File Systems

Peizhou Wu Zhenyu Wang Zihua Zeng

December 19, 2025

Southern University of Science and Technology

Outline

Log-structured File Systems

Flash-based SSDs

Crash Consistency: FSCK and Journaling

Log-structured File Systems

Motivation of LFS

- *System memories are growing:* FS performance largely depends on write performance.
- *Large gap between random I/O and sequential I/O performance*
- *Existing file systems perform poorly on many common workloads*
- *File systems are not RAID-aware*

Writing Sequentially And Effectively

Gap between sequential writes will incur delay caused by rotation.

Solution: you must issue a large number of contiguous writes to the drive in order to achieve good write performance.

How LFS works: When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory segment; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk.

Writing Sequentially And Effectively

Write buffering: Accumulate updates in memory and write them to the disk all at once.

The large chunk of updates LFS writes at one time: segment.

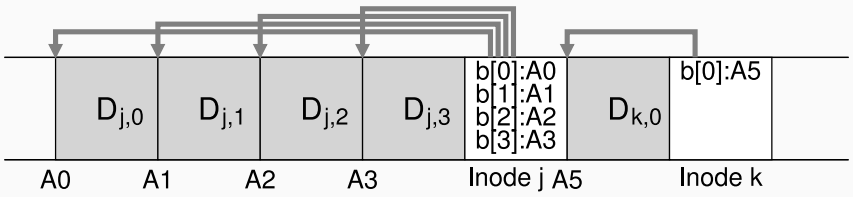


Figure 1: Segment example

How Much To Buffer?

Depends on how high the positioning overhead is in comparison to the transfer rate.

How much do you have to write in order to amortize that cost?

- *Positioning time:* $T_{position}$
- *Peak transfer rate:* R_{peak}
- *Target* $R_{effective}/R_{peak}$ *you want to achieve:* F

$$T_{\text{write}} = T_{\text{position}} + \frac{D}{R_{\text{peak}}}$$

$$R_{\text{effective}} = \frac{D}{T_{\text{write}}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}} = F \times R_{peak}$$

Solve it:

$$D = \frac{F}{1-F} \times R_{\text{peak}} \times T_{\text{position}}$$

How to find inodes?

- *old UNIX FS:* Fixed portion $\rightarrow$ direct calculation for address.
- *FFS*: Find which chunks then direct calculation for address.

What about LFS?

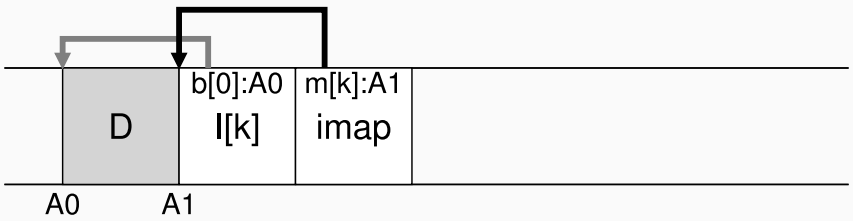
Problem: We've managed to scatter the inodes all throughout the disk!

We never overwrite in place, and thus the latest version of an inode keeps moving.

Solution: Through Indirection: The Inode Map(imap).

Imap takes an inode number as input and produces the disk address of the most recent version of the inode.

Put a piece of the inode map all together with updates onto the disk!



Where to put imap?

Imap needs to be kept persistent. Doing so allows LFS to recover after crash happens.

Put it at fixed part?: NO! More disk seeks, between each update and the fixed location of the imap!

Put a piece of the inode map all together with updates onto the disk!

Find imap piece: checkpoint region (CR)

The checkpoint region contains pointers to the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first.

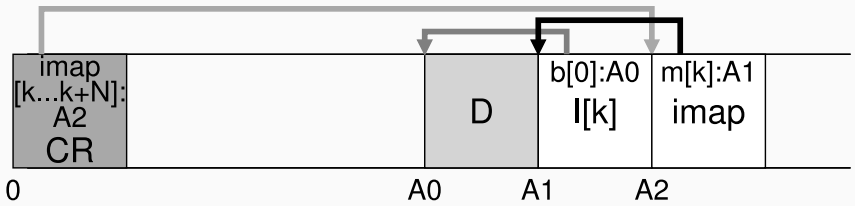


Figure 2: CR

CR

Only updated periodically (e.g.30s) (Performance not ill affected)

Real FS: much bigger CR.

Two CR: for crash recovery!

Directories

LFS with imap solves recursive update problem.

The change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inode-number mapping.

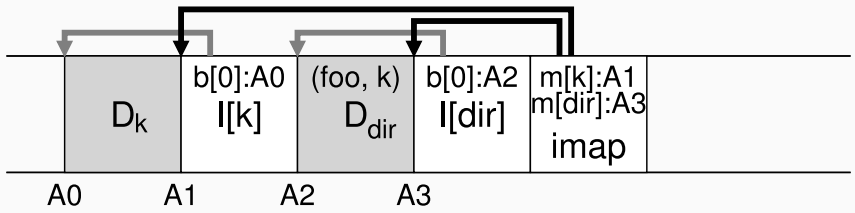


Figure 3: Directories

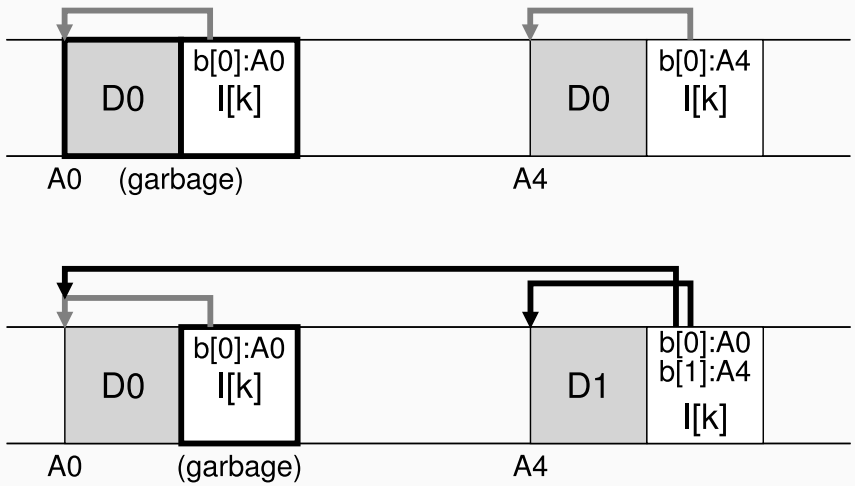
Garbage Collection

Note: could keep older versions for rollback.

LFS instead keeps only the latest live version of a file.

Segment-by-segment basis. (Otherwise useless free holes will appear)

- reads in a number of old (partially-used) segments (N);
- determines which blocks are live;
- collect live blocks and pack them to a new set of segments(M);
- then free the old segments
- N $\rightarrow$ M $N > M$



Block Liveness (segment summary block)

For each segment, at the head, we have a segment summary block.

For each data block D

- inode number
- offset (which block)

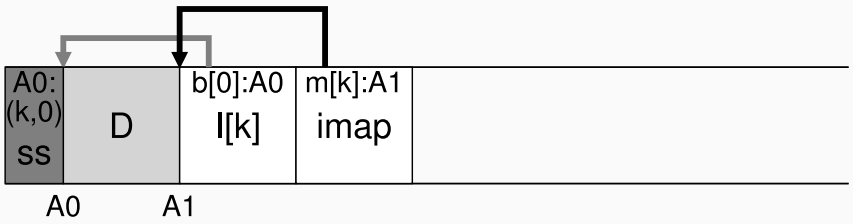


Figure 4: segment summary block

Block Liveness

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Shortcut:short circuit the longer check described above by comparing on-disk version number with a version number in the imap.

Cleaning Strategy

In the original LFS paper:

- hot segment (frequently over-written): wait a long time before clean it
- clod segment: clean it sooner

This isn't perfect! There is a lot of researches on it.

Crash Recovery (Back to Check-point)

Two check-point regions (CR): at two ends of the disk.

Protocol of writing CR:

- write a header (with timestamp);
- body of CR;
- last block (also with a timestamp);

Inconsistent pair of timestamps $\rightarrow$ crash.

Crash recovery: choose to use the most recent CR that has consistent timestamps.

Problem: CR 30 s per write $\rightarrow$ last consistent snapshot of the file system may be quite old.

Crash Recovery (Roll Forward)

- Read last CR and find the end of log;
- read through the next segments and see if there are any valid updates within it;
- Recover as much data as we can;

Flash-based SSDs

Basic Flash Operations: Read / Erase / Program

Flash (NAND) is accessed in two units: **pages** (a few KB) and **blocks** (erase units, much larger). A flash chip supports three low-level operations.

Read (a page). Read can access any page; it is typically fast (tens of microseconds).

Erase (a block). Before programming a page, the entire containing block must be erased, which destroys the block's contents and is expensive (a few milliseconds).

Program (a page). After erase, a page can be programmed by changing some 1's to 0's; once programmed, the page cannot be programmed again without erasing the whole block.

Page-state intuition: $\texttt{INVALID} \xrightarrow{\text{Erase(block)}} \texttt{ERASED} \xrightarrow{\text{Program(page)}} \texttt{VALID}$.

Why Writes Are Hard on Flash

Flash cannot perform in-place overwrite: once a page is programmed, updating it requires erasing the *entire* containing block. Because erase is expensive, write performance is dominated by block erases.

Device	**Read (μs)**	**Program (μs)**	**Erase (μs)**
SLC	25	200–300	1500–2000
MLC	50	600–900	$\sim$3000
TLC	$\sim$75	$\sim$900–1350	$\sim$4500

Table 1: Raw flash performance characteristics.

Question: how can an SSD/FTL optimize writes despite page-level programming and high block-erase cost?

Why Writes Are Hard on Flash (cont'd)

Wear out: each erase/program (P/E) cycle slightly degrades a block; over time it becomes harder to distinguish 0 from 1, and the block becomes unusable.

Typical ratings: MLC $\approx$ 10,000 P/E cycles; SLC $\approx$ 100,000 P/E cycles.

Disturbance: accessing one page can flip bits in neighboring pages, known as read disturbs and program disturbs.

A common mitigation: *program pages within an erased block in order, from low page to high page.*

From Raw Flash to a Flash-Based SSD

A flash-based SSD presents the same block interface as a disk: read or write 512-byte sectors (or larger) by block address.

Inside, it combines multiple flash chips, a small amount of volatile memory (e.g., SRAM) for caching and mapping metadata, and a controller.

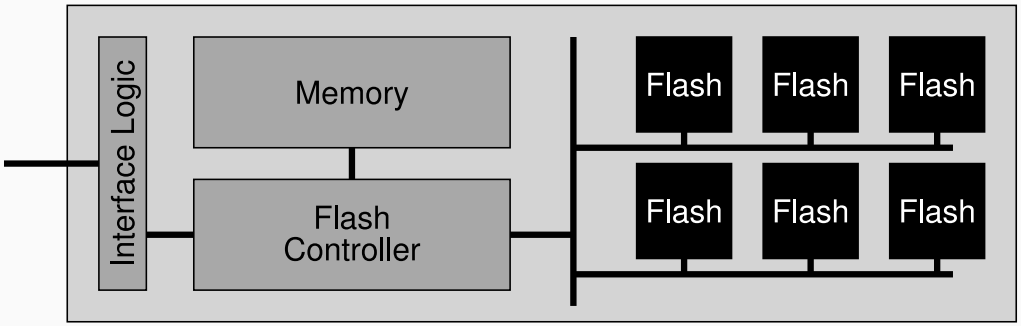


Figure 5: Raw Flash Performance Characteristics

The *flash translation layer* (FTL) translates logical-block requests into read/erase/program operations on physical pages and blocks, targeting performance and reliability.

FTL Organization: Direct-Mapped Is a Bad Approach

Direct-mapped FTL translates logical page N to physical page N for reads.

For a write to logical page N, it must read the entire containing block, erase that block, and then program the old pages plus the new one.

This makes updates expensive: block-level read/erase/program causes severe write amplification and concentrates wear on popular blocks.

Therefore, we need a different organization for writes, which leads to log-structured FTLs.

Log-Structured FTL

Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block; this style of writing is called logging.

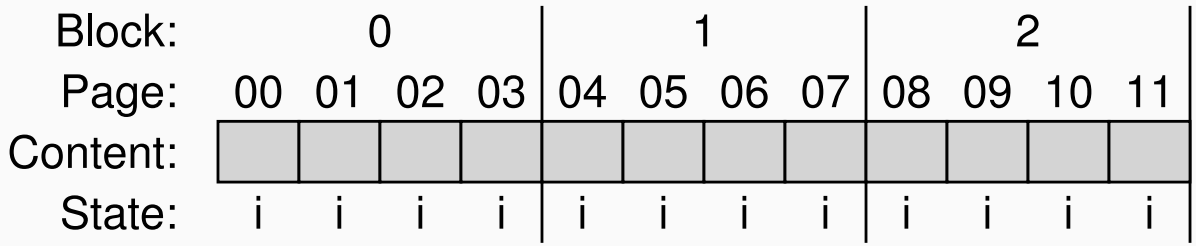
To allow for subsequent reads of block N, the device keeps a mapping table (in memory, and persistent in some form on the device).

The table stores the physical address of each logical block in the system.

Log-Structured FTL: An Example

Assume the client issues the following sequence of operations:

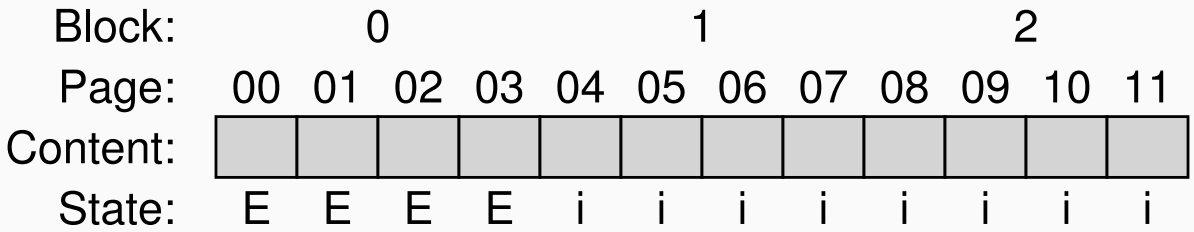
- Write(100) with contents `a1`
- Write(101) with contents `a2`
- Write(2000) with contents `b1`
- Write(2001) with contents `b2`



Log-Structured FTL: An Example

Assume the client issues the following sequence of operations:

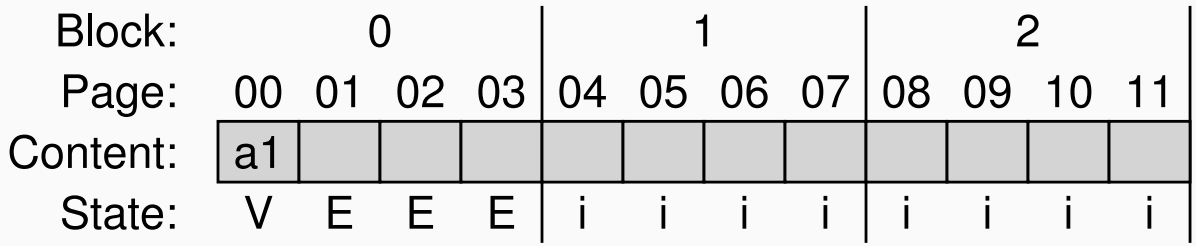
- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2



Log-Structured FTL: An Example

Assume the client issues the following sequence of operations:

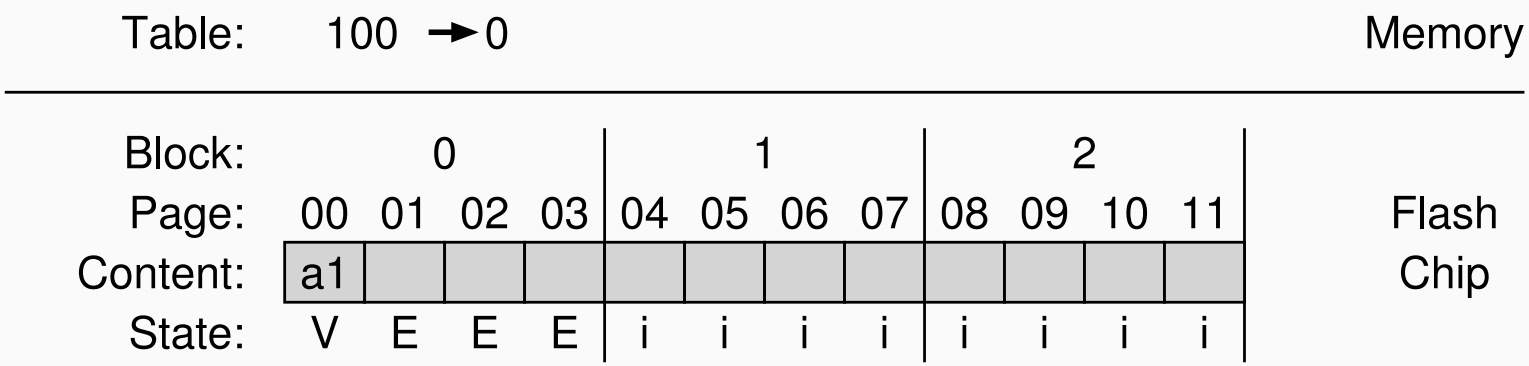
- Write(100) with contents `a1`
- Write(101) with contents `a2`
- Write(2000) with contents `b1`
- Write(2001) with contents `b2`



Log-Structured FTL: An Example

Assume the client issues the following sequence of operations:

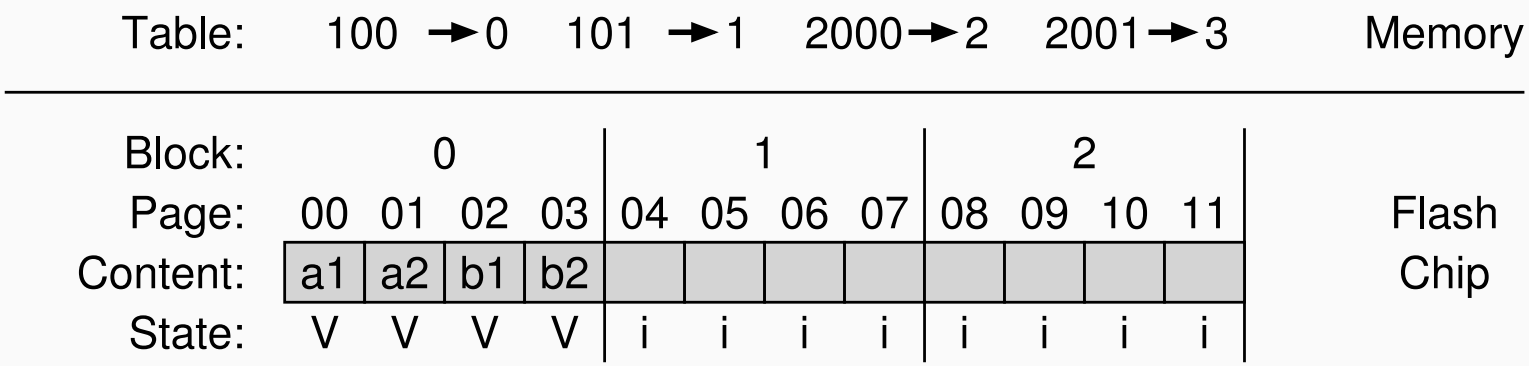
- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2



Log-Structured FTL: An Example

Assume the client issues the following sequence of operations:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2



Garbage Collection

In a log-structured FTL, overwrites leave old physical pages behind; these pages become garbage, while some pages in the same block may still be live.

GC creates free space by cleaning a victim block and reclaiming it for future writes.

The basic process is:

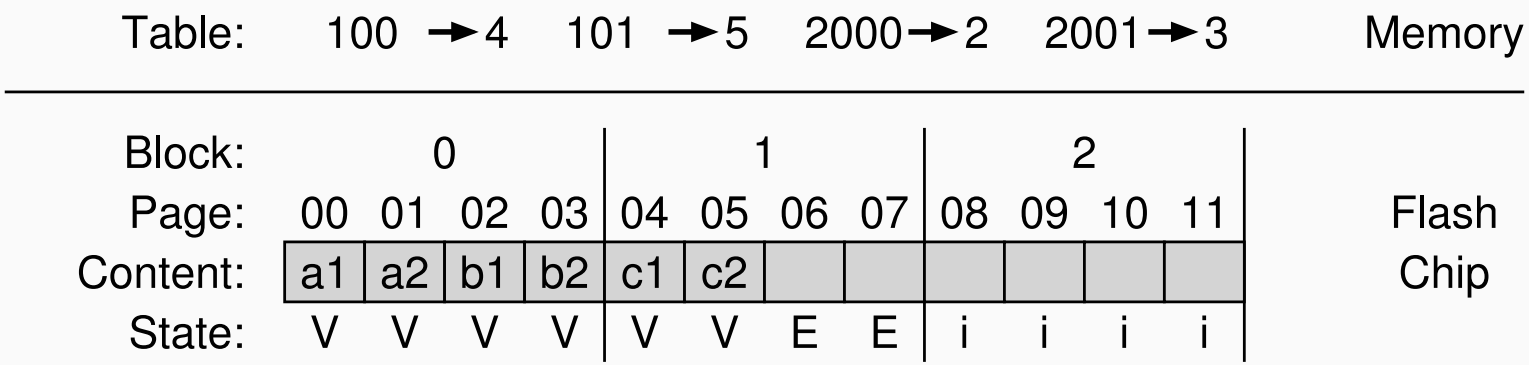
1. find a block that contains one or more garbage pages,
2. read the live (non-garbage) pages from that block,
3. write those live pages out to the log (to new free pages),
4. and finally erase and reclaim the entire block for use in writing.

GC is unavoidable in a log-structured design, but it is expensive: cleaning a block requires a read/erase/program operation. The ideal candidate for reclamation is a block that consists of only dead pages;

Garbage Collection: An Example

Assume the client issues the following sequence of operations:

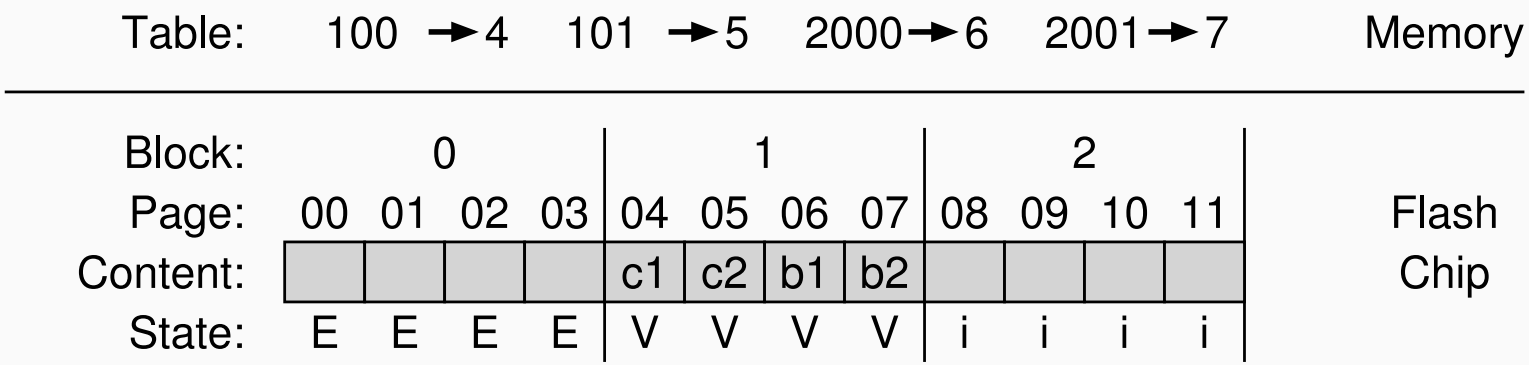
- Write(100) with contents c1
- Write(101) with contents c2



Garbage Collection: An Example

Assume the client issues the following sequence of operations:

- Write(100) with contents c1
- Write(101) with contents c2



Garbage Collection: TRIM and Overprovision

TRIM

A storage interface that allows the operating system to inform the SSD that certain logical blocks have been deleted and are no longer needed.

For a log-structured SSD, this knowledge lets the FTL drop obsolete mappings and treat the corresponding pages as garbage, reducing the amount of live data that must be moved during garbage collection.

Overprovision

Reserving extra flash capacity inside the SSD that is not exposed to the user.

With more free space available, GC can be delayed, run in the background, and move less data, making cleaning cheaper and reducing performance disruption.

Mapping Table Size

Page-level mapping is the simplest design: the FTL keeps one entry per logical page (e.g., 4KB) to translate to a physical page. At SSD scale, the table becomes too large to keep in SRAM.

Example

A 1TB device with 4KB sized pages has about 2^{28} pages, so even 4 bytes per entry requires roughly 1GB of memory.

Idea: Shrink the table by using block-level mapping, reducing the amount of mapping information by a factor of $\frac{Size_{block}}{Size_{page}}$.

Issue

Small updates become expensive: updating a single page inside a mapped block forces the FTL to copy the other live pages and rewrite the block elsewhere.

Hybrid Mapping

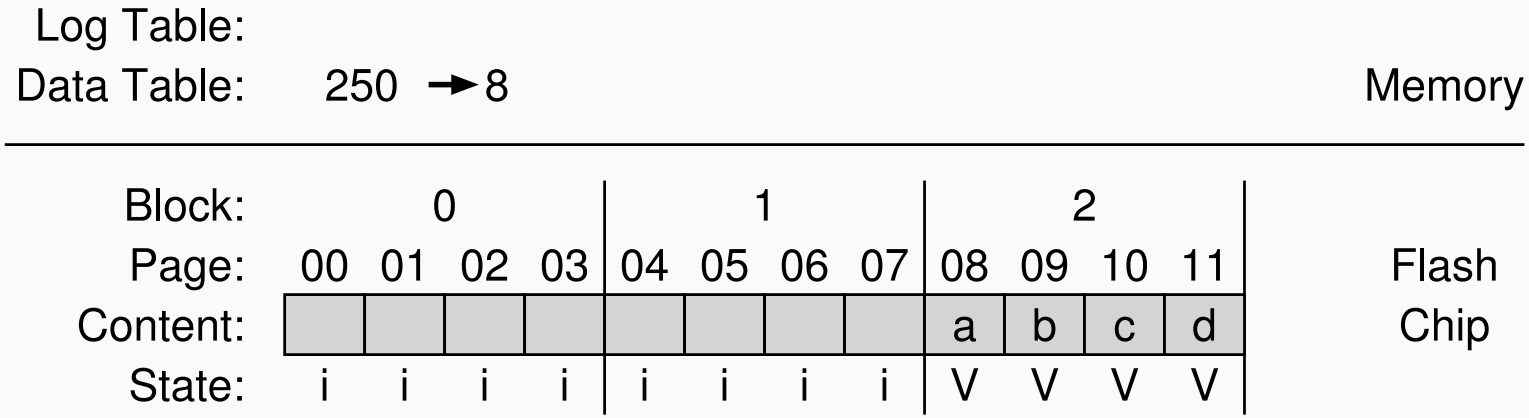
Motivated by page-level mapping and block-level mapping, hybrid mapping mixes the two to balance memory cost and write efficiency; it reserves dedicated log blocks for writes and keeps them erased.

For active data, the FTL maintains a page log table for pages written into log blocks; for cold data, it uses a block data table to map logical blocks to data blocks.

Question: *how do we control the size of the log table (and thus the number of log blocks) without making merges too expensive?*

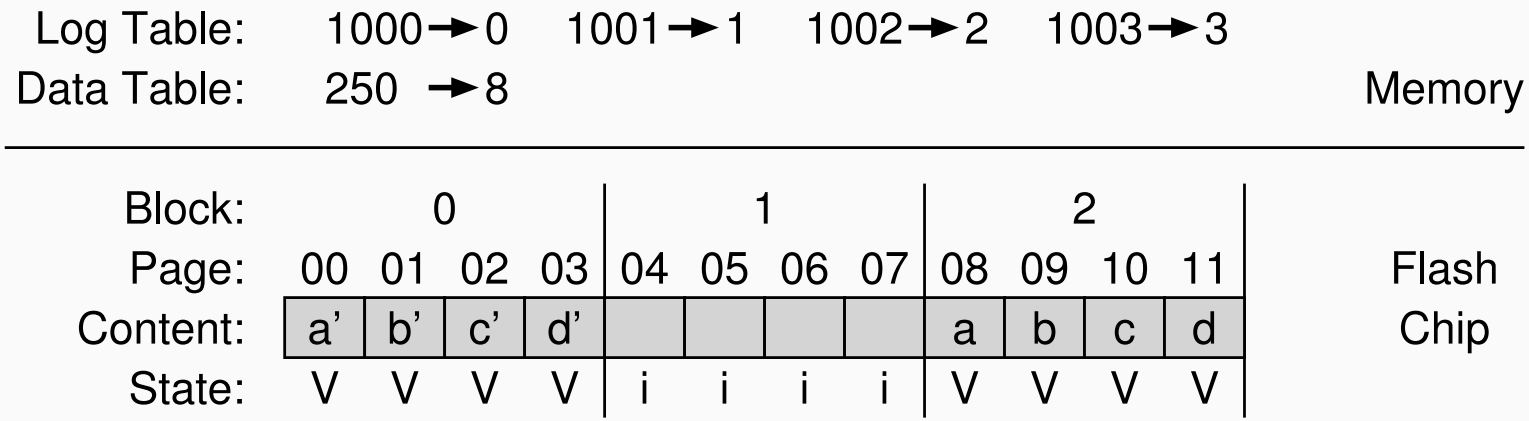
FTL has to periodically examine log blocks (which have a pointer per page) and switch them into blocks that can be pointed to by only a single block pointer.

Switch Merge: An Example



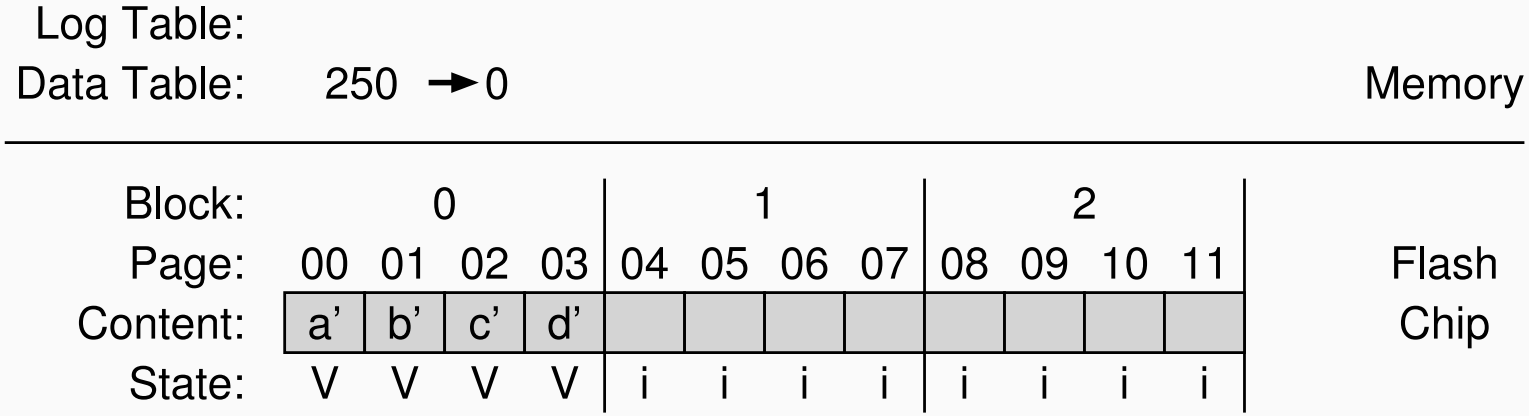
Previously, logical pages 1000–1003 are stored together in a single physical data block (block 2). The data table points to that block; the log table is empty.

Switch Merge: An Example



The client overwrites 1000–1003 in the same order; the FTL appends them into an erased log block (block 0). The log table records page-level pointers 1000→0, 1001→1, 1002→2, 1003→3.

Switch Merge: An Example

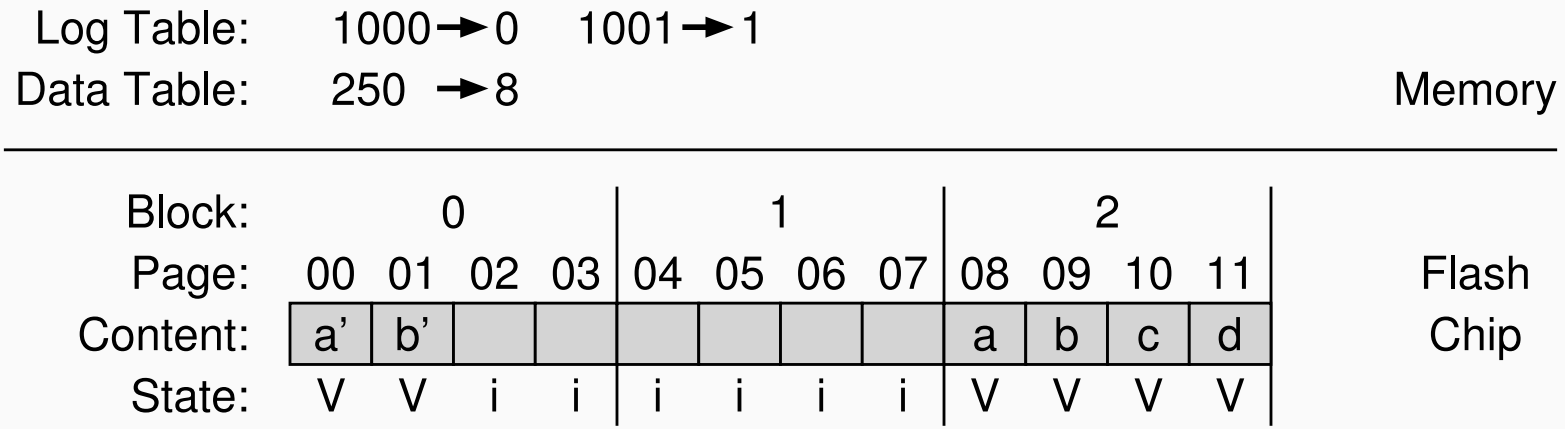


Switch merge: the log block now contains a full, up-to-date copy of the block, so it becomes the new data block. The data table switches to block 0; the old data block is erased and can be reused as a log block.

Partial Merge: An Example

When only part of a logical block is overwritten (e.g., 1000 and 1001), the log block does not yet contain a full up-to-date copy.

To regain a single block pointer, the FTL performs a partial merge: it reads the remaining live pages (e.g., 1002 and 1003) from the old data block and appends them to the log.



After the missing pages are appended, the log block can be switched into a data block; the extra reads/writes increase write amplification.

Full Merge

Full merge happens when one log block mixes pages from many logical blocks, so it cannot be switched into a single block-mapped data block.

Example

- Logical blocks 0, 4, 8, and 12 are written into the same log block A;
- build $(0 \sim 3)$ by reading $1 \sim 3$ from elsewhere and writing $0 \sim 3$ together;
- then build $(4 \sim 7)$, $(8 \sim 11)$, and $(12 \sim 15)$ similarly.

This requires many extra reads/writes, so frequent full merges can severely hurt performance.

Wear Leveling

Flash blocks wear out after many erase/program (P/E) cycles, so the FTL should spread this work evenly across the device.

Log-structured writes and garbage collection already help distribute wear, but blocks holding long-lived data may never be reclaimed and thus receive too few P/E cycles.

To fix this imbalance, the FTL periodically migrates live data out of such blocks and rewrites it elsewhere, freeing the blocks for future writes.

This improves lifetime uniformity but adds extra I/O, increasing write amplification and reducing performance.

Performance

Flash-based SSDs have no mechanical components and behave much more like random-access devices. The largest performance gap appears in random reads and writes.

Device	Random (MB/s) Reads	Writes	Sequential (MB/s) Reads	Writes
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Table 2: SSDs and hard drives: performance comparison.

Performance

What the table shows?

- Random I/O: SSDs reach tens to hundreds of MB/s, while the HDD is only a couple MB/s.
- Sequential I/O: the gap is much smaller; HDDs remain competitive when access is mostly sequential.
- SSD random writes can be unexpectedly strong because many SSDs use log-structured designs that turn random writes into more sequential internal writes.

Reducing random I/O can still matter on SSDs, but the penalty is much smaller than on HDDs.

Cost

Performance alone does not decide deployment; cost per GB is the key limiter.

Example

- SSD $\approx$ \$150 for 250GB ($\approx$ \$0.60/GB);
- HDD $\approx$ \$50 for 1TB ($\approx$ \$0.05/GB).

This $>$ 10 times cost gap keeps HDDs attractive for large-capacity storage.

Hybrid storage:

- SSDs for hot data (performance-critical);
- HDDs for cold data (capacity-focused).

Crash Consistency: FSCK and Journaling

Crash Consistency

File systems manage persistent data structures:

- data blocks
- inodes
- bitmaps
- directories

Unlike memory data structures, disk data is persistent.

i.e., they must survive over the long haul, stored on devices that retain data despite power loss

Why Persistence Is Hard

- Disk writes are *not atomic*
- A single file system operation updates multiple on-disk structures
- The disk only services one write at a time
- A crash may occur between any two writes

Question: What happens if the system crashes in the middle of an update?

Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power?

The Crash-Consistency Problem

THE CRUX

How to update the disk despite crashes?

- Crashes can happen at arbitrary times
- On-disk state may be only partially updated
- After reboot, the file system must keep the on-disk image in a reasonable state

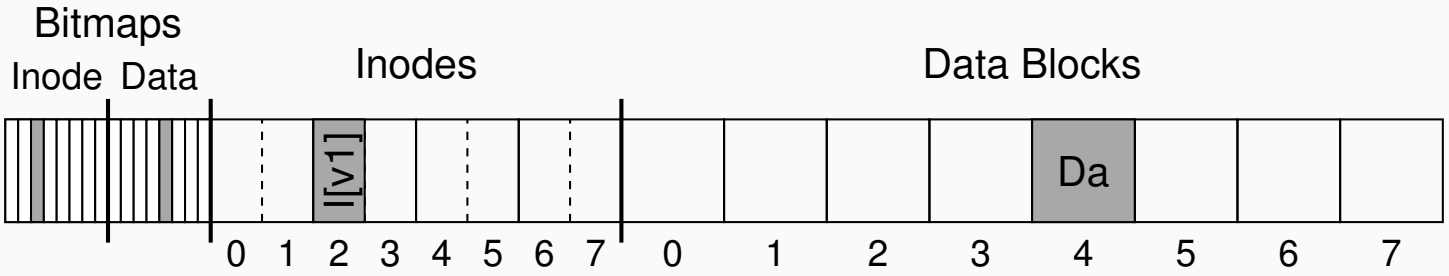
An Example : File Append

Assume here that the workload is simple: the append of a single data block to an existing file.

The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single `4KB` write to the file before closing it.

An Example : File Append

A diagram of this file system:

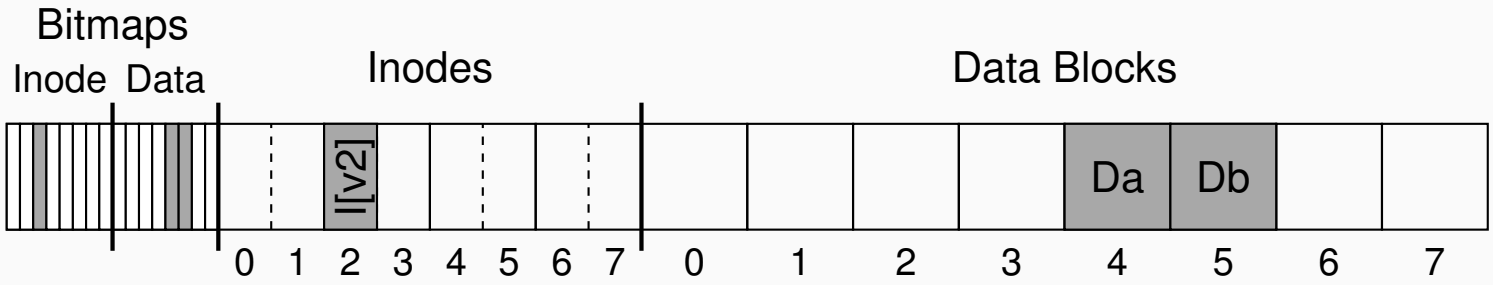


The inode is:

```
owner       : xuezhe
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

An Example : File Append

After we append to the file:



And the inode is:

```
owner       : xuezhe
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

An Example : File Append

Appending a block to a file typically updates:

- **Data block** (file contents)
- **Inode** (file size and pointers)
- **Data bitmap** (allocation status)

Crash during these updates may leave the file system inconsistent

Crash Scenarios

If only *one* write reaches disk before a crash:

- **Only Db written**
  - Data is on disk, but no inode or bitmap refers to it
  - Effectively, the write never happened
  - *No crash-consistency problem*
- **Only I[v2] written**
  - Inode points to block 5, but Db was never written
  - Reading block 5 returns garbage data
  - Bitmap says block 5 is free $\Rightarrow$ inconsistency
- **Only B[v2] written**
  - Bitmap marks block 5 as allocated
  - No inode points to it
  - Results in a space leak

Crash Scenarios

If *two* writes succeed and the last one fails:

- **I[v2] + B[v2], but no Db**
  - Metadata is fully consistent
  - Inode points to block 5, bitmap marks it allocated
  - Block 5 contains garbage data
- **I[v2] + Db, but no B[v2]**
  - Inode points to correct data
  - Bitmap still says block 5 is free
  - Inode and bitmap are inconsistent
- **B[v2] + Db, but no I[v2]**
  - Block 5 is allocated and contains data
  - No inode refers to it
  - File ownership is unknown

The Crash Consistency Problem

Goal: move the file system **atomically** from one consistent state to another.

This is known as the crash-consistency problem.

Solution 1: FSCK

FSCK (File System Checker)

- Allow inconsistencies to happen after crashes
- Detect and repair them during reboot
- Used by early UNIX file systems

Key idea: fix problems **after** the crash, not prevent them before.

FSCK

What dose FSCK do?

Superblock

Fsck first checks if the superblock looks reasonable

If FSCK found a suspect superblock, the system may decide to use an alternate copy of the superblock.

Free blocks

Fsck scans the inodes, indirect blocks, double indirect blocks, etc., to know which blocks are currently allocated within the file system.

If there is any inconsistency between bitmaps and inodes, FSCK will trust the information within the inodes.

FSCK

Inode state

FSCK checks each allocated inode for corruption or invalid fields.

For example, FSCK verifies that the inode type is valid (e.g., regular file, directory, symbolic link).

If an inode is clearly corrupted and cannot be fixed, FSCK clears the inode and updates the inode bitmap accordingly.

FSCK

Inode links

FSCK verifies the link count of each allocated inode.

It scans the entire directory tree, starting from the root directory, and recomputes the number of directory entries that reference each inode.

If the link count is incorrect, FSCK fixes the value stored in the inode. If an allocated inode is not referenced by any directory, it is moved to `lost+found`.

Duplicates

FSCK checks for duplicate block pointers, where multiple inodes refer to the same data block.

If one inode is clearly incorrect, FSCK may clear it. Otherwise, the data block may be copied so that each inode has its own separate copy.

FSCK

Bad blocks

FSCK checks for bad block pointers while scanning all block references.

A block pointer is considered invalid if it points outside the valid range of the file system.

In such cases, FSCK removes (clears) the invalid pointer.

Directory checks

FSCK performs additional integrity checks on directory contents.

It verifies the presence of "." and "..", ensures that directory entries refer to allocated inodes, and checks that directories are not linked more than once.

Limitations of FSCK

Although FSCK can repair file system inconsistencies, it has major drawbacks:

- Requires scanning the entire disk
- Recovery time grows with disk size
- File system may be unavailable for a long time after a crash

Motivation: Can we recover faster after crashes?

Solution 2: Journaling (Write-Ahead Logging)

The most popular solution to the consistency problem.

Borrowed from the DBMS world, *Journaling* addresses the cost of `FSCK` by adding a "note" before updating the disk.

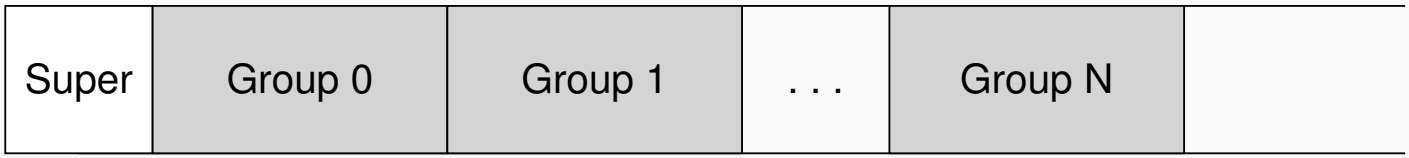
- **Key Idea:** Before overwriting structures in place, write a small note to a well-known location (the **log**) describing the intended change.
- **Write-Ahead:** The "note" must reach the disk before the actual update.
- **Recovery:** If a crash occurs, we only need to check the log and "replay" or "retry" the operations, rather than scanning the entire disk.

Examples: Linux ext3/ext4, XFS, NTFS, JFS.

Linux ext3

- a popular journaling file system
- Most of the on-disk structures are identical to **Linux ext2**

An ext2 file system (without journaling) looks like this:

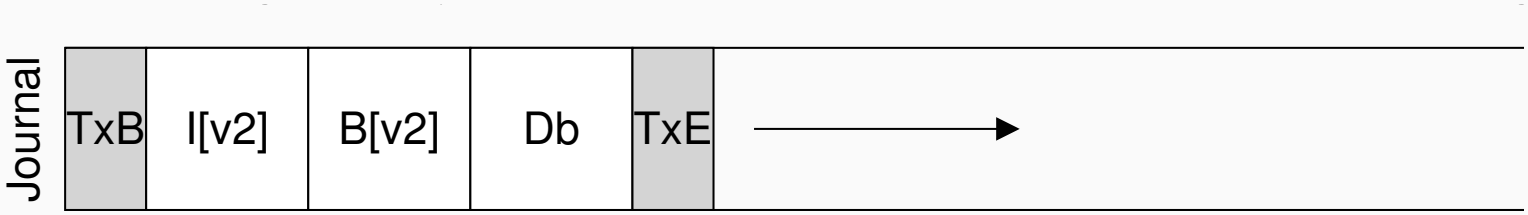


The new key structure of **Linux ext3** is the journal itself, which occupies some small amount of space within the partition or on another device.

An ext3 file system with a journal looks like this:

How does data journaling work?

The journal looks like:



The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), and some kind of **transaction identifier (TID)**.

The final block (TxE) is a marker of the end of this transac tion, and will also contain the TID.

Linux ext3

Our initial sequence of operations is:

- **Journal write:** Write the transaction.
- **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

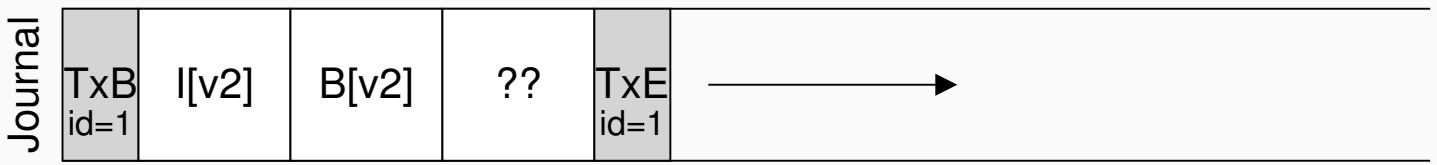
Have we completely solved the problem now?

Problem

If all blocks of a transaction (TxB, I[v2], B[v2], Db, TxE) are issued to the disk simultaneously:

- **Original Intent:** To improve performance by merging multiple small writes into a single large sequential write.
- **Risk:** The disk controller may **schedule** the write operations internally to optimize performance.

if the disk loses power after the TxE and before Db, there will be a problem.



Solution

Split the process into 2 steps:

- write all blocks except the TxE block to the journal
- write TxE block

Thus, our current protocol to update the file system is:

- **Journal write** : write other blocks
- **Journal commit** : write TxE
- **check point**

Recovery: Redo Logging

If a crash occurs, the system uses the journal to recover:

- **Scenario 1: Crash before Log Commit**
  - The transaction (including TxE) is not fully written to the log.
  - *Action:* The pending update is simply **skipped**.
- **Scenario 2: Crash after Log Commit, before Checkpoint**
  - The transaction is safely in the log but not yet at its final location.
  - *Action:* **Scan** the log for committed transactions and **replay** them in order.

Redo Logging

By re-writing the blocks from the journal to their final disk locations, the FS ensures consistency. This is known as *redo logging*.

Batching Log Updates

Problem: The Cost of Frequent Commits

- Each system call (e.g., `write()`) triggering a full journal protocol is too slow.
- Redundant I/O: Repeatedly writing the same metadata blocks (bitmaps, directories) for small, rapid updates.

The Solution: Global Transactions

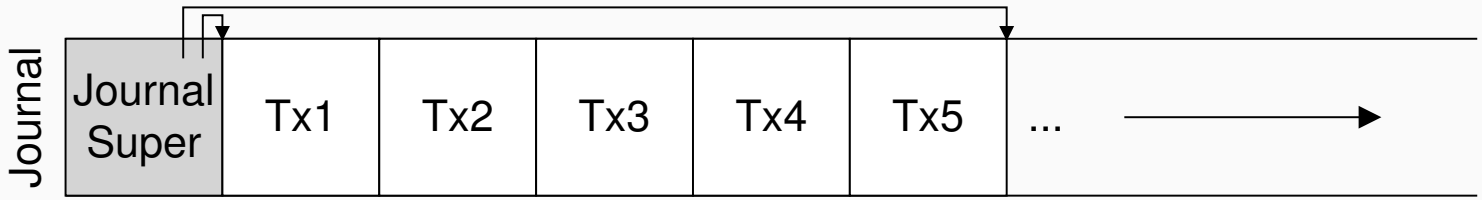
- **Buffer:** Hold all updates in memory first.
- **Batch:** Group multiple individual updates into one large transaction.
- **Commit:** Write to the journal only periodically (e.g., every 5s) or when the buffer is full.

Making The Log Finite

The journal is a **fixed-size circular buffer** on disk.

Managing the Log: Journal Superblock

- A simple structure that marks the **start** and **end** of valid transactions in the circular log.
- After a checkpoint, the FS updates the **Journal Superblock** to free space, allowing the log to be reused.



The 4-Step Protocol

1. Journal write $\rightarrow$ 2. Journal commit $\rightarrow$ 3. **Checkpoint** $\rightarrow$ 4. **Free**

Metadata Journaling

The user data is **not written to the journal**, but is written directly to its final location on disk.

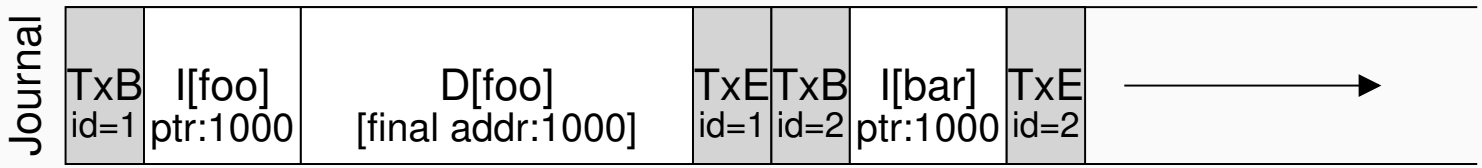
The 5-Step Protocol

1. Data write $\rightarrow$ 2. Journal write $\rightarrow$ 3. Journal commit $\rightarrow$ 4. **Checkpoint** $\rightarrow$ 5. **Free**

Tricky Case: Block Reuse

The problem: directory is recognized as metadata, and it would be written to the journal.

Assume a directory `foo` is modified, and then deleted. After that, a file `bar` is created and it used the same block. The journal will be:



Tricky Case: Block Reuse

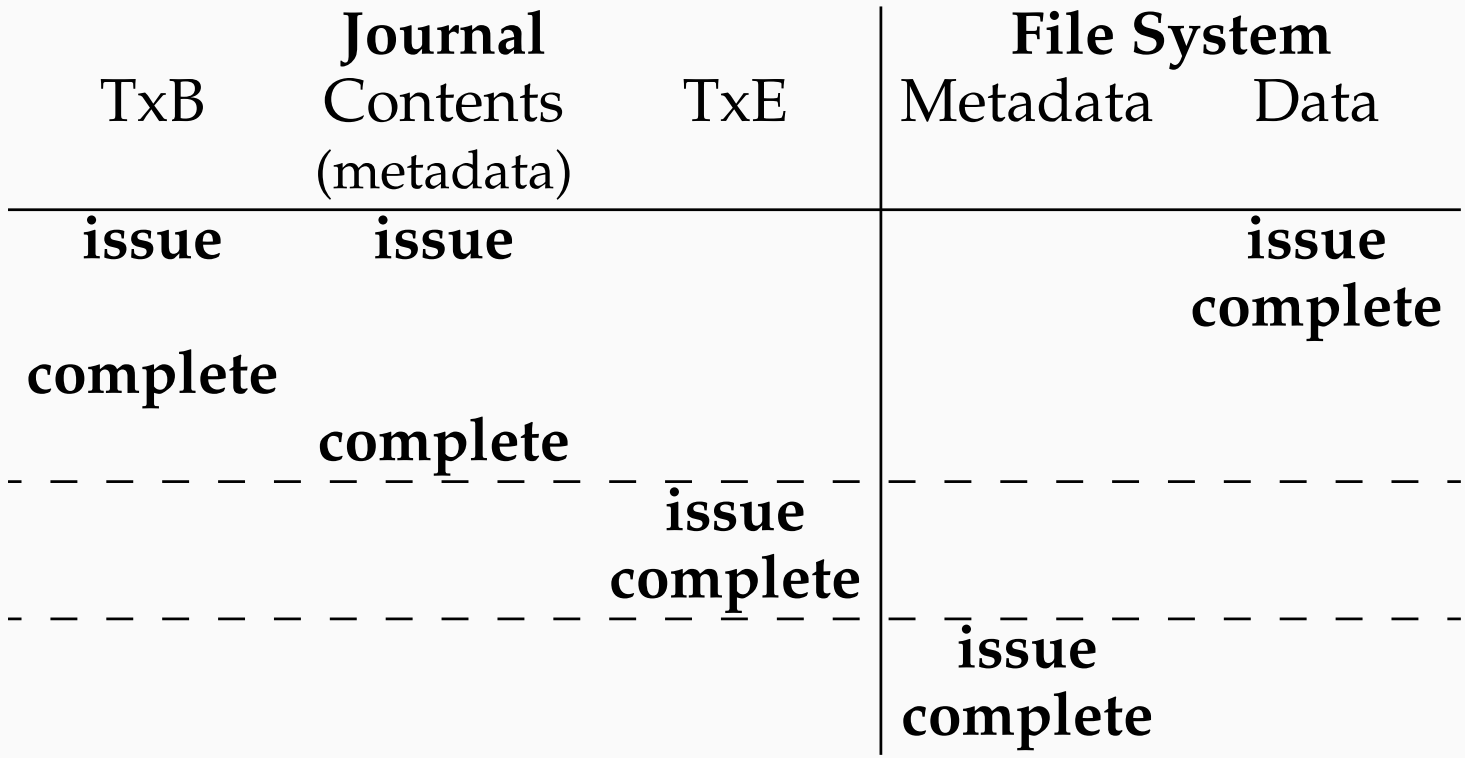
The Solution: Revoke Records

- **Revoke Record:** When a block is deleted, a "revoke" entry is added to the log.
- **Smart Replay:**
  1. System first scans the log for all Revoke Records.
  2. During replay, any update to a revoked block is **skipped**.

Alternative Solution

Never reuse blocks until the deletion of those blocks is **checkpointed** out of the journal.

Timeline


Journal
File System
TxB
Contents
(metadata)
TxE
Metadata
Data
issue
issue
issue
complete
complete
complete
issue
complete
issue
complete

Soft Updates

Next, we will introduce some other methods.

By carefully arranging the order of all writes to the file system, it is ensured that the disk structure will never be in an inconsistent state.

It requires intricate knowledge of each file system data structure. And there will be problems like space leak.

Copy-on-write (COW)

Core Concept: Never Overwrite

- Instead of updating in place, COW writes new data and metadata to **unused blocks**.
- Old versions are kept intact until explicitly freed (enables snapshots).

Mechanism: Root Flipping

- Updates propagate up the file system tree (Path Selection).
- **Atomic Update:** The system completes a batch of updates by "flipping" the **root structure** to point to the new tree.

Examples: Sun's ZFS, Linux Btrfs, Apple APFS.

Backpointer-based Consistency

The Concept: Reverse Dependency

- Traditional systems use **Journaling** to order writes (Data $\rightarrow$ Metadata).
- Backpointer-based systems add a **backpointer** to every data block, pointing back to the Inode that owns it.

How it Works

- A data block D is only considered **valid** if:
- `Inode(Owner)` $\rightarrow$ `Block(D)` **AND** `Block(D)` $\rightarrow$ `Inode(Owner)`

Optimistic Crash Consistency

The Problem: The Cost of Barriers

- Traditional journaling waits for Data to commit before issuing TxE.
- These "Write Barriers" significantly reduce disk throughput.

The Optimistic Approach

- **No Wait:** Issue all transaction blocks (Data, Metadata, TxE) simultaneously.
- **Checksums:** Include a checksum of the entire transaction in the TxE block.
- **Validation:** During recovery, if `Checksum(Log)` $\neq$ `TxE.Checksum`, the transaction is ignored.