

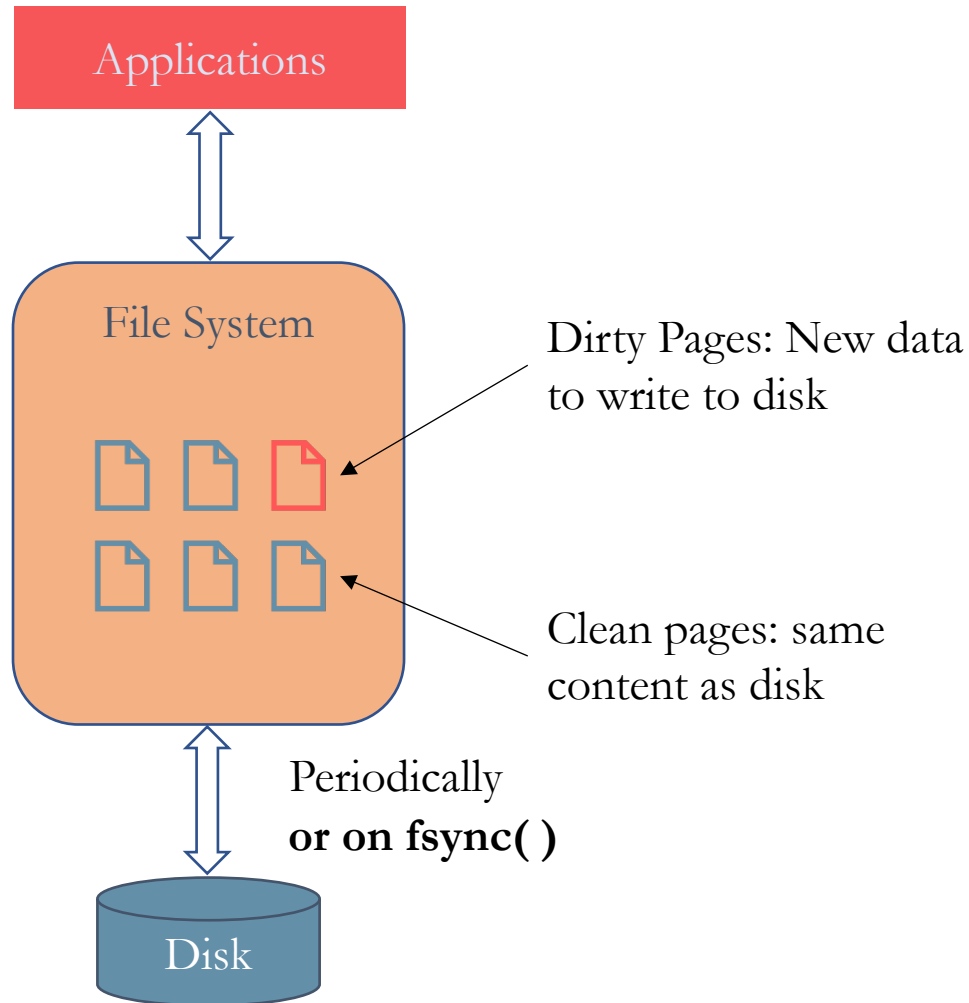


Can Applications Recover from `fsync` Failures?

Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan,
Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison

How does data reach the disk?



Applications use the file system

- System calls – `open()`, `read()`, `write()`

For Performance

- Data buffered in the page cache
- Modified pages are marked dirty
- Periodically flushed to disk
 - Vulnerable to data loss while in RAM

For Correctness

- Dirty pages can be flushed immediately using `fsync()`

fsync is really important

Many applications care about durability

- Ensure data on non-volatile storage before acknowledging client

Devices have volatile storage

- Direct IO: fsync can issue a FLUSH command

Ordering of writes is important

- Force to disk with fsync before writing the next
- Optimistic Crash Consistency Chidambaram et al. [SOSP'13]
 - Decouples ordering from durability

It's hard to get durability correct

Applications find it difficult

- Even when fsync works correctly

Example: persisting a newly created file

```
creat(/d/foo)
```

```
write(/d/foo, "abcd")
```

```
fsync(/d/foo)
```

```
fsync(/d)
```

← Ensure that directory entry is persisted

All File Systems Are Not Created Equal Pillai et al. [OSDI'14]

- Studied 11 applications
- Update protocols are tricky
- More than 30 vulnerabilities under ext3, ext4, btrfs

fsync can fail

Durability gets harder to get right

Failures **before** interacting with disk

- Invalid arguments, insufficient space
- Easy to handle

Failures **while** interacting with disk

- EIO: An error occurred during synchronization
- Transient disk errors, network disconnects
- In-memory data structures may need to be reverted

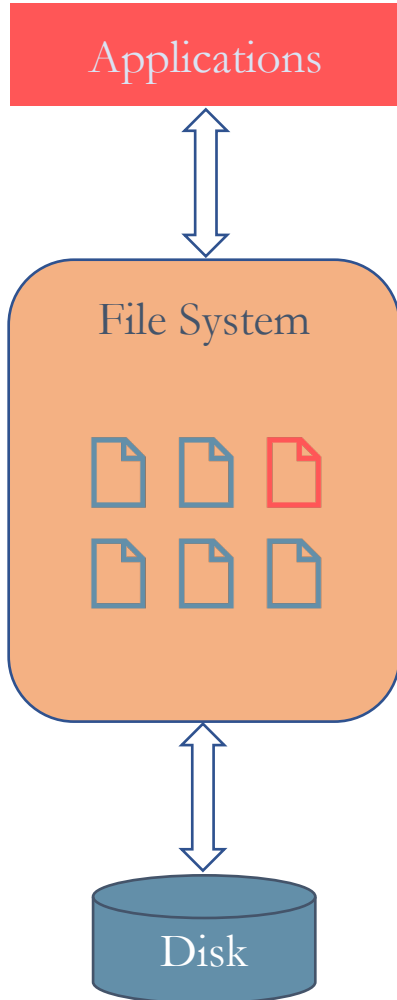
Why care about fsync failures?

“About a year ago the PostgreSQL community discovered that fsync (on Linux and some BSD systems) may not work the way we always thought it is [sic], with possibly disastrous consequences for data durability/consistency (which is something the PostgreSQL community really values).”

- Tomas Vondra, FOSDEM 2019

Our work

Systematically understand fsync failures



- ② **Application reactions** to fsync failures
 - Redis, LMDB, LevelDB, SQLite, PostgreSQL
- ① **File system reactions** to fsync failures
 - Ext4, XFS, Btrfs

File System Results

All file systems mark dirty pages clean on fsync failure

- Retries are ineffective

File systems do not handle errors during fsync uniformly

- Content in pages is different
 - Latest data (ext4, XFS), Old data (Btrfs)
- Failure notifications not always immediate
 - Ext4 data mode reports errors later

In-memory data structures are not entirely reverted after fsync failure

- Garbage/Zeros in the files
 - Free space and block allocation unaltered (ext4, XFS)
 - User-space file descriptor offset unaltered (Btrfs)

Application Results

Simple strategies fail

- Retries are ineffective
- Crash/Restart can be incorrect
 - False Failures: Indicate failure but actually succeed
 - Incorrect recovery from WAL using the page cache

Defenseless against late error reporting

- Ext4 data mode
 - Every application faced data loss
 - Most faced corruption (all except PostgreSQL)

Copy-on-write is good, but not invincible

- Btrfs is bad for rollback strategies
 - But seems good for WAL recovery

Outline

Introduction

File Systems

- Methodology (dm-loki, workloads)
- Results

Applications

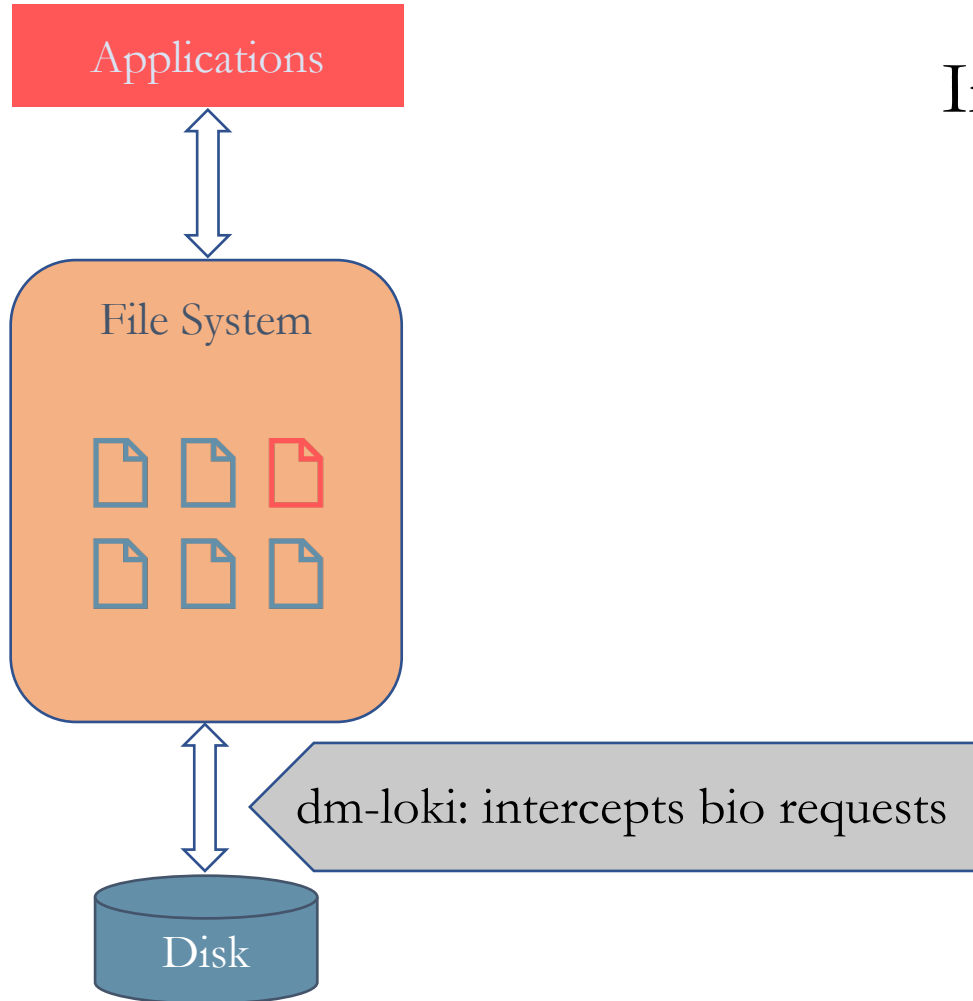
- Methodology (CuttleFS)
- Results

Challenges and Directions

Summary

File System | Methodology: Fault Injection

Goal: Understand file system reactions to fsync failures without modifying the kernel



Intercept all block requests that go to disk

- Custom device mapper target – dm-loki
 - Trace bio requests
 - Fail i^{th} write to sector/block

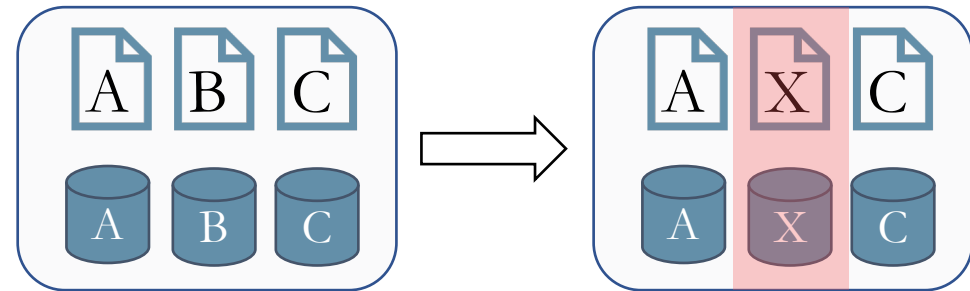
File System | Methodology: Workloads

Common write patterns in applications

- Reduced to simplest form

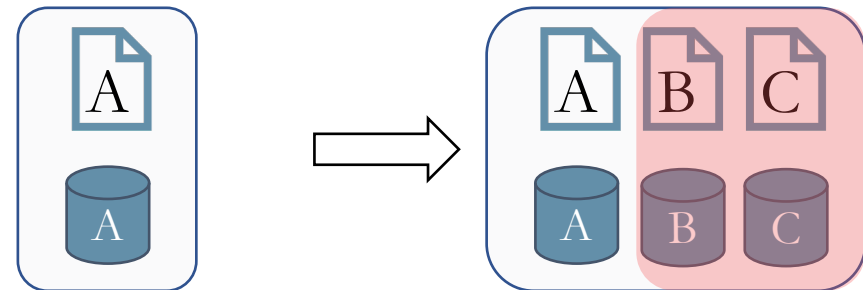
Single Block Update

- Modify a single block in a file
- Examples:
 - LMDB, PostgreSQL, SQLite



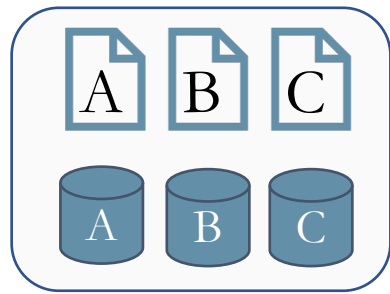
Multi Block Append

- Add new blocks to the end of a file
- Examples:
 - Redis append-only file
 - Write-ahead logs
 - PostgreSQL, LevelDB, SQLite



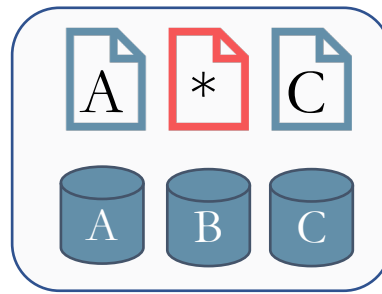
File System | Result #1: Clean Pages

Dirty page is marked clean after fsync failure on all three file systems



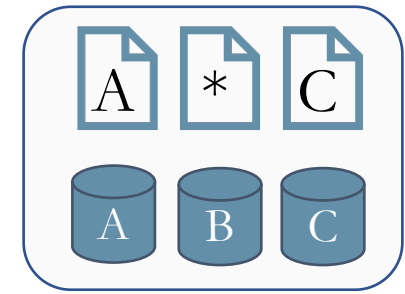
1

Modify middle page



2

fsync() fails
Page is marked clean



3

Feature, not bug

- Avoids memory leaks when user removes USB stick

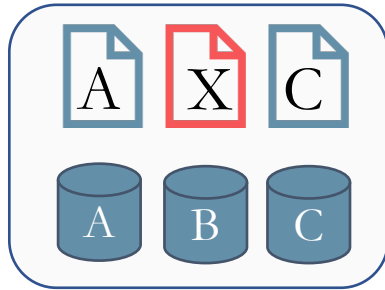
Retries are ineffective

- No more dirty pages on the next fsync

File System | Result #2a: Page Content

File systems do not handle fsync errors uniformly

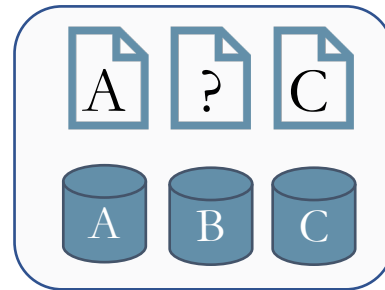
- Page content depends on file system



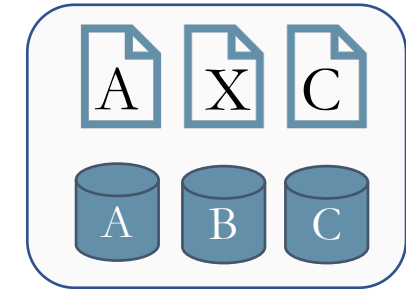
Middle page modified

1

fsync() fails
Page is marked clean

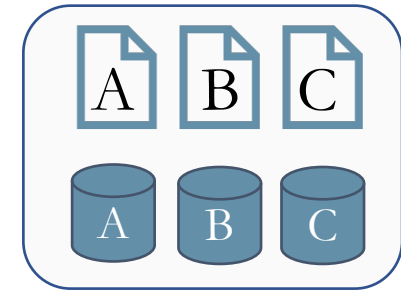


2



2a

Ext4 and XFS
Keep latest data



2b

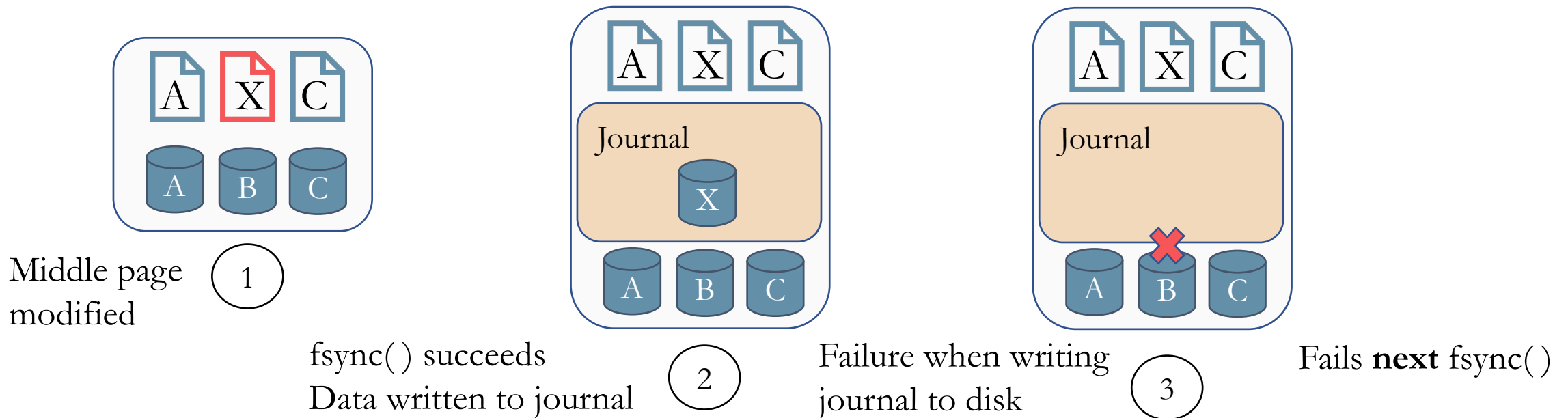
Btrfs reverts state

Cannot reliably depend on page cache content after an fsync failure

File System | Result #2b: Notifications

File systems do not report fsync failures uniformly

- Ext4 data mode reports failures later
- Ext4 ordered mode, XFS, Btrfs report immediately



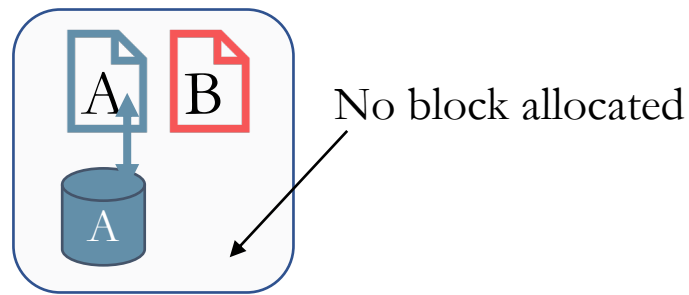
Ext4 data mode reports success too early

- Two fsyncs can solve the problem

File System | Result #3: In-memory state

In-memory data structures are not entirely reverted

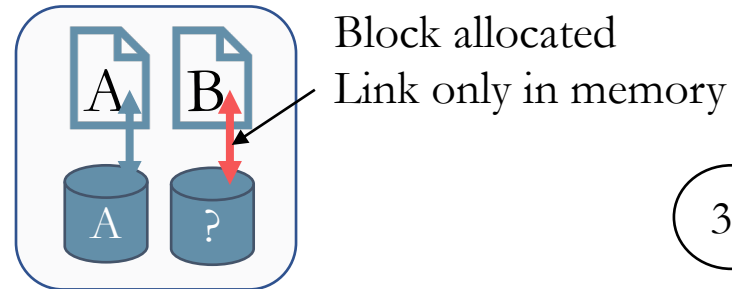
- Free space and block allocation unaltered in ext4, XFS



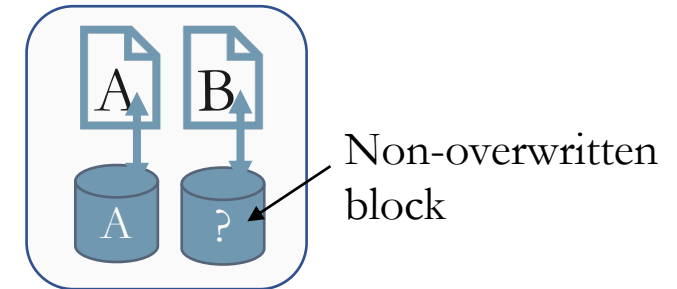
Write to end of file

1

fsync() fails
No metadata persisted

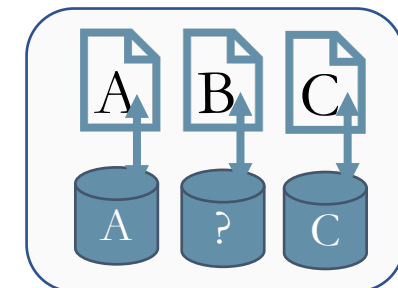


2



3a

Link persisted after some time or unmount



3b

Link persisted if future writes + fsync succeeds

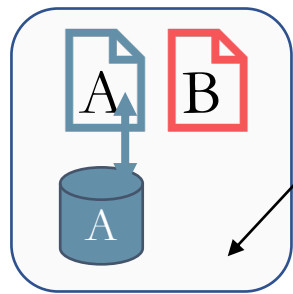
On EXT4 and XFS -

Applications read block's old contents - corruption

File System | Result #3: In-memory state

In-memory data structures are not entirely reverted

- Holes in Btrfs as file descriptor offset is not reverted

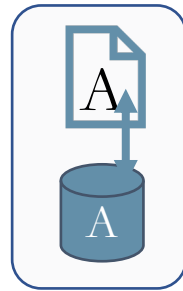


No block allocated

Write to end of file

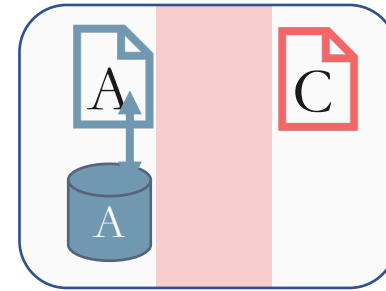
1

fsync() fails
State is reverted



2

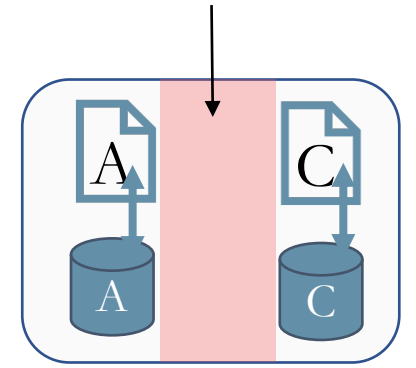
Next write is at updated offset



3

fsync() persists at updated offset

Hole in place of B



4

On Btrfs -

Application reads zeroes at the hole offset - corruption

File System | Results Summary

After fsync failure ...

Dirty pages are marked clean

- Retries are ineffective

Errors are not handled uniformly

- Page content varies across file systems
- Notifications are not always immediate


In-memory data structures are not correct

- Future operations cause non-overwritten blocks (ext4, XFS), holes (Btrfs)
- Both are corruptions to the application

Applications

Applications

Five widely used applications

	Key Value Store	Relational Database
Embedded	 symas LMDB v0.9.24	 SQLite v3.30.1
Server	 redis v5.0.7	 PostgreSQL v12.0

Applications | Methodology

Goal: Are application strategies effective when fsync fails

Simple workload

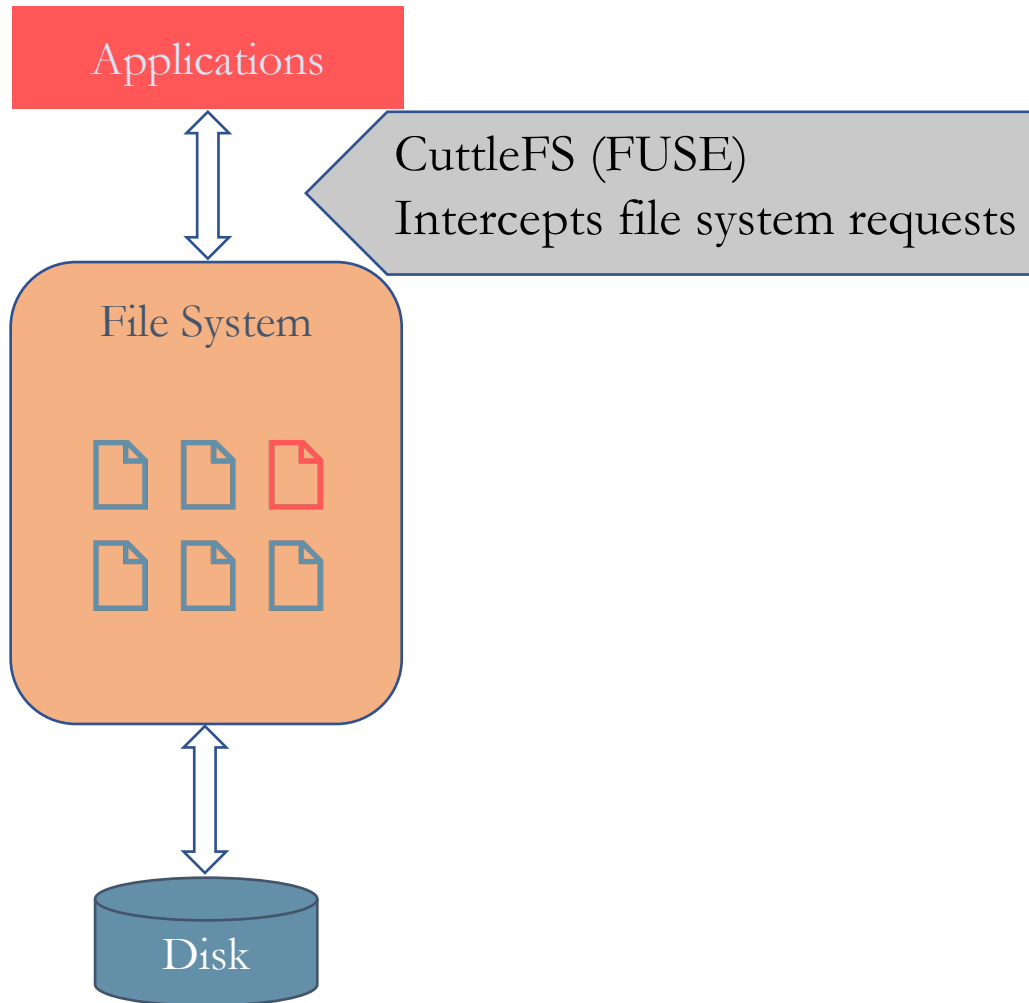
- Insert/Update a key-value pair
- Use two-column table for RDBMS

Make fsync fail

Dump all key-value pairs

- When running
- On application restart
- On page eviction
- On machine restart

Applications | Methodology: CuttleFS



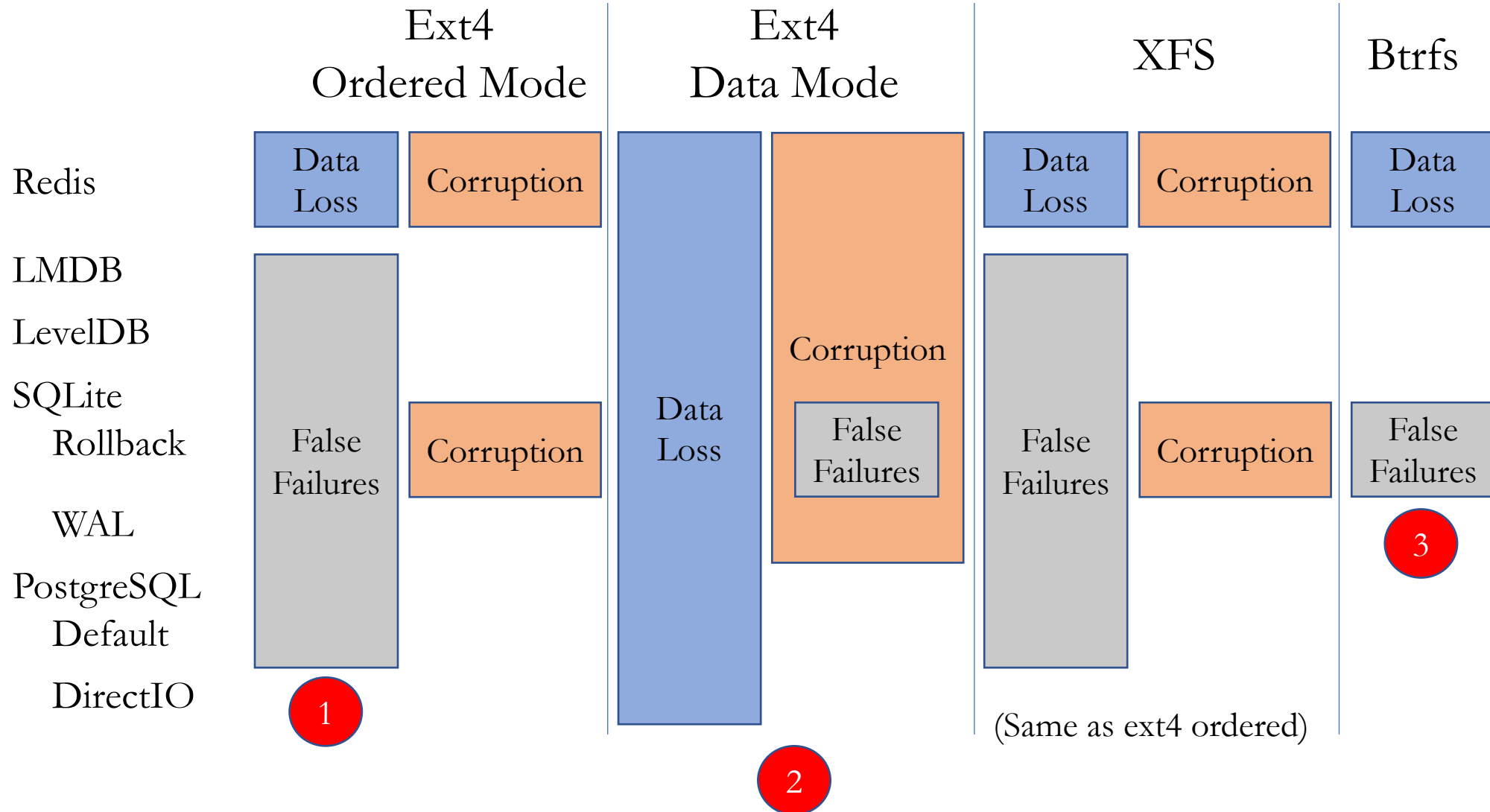
Deterministic fault injection with configurable post-failure reactions

- Fail file offsets, not block numbers

User-space page cache

- Easy to simulate different post-failure reactions
 - Dirty or clean pages
 - New or old content
 - Immediate or late error reporting
- Fine grained control over page eviction

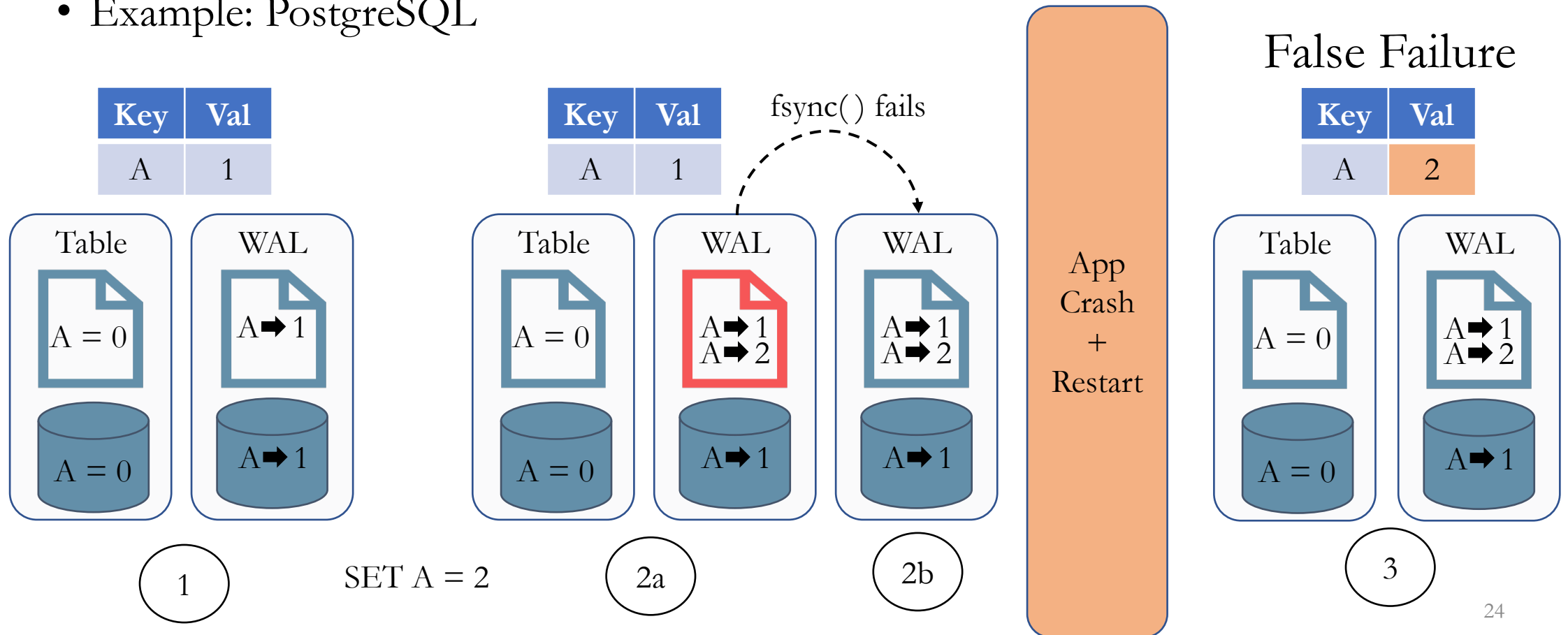
Applications | Results: Overview



Applications | Results #1: Crash/Restart

Simple strategies fail

- Crash/restart is incorrect: recovers wrong data from page cache
- Example: PostgreSQL



Applications | Results #1: False Failures

False Failures: Indicate failure but actually succeed

	Expected State	Actual State
Initially	A=100	A=100
UPDATE Table SET A = A - 1		
Reports failure	A=100	A=99
Retry...		
UPDATE Table SET A = A - 1	A=99	A=98

False Failure

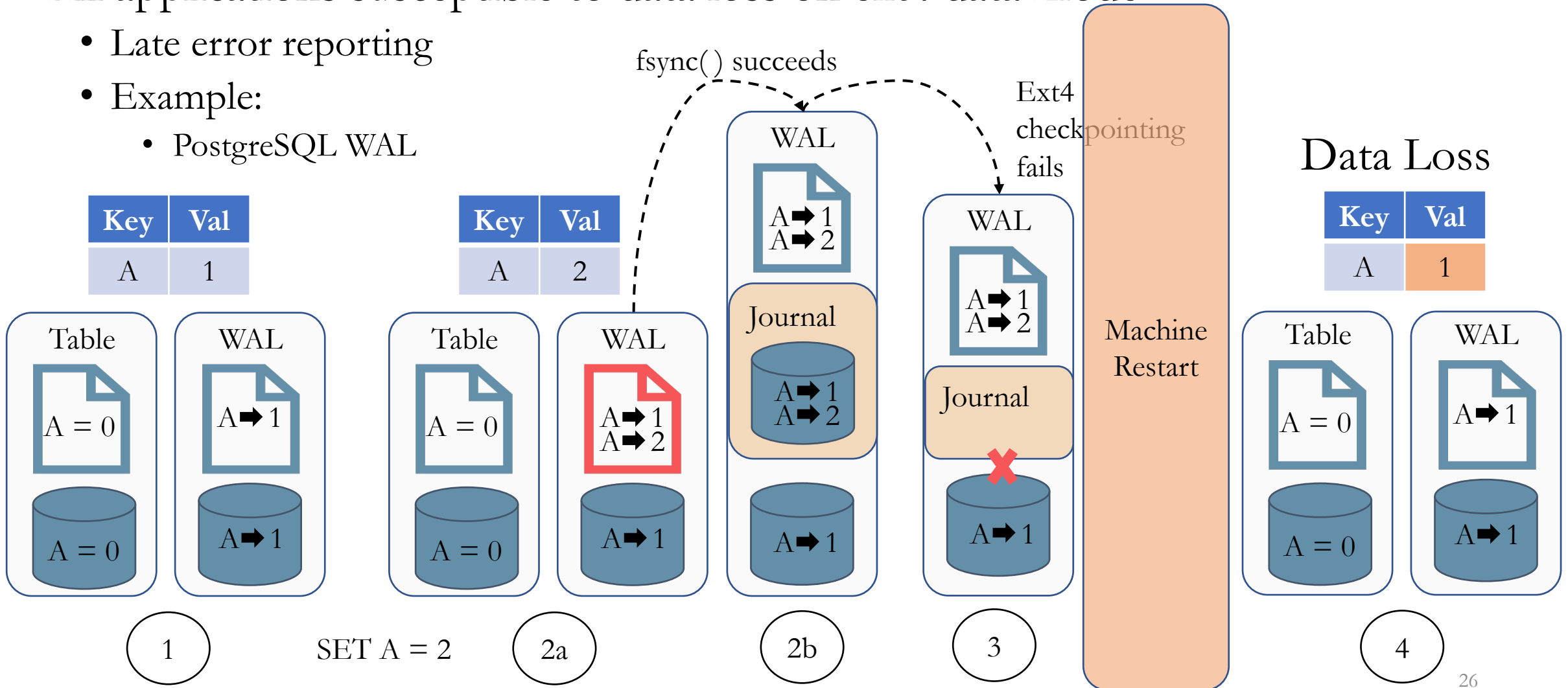
Double Decrement

PostgreSQL, SQLite, LevelDB WAL are affected

Applications | Results #2: Late Error Reporting

All applications susceptible to data loss on ext4 data mode

- Late error reporting
- Example:
 - PostgreSQL WAL

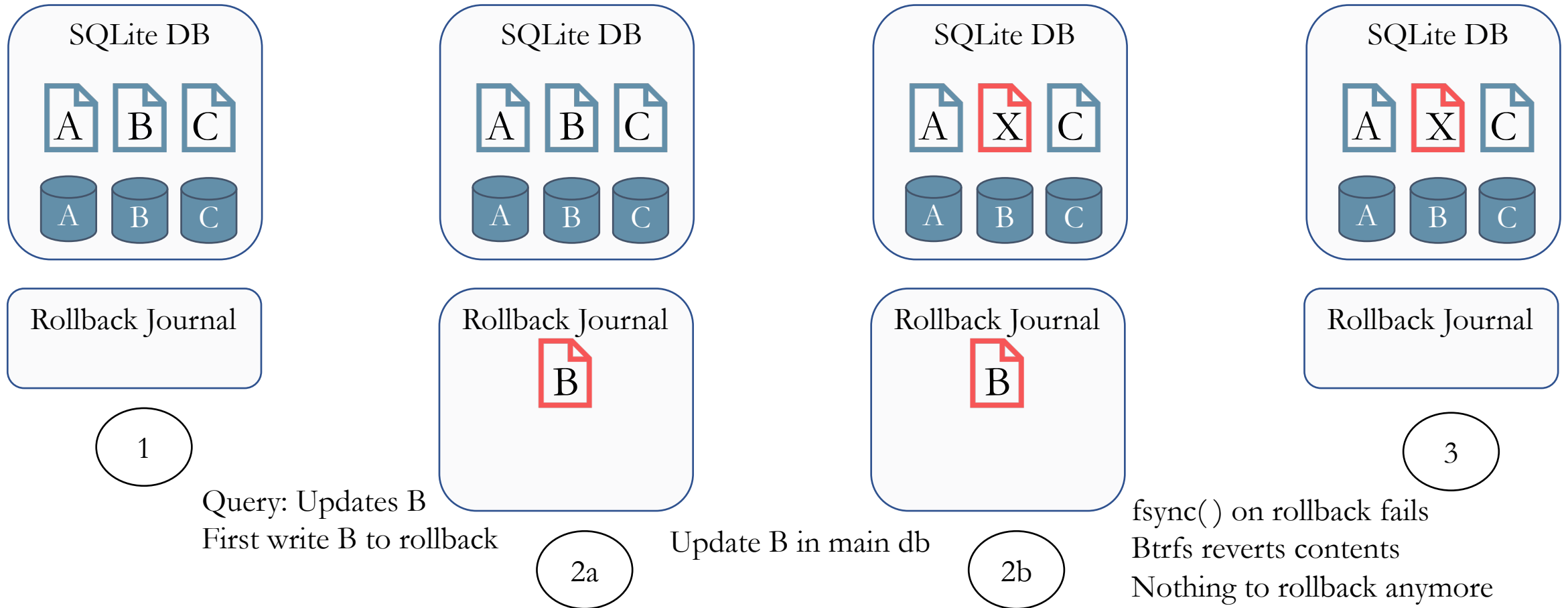


Applications | Results #3: Btrfs winning?

Btrfs copy-on-write strategy is good, but not entirely

- Reverts page cache to match disk
 - Works well for recovery from WAL
 - Bad for rollback techniques
 - Example: SQLite rollback mode

Applications | Results #3: Btrfs winning?



Rollback should not assume page-cache contents
Corruptions in ext4 ordered mode / XFS.

False Failure

Applications | Results Summary

Simple strategies fail

- Applications have moved away from retries
- Crash/Restart not entirely correct
 - Don't trust the page cache while recovering

Defenseless against late error reporting

- Ext4 Data Mode
 - Data loss in all applications
 - Corruptions in some
 - Double fsync should help

Copy-on-write file systems look promising

- Btrfs
 - Works well with write-ahead logs
 - Problematic with rollback journals

Wrapping Up

Can applications recover from fsync failures?

- Maybe, if ...
 - Developers write file-system specific code

Need to standardize file-system behavior for fsync failures

Challenges and Directions

How should post-failure behavior be standardized?

- FreeBSD re-dirties pages

Should applications code for specific file systems?

- Currently, OS-specific

We need a stronger contract for failed intentions (ext4 data mode)

Fault injection

- Don't mock system calls
 - Exercise file-system error handling
 - dm-loki: <https://github.com/WiscADSL/dm-loki>
- Mock the file-system error handling
 - CuttleFS: <https://github.com/WiscADSL/cuttlefs>

Summary

Durability is important

- Hard to get right
- fsync is essential

Failures are inevitable

- We don't handle them uniformly

Applications have different strategies to achieve durability

- No single strategy works well on all file systems

Questions?

Anthony Rebello

- arebello@wisc.edu
- <https://github.com/WiscADSL/cuttlefs>
- <https://github.com/WiscADSL/dm-loki>

Thank You