RocksDB

An LSM engine in production

CSE 532 · Database Systems

September 21, 2026

Dong, Kryczka, Jin, and Stumm

The RocksDB Experience · FAST 2021

LevelDB at Google, 2011

Who	Jeff Dean and Sanjay Ghemawat at Google
When	Open-sourced July 27, 2011
Why	Portable, embedded storage for ordered key/value data
Workload	Efficient batch updates over a large key space
Roots	Ideas from Bigtable’s tablet storage

Early uses: Chrome’s IndexedDB and Riak’s per-node storage.

An LSM storage library that applications could embed directly.

Facebook's fork: RocksDB

2012: Facebook's database team forks LevelDB.

November 21, 2013: RocksDB released as open source.

Why fork?

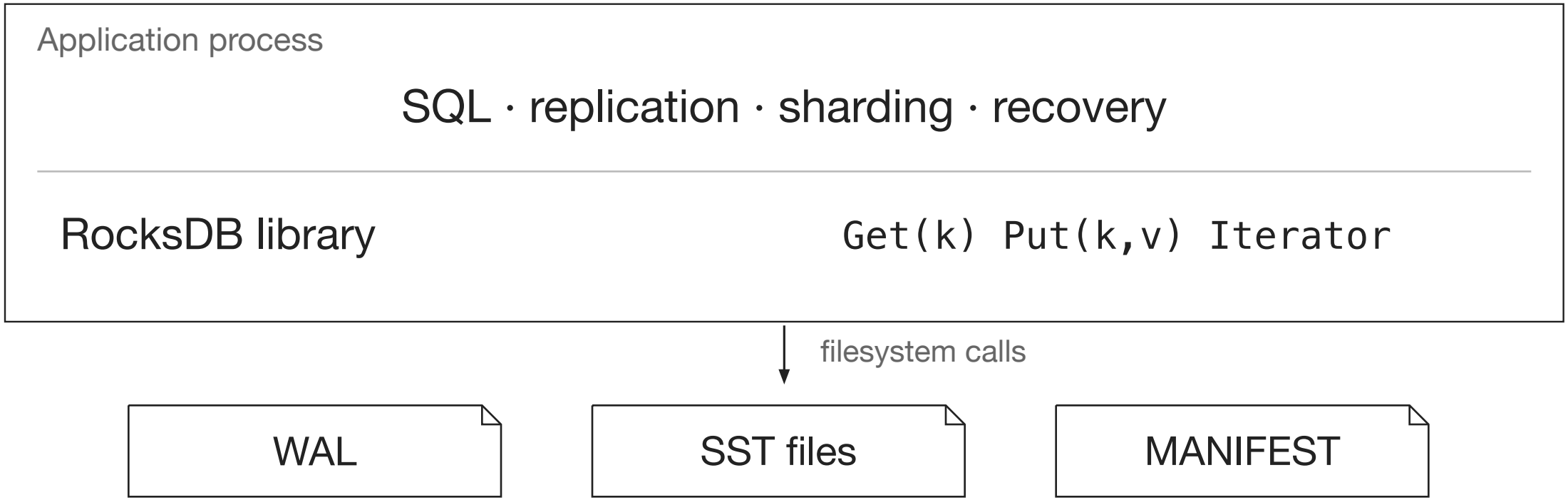
LevelDB's single compaction thread caused write stalls on some server workloads.

What changed?

Parallel compaction, less write amplification, and tunable components for fast SSDs and many CPU cores.

RocksDB retained LevelDB's embedded, ordered key/value interface.

Where RocksDB sits



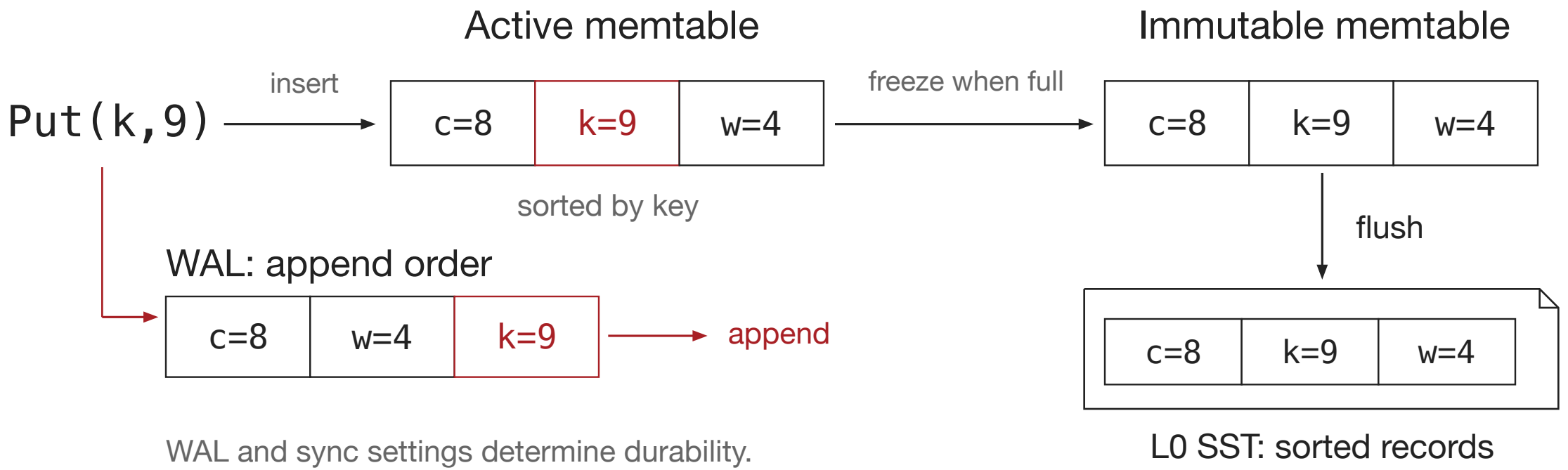
One embedded library, many application architectures.

Targets high throughput on SSDs and multicore CPUs; replication stays in the application.

From the previous LSM lecture

1996 LSM concept	RocksDB representation
In-memory component C_0	Memtable, typically a skiplist
A frozen memory component	Immutable memtable, then an L0 SSTable
Disk components C_1 … C_k	Levels of immutable SSTable files
One-pass merge	Compaction of selected files
Deletion record	Tombstone

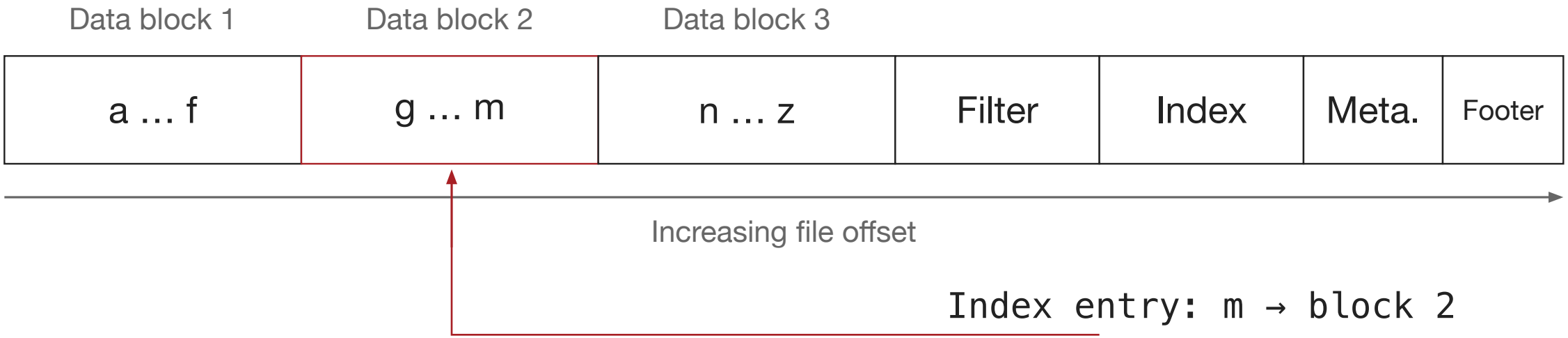
The write path



Flush turns a mutable buffer into an immutable sorted file.

Inside an SSTable

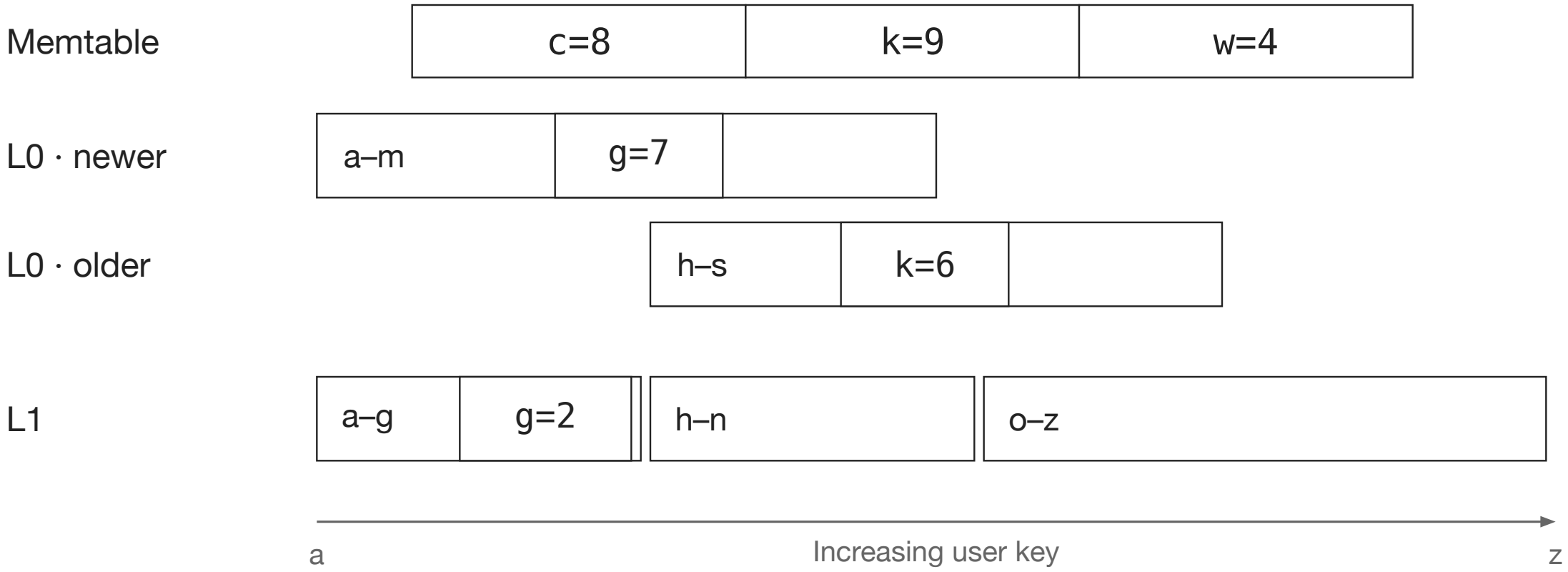
One immutable file



The index locates a data block. The filter can rule out a file.

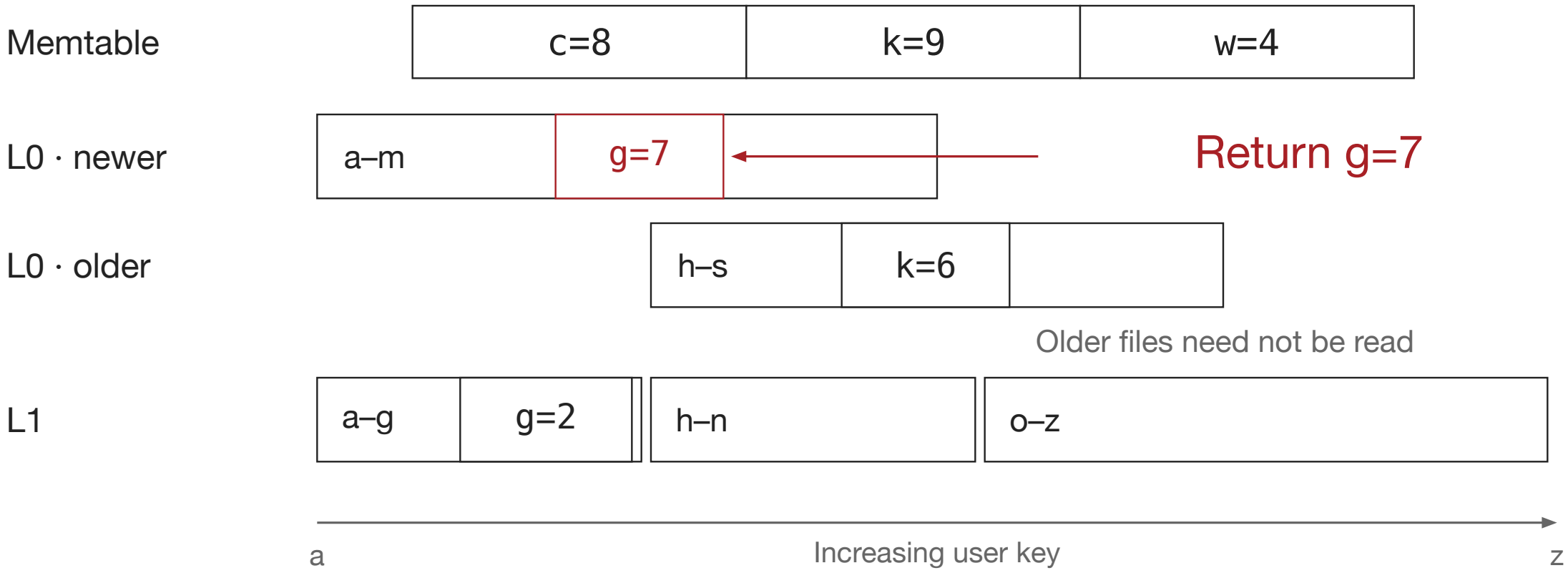
Conceptual structure. Block sizes and metadata layout are simplified.

L0 overlap and L1 partitioning



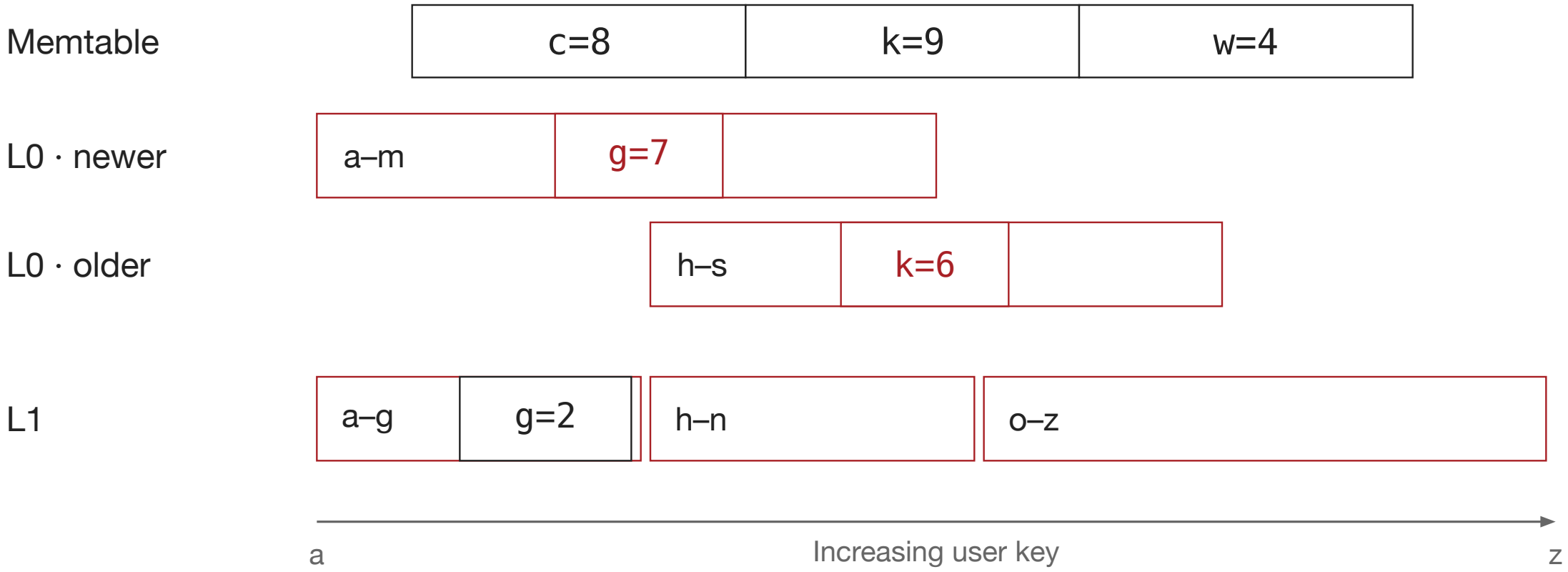
Key ranges overlap in L0. Within each later level, they do not.

Point lookup: Get(g)



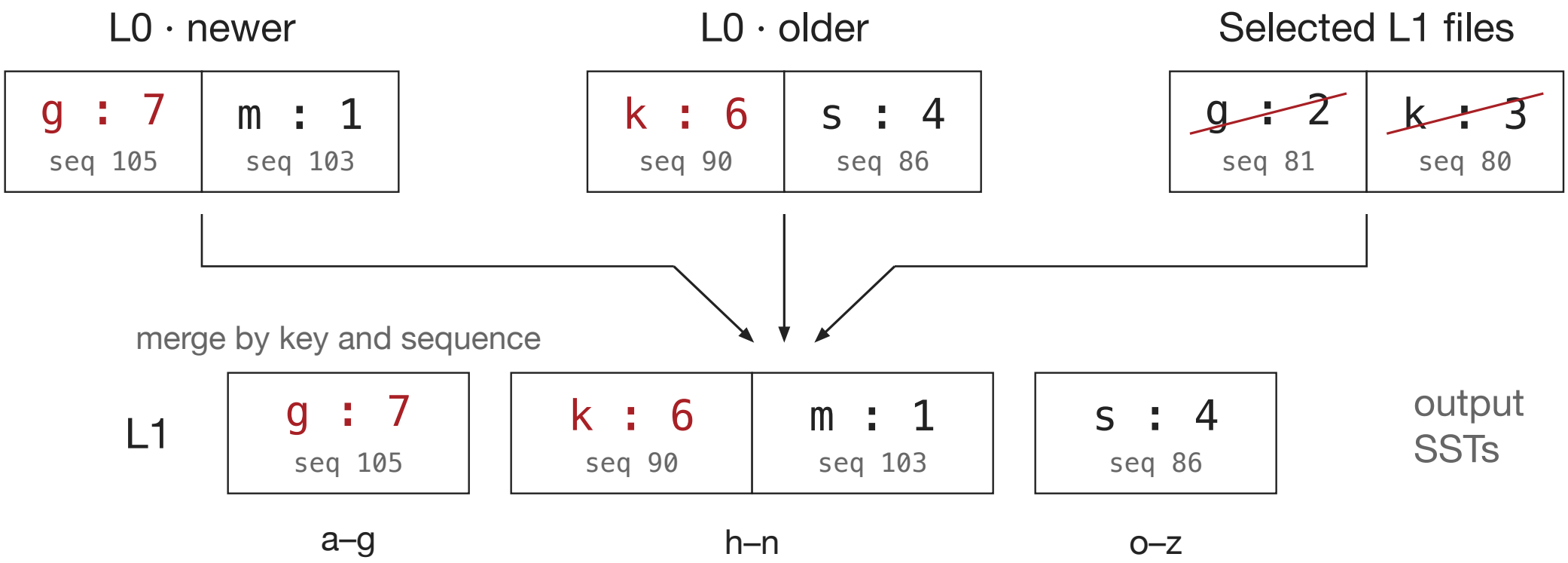
Search memtables, relevant L0 files, then at most one file per later level.

Compaction inputs



Select overlapping L0 inputs and every L1 file their ranges overlap.

Compaction outputs

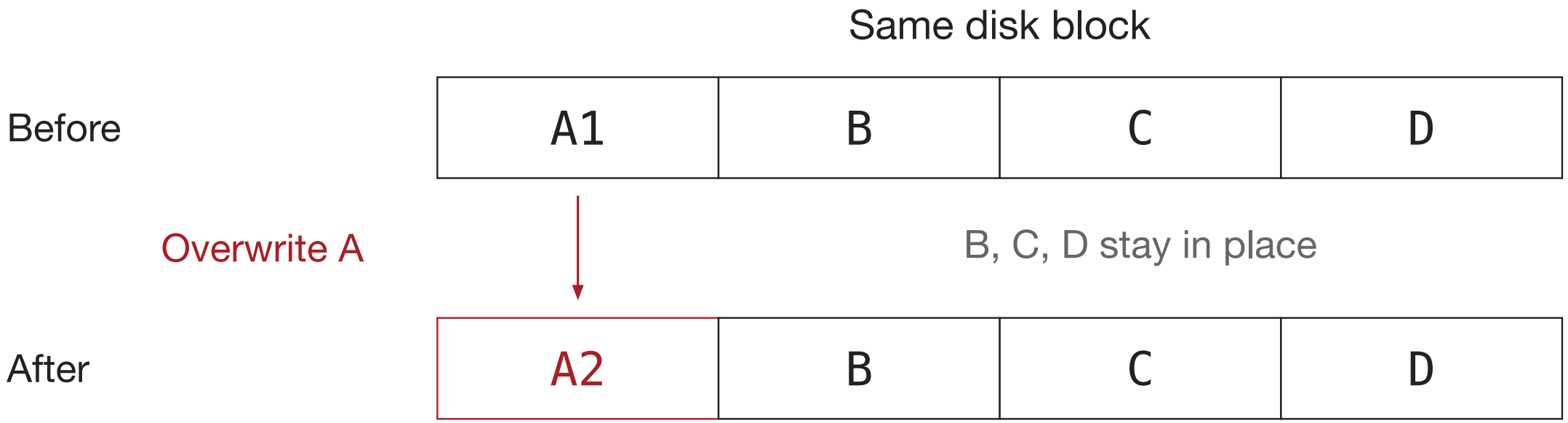


Selected records shown; other keys omitted. No snapshot needs the old versions.

Why do we need LSM on SSD?

HDD: update one page in place

Change page A from A1 to A2. Each cell below is one page.

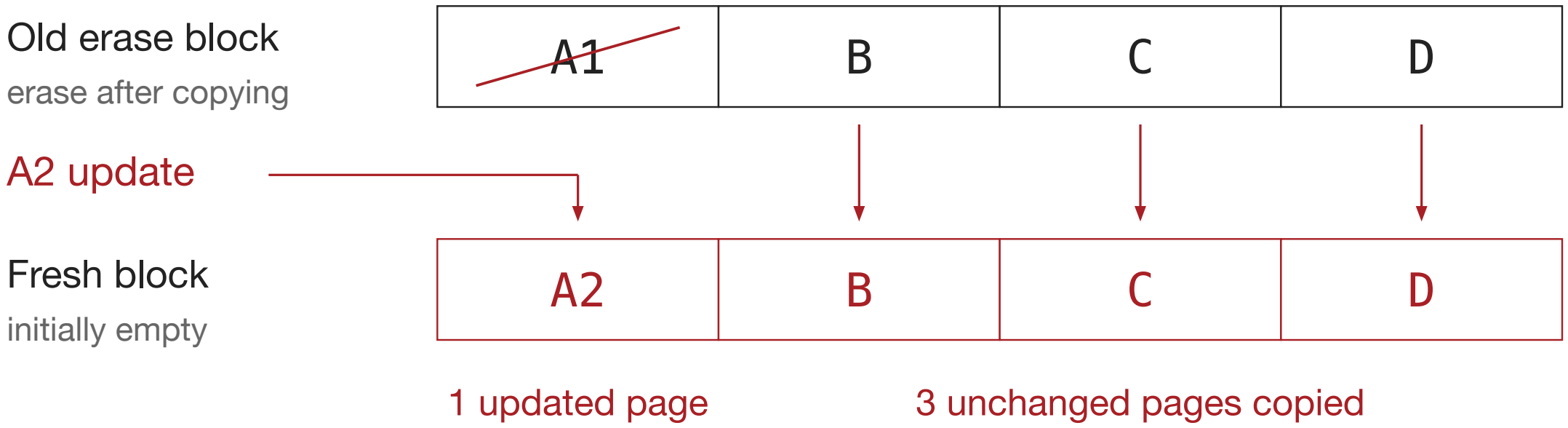


One page update writes one page; the other pages are untouched.

Four pages shown for illustration.

SSD: copying creates write amplification

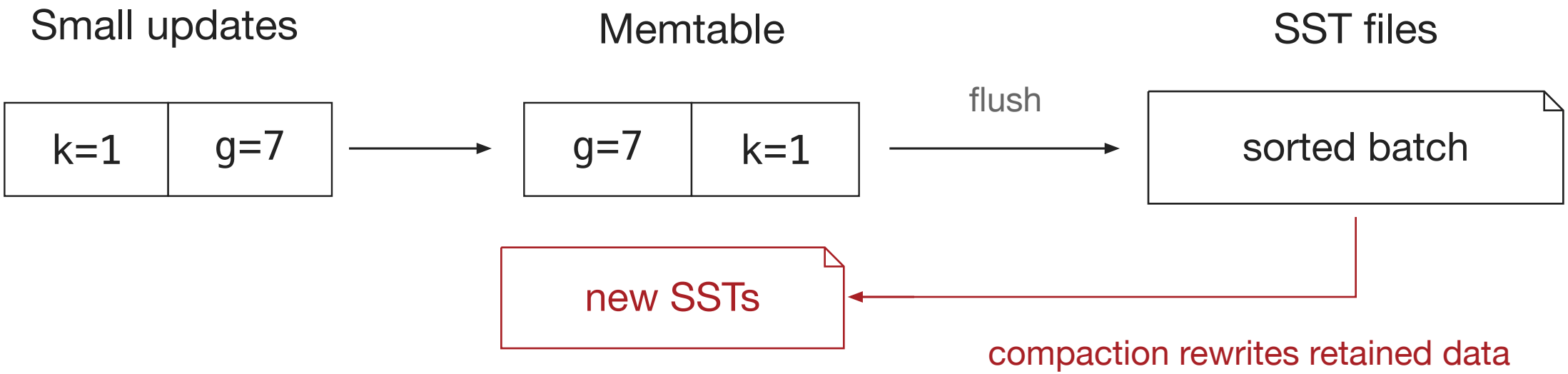
Flash erases whole blocks. Live pages must be saved before erasing.



1 updated page + 3 copied pages = 4 page writes (4×).

Four-page example, including later reclamation. The SSD can defer this copying.

Why LSM on flash?

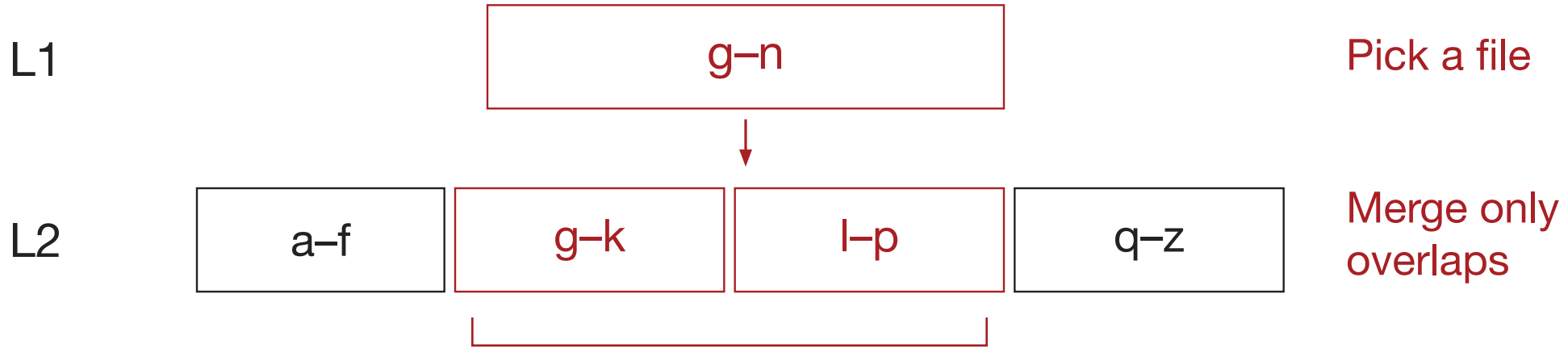


- Batch small updates instead of rewriting a database page for each one.
- Compaction and internal SSD copying both create extra writes.

LSM changes the write pattern; it does not eliminate write amplification.

Compaction policies

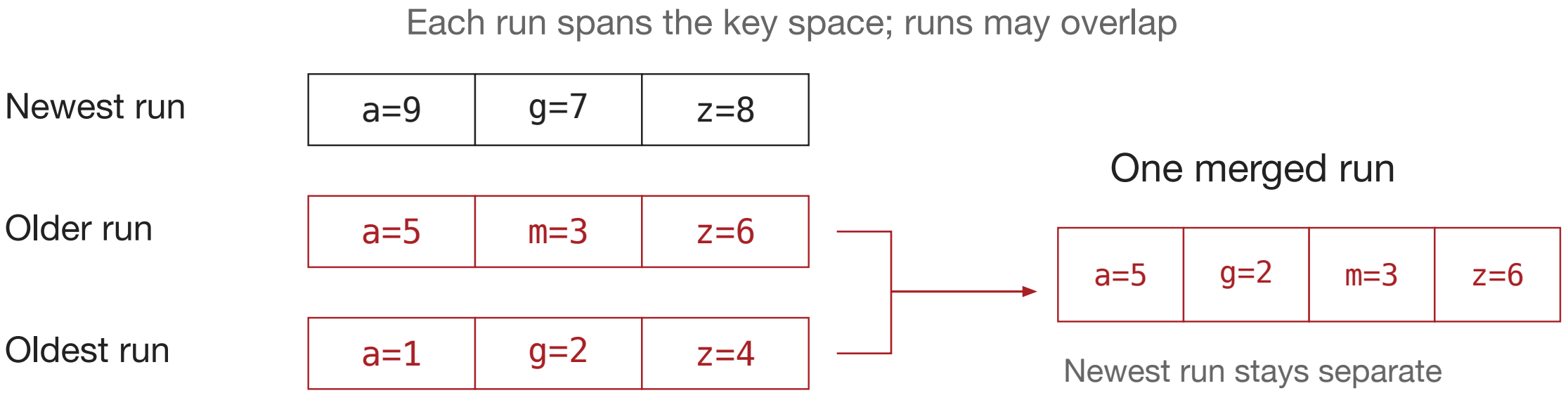
Leveled: merge overlapping key ranges



- Keep nonoverlapping file ranges within each level below L0.
- Merge selected files with the overlapping files in the next level.

More rewriting buys lower read and space amplification.

Universal / tiered: merge whole sorted runs

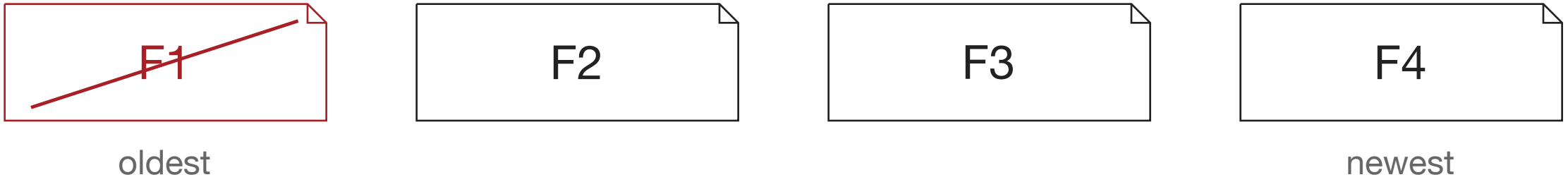


- Let overlapping sorted runs accumulate; merge selected runs together.
- Defer merges to reduce rewriting, while keeping more runs to read.

Usually cheaper writes, with more read work and retained space.

FIFO: discard the oldest files

Four equal-size SSTs; capacity limit = three files



Delete F1

Keep F2, F3, F4 without rewriting them

- Reclaim space by deleting whole files, without merging their records.
- Use only when the application permits old data to disappear.

Low write cost comes from a different retention contract.

Measured compaction tradeoffs

Policy	Write amp.	Avg. space overhead	Point Get I/O with Bloom
Leveled	16.07×	9.5%	0.99
Tiered	4.80×	45.5%	1.03
FIFO	2.14×	N/A	1.16

Write amp. = SST bytes written / memtable bytes flushed.

Space overhead = (current size / fully compacted size – 1) × 100%.

RocksDB 5.9. Write amp. includes flush and compaction, excluding WAL and SSD-internal writes. FIFO discards data.

Which policy fits the application?

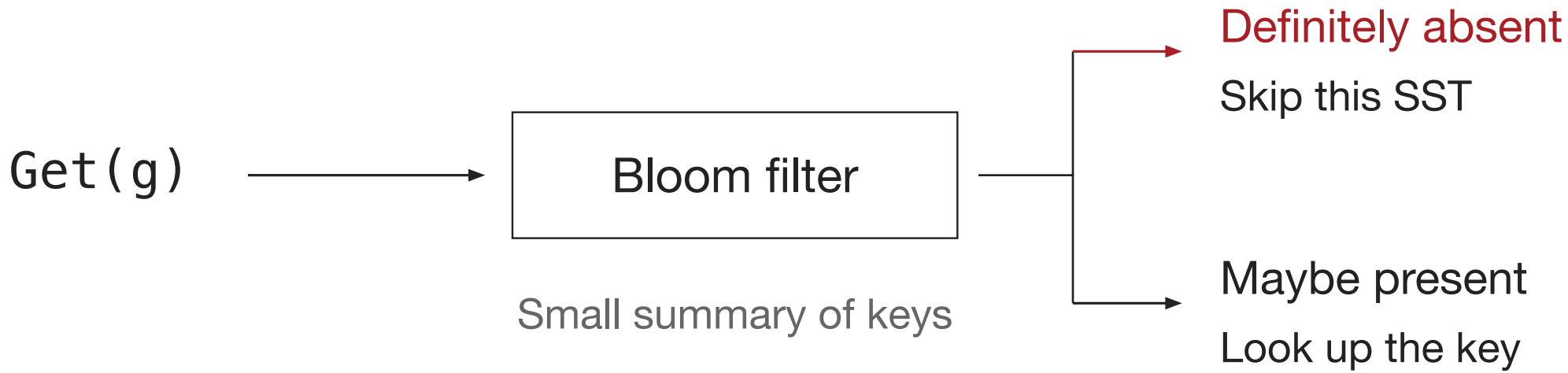
Application	Suggested policy	Why it fits
User database	**Leveled**	Frequent reads and tight capacity favor fewer files and lower space overhead.
Stream state	**Universal / tiered**	Heavy ingestion favors fewer rewrites, with room for more retained runs.
Overflow cache	**FIFO**	Fixed capacity, old data may disappear. Delete old files without merging.

Bloomfilter

A small test before reading an SST

A Bloom filter summarizes a set of keys using a small array of bits.

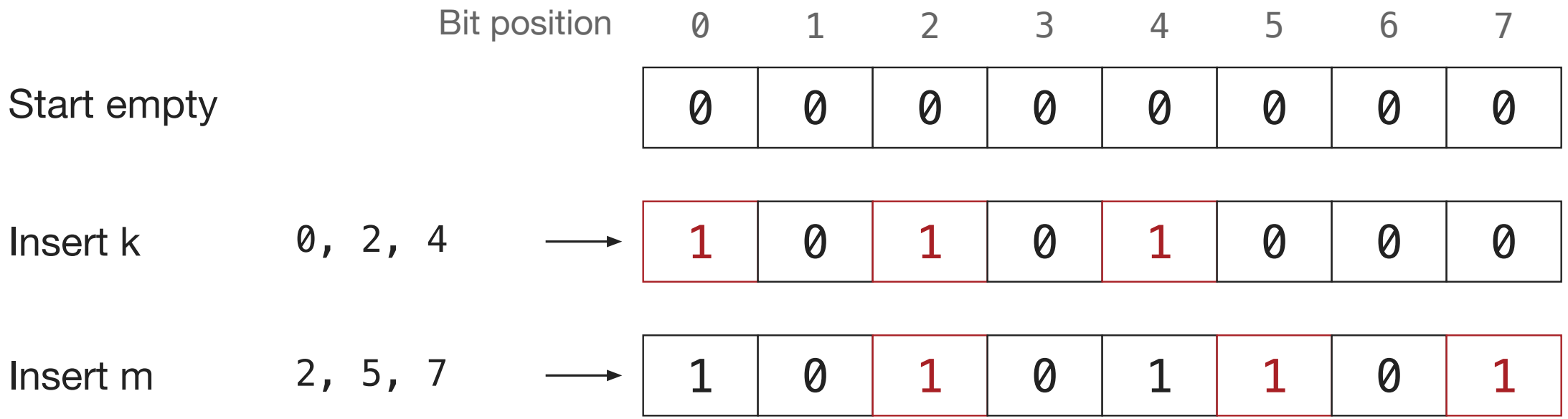
Question: does this SST contain key g?



It tests membership; it does not return the value.

Building a filter: set bits with hashes

Three hash functions each map a key to one bit position.



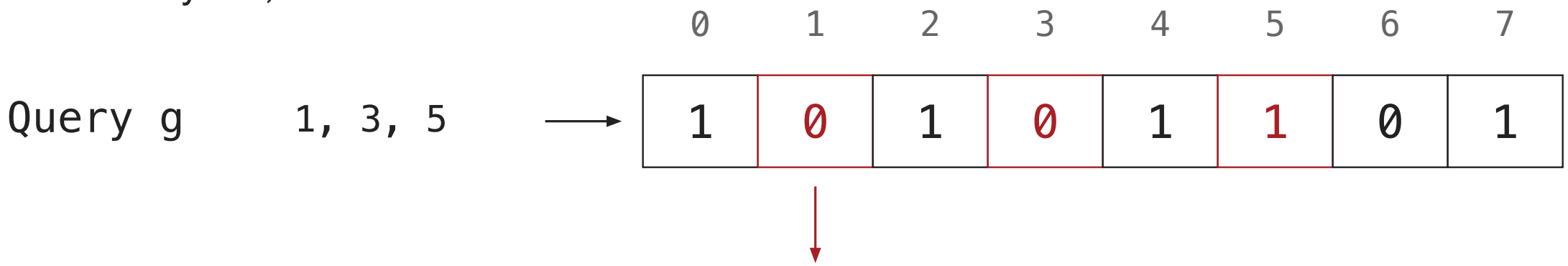
Each key sets three positions to 1. Existing 1s stay 1.

Eight-bit example. The same key always gives the same positions.

A zero means “definitely absent”

Query g uses the same hash functions as insertion.

Stored keys: k, m



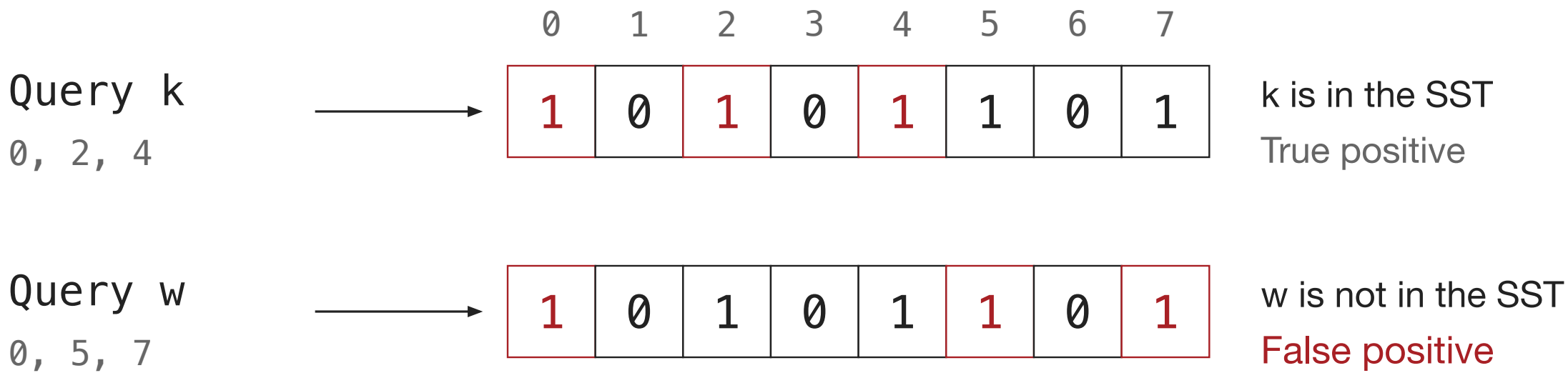
No false negatives: an inserted key cannot be ruled out.

Assumes the same hashes and an intact filter.

All ones mean “maybe present”

Every checked bit is 1 in both queries below. Only one key is present.

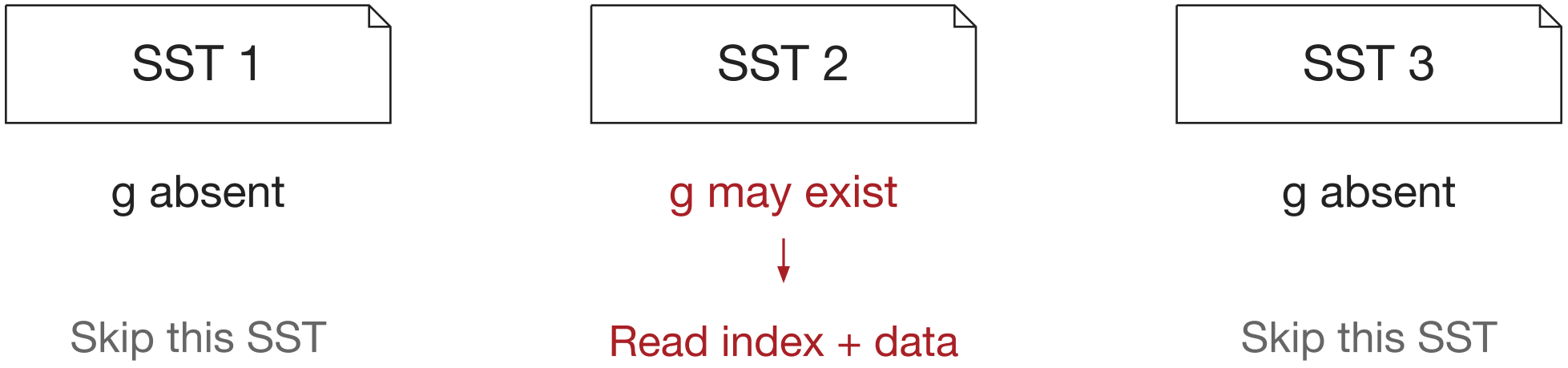
Stored keys: k, m



A false positive causes extra work, not a wrong database answer.

Bloom filters in the RocksDB read path

After checking memory, Get(g) tests the filters of candidate SSTs.



- More bits per key use more memory and reduce false positives.
- Whole-key filters help point lookups; scans still merge runs.

Filters must themselves be available in memory or cache.

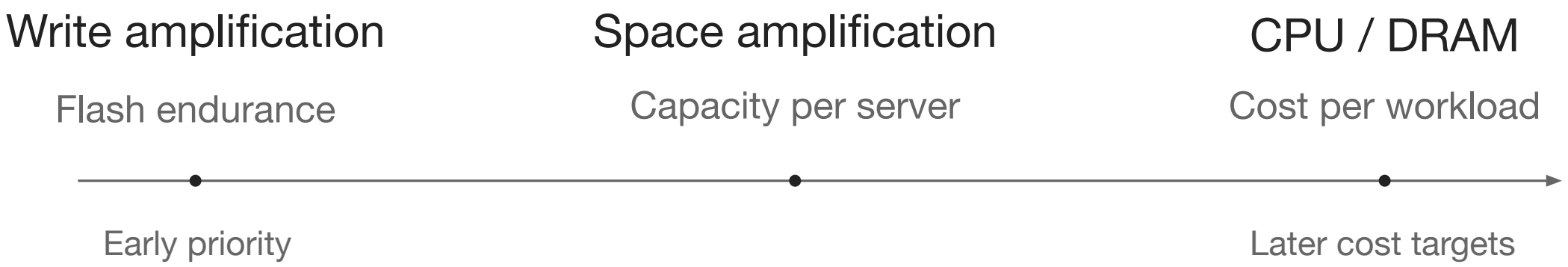
Resource optimization

What the deployed workloads used

Example	CPU	Space	Flash endurance	Read bandwidth
Stream processing	11%	48%	16%	1.6%
Logging / queues	46%	45%	7%	1.0%
Index services	47%	61%	5%	10.0%
Cache	3%	78%	74%	3.5%

Table 2 examples. The separate 42-deployment survey found most workloads space constrained.

The bottleneck moved



Priorities accumulated as deployments changed.

The data structure survived. The optimization target changed.

Dynamic level sizes reduce space overhead

Set level targets from the **actual last-level size**.

Keys	Dynamic targets	Static targets
200 million	12.4%	25.6%
400 million	11.8%	12.2%
600 million	12.2%	16.9%
800 million	12.7%	19.7%
1 billion	12.4%	22.4%

Space overhead in Table 4's random-update experiment. Lower is better.

CPU and DRAM became cost targets

CPU and DRAM became more expensive relative to flash.

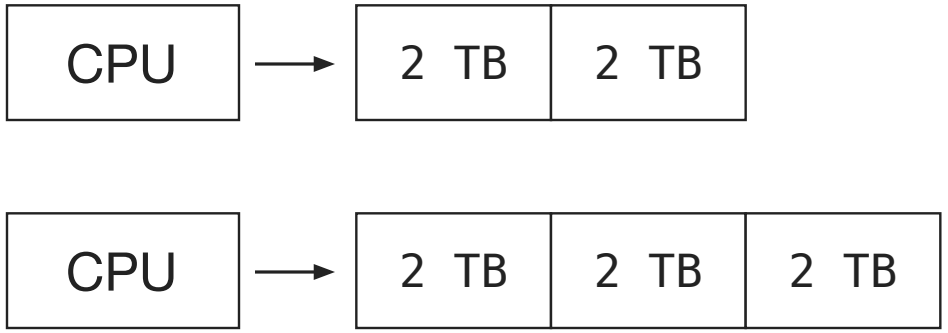
- Hosts reserve spare capacity for growth and failures, so low CPU use can be intentional.
- Checking a filter before the index avoids searches and unnecessary data reads.
- Batched lookups and parallel I/O overlap independent work instead of waiting in sequence.

Optimize cost per request as well as peak throughput.

Disaggregation separates compute and capacity

Disaggregation puts SSDs on storage servers reached over a network.

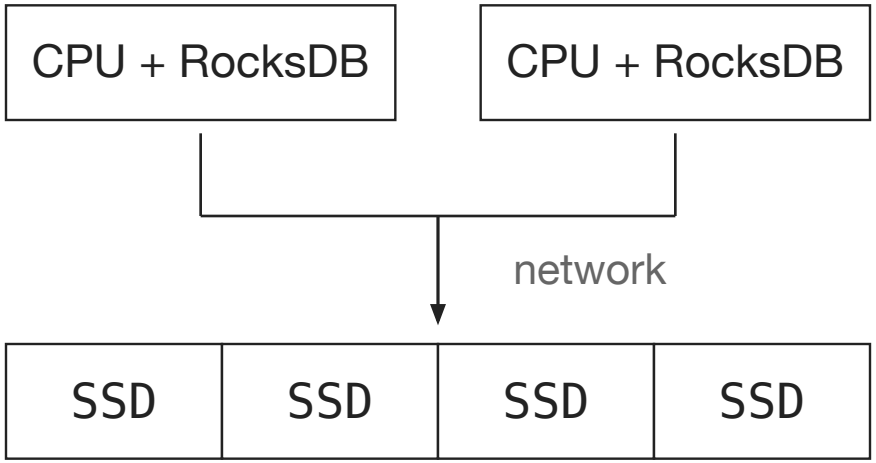
Locally attached SSDs



More disks shift the CPU/space ratio.

One resource can still sit idle.

Disaggregated storage



Provision CPU and flash separately

Compute and flash can grow independently. Network delays and failures now require retries, batching, and I/O priorities.

Data integrity and recovery

What a checksum can detect

A checksum is a small number computed from bytes to detect accidental changes.

1. **Create:** for data x, compute C(x) and save it with the data or its metadata.
2. **Verify:** read or receive bytes y, then compute C(y) using the same function.
3. **Compare:** if C(y) differs from the saved C(x), reject the data as corrupted.

A mismatch detects a change. It cannot reconstruct the original bytes.

A match can miss a checksum collision. The expected checksum must also be protected.

Rare corruption matters at scale

Silent corruption means wrong bytes without an obvious I/O failure.

1 / 3 months

One detected RocksDB corruption per three months across about 100 petabytes (PB).

40% had already reached another replica.

17 / PB

Checksum mismatches per petabyte of files transferred in one observed period.

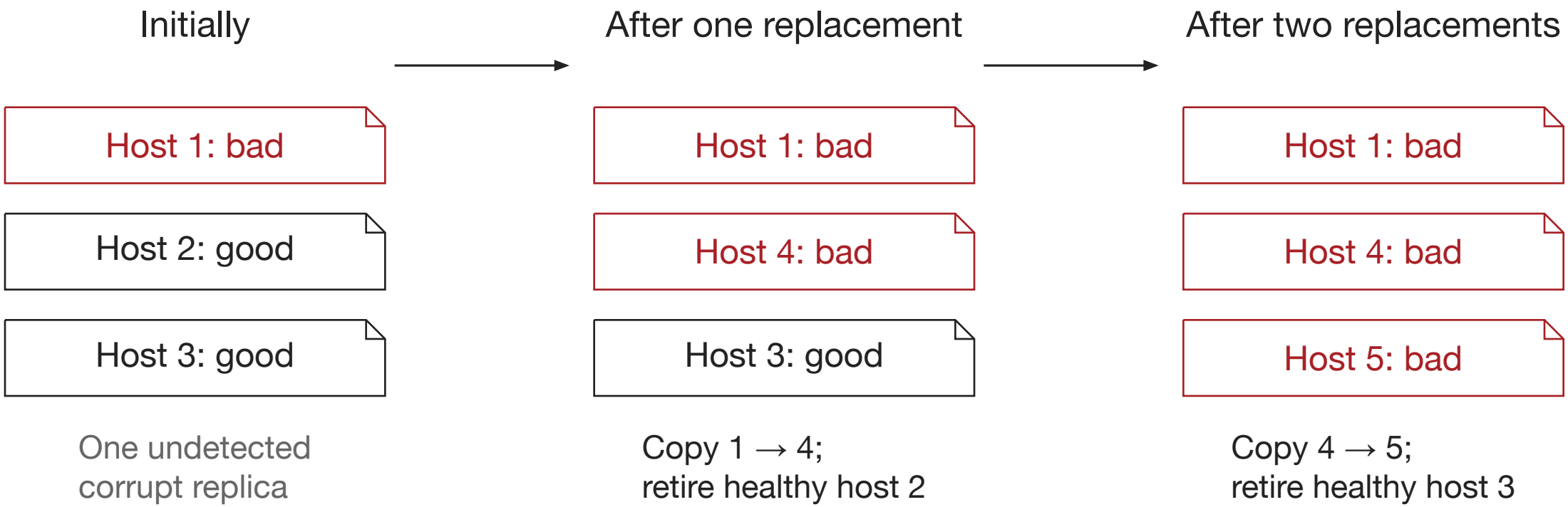
Software bugs were an important cause.

Replication helps only while a correct copy remains.

These are production observations, not universal hardware error rates.

Unchecked copies can replace every good replica

Three hosts hold copies of the same shard. Red marks corrupted data.



Validate the source before replacing a healthy replica with its copy.

Integrity checks at different boundaries

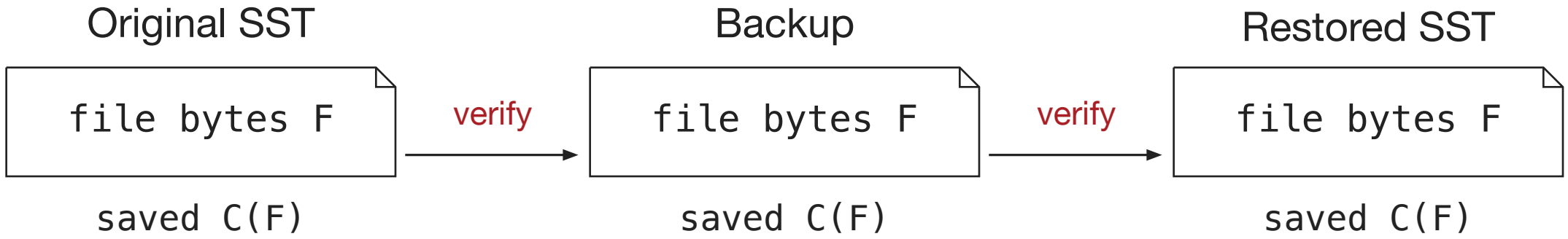
A checksum covers a specific set of bytes between creation and verification.

Boundary	Protected data	Check
In-memory records	`key` `value` `check`	Per-KV protection In progress in 2021
Write handoff	`WAL bytes` `check`	Storage API verifies
Data-block read	`block data` `check`	Compare on read
File transfer	all SST bytes `C(file)`	Verify complete file

KV means key/value. Detection still needs a valid copy for recovery. Status shown as in 2021.

Verify SSTs during copy, backup, and restore

Save C(F), the checksum of file F computed at SST creation.



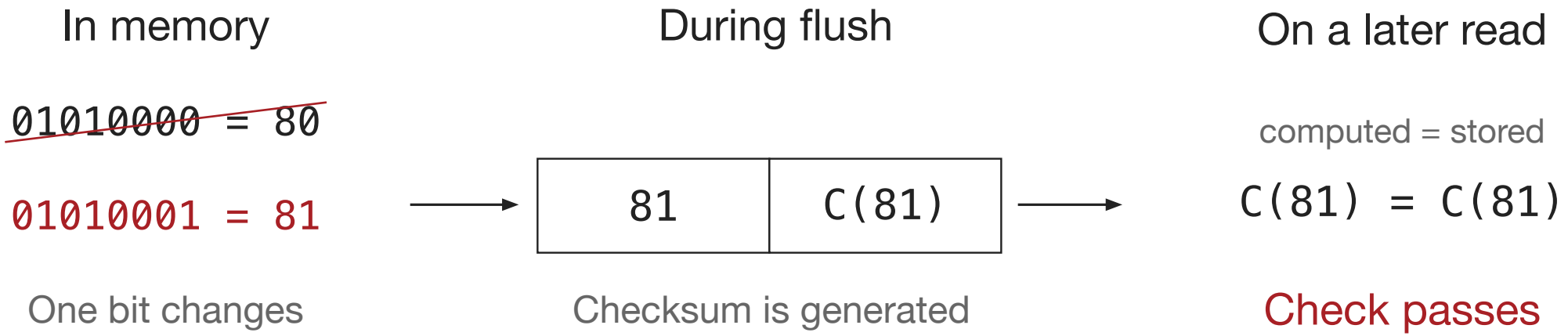
C(received bytes) must equal the saved C(F)

Use this check for replica file copies, backup, and restore.

Reject a mismatch before accepting the copied SST.

2021: whole-file checksums protect SSTs; WAL files require separate protection.

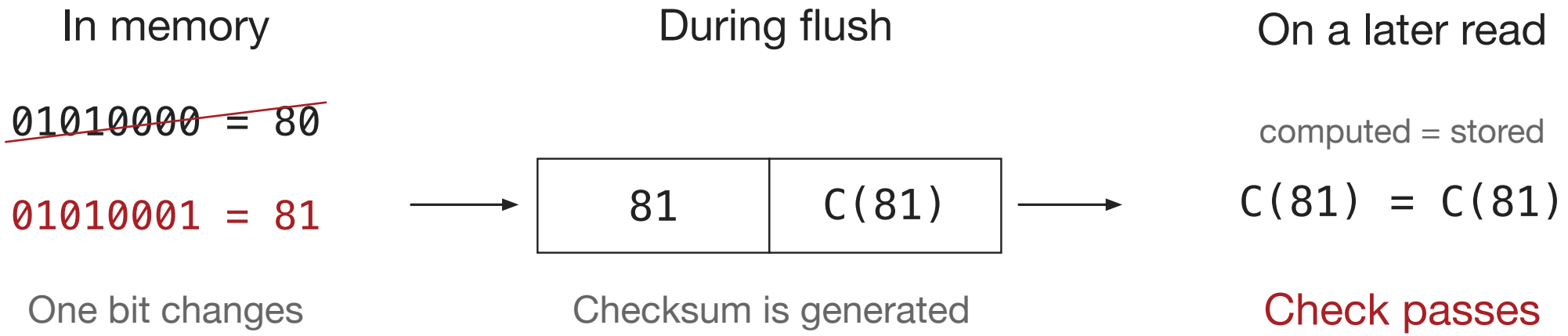
A checksum can protect the wrong value



The file faithfully preserves an already-corrupted value.

Why did verification pass?

A checksum can protect the wrong value



The file faithfully preserves an already-corrupted value.

Why did verification pass?

The checksum was computed after corruption. An earlier record checksum could catch the change before flush.

Carry record checksums through memory

Compute a checksum per key/value record and carry it with each memory copy.

Boundary	Where to verify the record
Write	Check the key/value checksum passed into the engine.
Flush	Check memtable records before writing SST blocks.
Compaction	Check decoded records while merging and rewriting files.
Read	Check memtable and block-cache records before returning them.

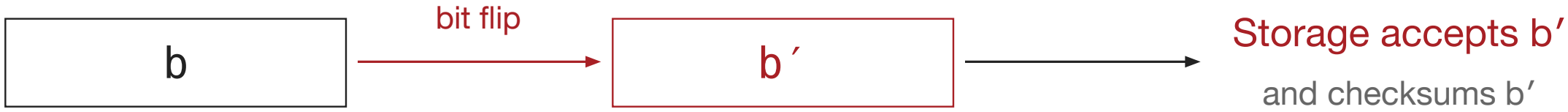
Keep block checksums on SSTs. Record checksums cover the memory paths.

Proposed / in progress in the 2021 paper. A mismatch detects, but does not repair, corruption.

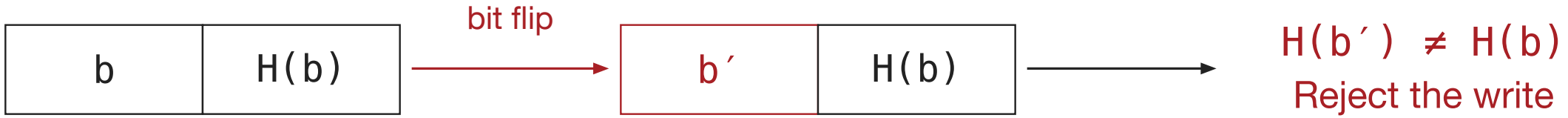
Protect the write across the storage API

H(b) is a checksum made before RocksDB passes write buffer b to storage.

Without a caller-supplied handoff checksum



With a checksum computed before copying



b is the write buffer, including the WAL record's internal checksum.

Storage recomputes H from the received bytes b′ and rejects a mismatch before acknowledging the write. This requires a supporting API.

Errors need different recovery policies

The response must address the cause. Retrying cannot repair corrupted bytes.

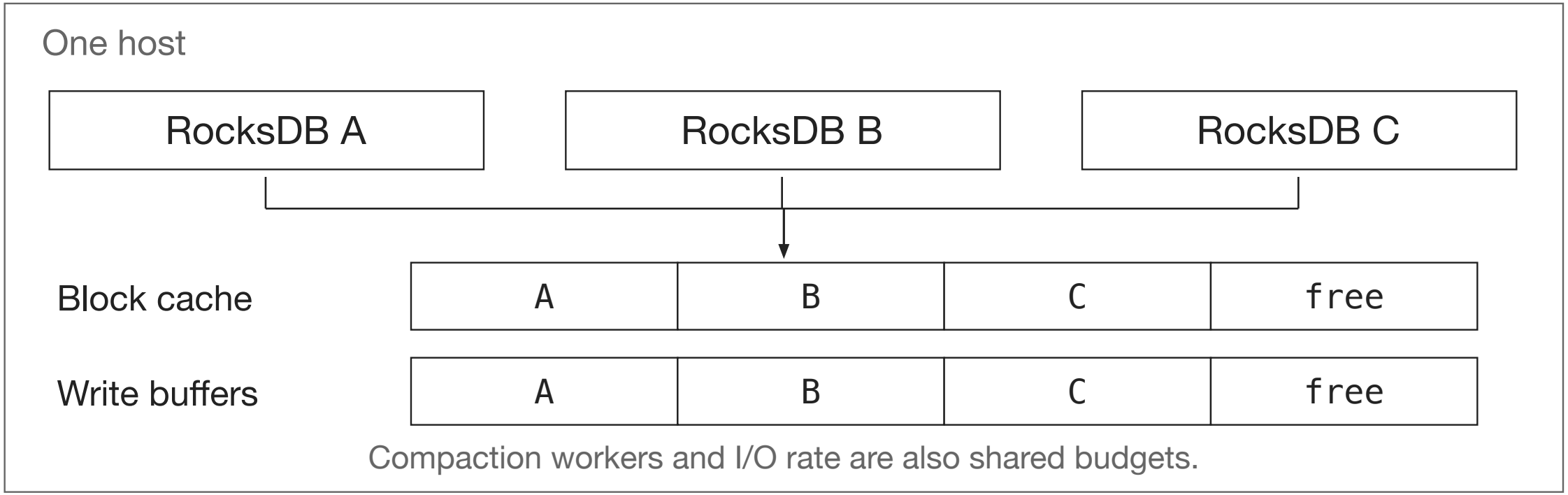
Condition	Response
Transient remote I/O failure	Retry or resume once connectivity or the service recovers.
Out of space	Pause writes, reclaim capacity, then resume safely.
Confirmed corruption	Isolate the bad data and rebuild from a validated replica or backup.

Automatic retry fits a temporary failure. Corruption needs a known-good recovery source.

Operating RocksDB at scale

Many instances share one host

One data partition (shard) per instance. A host may run 100 instances.



Share the block cache (read data) and write buffers (recent writes). Per-instance caps keep one busy shard from exhausting the host.

What the write-ahead log protects

The WAL is an append-only record used to recover recent writes.

1. **Before flush:** Put(k, 9) updates a memtable and records the write in the WAL.
2. **After a crash:** RAM is lost. Replay surviving WAL records to rebuild the memtable.
3. **After durable flush:** SSTs contain the data. Retire a WAL segment when none of its records are still needed.

A write in memory needs a durable recovery source.

The WAL follows the durability contract

WAL mode	What success means
Synchronous	Sync the WAL before acknowledging a durable write.
Buffered	Return before sync. Unsynced writes may be lost in a machine crash.
Disabled	Write no RocksDB WAL. Recover using another durable source.

When is a replicated consensus log enough?

The WAL follows the durability contract

WAL mode	What success means
Synchronous	Sync the WAL before acknowledging a durable write.
Buffered	Return before sync. Unsynced writes may be lost in a machine crash.
Disabled	Write no RocksDB WAL. Recover using another durable source.

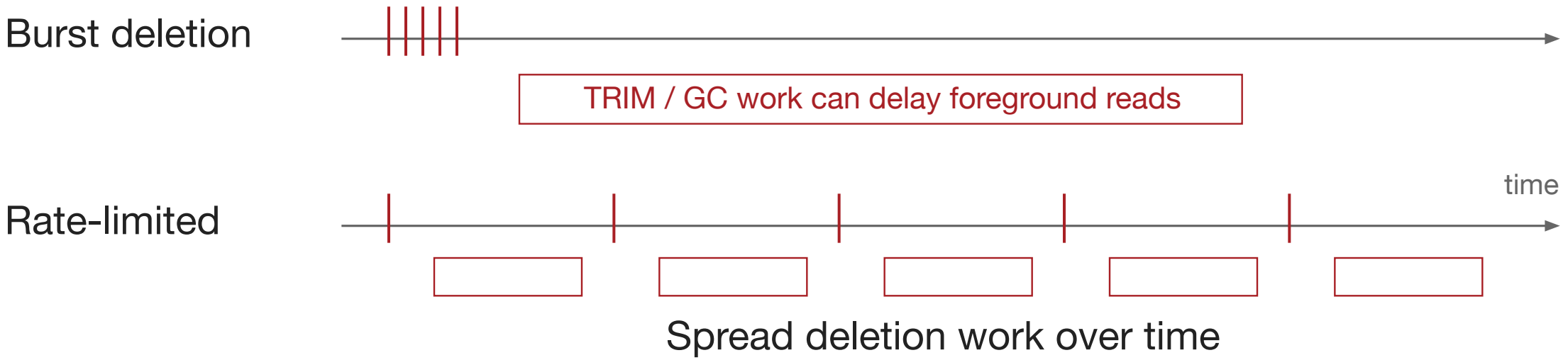
When is a replicated consensus log enough?

It must replay every required write and retain the log until RocksDB has persisted the data.

Deleting files can stall reads

Compaction can make many old SST files obsolete at once.

Illustrative timing: each tick is one file deletion



TRIM tells the SSD which ranges are unused. Garbage collection (GC) reclaims flash. Spreading deletions out avoids a burst of this work.

This coupling depends on filesystem settings and SSD firmware.

Rolling upgrades constrain file formats

A rolling upgrade replaces servers gradually, so old and new code coexist.

Operation	Why the reader must handle this format
Upgrade	New code must open files written by old code.
Rollback	Old code must open files written during the upgrade.
Replica bootstrap	A new replica may receive files from a different version.

Example: rolling back the binary is unsafe if it cannot read yesterday's SSTs.

2021 policy: keep old formats readable and aim for at least one year of forward compatibility, with exceptions for new features.

Configuration becomes an operational interface

25+ configurations

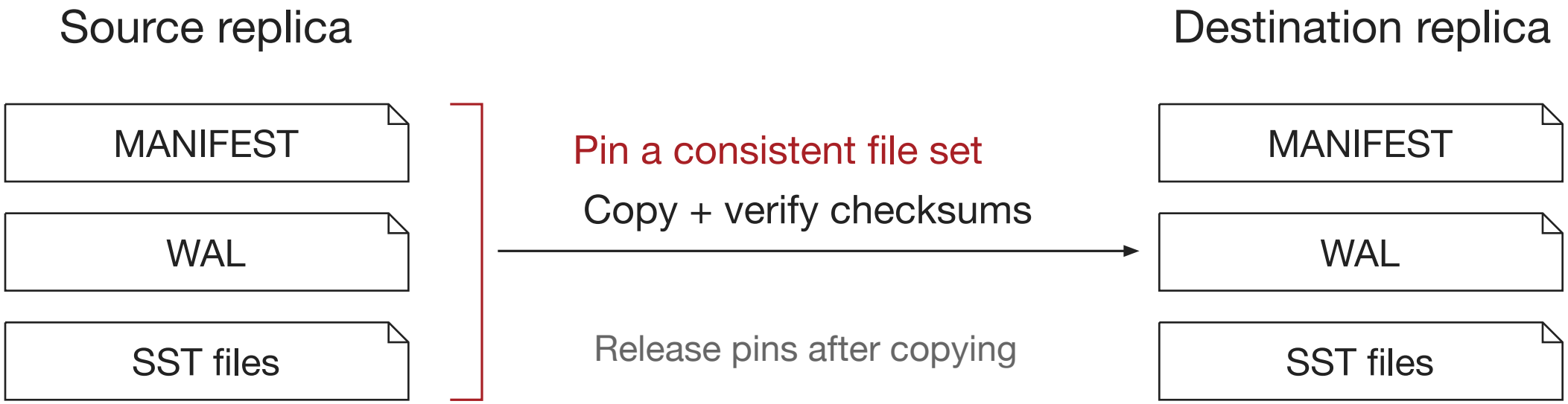
across 39 sampled ZippyDB deployments

- Save chosen options beside the database so restarts can reproduce them.
- Check compatibility before opening existing files with changed options.
- Pin intended settings during upgrades: a new default may reserve more RAM.

Examples: memtable size, cache budget, compression, and compaction policy.

Replication needs storage-engine support

A new replica needs a consistent starting copy of an existing database.



MANIFEST identifies the SST set. WAL holds writes not yet in SSTs.

Pinning keeps the selected files alive while compaction continues.

Logical copying instead reads key/value records and bulk-loads fresh files.

Versions and timestamps

Sequence numbers and timestamps

Property	Sequence number	User timestamp
Assigned by	RocksDB	The application
Meaning	Write order within one RocksDB instance	Application version time, physical or logical
Read limit	Snapshot at sequence S	Read at timestamp T

A version can carry both: sequence 101 and timestamp 30.

For one key, timestamps cannot decrease as sequence numbers increase.

Sequence and timestamp limits both apply

Two versions of key k. Take snapshot S=100 before the second write.

Write	Sequence	Timestamp	Value
First	100	20	80
Second	101	30	81

- Latest view, timestamp 25: return **80**.
- Latest view, timestamp 35: return **81**.
- Snapshot S=100, timestamp 35: return **80**.

An active snapshot preserves its view. Timestamp reads also need retained history.

Local sequence numbers are not a shared clock

Each RocksDB instance numbers its own writes independently.

Instance	Local snapshot	Application time reached
Shard A	Sequence 100	t = 20
Shard B	Sequence 100	t = 35

Equal sequence numbers can represent different application times.

Reading all shards at time 25 needs a shared timestamp and a coordination protocol.

Illustrative histories. Timestamp support helps express the read; the application still coordinates it.

Where should a version timestamp live?

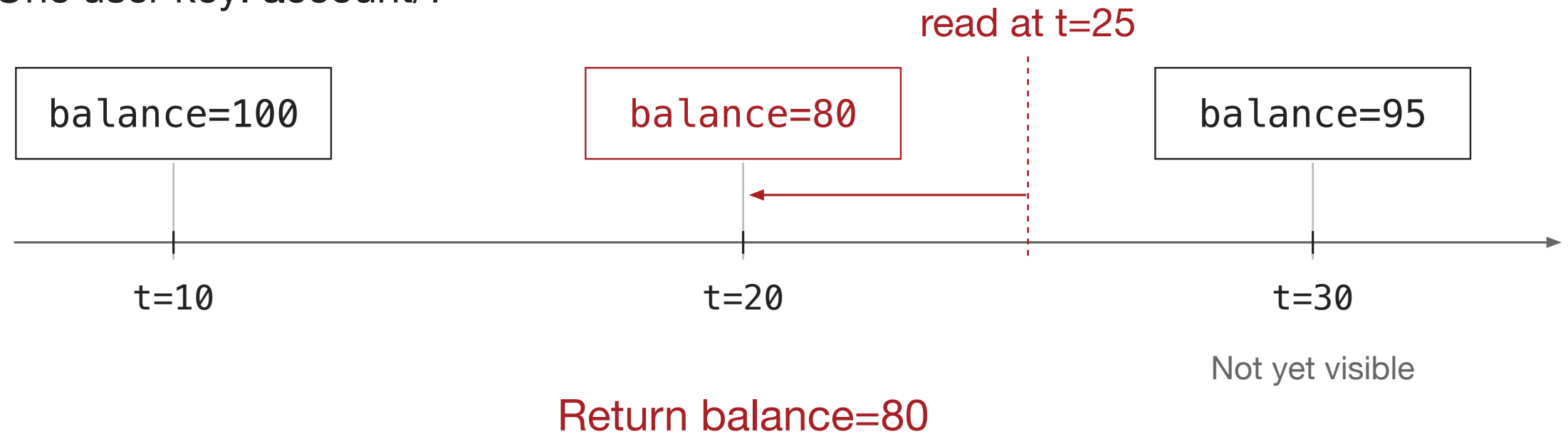
Query: return the version of account/7 visible at time 25.

Placement	Example	How to find the version
In the key	`account/7@20`	Seek among versioned keys for the latest timestamp ≤ 25.
In the value	`(20, balance=80)`	The application must store and search a history of values.
Separate metadata	`key, ts, value`	A timestamp-aware Get selects the visible version for the user key.

Separate timestamp semantics let the engine filter by user key and skip files outside the requested time range.

A timestamp-aware point lookup

One user key: account/7



Choose the newest retained version with timestamp ≤ 25: t=20.

The application controls timestamp order, history retention, and coordination across shards.

Timestamp metadata improved this benchmark

Baseline: timestamps encoded in user keys. Results are throughput ratios.

Populate, then measure	Throughput gain
Sequential fill, random reads	1.2×
Sequential fill, reads during writes	1.9×
Random fill, random reads	1.9×
Random fill, reads during writes	2.0×

2.0× means twice as many operations per second in this DB_bench experiment.

Engine-aware timestamps improve lookup and filtering, at the cost of space and API complexity.

Discussion and takeaways

Design discussion

A host runs 100 shards, each with a RocksDB instance. A consensus log durably records writes. New replicas copy data from existing hosts.

1. Which resources need a shared budget?
2. When is it safe to disable the RocksDB WAL?
3. What must a new replica verify before using a copied database?

Design discussion

A host runs 100 shards, each with a RocksDB instance. A consensus log durably records writes. New replicas copy data from existing hosts.

1. Which resources need a shared budget?
2. When is it safe to disable the RocksDB WAL?
3. What must a new replica verify before using a copied database?

Use host budgets, replayable logs, and a verified file set.

Design discussion: one possible design

1. **Shared resources:** cap cache, memtable memory, and background I/O. Give each shard a bounded share.
2. **Recovery:** disable the RocksDB WAL only if the durable consensus log can replay lost writes. Retain it until engine state is durable.
3. **Replica copy:** pin a consistent file set, check format compatibility, and verify file checksums before using the copy.

Replication provides recovery only when these conditions hold.

What to remember

- Memtables batch small writes. SSTs keep data sorted. Compaction removes obsolete versions.
- Leveled compaction spends more writes to reduce retained space and read work.
- Measure the limiting resource: flash endurance, capacity, CPU, or memory cost.
- Set host-wide budgets and identify which durable log can replay lost memory state.
- Check data before trusting a copy. Retain the versions needed for timestamped reads.

Appendix

Appendix: hardware priorities in 2021

Direction	What it offers
Host-managed flash	Open-channel, zoned namespaces (ZNS), and multi-stream SSDs expose placement or write-order controls.
Remote SSDs	Access flash over a network to scale storage capacity separately from compute.
Storage-class memory	Persistent memory between DRAM and SSDs in cost and latency. Possible roles: cache, main store, or WAL.

The paper asks whether each feature improves the workload’s limiting resource enough to justify its cost.

Appendix: separating keys and large values

Example: a small key points to a 4 KB value.

Values inside SSTs

SST entry: key + 4 KB value

Every compaction copies the large value along with the key.

Values in separate files

SST entry: key + location

The value stays elsewhere. Reads follow its location to fetch it.

Less compaction copying, but extra value reads and garbage collection.

Value garbage collection reclaims bytes that no live key references.

Appendix: read costs depend on the operation

Get looks up one key. An iterator seek positions a scan at a starting key.

Policy	Get I/O with Bloom	Get I/O without Bloom	Iterator seek I/O
Leveled	0.99	1.70	1.84
Tiered	1.03	3.39	4.80
FIFO	1.16	528	967

Filters can reject files for an exact key. A seek must find the next key across runs, exposing FIFO's many-file read cost.

RocksDB 5.9, direct I/O, cache at 10% of compacted size. Tiered: 12 runs. FIFO: 20 filter bits/key.

Appendix: project milestones, 2012–2016

Year	Milestone and purpose
2012	LevelDB fork. Multiple compaction threads use more CPU cores.
2013	Open source and production use. Pluggable parts allow customization.
2014	Backup support creates restorable database copies.
2015	MongoRocks for MongoDB at Facebook. Transactions and bulk loading.
2016	MyRocks for MySQL at Parse. DeleteRange removes a key interval.

Appendix: project milestones, 2017–2020

Year	Milestone and purpose
2017	Experiments with nonvolatile-memory (NVM) caching.
2018	Rocksandra for Cassandra at Instagram. Work on remote, disaggregated storage.
2019	MultiGet issues independent reads in parallel for a batch of keys.
2020	Partial support for timestamps supplied by the application.

Historical status in the talk: NVM work was experimental and timestamp support was partial.

Appendix: the 42-deployment survey

ZippyDB and MyRocks

One bar per deployment; averages across its hosts.

CPU / read bandwidth: busiest hour in the month.

Space / endurance: average over the month.

The paper identifies storage capacity as the common constraint.

Endurance is write-budget usage, not the fraction of the SSD lifetime already worn out.

Appendix: amplification at each layer

Write amplification compares output bytes with input bytes at one layer.

Layer	Talk's rough scale	Why bytes multiply
Storage engine	Often hundreds×	Rewrite a full database page for a small record update.
Filesystem	About 1×	Pass the writes to the SSD.
Inside the SSD	About 1–3×	Copy live pages while reclaiming flash blocks.

Example: rewriting a 4 KB page for a 50-byte change is about 82×, before any extra SSD writes.

These are motivating estimates, not Table 3 results. The paper reports software amplification up to about 100× and SSD amplification of 1.1–3×.