

Guest Lecture • University of Waterloo • Cheriton School of Computer Science

Modern Transactional Storage Engines & The LSM-Tree

From Planetary-Scale Distributed Bigtable to High-Performance Single-Node Storage Kernels

Zichen Zhu

Software Engineer, Google (BigQuery Core) • PhD, Boston University (DiSC Lab)

Affiliations & Core Technologies



Boston University



Google



BigQuery Core



BigQuery Vortex

What is a Storage Engine? Anatomy of a DBMS

1. Query Engine & Optimizer

SQL Parsing • AST Generation • Logical & Physical Cost Optimization

2. Transaction & Concurrency Manager

ACID Guarantees • MVCC / 2PL • Raft / Paxos Distributed Consensus

3. Storage Engine (Physical Kernel)

Physical Page & Block Layout • Buffer Pool & WAL • In-Memory Indexes & SSTables

The storage engine is the persistent physical kernel that maps logical records to hardware bytes.

2006: Jeff Dean & Google's Bigtable

Bigtable Data Model (OSDI 2006)

A sparse, distributed, persistent sorted map over a 3-part key:

(row:string, column:string, time:int64) → string

- One total order: row, then column, then timestamp
- Petabytes of data across 10,000s of commodity PCs



Bigtable replaced in-place disk overwrites with immutable, sequentially written SSTables.

Bigtable Architecture: Tablets & Request Routing

Logical Table: Rows × Column Families

	Row Key	CF: "contents"	CF: "anchor"
Tablet 1 [com.a .. com.f] (~100-200 MB)	com.apple	html: "..."	cnnsi: "Apple"
	com.cnn	html: "..."	my.look: "CNN"
Tablet 2 [com.g .. com.z] (~100-200 MB)	com.google	html: "..."	stanford: "Google"
	com.yahoo	html: "..."	cnet: "Yahoo"

Bigtable Architecture: Tablets & Request Routing

Logical Table: Rows × Column Families

	Row Key	CF: "contents"	CF: "anchor"
Tablet 1 [com.a .. com.f] (~100-200 MB)	com.apple	html: "..."	cnnsi: "Apple"
	com.cnn	html: "..."	my.look: "CNN"
Tablet 2 [com.g .. com.z] (~100-200 MB)	com.google	html: "..."	stanford: "Google"
	com.yahoo	html: "..."	cnet: "Yahoo"

Request Routing & Serving

Client Request: Row "com.google"
➔ Routes RPC to Tablet Server 2 [com.g .. com.z]

🖥️ Tablet Server 1 (Node A)
• Serves Tablet 1 [com.a .. com.f]

🖥️ Tablet Server 2 (Node B) 🎯 Target
• Buffers Tablet 2 writes in MemTable (RAM)

Bigtable Architecture: Tablets & Request Routing

Logical Table: Rows × Column Families

	Row Key	CF: "contents"	CF: "anchor"
Tablet 1 [com.a .. com.f] (~100-200 MB)	com.apple	html: "..."	cnnsi: "Apple"
	com.cnn	html: "..."	my.look: "CNN"
Tablet 2 [com.g .. com.z] (~100-200 MB)	com.google	html: "..."	stanford: "Google"
	com.yahoo	html: "..."	cnet: "Yahoo"

Request Routing & Serving

Client Request: Row "com.google"
➔ Routes RPC to Tablet Server 2 [com.g .. com.z]

🖥️ Tablet Server 1 (Node A)
• Serves Tablet 1 [com.a .. com.f]

🖥️ Tablet Server 2 (Node B) 🎯 Target
• Buffers Tablet 2 writes in MemTable (RAM)

💾 Colossus / GFS Shared Storage
• Stores Commit Logs (WAL) & Immutable SSTables
• Decouples compute & storage for fast failover

Bigtable splits sorted rows into Tablets; Tablet Servers buffer writes in RAM and flush SSTables to GFS.

The Key-Value Abstraction & The B-Tree Dilemma

Put(K, V)

Get(K)

Delete(K)

Scan(Min, Max)

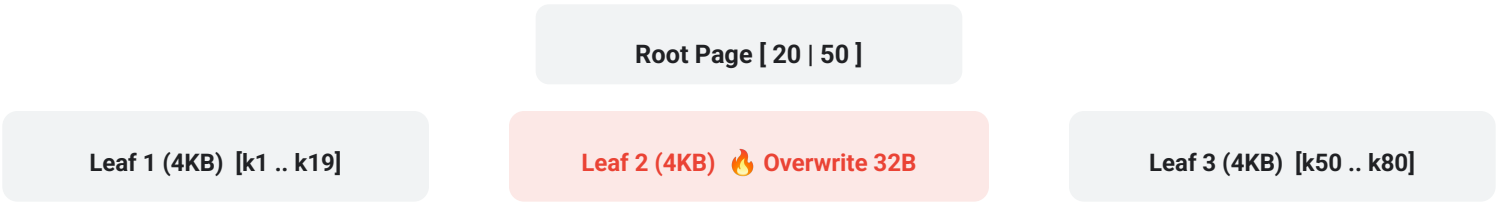
The Key-Value Abstraction & The B-Tree Dilemma

Put(K, V)

Get(K)

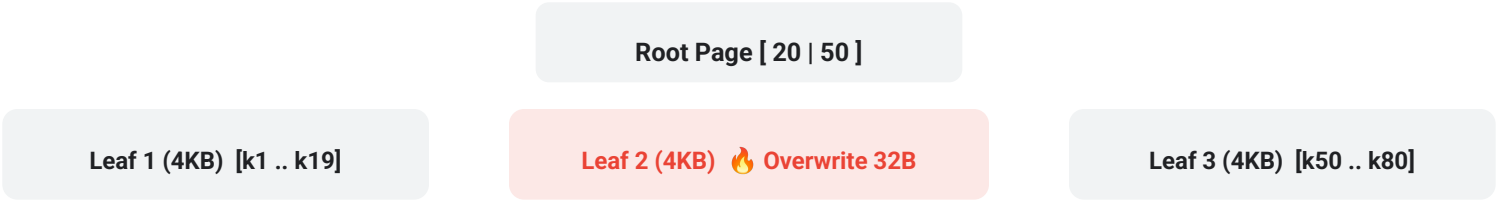
Delete(K)

Scan(Min, Max)



The Key-Value Abstraction & The B-Tree Dilemma

Put(K, V) | Get(K) | Delete(K) | Scan(Min, Max)



⚠️ **In-Place Overwrite Cost: Modifying 32B rewrites an entire 4KB page to flash SSD**

$$\mathrm{WA}_{\text{B-Tree}} = \frac{\text{Page Size}}{\text{Updated Record Size}} = \frac{4{,}096\ \text{B}}{32\ \text{B}} = 128\times$$

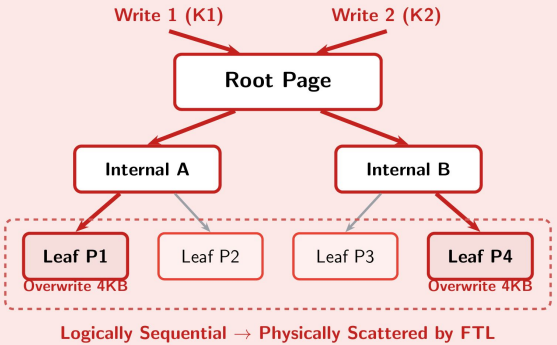
To identify and modify a value on a single node, B-Trees traverse the page index, load the 4KB leaf page, modify in RAM, and write back 4,096 bytes.

In-place updates in B-Trees cause severe Write Amplification (WA > 100x) for small key-value writes.

The Physics of Storage: Sequential vs. Random I/O

B-Tree: Random In-Place Overwrites

🔥 ~250 – 400 MB/s (FTL GC & Up to 25× Penalty)

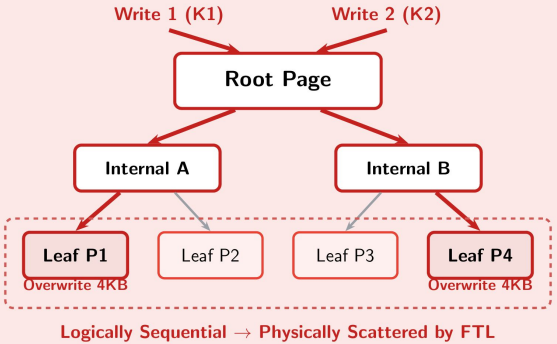


- **Scattered I/O:** In-place mutations force **random 4KB rewrites**.
- **Two-Layer WAF:** Total WAF = **WA_DBMS × WA_SSD** (Doublewrite + GC).
- **Flash Burnout:** Burns out 5-yr SSD in **6 weeks** (**VLDB '26 Best Paper**)!

The Physics of Storage: Sequential vs. Random I/O

B-Tree: Random In-Place Overwrites

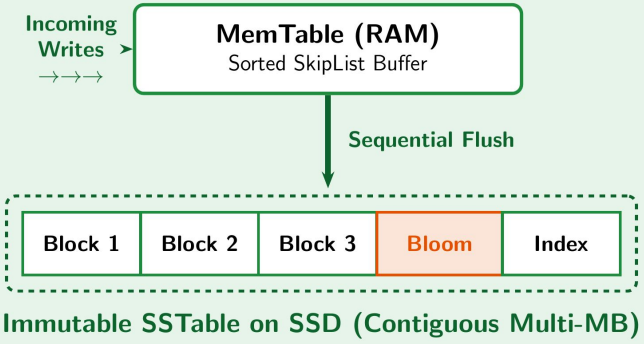
🔥 ~250 – 400 MB/s (FTL GC & Up to 25× Penalty)



- **Scattered I/O:** In-place mutations force **random 4KB rewrites**.
- **Two-Layer WAF:** Total WAF = **WA_DBMS × WA_SSD** (Doublewrite + GC).
- **Flash Burnout:** Burns out 5-yr SSD in **6 weeks** (**VLDB '26 Best Paper**)!

LSM-Tree: Sequential Append-Only Writes

⚡ ~6,500 – 10,000 MB/s (Full Sequential SSD Bandwidth)

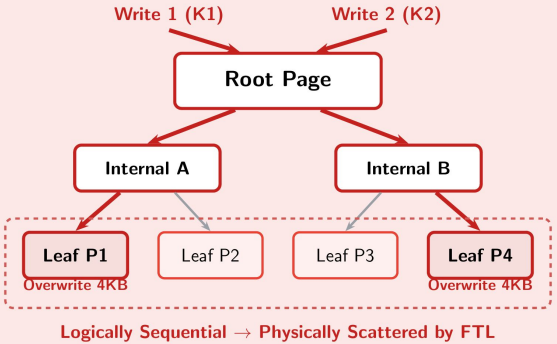


- **Sequential Streaming:** Buffers in RAM, flushes **multi-MB SSTables**.
- **Minimal FTL GC:** Avoids 4KB random overwrites; **keeps WA_SSD low**.
- **Hardware Longevity:** Unlocks **SSD sequential speed** & 5+ yr SSD life!

The Physics of Storage: Sequential vs. Random I/O

B-Tree: Random In-Place Overwrites

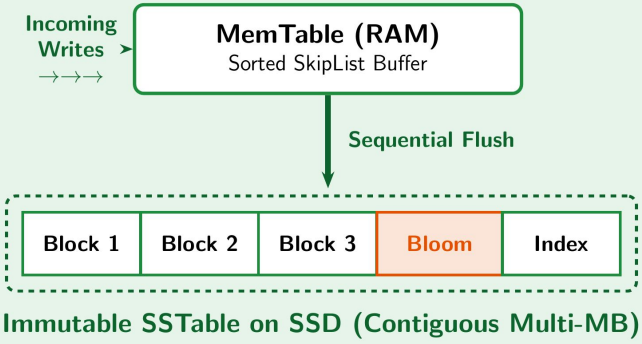
🔥 ~250 – 400 MB/s (FTL GC & Up to 25× Penalty)



- **Scattered I/O:** In-place mutations force **random 4KB rewrites**.
- **Two-Layer WAF:** Total WAF = **WA_DBMS × WA_SSD** (Doublewrite + GC).
- **Flash Burnout:** Burns out 5-yr SSD in **6 weeks** (**VLDB '26 Best Paper**)!

LSM-Tree: Sequential Append-Only Writes

⚡ ~6,500 – 10,000 MB/s (Full Sequential SSD Bandwidth)



- **Sequential Streaming:** Buffers in RAM, flushes **multi-MB SSTables**.
- **Minimal FTL GC:** Avoids 4KB random overwrites; **keeps WA_SSD low**.
- **Hardware Longevity:** Unlocks **SSD sequential speed** & 5+ yr SSD life!

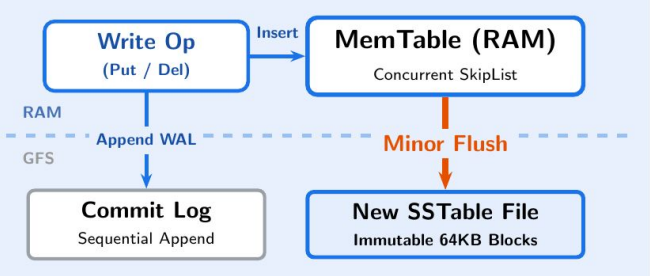
Hardware Baseline: Modern SSDs (Enterprise PCIe 4.0/5.0, Samsung PM9A3 / Micron 9550, 3D TLC) • References: VLDB '26 Best Paper (Lee et al., 'How to Write to SSDs'), FAST '24 (Jun et al.)

LSM-trees turn random writes into sequential I/O—unlocking SSD sequential bandwidth and preventing flash burnout.

Bigtable (OSDI 2006): MemTable & SSTables

Minor Compaction: Ingestion & Flush

⚡ Zero Disk Seeks (RAM SkipList → Sequential SSTable)

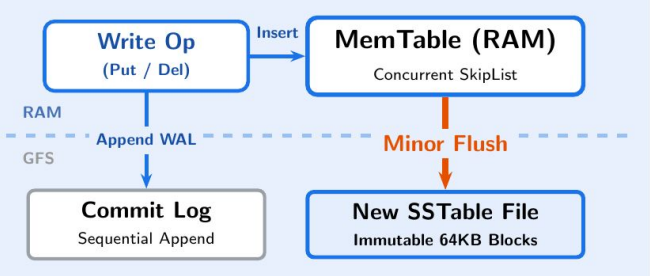


- **Active MemTable:** Ingests **Put/Delete in RAM** via **sorted SkipList**.
- **Commit Log (WAL):** **Append-only log** on GFS ensures **crash durability**.
- **Minor Compaction:** Flushes full MemTable (**~64MB**) as a **new SSTable**.

Bigtable (OSDI 2006): MemTable & SSTables

Minor Compaction: Ingestion & Flush

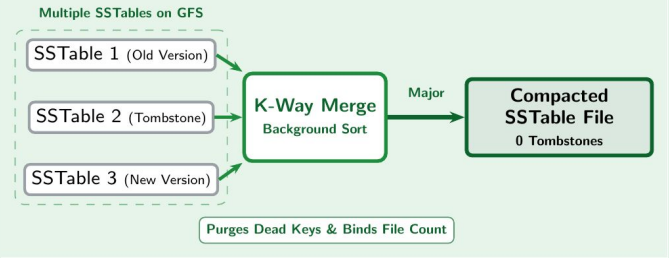
⚡ **Zero Disk Seeks (RAM SkipList → Sequential SSTable)**



- **Active MemTable:** Ingests **Put/Delete in RAM** via **sorted SkipList**.
- **Commit Log (WAL):** **Append-only log** on GFS ensures **crash durability**.
- **Minor Compaction:** Flushes full MemTable (**~64MB**) as a **new SSTable**.

Major Compaction: Background Consolidation

🔄 **K-Way Merge Sort (Garbage Collection & Tombstone Purge)**

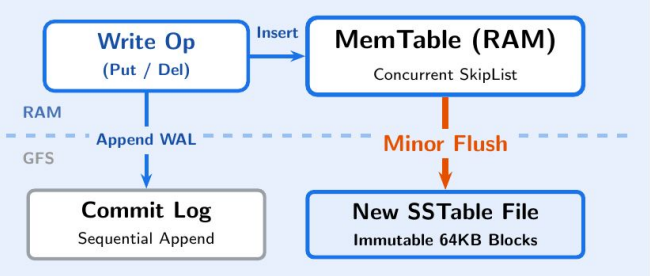


- **Immutable SSTables:** **64KB compressed blocks** + **in-memory index**.
- **Merging Compaction:** **Binds file count** by **merging subset** of files.
- **Major Compaction:** **Rewrites all runs to 1 file**; **purges tombstones**.

Bigtable (OSDI 2006): MemTable & SSTables

Minor Compaction: Ingestion & Flush

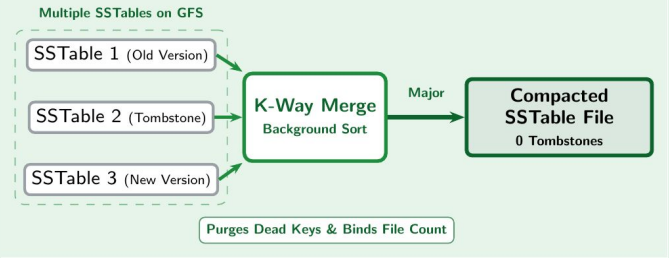
⚡ Zero Disk Seeks (RAM SkipList → Sequential SSTable)



- **Active MemTable:** Ingests **Put/Delete in RAM** via **sorted SkipList**.
- **Commit Log (WAL):** **Append-only log** on GFS ensures **crash durability**.
- **Minor Compaction:** Flushes full MemTable (**~64MB**) as a **new SSTable**.

Major Compaction: Background Consolidation

🔄 K-Way Merge Sort (Garbage Collection & Tombstone Purge)



- **Immutable SSTables:** **64KB compressed blocks + in-memory index**.
- **Merging Compaction:** **Binds file count** by **merging subset** of files.
- **Major Compaction:** **Rewrites all runs to 1 file**; **purges tombstones**.

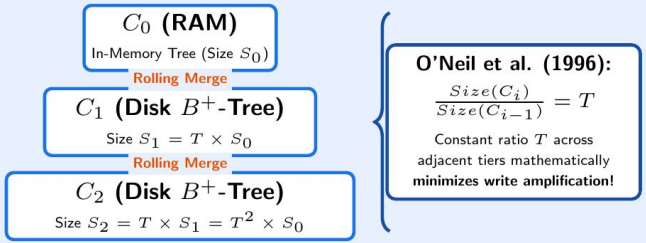
Architectural Foundation: Chang et al., 'Bigtable: A Distributed Storage System for Structured Data', OSDI 2006 (Figure 5)

Bigtable decouples writes from disks: MemTables absorb mutations, flushes and compactions maintain order.

History of the LSM-Tree: 1996 to Modern SSTables

1996: Multi-Component LSM (O'Neil et al.)

Multi-Tier B⁺-Trees with Continuous Rolling Merge

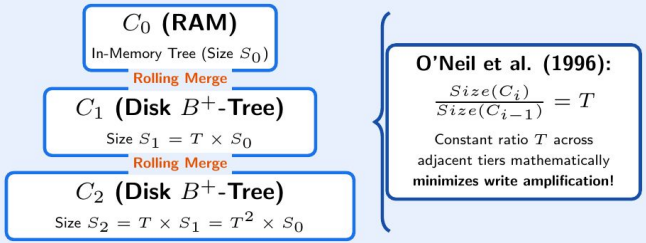


- **Multi-Tier Memory/Disk:** **C_0 in RAM** flushes to **disk B⁺-trees (C_1..Ck)**.
- **Rolling Merge:** Merges **leaf pages sequentially** via **moving cursors**.
- **Optimal Size Ratio:** Proved **$S(C_i)/S(C_{i-1}) = T$** **minimizes total WA!**

History of the LSM-Tree: 1996 to Modern SSTables

1996: Multi-Component LSM (O'Neil et al.)

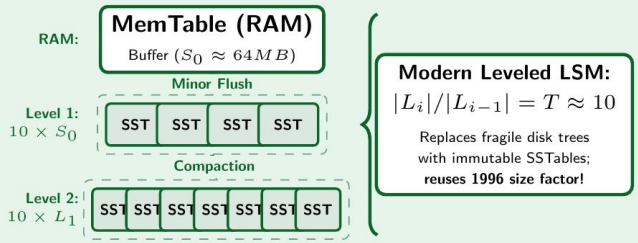
📐 **Multi-Tier B⁺-Trees with Continuous Rolling Merge**



- **Multi-Tier Memory/Disk:** **C_0 in RAM** flushes to **disk B⁺-trees (C_1..Ck)**.
- **Rolling Merge:** Merges **leaf pages sequentially** via **moving cursors**.
- **Optimal Size Ratio:** Proved **$S(C_i)/S(C_{i-1}) = T$** **minimizes total WA!**

2011: Modern Leveled LSM (LevelDB)

⚡ **Multi-Level Partitioned SSTables (Constant Factor T ≈ 10)**

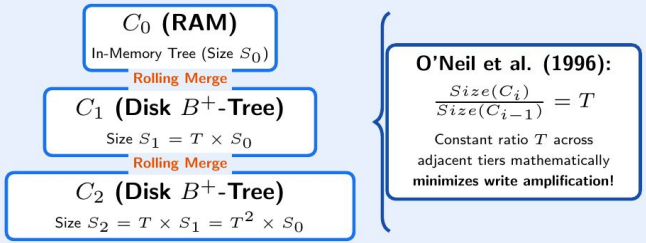


- **Replaced Disk Trees:** Swapped **fragile B⁺-trees** for **immutable SSTables**.
- **Partitioned Levels:** Each tier is split into **sorted, disjoint files**.
- **Reusing 1996 Proof:** Maintains **constant factor T ≈ 10** to **bound WA**.

History of the LSM-Tree: 1996 to Modern SSTables

1996: Multi-Component LSM (O'Neil et al.)

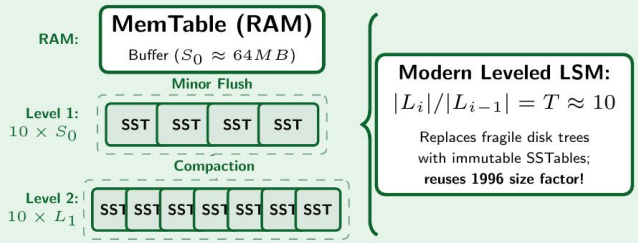
📐 **Multi-Tier B⁺-Trees with Continuous Rolling Merge**



- **Multi-Tier Memory/Disk:** **C_0 in RAM** flushes to **disk B⁺-trees (C_1..Ck)**.
- **Rolling Merge:** Merges **leaf pages sequentially** via **moving cursors**.
- **Optimal Size Ratio:** Proved **$S(C_i)/S(C_{i-1}) = T$** **minimizes total WA!**

2011: Modern Leveled LSM (LevelDB)

⚡ **Multi-Level Partitioned SSTables (Constant Factor T ≈ 10)**



- **Replaced Disk Trees:** Swapped **fragile B⁺-trees** for **immutable SSTables**.
- **Partitioned Levels:** Each tier is split into **sorted, disjoint files**.
- **Reusing 1996 Proof:** Maintains **constant factor T ≈ 10** to **bound WA**.

Historical Evolution: O'Neil et al. (Acta Informatica 1996) • Dean & Ghemawat (LevelDB 2011)

Modern LSMs replace fragile disk B-trees with SSTables while preserving 1996's optimal constant size factor T.

The Modern Ecosystem: LevelDB → RocksDB & Pebble



Google (2011)

- Single-threaded baseline engine
- Global mutex on write path
- Embedded in Chrome IndexedDB

The Modern Ecosystem: LevelDB → RocksDB & Pebble



Google (2011)

- Single-threaded baseline engine
- Global mutex on write path
- Embedded in Chrome IndexedDB



Meta (2013)

- Multi-threaded flush & compaction
- Column Families & rate limiters
- Powers MyRocks, Kafka, TiKV

The Modern Ecosystem: LevelDB → RocksDB & Pebble



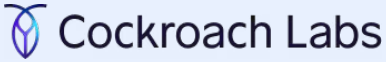
Google (2011)

- Single-threaded baseline engine
- Global mutex on write path
- Embedded in Chrome IndexedDB



Meta (2013)

- Multi-threaded flush & compaction
- Column Families & rate limiters
- Powers MyRocks, Kafka, TiKV



Pebble Storage Engine (2019)

- Pure Go (zero CGo call overhead)
- First-class Range Deletions
- Core engine for Raft SQL nodes

The Modern Ecosystem: LevelDB → RocksDB & Pebble



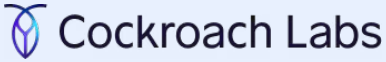
Google (2011)

- Single-threaded baseline engine
- Global mutex on write path
- Embedded in Chrome IndexedDB



Meta (2013)

- Multi-threaded flush & compaction
- Column Families & rate limiters
- Powers MyRocks, Kafka, TiKV



Pebble Storage Engine (2019)

- Pure Go (zero CGo call overhead)
- First-class Range Deletions
- Core engine for Raft SQL nodes

🎯 Specialized Workload-Tailored Storage Engines



Speedb (2022) • RocksDB Redesign

- Auto-tuned memory allocation
- Paired Bloom filters
- Eliminates write stalls under load



Microsoft (2018) ➔ F² (SIGMOD '24)

- In-memory hash + append disk log
- Millions ops/sec point analytics



VMware (2020) • USENIX ATC '20

- Combination of Bᵋ-Tree & LSM-Tree
- Quotient maplets for branch routing

The core multi-tier data organization & compaction remains universal across modern engines.

How KV Stores Power Relational Databases

1. Ordered Key Encoding

SQL Table Schema:
User (ID PK, Country, Age)

- Primary Key Encoding:
  Key: /User/Primary/<ID>
  Val: (Country, Age)

- Secondary Index (Country):
  Key: /User/idx/<Country>/<ID>
  Val: ∅ (Empty)

Ordered KV prefixes preserve relational sort order across SSTables.

How KV Stores Power Relational Databases

1. Ordered Key Encoding

SQL Table Schema:
User (ID PK, Country, Age)

• Primary Key Encoding:
  Key: /User/Primary/<ID>
  Val: (Country, Age)

• Secondary Index (Country):
  Key: /User/idx/<Country>/<ID>
  Val: ∅ (Empty)

Ordered KV prefixes preserve relational sort order across SSTables.

2. SQL Query ➔ KV Iterators

• Point Query:
  SELECT * WHERE ID = 42
  ➔ KV.Get("/User/Primary/42")

• Secondary Range Scan:
  SELECT * WHERE Country = 'CA'
  1. KV.Seek("/User/idx/'CA'/")
  2. Stream keys to collect IDs
  3. MultiGet primary table for rows

Translates declarative WHERE into contiguous 1D range scans.

How KV Stores Power Relational Databases

1. Ordered Key Encoding

SQL Table Schema:
User (ID PK, Country, Age)

• Primary Key Encoding:
  Key: /User/Primary/<ID>
  Val: (Country, Age)

• Secondary Index (Country):
  Key: /User/idx/<Country>/<ID>
  Val: ∅ (Empty)

Ordered KV prefixes preserve relational sort order across SSTables.

2. SQL Query ➔ KV Iterators

• Point Query:
  SELECT * WHERE ID = 42
  ➔ KV.Get("/User/Primary/42")

• Secondary Range Scan:
  SELECT * WHERE Country = 'CA'
  1. KV.Seek("/User/idx/'CA'/")
  2. Stream keys to collect IDs
  3. MultiGet primary table for rows

Translates declarative WHERE into contiguous 1D range scans.

3. Mutations & Tombstones

• UPDATE Statement:
  UPDATE User SET Country='US'
  WHERE ID = 42

1. Read old state: Country='CA'
2. Primary Put("/User/Primary/42", 'US')
3. Delete old index: Delete(".../'CA'/42")
4. Insert new index: Put(".../'US'/42")

Atomically committed via WriteBatch into WAL & active MemTable.

Distributed SQL engines map SQL plans onto ordered KV prefixes: fast Seek, Scan, and Put.

The Storage Engine Spectrum: Counting Page I/Os

💡 **Why retire Big-O? In storage engines, 6 page I/Os means 6× higher latency than 1 page I/O — constant factors ARE the design!**

Storage Paradigm	Write Cost (Ingestion)	Point Read (Exact Key)	Range Scan (Key Range)	Trade-off & Architecture

The Storage Engine Spectrum: Counting Page I/Os

💡 **Why retire Big-O? In storage engines, 6 page I/Os means 6× higher latency than 1 page I/O — constant factors ARE the design!**

Storage Paradigm	Append-Only Log
Write Cost (Ingestion)	**1 / B page I/Os** Batched append (1 if sync per write)
Point Read (Exact Key)	**N / B page I/Os** Full table scan required
Range Scan (Key Range)	**N / B page I/Os** Full table scan required
Trade-off & Architecture	⚡ **Max Write Throughput** ❌ **Unusable for Reads (no index)**

The Storage Engine Spectrum: Counting Page I/Os

💡 **Why retire Big-O? In storage engines, 6 page I/Os means 6× higher latency than 1 page I/O — constant factors ARE the design!**

Storage Paradigm	Append-Only Log	B-Tree
Write Cost (Ingestion)	**1 / B page I/Os** Batched append (1 if sync per write)	**1 page rewrite** Up to 128× WA (In-place & GC)
Point Read (Exact Key)	**N / B page I/Os** Full table scan required	**1 page I/O (leaf)** Internal nodes cached in RAM
Range Scan (Key Range)	**N / B page I/Os** Full table scan required	**1 seek + sequential scan** Contiguous leaf page pointers
Trade-off & Architecture	⚡ **Max Write Throughput** ❌ **Unusable for Reads (no index)**	🔍 **Optimal Read Locality** ❌ **High Write Amp**

The Storage Engine Spectrum: Counting Page I/Os

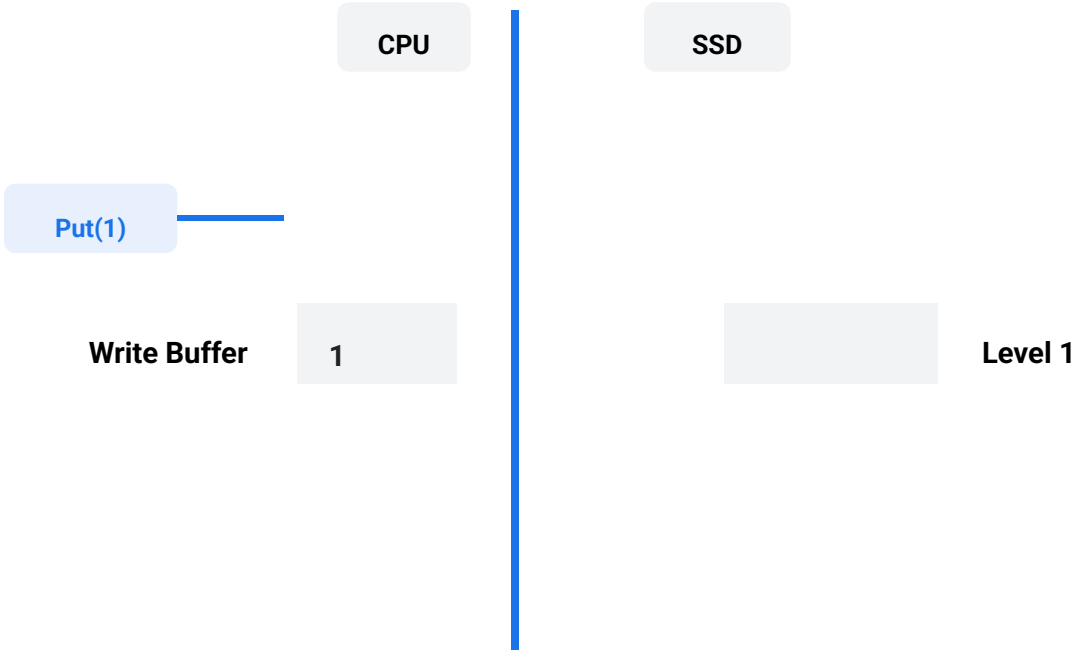
💡 **Why retire Big-O? In storage engines, 6 page I/Os means 6× higher latency than 1 page I/O — constant factors ARE the design!**

Storage Paradigm	Append-Only Log	Traditional LSM-Tree	B-Tree
Write Cost (Ingestion)	**1 / B page I/Os** Batched append (1 if sync per write)	**(T/2)·L / B amortized** WAL (1/B) + RAM buffer & compactions	**1 page rewrite** Up to 128× WA (In-place & GC)
Point Read (Exact Key)	**N / B page I/Os** Full table scan required	**≤ L ➔ Bloom: 1 + L·ε ≈ 1** RAM filters skip empty levels	**1 page I/O (leaf)** Internal nodes cached in RAM
Range Scan (Key Range)	**N / B page I/Os** Full table scan required	**L seeks + L-way merge** Parallel iterators across levels	**1 seek + sequential scan** Contiguous leaf page pointers
Trade-off & Architecture	⚡ Max Write Throughput ❌ Unusable for Reads (no index)	⚡ High Ingest + Sub-ms Reads 🔍 Balanced via RAM & Filters	🔍 Optimal Read Locality ❌ High Write Amp

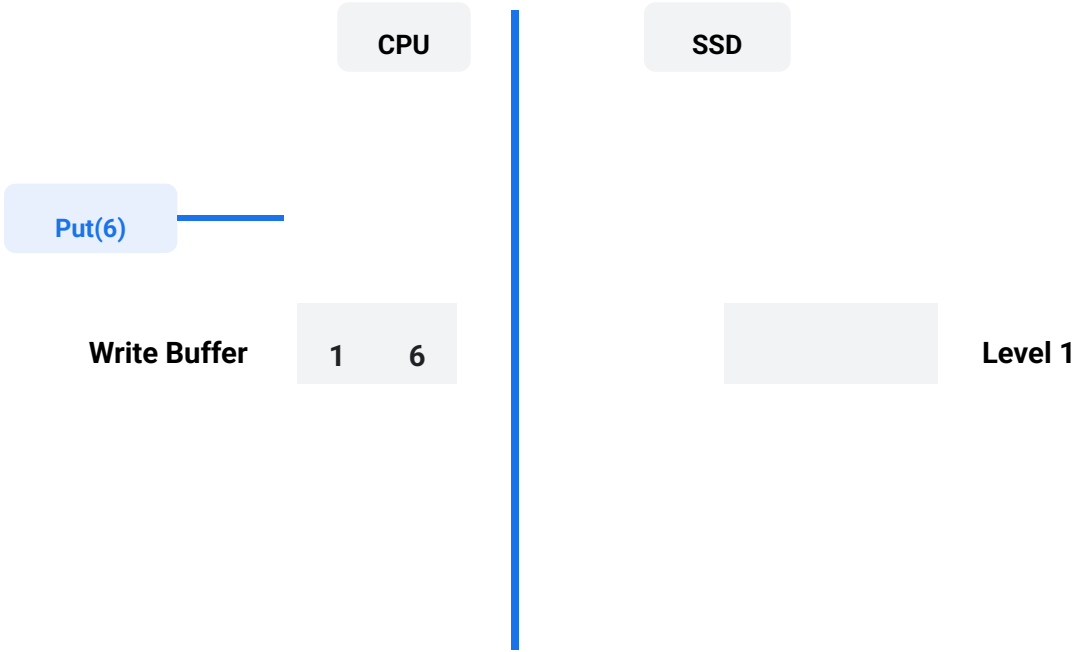
N = entries • B = entries per 4 KB page • T = size ratio • L = ⌈log_T(N/B)⌉ levels (6–7 in RocksDB) • ε = Bloom false positive rate

Cost Model: Aggarwal & Vitter (CACM '88); Dayan & Athanassoulis, Dostoevsky (ACM SIGMOD '18, DOI: 10.1145/3183713.3196927)

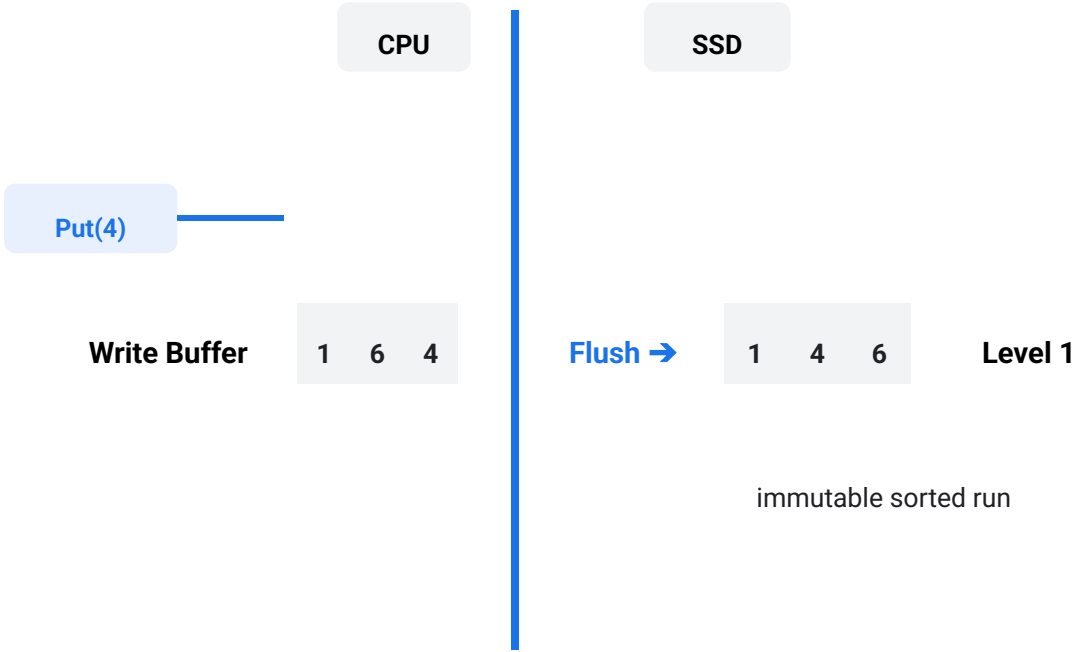
LSM Basics - Ingestion & MemTable



LSM Basics - Ingestion & MemTable

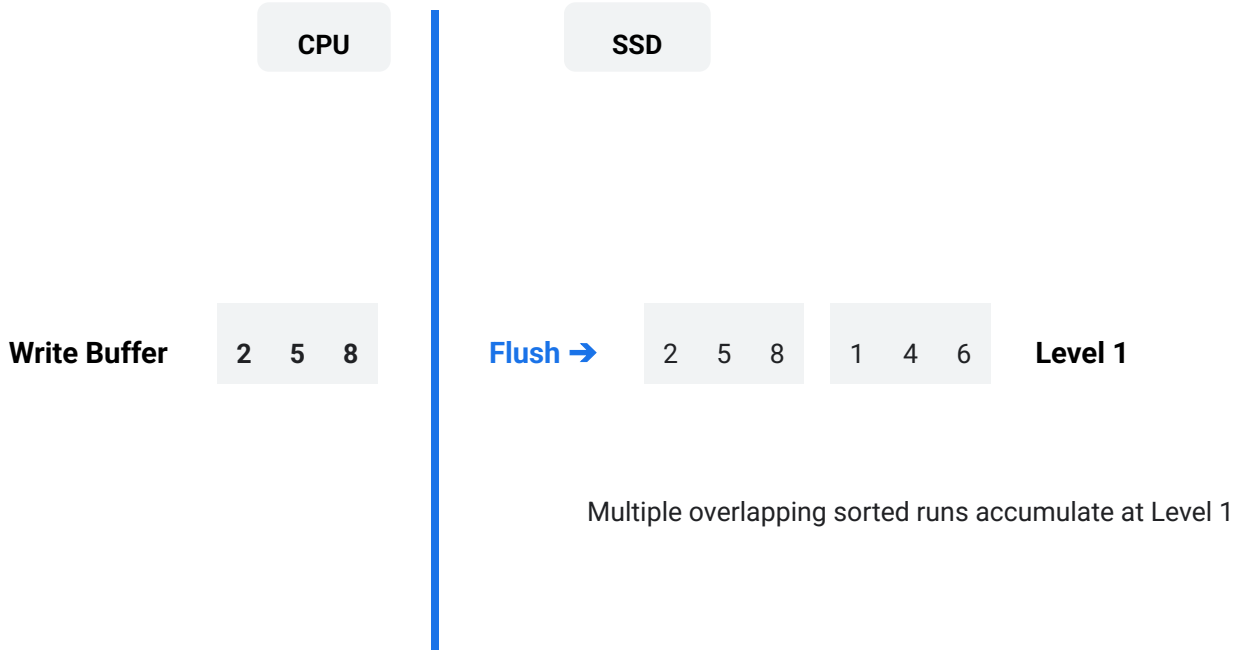


LSM Basics - Flush



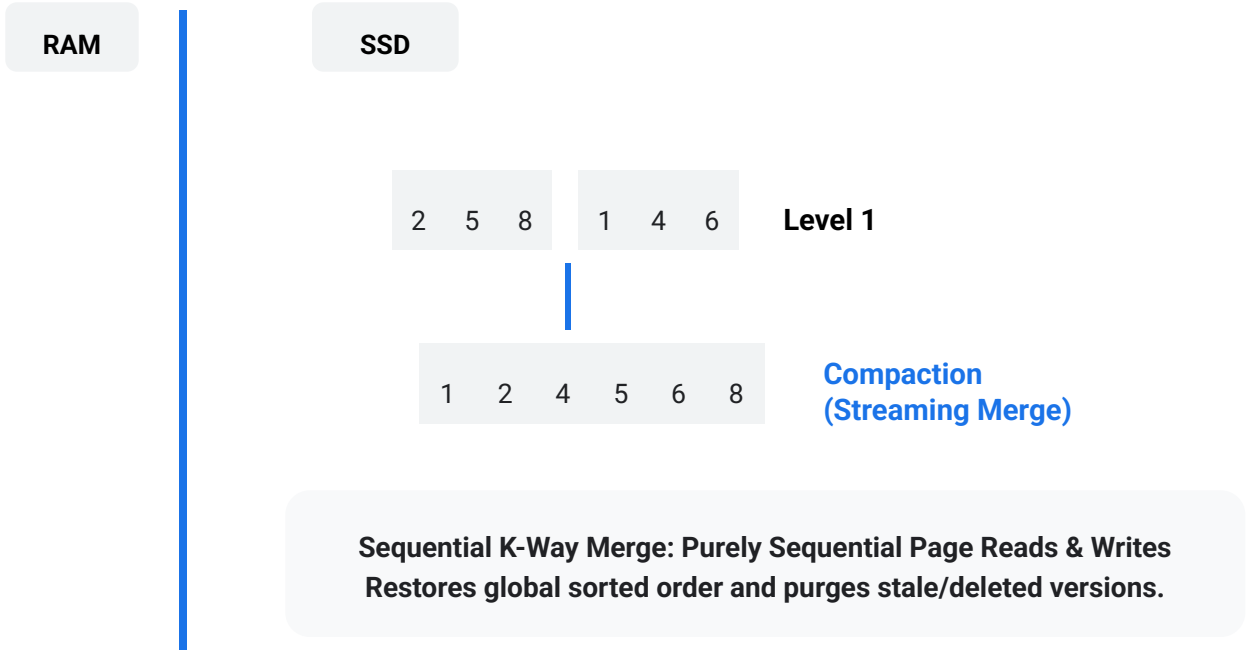
Flushing sorts keys in RAM and writes an immutable sorted SSTable to disk.

LSM Basics - Second Flush



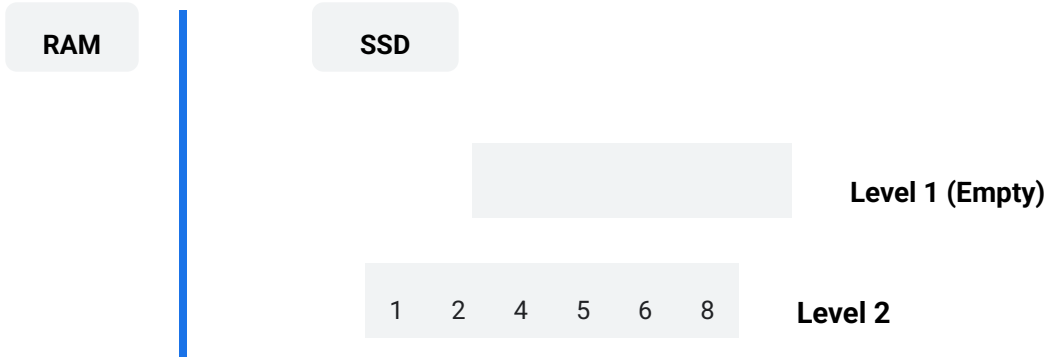
Each flush produces a new sorted run; overlapping runs trigger background compaction.

LSM Basics - Compaction



Compaction restores sorted order via streaming merge sort without random seeks.

LSM Basics - Level Overflow & Write Amplification



$$\text{Write Amplification (WA)} = \frac{\text{Total Bytes Written to Storage Media}}{\text{Bytes Ingested by Application}} \approx \frac{T}{2} \cdot L$$

$$\text{Amortized Write Cost (Page I/Os): } U = \frac{\text{WA}}{B} \approx \frac{T}{2} \cdot \frac{L}{B} \qquad L = \left\lceil \log_T \left(\frac{N}{B} \right) \right\rceil$$

each entry is rewritten $\approx T/2$ times per level across L levels (B entries share 1 page I/O)

Each key is rewritten ≈ (T/2)·L times across levels ➔ (T/2)·L / B page I/Os per write.

Compaction Strategy: Tiering

RAM

SSD

Tiering:

- **Write Cost: L / B I/Os (WA ≈ L)**
- **Point Read: up to T·L runs**
  ➔ Bloom: 1 + T·L·ε
- **Range Scan: T·L seeks + stream**
- **Space Amp: SA ≈ T**
  (T = 10 ➔ ~10× the live data)
- **Merges T overlapping runs at a time**

Level 1: T runs — each sorted, but key ranges overlap

Run 1:	2	6	9	14
Run 2:	1	6	11	15
Run 3:	3	5	9	12
Run 4:	2	7	10	16

Level 2: T runs — same key space, each ≈ T × more data

Run 1:	1	3	5	6	8	11	13	15
Run 2:	2	4	6	9	10	12	14	16
Run 3:	1	4	7	8	11	13	14	15
Run 4:	2	5	6	9	12	13	15	16

Tiering minimizes Write Cost (L / B page I/Os, WA ≈ L) at the cost of higher Read & Space Amplification.

The RUM Conjecture in Storage Engines

The RUM Tradeoff Matrix (Athanassoulis et al. EDBT 2016)

Cost measured in **page I/Os per operation** $L = \lceil \log_T(N/B) \rceil$, B = entries/page, ϵ = filter FPR

Access Metric	B-Tree (In-Place)	LSM Tiering	LSM Leveling
Point Read (R)	1	$1 + T \cdot L \cdot \epsilon$	$1 + L \cdot \epsilon$ [~ 1]
Update (U, amortized)	1 [WA $> 50\times$]	L/B [Lowest]	$\frac{T}{2} L/B$ [High]
Memory / Space (M)	$\sim 44\%$ [pages $\sim 69\%$ full]	$(T-1)\times$ [T=10 $\Rightarrow \sim 900\%$]	$1/T$ [$\approx 10\%$]

The RUM Conjecture:
Optimizing any two of Read (R), Update (U), and Memory (M) comes at the expense of the third — no single design achieves the best in all three.

On these three metrics alone, one design wins every column. Which design — and what did we leave out?

The RUM Conjecture in Storage Engines

The RUM Tradeoff Matrix (Athanassoulis et al. EDBT 2016)

Cost measured in **page I/Os per operation** $L = \lceil \log_T(N/B) \rceil$, B = entries/page, ϵ = filter FPR

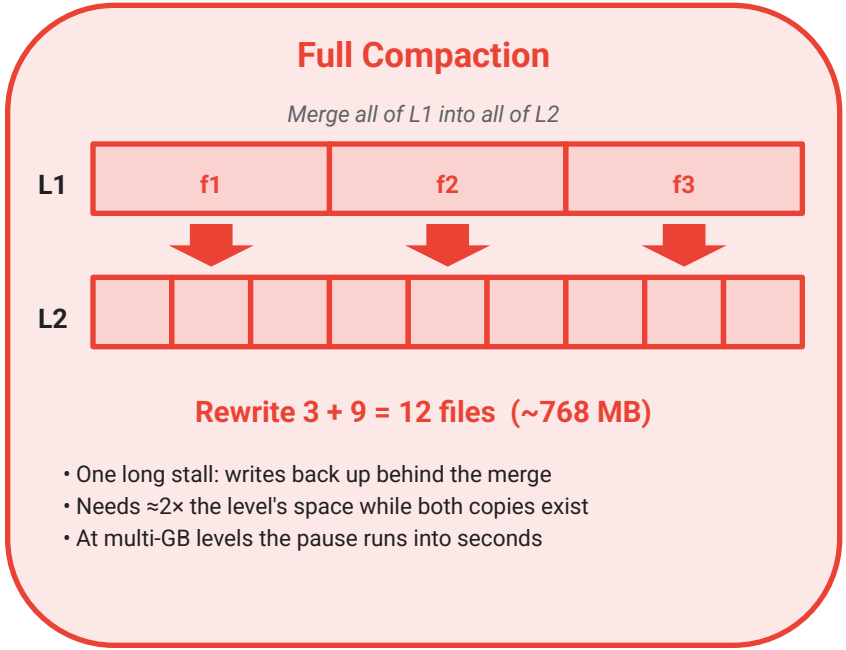
Access Metric	B-Tree (In-Place)	LSM Tiering	LSM Leveling
Point Read (R)	1	$1 + T \cdot L \cdot \epsilon$	$1 + L \cdot \epsilon$ [~ 1]
Range Scan (R)	1 seek + stream	$T \cdot L$ seeks + stream	L seeks + stream
Update (U, amortized)	1 [WA $> 50\times$]	L/B [Lowest]	$\frac{T}{2} L/B$ [High]
Memory / Space (M)	$\sim 44\%$ [pages $\sim 69\%$ full]	$(T-1)\times$ [T=10 $\Rightarrow \sim 900\%$]	$1/T$ [$\approx 10\%$]

If the workload is **point lookups only**, do not pay for ordered access: a hash-based engine (e.g. FASTER, SIGMOD '18) is the right choice.

The catch: R is read overhead over EVERY read you support — point AND range.
A hash index swept the first table only because the one row it loses was missing.
It answers a smaller question; it does not beat the conjecture.

You never escape the trilemma — you relocate it by changing which operations you promise to support.

Full vs. Partial Compaction (Leveling)

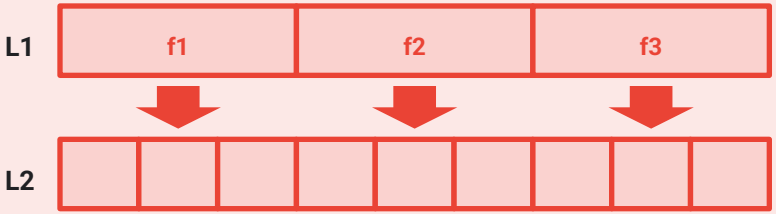


Naive full compaction rewrites entire levels at once — causing multi-second write stalls and doubling scratch space.

Full vs. Partial Compaction (Leveling)

Full Compaction

Merge all of L1 into all of L2

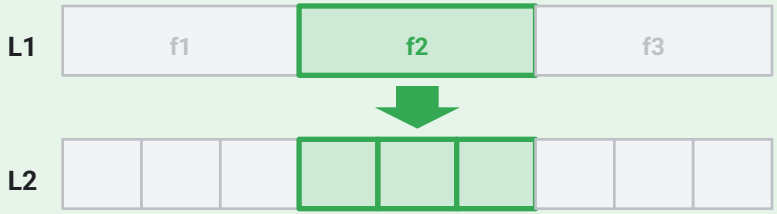


Rewrite 3 + 9 = 12 files (~768 MB)

- One long stall: writes back up behind the merge
- Needs ≈2× the level's space while both copies exist
- At multi-GB levels the pause runs into seconds

Partial Compaction

Merge ONE L1 file with the L2 files it overlaps



Rewrite 1 + 3 = 4 files (~256 MB)

- Bounded pause: one small merge, then back to serving
- Scratch space ≈ one compaction, not one whole level
- Free choice of WHICH file ➔ room to optimize (next slide)

Cells are 64 MB SSTables laid out across the key space; T = 3. LevelDB, RocksDB and Pebble all compact partially.

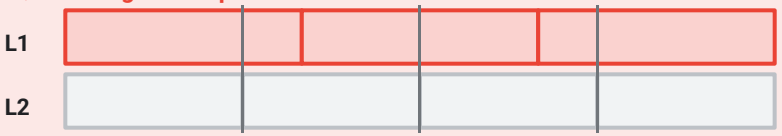
Same amortized write amplification — partial compaction replaces one huge stall with many small merges.

Partial Compaction: Where to Cut

1A · Naive Cut: Size Target Alone

Cut strictly at 64 MB target regardless of L2 boundaries

✗ Misaligned Output Boundaries



Every L1 file straddles 2 files below

- Fixed cuts ignore L2: every L1 file straddles 2 below.
- Doubles future merge I/O with cascading fan-out into L3.

1B · Aligned Cut: Stop at L2 Boundaries

Cut early whenever an L2 file boundary is crossed

✓ Boundary-Aligned Partitioning



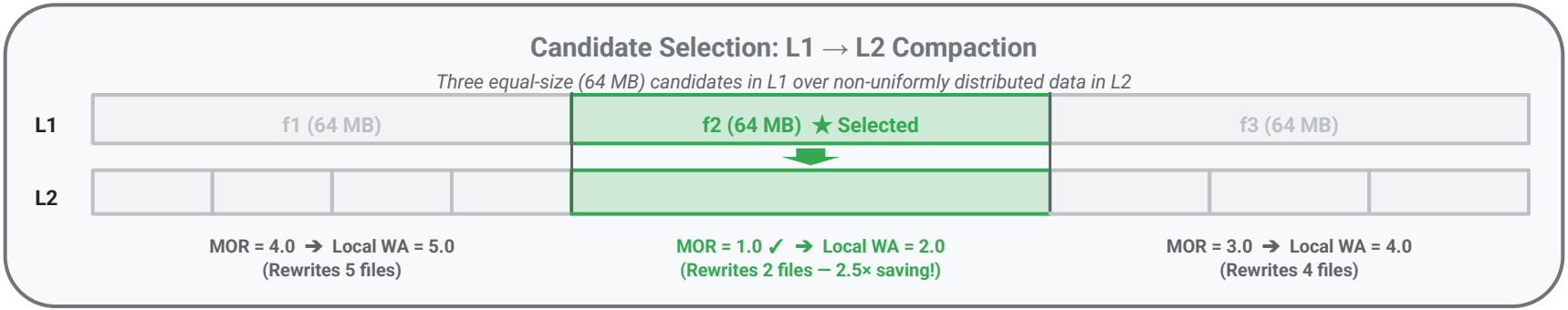
Clean 1:1 alignment with L2 file boundaries

- Cuts early at L2 boundaries: guarantees 1:1 alignment.
- Halves future compaction I/O with low metadata cost.

RocksDB CompactionOutputs::ShouldStopBefore cuts output files early once size + grandparent overlap exceeds max_compaction_bytes.

Aligning output files with the level below eliminates boundary straddling for future compactions.

Partial Compaction: Which File to Pick



Formal Definitions: WA & MOR

• Overlapped bytes in L(i+1) ÷ candidate file size in L(i):

$$\mathrm{MOR}(f) = \frac{\sum_{f' \in \mathrm{Overlap}(f, L_{i+1})} \mathrm{Size}(f')}{\mathrm{Size}(f)}$$

$$\text{Local WA}(f) = 1 + \mathrm{MOR}(f)$$

• **Rewrites 1 file in Li + MOR(f) files in Li+1**

★ **Minimizing MOR directly minimizes local write amp!**

Greedy File Selection Policy

• Density Skew: Equal-size files span unequal key ranges:

$$f^* = \arg\min_{f \in L_i} \mathrm{MOR}(f)$$

$$\Longrightarrow \min \mathrm{WA}_{\mathrm{step}} = 1 + \mathrm{MOR}(f^*)$$

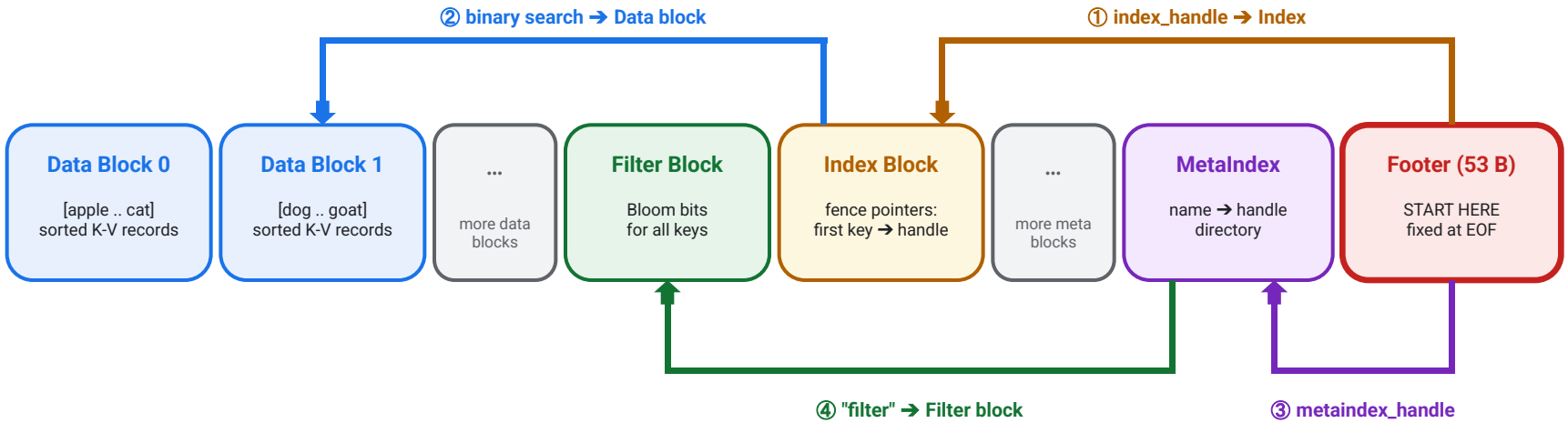
• Drains cheapest key ranges first (2.5× I/O saving)

• Production Default: RocksDB uses kMinOverlappingRatio.

Greedy MinOverlappingRatio minimizes write amplification by prioritizing the least-dense key ranges.

SSTable Anatomy: Modern File Layout

Physical layout on disk: offset 0 (left) ➔ EOF (right). The footer is the only block at a known position.



RocksDB BlockBasedTable layout; footer sits at fixed EOF.

Only the footer sits at a known offset — every other block is reached by following a handle.

Fence Pointers: In-Memory Block Indexing

RAM Fence Pointer Table

```
Block 1: [ 'apple'  .. 'cat'   ]
Block 2: [ 'dog'    .. 'goat'  ]
Block 3: [ 'horse'  .. 'lion'  ]
Block 4: [ 'monkey' .. 'zebra' ]
```

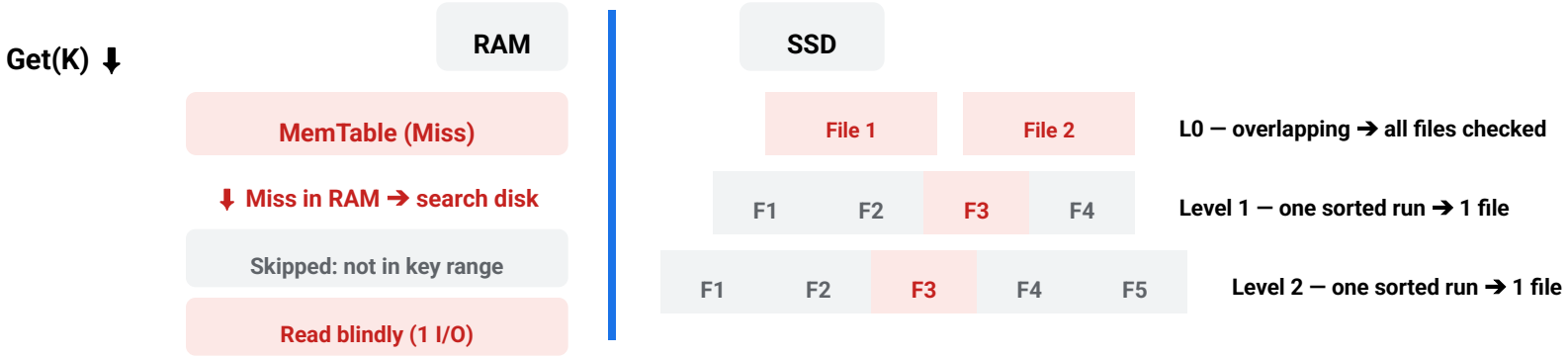
RAM Search (0 I/O)
➔ ➔
At Most 1 Disk I/O

SSD (4KB Blocks)

Block 1 (Offset 0x0000)
Block 2 (Offset 0x1000)
🎯 Block 3 (Read 4KB: ≤ 1 Disk I/O)
Block 4 (Offset 0x3000)

Fence pointers pinpoint candidate 4KB blocks in RAM (0 I/O) — bounding disk cost to at most 1 I/O per SSTable.

LSM Basics - Get(K) & Bloom Filter Math



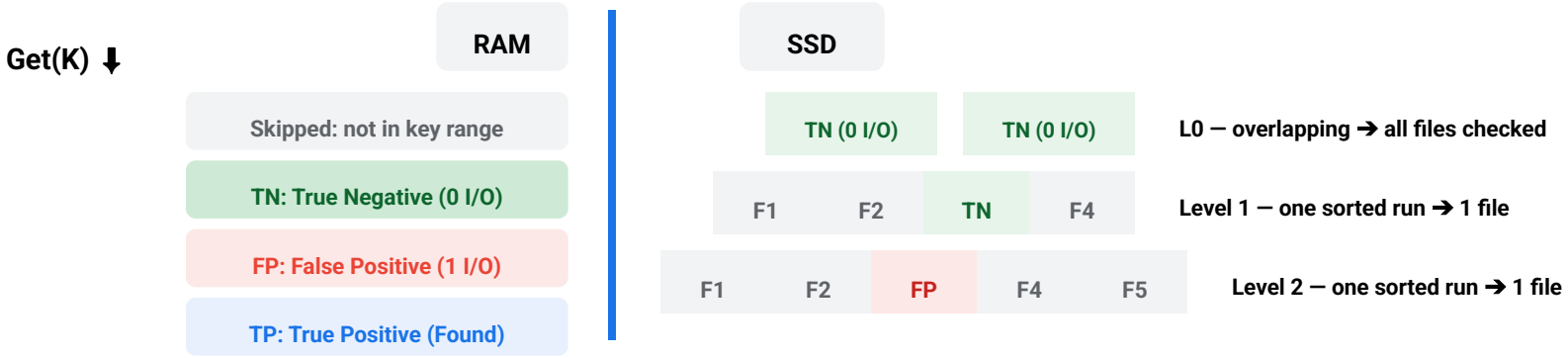
The Blind Point Read Penalty (No Bloom Filters):

- A sorted level is ONE run: file-level fences (min/max key per SSTable, already in RAM) leave at most ONE candidate.
- L0 is the exception — its runs overlap, so every L0 file stays a candidate.
- Without filters each survivor is read blindly: 2 + 1 + 1 = 4 random I/Os — cost scales as |L0| + L.

More advanced filters, such as Chucky and maplets, can be found in recent top conferences.

A sorted level is one run: fences pick ≤ 1 file; 10 bits/key ⇒ <1% FPR erases the rest.

LSM Basics - Get(K) & Bloom Filter Math



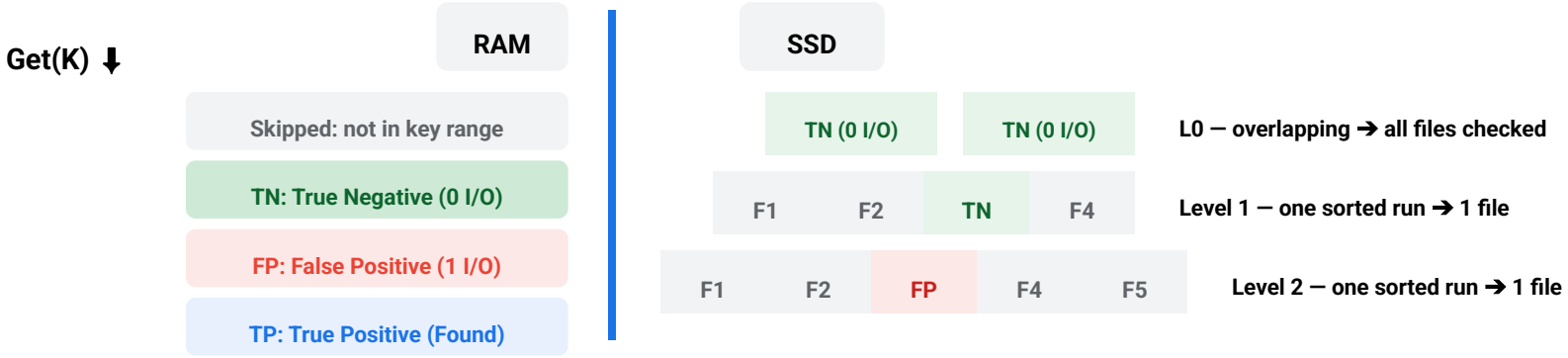
Bloom Filters (Pruning What the Fences Left):

- The filter is probed only for candidates the fences could not rule out — in RAM, k hashes, 0 disk I/O.
- True Negative (TN): filter returns 0 ➔ key is guaranteed absent ➔ the disk read is cancelled.
- False Positive (FP): a hash collision costs 1 wasted read. The 4 I/Os above collapse to just 1.

More advanced filters, such as Chucky and maplets, can be found in recent top conferences.

A sorted level is one run: fences pick ≤ 1 file; 10 bits/key ⇒ <1% FPR erases the rest.

LSM Basics - Get(K) & Bloom Filter Math



$$\text{False Positive Rate (FPR): } \epsilon = \left(1 - e^{-k \cdot n/m}\right)^k \xrightarrow{k=\frac{m}{n}\ln 2} \epsilon = e^{-b(\ln 2)^2} \approx 2^{-k}$$

$$\text{Bits-per-Key (bpk): } b = -\frac{\ln \epsilon}{(\ln 2)^2} \approx -1.44 \log_2 \epsilon \quad (10 \text{ bpk} \implies \epsilon \approx 0.82\%)$$

More advanced filters, such as Chucky and maplets, can be found in recent top conferences.

A sorted level is one run: fences pick ≤ 1 file; 10 bits/key ⇒ <1% FPR erases the rest.

The Flaw of Uniform Filter Allocation

Uniform Point Query Cost: $R = \sum_{j=1}^{L} \epsilon_j = 3 \times 0.01 = 0.03$ expected I/Os (or $1 + \sum_{j=1}^{L-1} \epsilon_j$ for lookup hits)

Exponential Data Skew: $N_j = T^{j-1} \cdot N_1 \implies L_{\max}$ holds $\approx 90\%$ of keys and consumes 90% of filter RAM

The Opportunity: L_1 holds only 0.1% of keys $\implies$ slashing ϵ_1 ($1\% \to 0.01\%$) costs ≈ 0 additional RAM!

L_max holds ~90% of the keys.

Upper-level filters can be 100× sharper for almost no extra RAM.

Spend filter RAM where it removes the most I/O — not evenly across levels.

Monkey: Optimal Filter Allocation (Best of SIGMOD '17)

RAM

Optimal Non-Uniform Sizing:

- ~90% of keys sit in the deepest level
- Shallow levels ➔ 14 bpk (ε = 0.1%)
- Deepest level ➔ 7 bpk (ε = 10%)
- Cost = Σ ε over levels ➔ minimize the sum

SSD

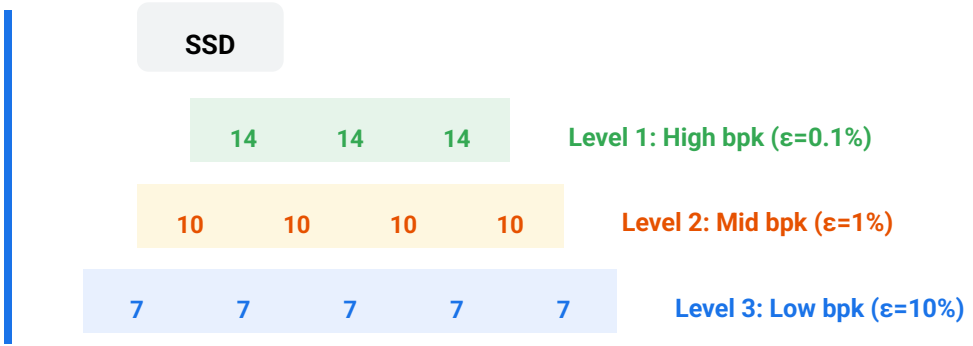


Key Insight: Spending bits on shallow levels buys exponential accuracy gains for almost zero RAM!

Monkey: Optimal Filter Allocation (Best of SIGMOD '17)

Optimal Non-Uniform Sizing:

- ~90% of keys sit in the deepest level
- Shallow levels ➔ 14 bpk (ε = 0.1%)
- Deepest level ➔ 7 bpk (ε = 10%)
- Cost = Σ ε over levels ➔ minimize the sum



$$\min_{\{\epsilon_j\}} \sum_{j=1}^{L} \epsilon_j \quad \text{s.t.} \quad \sum_{j=1}^{L} g(\epsilon_j, n_j) \le M,\ \epsilon_j \le 1 \implies \mathbf{b_j} = \max\left(\mathbf{0},\ \mathbf{b}_{\mathbf{L}_{\text{filt}}} + (\mathbf{L}_{\text{filt}} - \mathbf{j}) \log_{\mathbf{2}} \mathbf{T}\right) \quad (\epsilon_j \propto n_j)$$

$$\text{Query Cost:} \quad R_{\text{uniform}} = L \cdot \epsilon \propto \mathbf{L} \xrightarrow{\textbf{Monkey}} R^* = R_{\text{filt}} + R_{\text{unfilt}} \propto \mathbf{1} \text{ w.r.t. } \mathbf{L} \quad (R_{\text{unfilt}} \propto L_{\text{unfilt}})$$

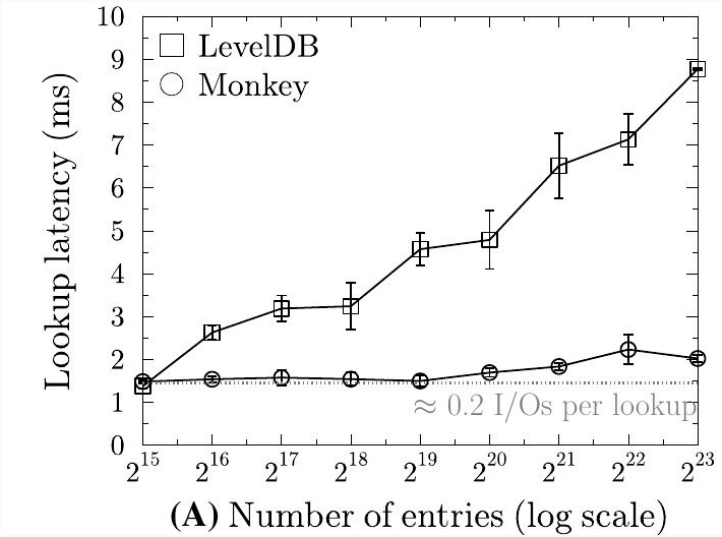
$$\text{Note: } L_{\text{unfilt}} = \left\lceil \log_T \left(\frac{m_{\text{thresh}}}{\text{bpk}} \right) \right\rceil \text{ depends only on bpk, not } L \ (L_{\text{unfilt}} = 0 \text{ when bpk } \ge 0.53)$$

Monkey shifts filter bits to shallow levels: worst-case lookup cost drops from O(L) to O(1)!

Monkey vs. Uniform: Empirical Proof & Practical Limits

Empirical Proof: O(1) vs. O(L)

Dayan et al., ACM SIGMOD 2017 (Fig. 11A)

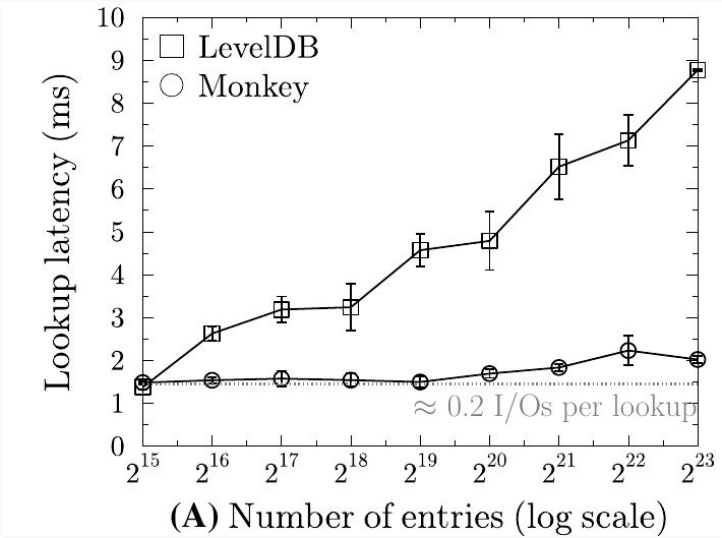


Empirical Proof: Monkey's lookup latency stays flat at ~1.5ms (O(1)), while LevelDB scales as O(L).

Monkey vs. Uniform: Empirical Proof & Practical Limits

Empirical Proof: O(1) vs. O(L)

Dayan et al., ACM SIGMOD 2017 (Fig. 11A)



The Experimental Workflow

Dayan et al., ACM SIGMOD 2017 (§7)

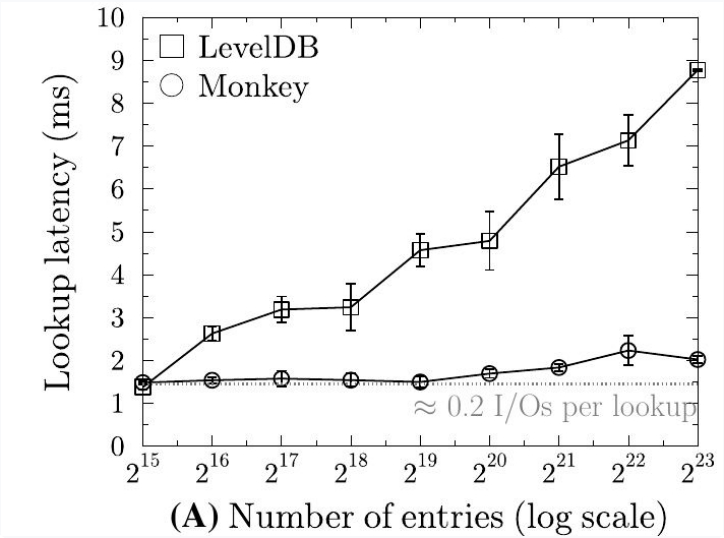
1	Ingest Baseline Tree	Uniform: 10 bpk
2	Run Query Benchmark	Scales O(L)
3	Offline Rebuild	Monkey: b_j
4	Replay Query Benchmark	Flat O(1)

The Experimental Workflow: Rebuilding and replaying queries validates O(1) vs. O(L) cost.

Monkey vs. Uniform: Empirical Proof & Practical Limits

Empirical Proof: O(1) vs. O(L)

Dayan et al., ACM SIGMOD 2017 (Fig. 11A)



? What About Practical Workloads?

Why static offline tuning breaks in 24/7 production systems

?	How to know total keys N?	Unknown Shape
?	What about mixed reads & writes?	Mixed Traffic
?	Can you freeze a 24/7 DB to rebuild?	Zero Downtime

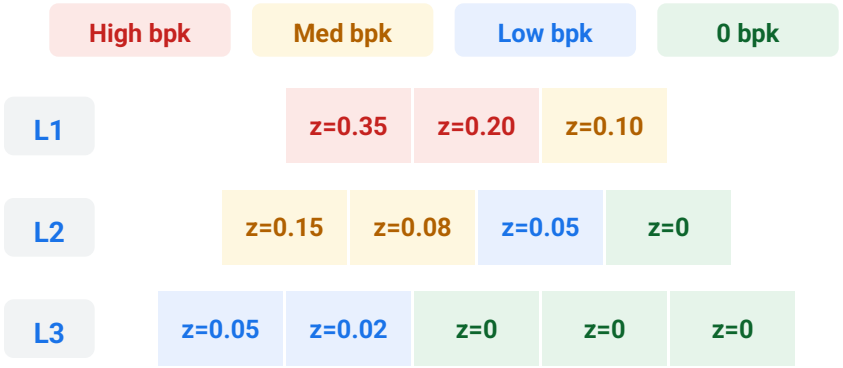
In production, N is unknown and traffic is mixed — motivating dynamic adaptation!

Mnemosyne: File-Wise Workload-Aware Filter Optimization (SIGMOD '25)

$$\min_{\{b_i\}} \sum_{i=1}^{F} z_i \cdot \epsilon_i \quad \text{s.t.} \quad \sum_{i=1}^{F} b_i \cdot n_i \leq M_{\text{budget}}$$

$$\text{Closed-Form: } \mathbf{b_i^*} \propto \log_2 \left(\frac{\mathbf{z_i}}{\mathbf{n_i}} \right)$$

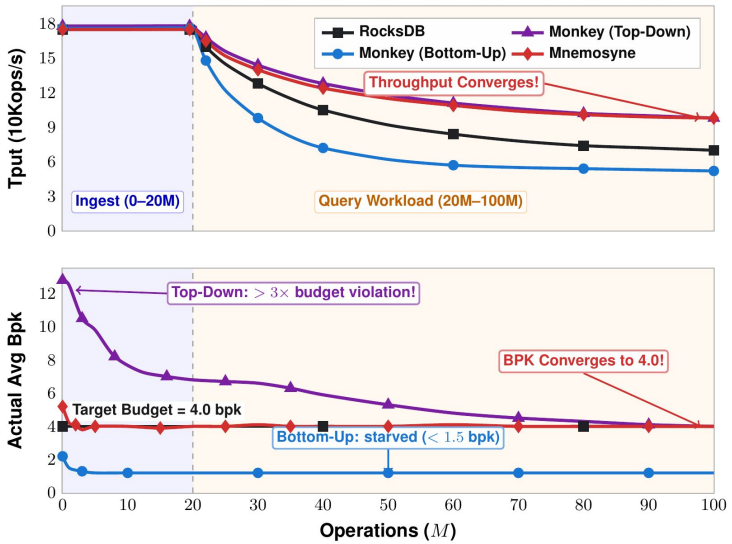
- **Shape-Agnostic: Leveling, Tiering, or hybrid**
- **Fast Solver: Nearly constant cost**



- **z_i: Zero-result query frequency per file**
- **Same z_i ➔ Same bpk across levels (z_i = 0 ➔ 0 bpk)**

File-wise optimization dynamically tunes filters based on each file's empty queries z_i.

Dynamic Filter Evaluation: Throughput & Memory



Operations (0–100M) • Uniform Workload (Target Budget = 4.0 bpk)

Static Allocation Dilemma

Top-Down exceeds budget (>3×); Bottom-Up starves deep filters.

Workload-Aware Budget Guarantee

Mnemosyne strictly enforces the 4.0 bpk target across all 100M ops.

Asymptotic Performance Convergence

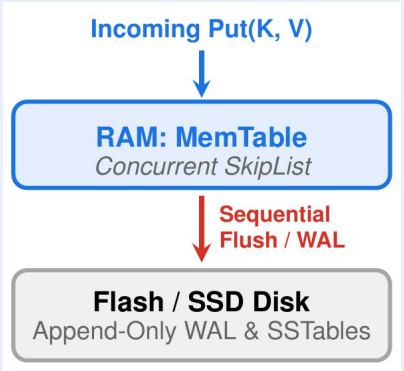
Mnemosyne matches Top-Down throughput with zero warmup delay.

Dynamic file-wise tuning guarantees strict memory budgets while achieving optimal throughput.

Summary & Key Takeaways

Sequential Updates

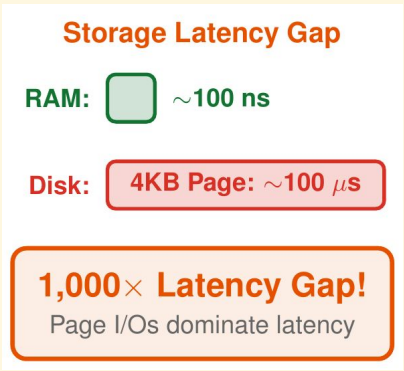
Physics Over In-Place



Random writes in RAM ➔ Streaming sequential I/O on Flash storage.

Beyond Big-O

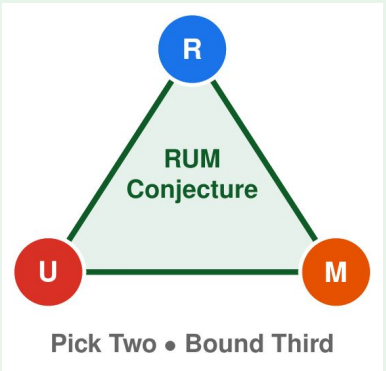
I/O Constants Dominate



1,000× latency gap: exact page I/Os dominate latency.

No Silver Bullet

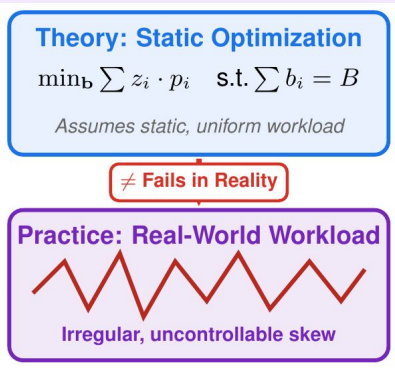
The RUM Conjecture



The RUM trade-off: optimizing reads sacrifices writes or RAM.

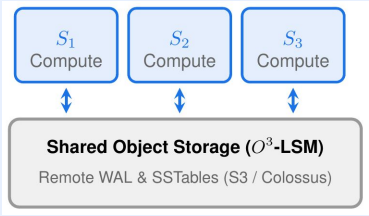
Theory vs. Practice

Dynamic Workload Reality



Ideal static math models fail under irregular, uncontrollable real workloads.

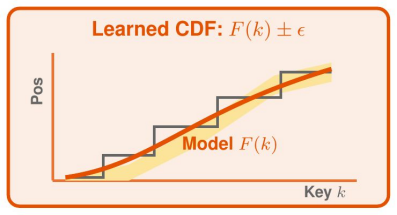
Open Research Frontiers



Disaggregated LSMs

Decoupled Cloud Storage

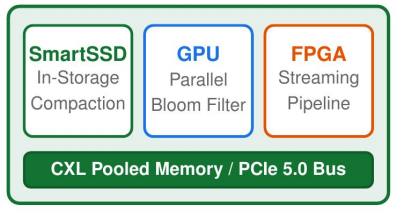
Compute tablet servers decoupled from shared cloud object storage (S3 / Colossus).



Learned Indexes & Filters

Neural CDFs & Data Models

Replacing static hash tables and fence pointers with neural regression CDF models.



Hardware Acceleration

Heterogeneous Compute

Offloading compaction and filtering to CXL memory, SmartSSDs, and FPGAs.