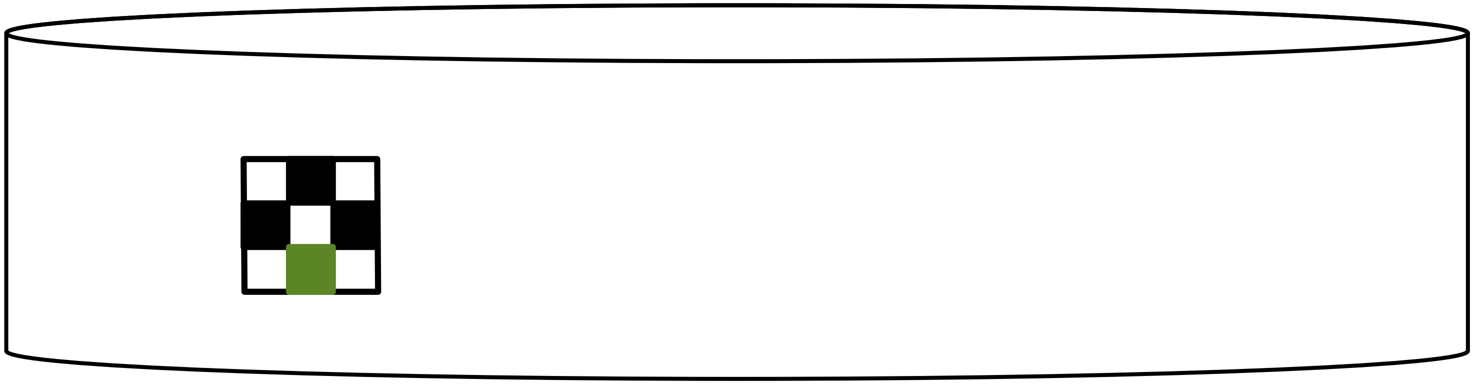


Bitmap for db3 is reverted.

Operational Log
for Atomic Propagation Group

App 1	App 2
✓ Turn on a bit 🟩	⊖ Turn on a bit 🟥
✓ cfs_commit()	⊖ cfs_abort()

Transactional Flash



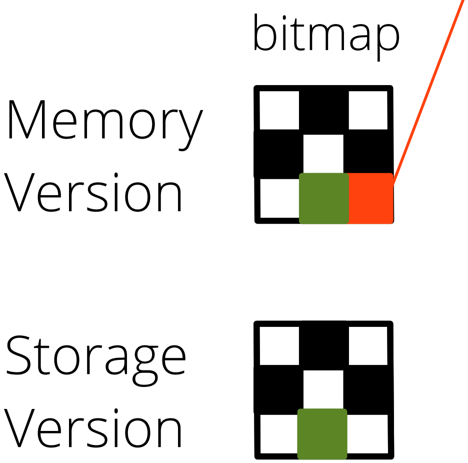
Example: Two apps. are running

***Bitmap for db3* is reverted.**

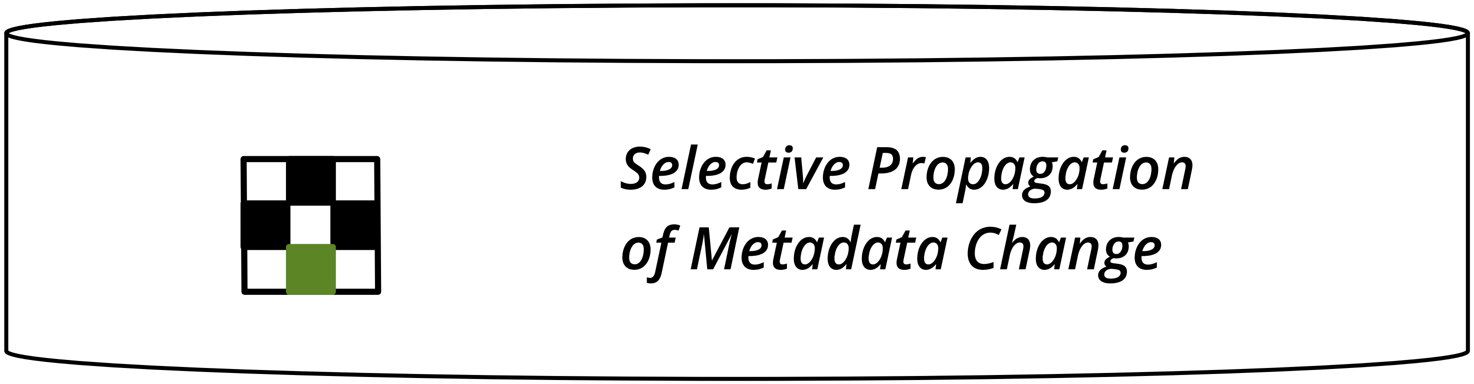
Operational Log
for Atomic Propagation Group

App 1	App 2
✓ Turn on a bit 🟩	⛔ Turn on a bit 🟥
✓ cfs_commit()	⛔ cfs_abort()

In-Memory Metadata



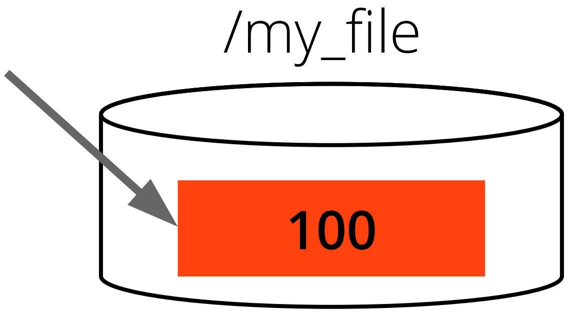
Transactional Flash



Selective Propagation of Metadata Change

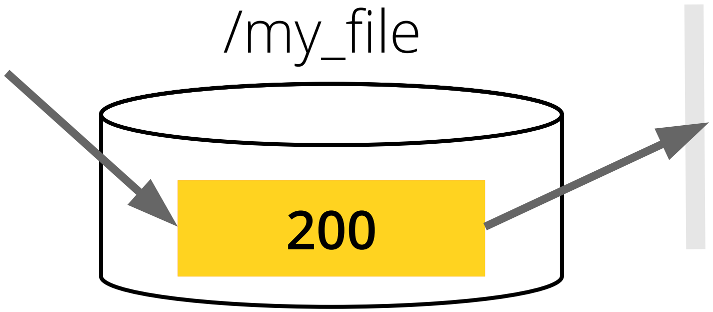
#3. Isolation? Concurrency Control?
: Leave It to Application

```
cfs_begin();
write(/my_file, “200”);
…
```



#3. Isolation? Concurrency Control?
: Leave It to Application

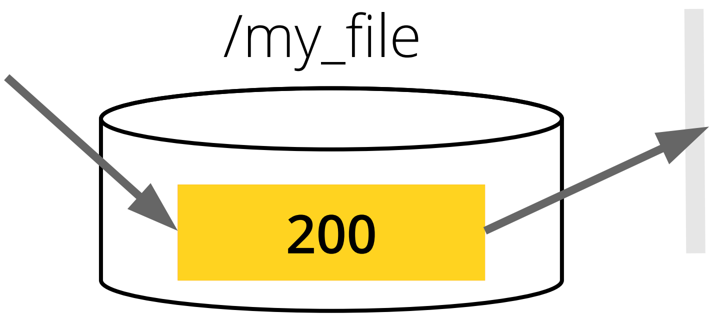
```
cfs_begin();
write(/my_file, “200”);
…
```



```
cfs_begin();
val = read(/my_file);
…
```

#3. Isolation? Concurrency Control?
: Leave It to Application

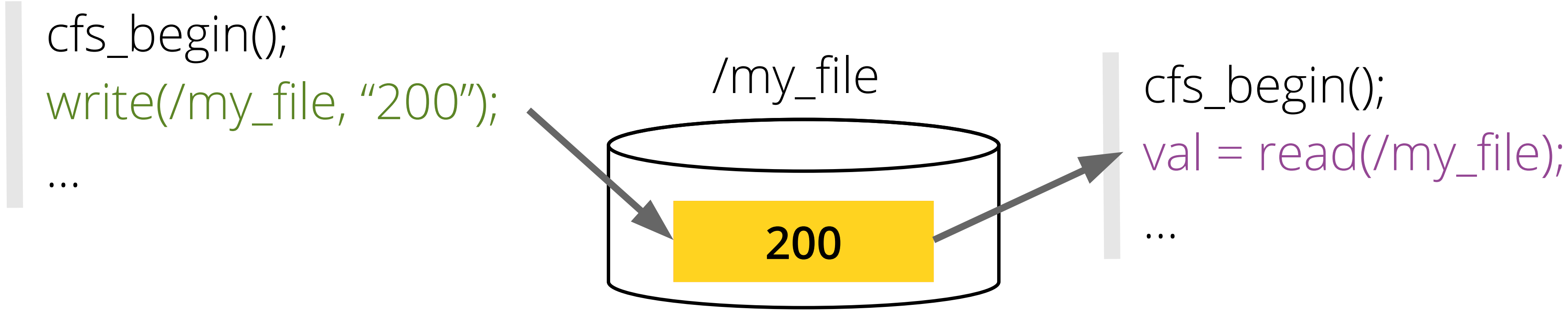
```
cfs_begin();
write(/my_file, “200”);
…
```



```
cfs_begin();
val = read(/my_file);
…
```

What *val* should be either of *100* or *200*?

#3. Isolation? Concurrency Control?
: Leave It to Application



```
cfs_begin();
write(/my_file, “200”);
…
```

/my_file

200

```
cfs_begin();
val = read(/my_file);
…
```

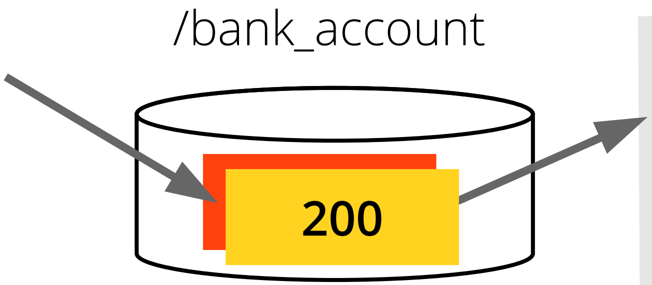
What *val* should be either of *100* or *200*?

It depends on application semantics.

#3. Isolation? Concurrency Control?
: Leave It to Application

MariaDB

```
> START TRANSACTION
> UPDATE account
  SET deposit='200'
> ...
> COMMIT
```



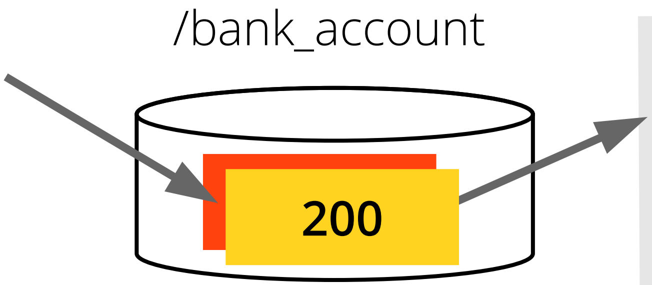
```
> START TRANSACTION
> SELECT deposit
  FROM account
> ...
> COMMIT
```

Deposit of bank account must be 200.

#3. Isolation? Concurrency Control?
: Leave It to Application

MariaDB

```
> START TRANSACTION
> UPDATE account
  SET deposit='200'
> ...
> COMMIT
```

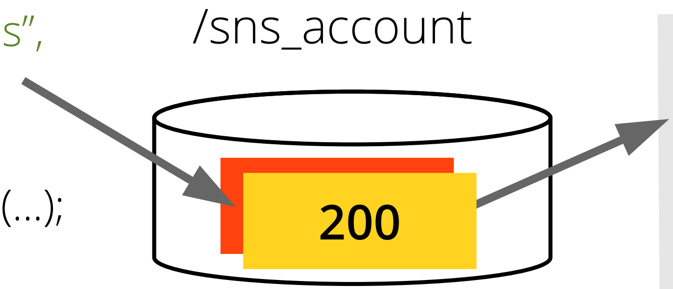


```
> START TRANSACTION
> SELECT deposit
  FROM account
> ...
> COMMIT
```

Deposit of bank account must be 200.

Tokyo Cabinet 8192PiB

```
KV→begin_transaction(...);
KV→set("account.likes",
        "200");
...
KV→end_transaction(...);
```



```
KV→begin_transaction(...);
KV→get("account.deposit");
...
...
KV→end_transaction(...);
```

Likes of SNS account can be either of 100 or 200.

#3. Isolation? Concurrency Control? : Leave It to Application

- Isolation and crash-consistency are orthogonal.
  - Even the SQL standard defines four different isolation levels.
- CFS does not provide its own concurrency control mechanism.
  - If needed, use existing synchronization primitives (e.g., mutex, RW lock, etc).

#4. Legacy Application Support

- System-Wide Atomic Propagation Group
  - Every update from legacy applications belongs.
  - Automatically committed by sync() or page flusher

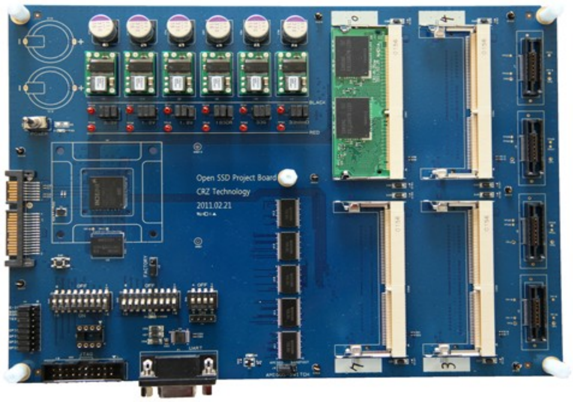
CFS supports legacy applications without any modification.

Outline

- Introduction
- CFS Design
- **Evaluation**
- Conclusion

Implementation

- CFS
  - 5.8k LoC modification of ext4 on Linux 3.10
  - Capture logs by inserting 182 places in ext4
- Transactional Flash
  - OpenSSD: 8KB, 128 pages/block, 8GB w/ SATA2
  - X-FTL/SSD [Kang:SIGMOD'13]



Real Application & Workloads

Mobile Database	SQLite	+ RLBench + Facebook
SQL Database	MariaDB	+ SysBench + LinkBench
Key/Value Store	Tokyo Cabinet 8192PiB	+ kctreetest + db_bench

Text Editor	Vim
Package Installer	debian apt-get

Real Application & Workloads

Mobile Database

SQLite

+ RLBench
+ Facebook

Key/Value Store

Tokyo Cabinet 8192PiB

+ kctreetest
+ db_bench

Text Editor

vim

Package Installer

debian

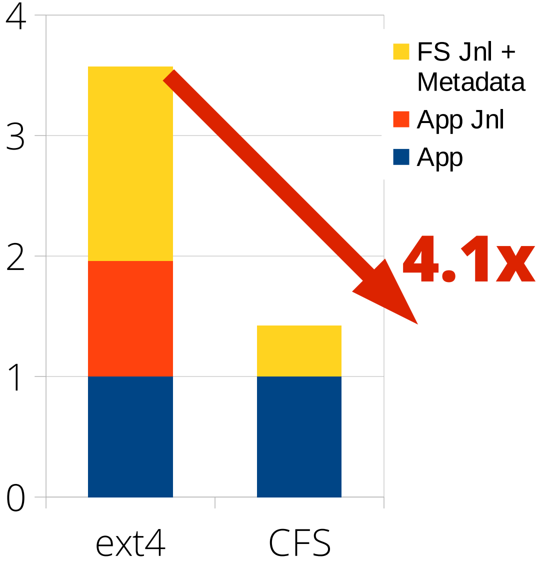
apt-get

CFS is easy-to-use!

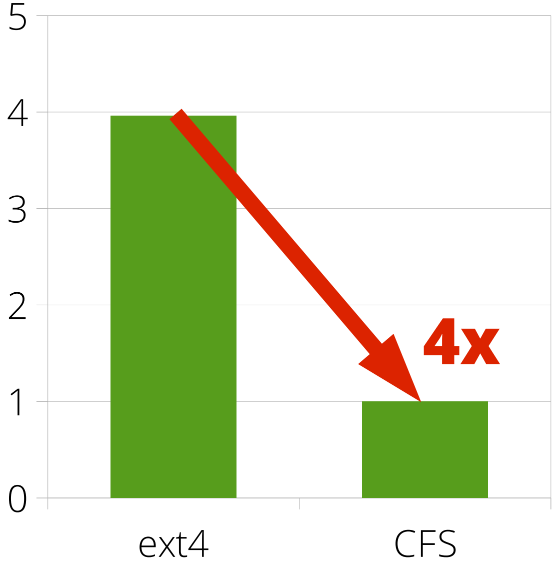
: 317 LoC changes out of 3.5 million

SQLite
+ Facebook App. SQL Trace

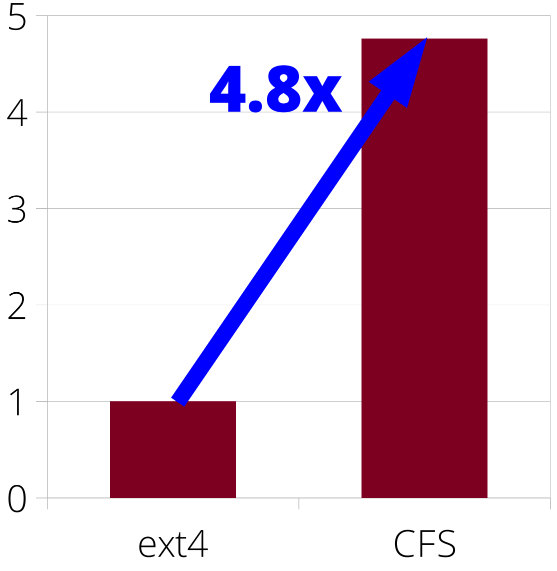
Write Amplification



Disk Flush



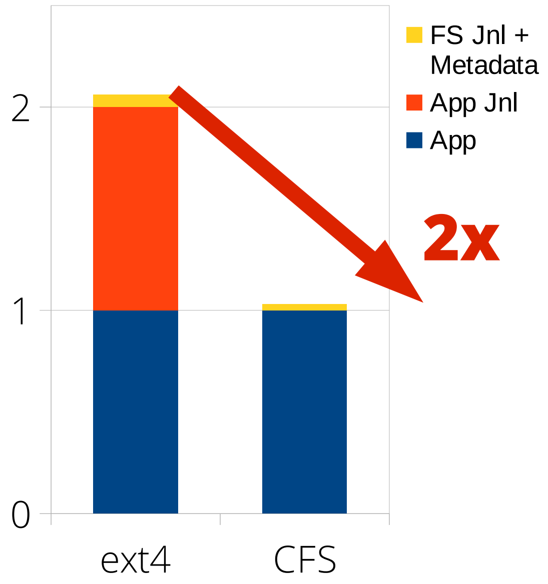
Performance



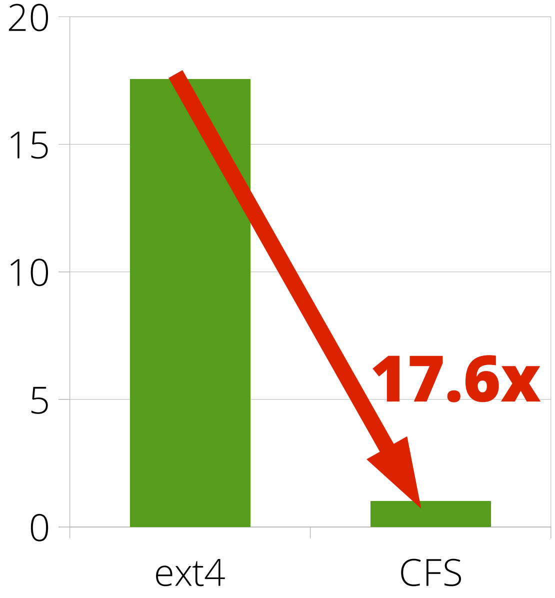
Ext4: ordered journal mode
SQLite: rollback journal mode

MariaDB
+ LinkBench

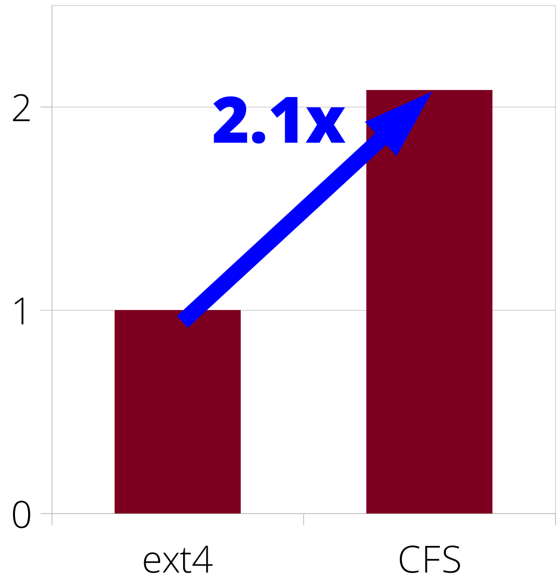
Write Amplification



Disk Flush

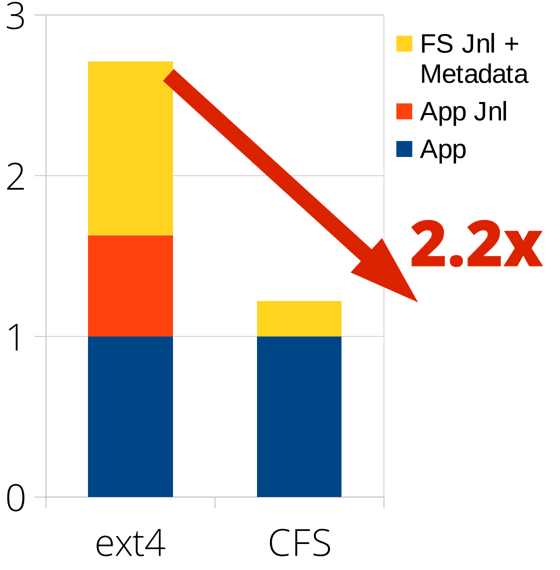


Performance

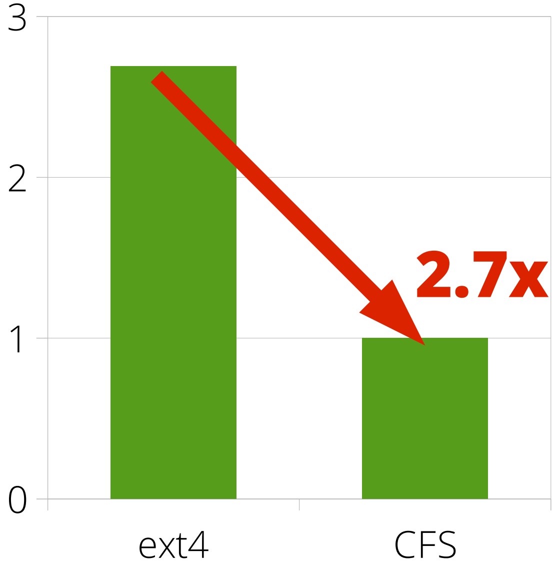


Ext4: ordered journal mode

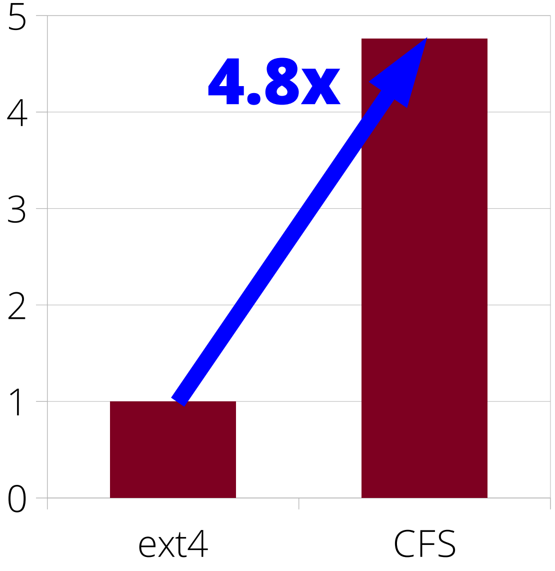
KyotoCabinet
+ db_bench



Write Amplification



Disk Flush



Performance

Ext4: ordered journal mode

Conclusion

- Current mechanisms for crash consistency is complex, slow, and error-prone.
- CFS simplifies application's crash consistency using transactional flash.
  - Atomic propagation group
  - In-memory metadata logging
- Our evaluation shows
  - Less write: 2 ~ 4x↓
  - Less disk flush: 3 ~ 17x↓
  - Higher performance: 2 ~ 5x↑

Thank you!

Changwoo Min
changwoo@gatech.edu
Woon-Hak Kang[†], Taesoo Kim, Sang-Won Lee[†], Young Ik Eom[†]

Georgia Institute of Technology
[†]Sungkyunkwan University

Questions?