# CakeMask: Enabling Fresh-Data HTAP Databases with High Performance and Low Memory Overhead

Baoyue Yan, Jianying Wang, Hao Chen, Xinjun Yang, Feifei Li
*Alibaba Cloud Computing*

## Abstract

HTAP systems increasingly need to support analytics over fresh transactional data. In row-columnar HTAP architectures, this raises a visibility-management problem: analytical queries must observe the latest updates. Addressing this, existing designs struggle with a three-way tradeoff among transactional update throughput, analytical scan efficiency, and memory overhead.

This paper presents CakeMask, a visibility-management design for fresh-data HTAP. To balance performance and memory overhead, CakeMask preserves the bitmap as the compact, scan-efficient base visibility representation, while moving recent visibility changes into a delta visibility layer to avoid concurrent update overhead on the bitmap. It further combines an LSM-inspired delta structure, an optimized concurrent B+ tree with non-blocking scans, optimized bitmap–delta merging via vectorization, parallelism, and caching, and dynamic consolidation to bound long-term scan overhead. We implement CakeMask in the commercial HTAP database, PolarDB-IMCI, and evaluate it on diverse workloads. The results show that CakeMask delivers scan performance comparable to an in-memory version array at 1.6% of its memory usage, and outperforms existing on-disk visibility-management solutions by up to 79× in write throughput. CakeMask has been validated in production PolarDB-IMCI under various real-world workloads over three years.

## 1 Introduction

Database workloads have long been divided into online transaction processing (OLTP) and online analytical processing (OLAP). Increasingly, however, modern applications require both capabilities over the same data, motivating hybrid transactional/analytical processing (HTAP) systems [47] that unify them within a single system. This need is especially acute when analytics must reflect fresh transactional data. For example, fraud detection, inventory management, and real-time logistics all depend on timely access to the latest operational state, making the traditional ETL pipeline too stale to support timely decisions. Meanwhile, as a cloud HTAP provider, we also observe strong customer demand for such freshness.

Realizing this promise is difficult because OLTP and OLAP favor fundamentally different access patterns and are therefore typically supported by different storage layouts. Row stores are efficient for transactional updates, whereas column stores are efficient for analytical queries [6, 39]. As a result, many HTAP systems adopt a row-columnar architecture, in which a row store absorbs transactional updates while a columnar store serves analytical queries. However, this architecture immediately raises a fundamental question: how can analytical queries continuously observe the latest transactional mutations without undermining transactional performance?

Because rewriting columnar data in place on the transactional path is prohibitively expensive, practical systems handle updates out of place [5]: they record mutations separately from the base columnar data, invalidate old versions through deletion marks, and reconstruct the latest analytical view at read time. As a result, freshness becomes a visibility problem: how to expose the latest correct state to analytical queries while sustaining high transactional update throughput.

HTAP systems mainly differ in how they manage this visibility metadata. LSM-based columnar stores [18, 20, 21, 30, 32, 35, 37, 45] absorb mutations efficiently through append-only tombstones and delete-then-insert updates, but analytical reads must reconstruct visibility by merging records across multiple levels, which becomes increasingly expensive for large scans. Heap-based columnar stores [2, 19, 22–24, 27, 28, 34] instead separate visibility metadata from base columnar data, typically maintaining visibility through bitmaps and thereby avoiding cross-level merging. However, this shifts the burden to the visibility structure itself. Monolithic bitmaps are scan-friendly but expensive to update, because maintaining isolation often requires copy-on-write and introduces substantial overhead. To further remove bitmap write bottlenecks, some systems use in-memory version arrays [41], which maintain a per-row in-memory version or timestamp entry for visibility checking. This design provides strong read and write performance, but at prohibitive memory cost. Although these approaches differ in form, they expose the same underlying

tension: the visibility structure must simultaneously absorb frequent mutations, support scan-efficient reads, and remain memory-efficient.

Supporting fresh-data HTAP efficiently requires a visibility-management design that balances mutation absorption, scan efficiency, and memory overhead. This is difficult because mixed workloads introduce contention on recent visibility state and costly scan-time reconstruction. Ideally, we would like a monolithic bitmap for direct visibility checks, yet it cannot isolate concurrent writes. The version array offers natural isolation but at prohibitive memory cost to track per-row versions. The key insight is that tracking all row versions is unnecessary. If a deletion is visible to all future readers, a single bit suffices. Motivated by this, we propose CakeMask, a visibility-management design for fresh-data HTAP to break the three-way tension. CakeMask retains a monolithic bitmap as the stable visibility representation for its compactness and scan efficiency, while organizing not-yet-visible changes into a delta layer with per-row version information. Updates are appended to the delta layer and then merged into the base bitmap asynchronously. Readers consult the bitmap first, then merge the delta layer to determine the final visibility state. This decoupling avoids direct write contention on the bitmap, allowing transactional updates to proceed efficiently while keeping the bitmap scan-friendly.

However, the design choice introduces three challenges. First, the delta layer now lies on the critical path of both transactional writes and analytical reads: it must absorb high-rate writes from the row store while supporting efficient scans from the columnar store to reconstruct the latest visibility view. CakeMask organizes the delta layer as an LSM-tree-inspired structure with immutable on-disk runs and a mutable in-memory non-blocking B+ tree to isolate concurrent writes and large scans. Second, analytical reads must efficiently merge the base bitmap with relevant delta entries, and this cost can grow with the amount of recent update state. CakeMask accelerates this path through vectorization, parallelism, and fine-grained caching. Third, if left unmanaged, accumulated delta entries would gradually increase scan overhead over time. CakeMask employs dynamic consolidation to safely promote all-visible deletion marks off the delta layer into the base bitmap to bound the delta size.

Together, these techniques enable CakeMask to achieve high performance and low memory overhead. We implement CakeMask and integrate it into PolarDB-IMCI [41], a commercial cloud HTAP database. Our evaluation shows that CakeMask achieves comparable analytical performance to an in-memory version array at a small fraction of its memory footprint, while substantially improving transactional throughput over prior deletion-vector designs. We summarize our main contributions as follows:

- We systematically analyze the visibility-management design space for fresh-data HTAP systems, identifying the fundamental tradeoff among transactional performance, analytical performance, and memory efficiency.
- We propose CakeMask, a visibility-management design that decouples scan-efficient visibility representation from write-efficient mutation buffering through a compact base bitmap and a delta structure with efficient concurrency and consolidation support.
- Our evaluations show that CakeMask matches the scan performance of an in-memory version array at only 1.6% of its memory cost, while delivering higher transactional throughput and better scalability.
- We demonstrate the practicality of CakeMask through a three-year production deployment in a commercial database on the cloud.

## 2 Background and Motivation

In a dual-store HTAP architecture, the row store typically organizes data in a persistent B+ tree, where updates and point lookups operate on a small number of B+ tree pages. The columnar store, by contrast, packs each column into large, densely encoded data blocks optimized for analytical scans. While mutations in the row store are quickly processed in place, applying them to the columnar store is particularly challenging: it must reconcile high write throughput, low read overhead, and bounded memory consumption simultaneously. Based on these three axes, we categorize existing designs into four representative strategies.

**Columnar LSM-tree**. A widely adopted approach organizes the columnar store as an LSM-tree to absorb writes via fast, append-only ingestion with high data freshness. As shown in Figure 1 (a), all incoming writes are first appended to an in-memory memtable, which is subsequently frozen and flushed to secondary storage as immutable columnar data blocks. Deletions are represented as tombstone keys, and updates are decomposed into a deletion followed by an insertion. The LSM-tree periodically compacts these blocks by purging obsolete entries and sort-merging data across levels of exponentially increasing capacity. While append-only writes achieve high ingestion throughput, the merge becomes a potential bottleneck. The cost of this merge scales with both the number of levels and the size of the primary key, so large keys can further degrade scan performance. Our evaluation in §4.6 shows that scan performance drops by over 97% under write-intensive workloads, due to the increasing merge cost.

**Columnar Chunk Rewriting**. This approach is viable only in pure OLAP settings that forgo full ACID compliance [16], prohibit concurrent writes [15, 48], or disallow deletions and updates altogether [10, 44]. As illustrated in Figure 1 (b), the columnar store is a heap store that is vertically partitioned into row groups (RGs). Within each row group, each column is independently encoded and compressed into a contiguous chunk. Different from the columnar LSM-tree, the heap store is append-only without background merging. Insertions are buffered in an append-only memtable and periodically

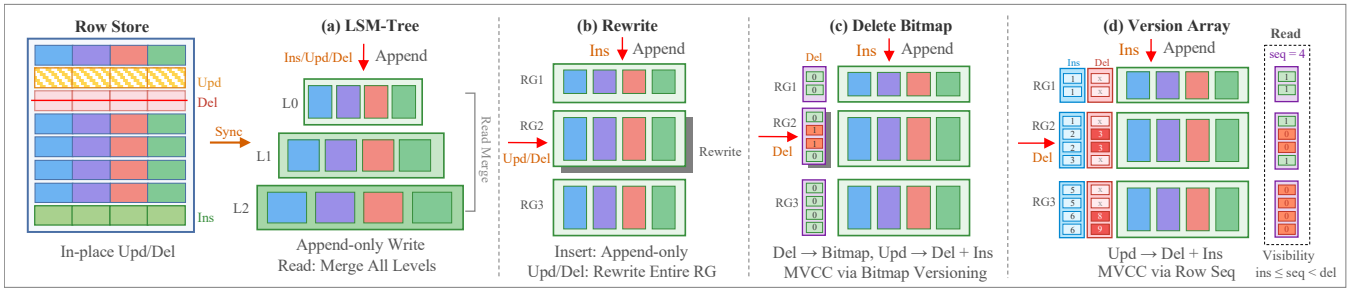



Figure 1: Existing strategies for synchronizing insertions and updates from the row store to the columnar store.

flushed to persistent storage as new row groups. Updates and deletions, however, require rewriting the affected row group entirely. Analytical queries can scan individual column chunks within a row group without cross-level merging, yielding excellent read performance. The fundamental drawback is severe read and write amplification for small mutations: a single-row update triggers a full row-group heavy rewrite. To amortize this cost, these systems typically batch mutations and apply them asynchronously, trading data freshness for throughput and incurring substantial data synchronization latency [4, 46]. This tradeoff is unacceptable for applications that require decisions based on current data.

**Delete Bitmap**. This is a form of deletion vector that decouples logical deletions from physical data rewriting by deferring reclamation to the background [2, 19, 22, 28, 34, 48]. As illustrated in Figure 1 (c), each row group maintains a per-row-group bitmap that records whether each row has been logically deleted. Insertions follow the same append-and-flush path as in chunk rewriting. For a deletion, the system first resolves the row offset of the target key—via the row store or a secondary index—and then sets the corresponding bit in the bitmap. Physically deleted rows are eventually reclaimed by a background process (e.g., `VACUUM` in PostgreSQL [38]). Updates are decomposed into a delete followed by an insert. To prevent read-write conflicts, bitmap updates are performed via copy-on-write. Analytical queries consult the bitmap to filter out deleted rows without any cross-level merging. The deletion vector exposes a fundamental tension governed by row group size: a large row group reduces the number of bitmaps a query must consult, but concentrates write contention; a small row group alleviates write contention but forces queries to traverse many small, randomly accessed bitmaps. It can be tuned to trade data freshness for costly synchronized writes or use asynchronous batch writes for higher write throughput.

**Version Array**. The version array traces row versions by memory-contiguous version numbers, as adopted in PolarDB-IMCI [41]. Because each row in a heap store carries a unique offset, the per-row bitmap degenerates into a scalar timestamp, yielding a per-row-group pair of arrays: an insert array and a delete array, each storing an 8-byte commit timestamp for MVCC isolation. As shown in Figure 1 (d), an insertion is first buffered in the memtable and its commit timestamp is later written into the insert array, which will be quickly purged after all rows become visible to future readers. A deletion resolves the row offset via the row store or a secondary index and writes the commit timestamp directly into the delete array. Deletions to distinct rows proceed concurrently without any locking. At query time, the reader obtains a read snapshot timestamp [43] and applies SIMD vector comparisons (e.g., AVX-512 [1]) over the in-memory arrays to construct the final visibility bitmap in a single pass. The version array thus achieves both high write throughput and scan performance with strong transactional consistency, but at the cost of a prohibitive memory footprint: a table with one billion rows may require up to 7.6 GB of memory for the deletion version arrays alone [41], two orders of magnitude larger than delete-bitmap-based solutions (seen in §4.5).

The four strategies above collectively span the design space and reveal a fundamental three-way tension among write throughput, read performance, and memory efficiency. The columnar LSM-tree favors write throughput but penalizes reads through expensive multi-level merging. Chunk rewriting delivers clean read performance but incurs high write latencies. The delete bitmap trades off write concurrency against scan efficiency as a function of row group granularity, yet cannot simultaneously optimize both. The version array resolves the read-write tension but demands memory resources that are impractical at scale. In this paper, we argue that these strategies need not be mutually exclusive: by selectively composing their strengths, it is possible to escape this trilemma. The design of CakeMask, described in the following section, instantiates this insight.

## 3 Design of CakeMask

Figure 2 illustrates the overall architecture of CakeMask. CakeMask employs three core components: a delta-tree (an LSM-tree structure) for deletion writes, an in-memory base bitmap for efficient scan processing, and version timestamps for MVCC-based concurrency control.

**Read and Write.** The transaction writes are first performed on the row store, then the deleted row ID and the newly inserted row are synchronously inserted into CakeMask and the columnar store, respectively. The columnar store is a heap store vertically partitioned by row groups. Given the physical

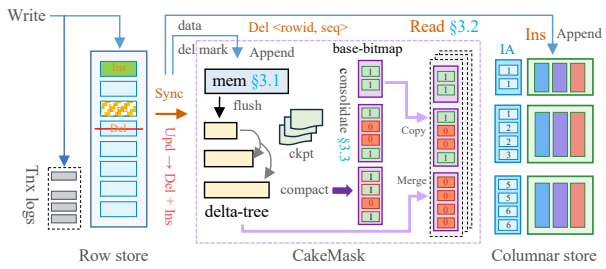



Figure 2: The high-level architecture of CakeMask.

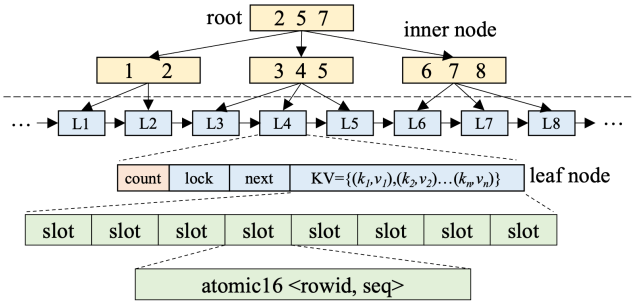



Figure 3: The page structure of the non-blocking-scan B+ tree

row ID $n$ and the row group size $k$, the row group is determined by $n/k$ and the row offset inside the group is $n\%k$. Insertions from the row store are appended into the columnar store as new immutable row groups, along with the commit timestamp into the insert version array (the IA in Figure 2), sharing the design in PolarDB-IMCI [41]. In CakeMask, updates are decomposed into a delete followed by an insert. For each deletion, the row ID is retrieved, and a compact 16-byte entry is appended to the delta-tree. A reader acquires a snapshot, copies the base bitmap into a thread-local one upon accessing a row group, and then merges the delta entries following the MVCC visibility rules. Deletion marks are retained until the corresponding row group is purged and the deleted rows are physically reclaimed in the background.

**Recovery.** Each component of the row store, CakeMask, and the columnar store has its own checkpoint. The foreground transactions always write persistent logs when they are committed to ensure durability. In case of crash recovery, the system checks the checkpoint and determines the start position in transaction logs, then scans the logs to recover the necessary in-memory data structures into a consistent state. Checkpoints are triggered by a configurable log size threshold to bound recovery time.

**Data Freshness.** CakeMask guarantees strong transactional consistency. The row store and the columnar store are synchronized within transactions—queries can always see the latest data. Additionally, with CakeMask, concurrent transactions are isolated from each other following the MVCC principle. The insertion sequence and the deletion sequence of each row are used to ensure that readers always see a consistent snapshot of the data.

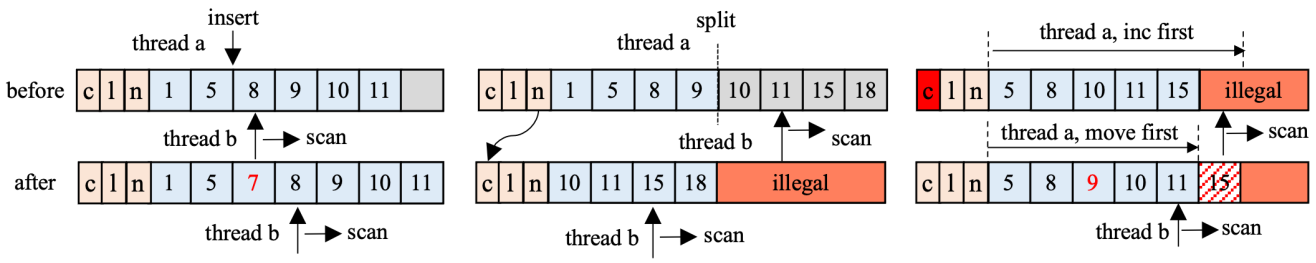
CakeMask pairs a delta LSM-tree for high-throughput append-only writes with a base bitmap for efficient scan processing. The memory-bounded memtable of the delta-tree caps peak memory consumption, while the base bitmap requires only one bit per row, making the overall memory footprint highly compact. Achieving this non-trivial combination calls for three key design contributions: (§3.1) a non-blocking-scan in-memory B+ tree that serves as the delta-tree memtable, eliminating lock contention between concurrent writes and long-running analytical scans; (§3.2) fine-grained, parallelized file merging within the delta-tree to minimize read cost for scanning queries; and (§3.3) dynamic consolidation, which progressively promotes deletion entries from the delta-tree into the base bitmap to bound delta-tree growth and sustain low scan overhead.

## 3.1 Non-blocking-scan In-memory B+ Tree

The B+ tree is widely recognized as the most efficient in-memory data structure for workloads that mix writes and range scans [42]. However, contention between concurrent scans and writes within a B+ tree presents a fundamental challenge for CakeMask's design. Concurrent access to an in-memory B+ tree is conventionally managed by the crab-locking protocol [25], which serializes all reads and writes to each leaf page behind a per-page lock. In a classical B+ tree, a scanner acquires the lock on a leaf page, scans its contents, and immediately releases it, keeping each lock short-lived. In the LSM-tree setting, however, each scan must merge data across all levels; when a scanner acquires the lock on a memtable leaf page, the page's contents may not be consumed until much later during the merge, causing the lock to be held for an extended duration and impeding concurrent writes. Since B+ tree scans access only leaf pages, and point lookups and insertions complete in negligible time, we confine the non-blocking design to leaf-page scans exclusively. Achieving this requires addressing two key challenges: (1) transaction isolation, and (2) data consistency under concurrent leaf-page modifications.

CakeMask follows the MVCC principle for transaction isolation: only committed data is written to the memtable, and each entry carries a unique commit sequence number as its visibility timestamp. With non-blocking scans, a reader and a concurrent writer may race on the same slot inside a leaf page; if each slot is accessed atomically, however, they are naturally serialized at slot granularity. The non-blocking-scan B+ tree therefore introduces the *atomic-16 slot* as the fundamental read and write unit, relying on hardware-guaranteed 16-byte aligned atomicity under the TSO memory model [14, 31]. Figure 3 illustrates the leaf page layout: three metadata fields govern concurrency control and sibling linking, followed by a contiguous slots array. The page lock serializes only writes and point lookups; scanners bypass it entirely, accessing each slot exclusively via SSE instructions.

To address the second challenge, we note that insertions may trigger slot shifts or page SMOs (Split-Merge Opera-




(a) data duplication in move (b) data duplication in split (c) illegal read

Figure 4: The race cases between leaf insertions and scans without lock protection.

**Algorithm 1:** Non-blocking leaf scan to avoid duplication, where *ls*:local slot, *slot_*:previous slot, *seq_*:read snapshot.

```
1  Function next():
2      while ls ← next_slot() ≠ none do
3          if ls.key ≤ slot_.key ∨ ls.seq > seq_ then
4              continue
5          end
6          slot_ ← ls
7          break
8      end
9  end
10 Function next_slot():
11     cursor_++
12     if cursor_ ≥ curr_leaf_.count then
13         curr_leaf_ ← curr_leaf_.right_sibling
14         cursor_ ← 0
15         if curr_leaf_ = nullptr then
16             return none
17         end
18     end
19     return curr_leaf_.slots[cursor_]
20 end
```

**Algorithm 2:** The multi-phase move protocol

```
1  Function insert_leaf(leaf, slot, pos):
       // leaf.count is atomic-8
2      cnt ← leaf.count
3      mv_cnt ← cnt − pos
4      leaf.slots_[cnt] ← leaf.slots_[cnt − 1]
5      leaf.count ← leaf.count + 1
6      for i ← mv_cnt − 2; i ≤ 0; i ← i − 1 do
7          leaf.slots_[pos + 1 + i] ← leaf.slots[pos + i]
8      end
9      leaf.slots_[pos] ← slot
10 end
11 Function split_leaf(leaf, new_leaf):
12     cnt ← N/2 + 1
13     new_cnt ← N − cnt
14     for i ← 0; i < new_cnt; i++ do
15         new_leaf.slots_[i] ← leaf.slots[cnt + 1]
16     end
17     new_leaf.count ← new_cnt
18     new_leaf.right_sibling ← leaf.right_sibling
19     leaf.right_sibling ← new_leaf
       // should be set in the last step
20     leaf.count ← cnt
21     return leaf.slots[cnt].key
22 end
```

tions), during which a scanner may observe transiently inconsistent data. Because the delta-tree is append-only, page merges never occur and only the split case must be handled. Figure 4 identifies three distinct race scenarios that arise when an insertion and a scan execute concurrently on the same leaf page, summarized as follows.

**Duplicate Reads During Slot Movement.** As shown in Figure 4 (a), thread *a* inserts key 7 between keys 5 and 8, shifting entries rightward. Concurrently, thread *b* has just read key 8; before *b* advances, *a* writes 7 and shifts 8 forward, causing *b* to re-encounter key 8 as a duplicate.

**Duplicate Reads During Page Split.** As shown in Figure 4 (b), thread *a* inserts key 7 into a full page, triggering a split that copies the right half to a new leaf. If *b* has passed key 11 and *a* completes the split before *b* follows the sibling pointer, *b* re-scans the relocated entries on the new leaf, again producing duplicates.

**Corrupt or Missing Reads During Slot Movement.** As shown in Figure 4 (c), thread *a* inserts key 9, requiring a two-phase rightward shift: phase 1 increments the slot-count metadata; phase 2 relocates the data. If *b* reads key 15 between the two phases, it sees a count of 6 and dereferences an uninitialized slot, yielding corruption. If instead phase 2 precedes phase 1, *b* sees only 5 slots and misses key 15 entirely.

In Figure 4 (a) and Figure 4 (b), a scanner may see duplicates but will never miss entries. Since leaf entries are totally ordered, any entry whose key does not strictly exceed the most recently accepted key is a duplicate and is discarded. Algorithm 1 jointly implements this de-duplication and snapshot isolation: the iterator accepts slot *ls* only if *ls.key* exceeds

the previous key and $ls.seq \leq seq_$ (lines 3–6); *next_slot*() advances the cursor, following the right-sibling pointer when the leaf is exhausted (lines 12–18).

Algorithm 1 cannot resolve Figure 4 (c), because incrementing the leaf's slot counter and relocating the slot data are separate non-atomic steps. We propose the *multi-phase movement protocol* in Algorithm 2, comprising a safe in-page movement and a safe page split. For in-page movement, the algorithm first copies the last slot to the extended position (line 4), invisible to scanners, then atomically increments the leaf counter (line 5). Any transient duplicate observed between lines 5–8 is discarded by Algorithm 1; after line 9, the page is fully consistent. The safe page split follows the same principle: the new leaf remains unreachable via the sibling chain until the split finalizes. The algorithm copies the upper half of entries to the new leaf (lines 12–16) and initializes its count and sibling pointer (lines 17–18). The critical invariant is that line 19—installing the new leaf into the sibling chain—must execute *before* line 20, which reduces the old leaf's count. Reversing this order would cause a scanner to follow the old sibling pointer past the new leaf, missing all relocated entries. Enforcing this ordering ensures every scanner discovers the new sibling before observing the reduced count.

## 3.2 Fine-grained File Merge

When a scan touches a row group, it determines the row offset range, seeks the corresponding range in the delta-tree, and performs a multi-level merge across all participating files—an operation that is typically CPU-intensive. To improve merge performance, we propose the following four optimizations.

**Vectorization.** Vectorization amortizes the overhead of virtual function calls across the merge pipeline. An LSM-tree scan spans multiple sorted levels—both memtable and file levels—each exposed through a uniform abstract iterator interface backed by virtual dispatch. When the merge loop invokes iterators millions of times per second, virtual dispatch becomes the dominant CPU bottleneck; we observe that it accounts for over 50% of CPU time at such rates. Analogous to the vectorized execution model over the Volcano model [9, 13], we adopt a batch iterator that returns one batch of entries per call, amortizing the virtual dispatch cost across the entire batch.

**Multi-level Cache.** During the merge, the merger fetches data from each iterator in batches and merges them into a globally sorted stream. Each file iterator maintains a prefetch buffer to hide sequential I/O latency. However, even with batch fetching, the merge loop remains suboptimal for two reasons. (1) A k-way merge selects the minimum value from one way per cycle and immediately requires the next value from that way; this per-element access pattern conflicts with the batch-fetch granularity of each iterator. (2) Each iterator maintains its own independent data buffer at a distinct memory location; the merger must access these buffers in an unpredictable order each cycle, resulting in poor cache locality. To address both issues, we introduce the multi-level cache hierarchy shown in Figure 5. Each file iterator is paired with a prefetch buffer; across all iterators, the merger maintains a single physically contiguous memory region that caches one batch from each iterator (the mini-batch in Figure 5). The mini-batch decouples value selection from iterator-level data fetching: the merger always reads from the contiguous mini-batch buffer, eliminating random cross-iterator memory accesses and enabling the batch iterator to fully amortize virtual dispatch overhead.

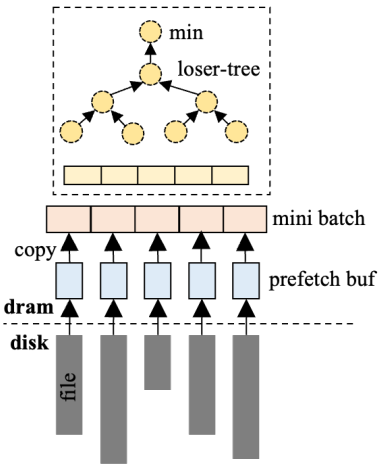



Figure 5: The file merge hierarchy with multi-level cache.

**Loser-tree for K-Way Merge.** K-way merge is commonly implemented with either a tournament tree (winner-tree or loser-tree) [11] or a binary heap. We find the loser-tree to be 1.5–2× faster than the heap-based approach in our scenario. In the heap, each adjustment requires the root to be compared against both its left and right children—two comparisons per cycle. In the loser-tree, adjustment proceeds bottom-up, with each node compared only against its parent—one comparison per cycle. The loser-tree is a complete binary tree that can be stored in a flat array. To maximize cache efficiency, we allocate this array adjacent to the mini-batch buffer in Figure 5, so that each merge cycle operates entirely within a small, physically contiguous memory region, minimizing cache misses.

**Parallelized Merge.** Parallelizing the merge across multiple CPU cores is a natural strategy to further accelerate merge throughput. In the LSM-tree, frozen memtables and on-disk files are immutable, so their scan performance scales linearly with CPU count up to the memory and I/O bandwidth limit. For the active memtable, the non-blocking-scan B+ tree (Section 3.1) enables concurrent scans to scale without interference from ongoing insertions. In the parallel vectorized execution model, the executor partitions a table at row-group granularity, with each row group processed by an independent thread. Visibility bitmap generation for each row group is therefore naturally parallelized.

## 3.3 Dynamic Consolidation

Dynamic consolidation progressively promotes deletion marks from the delta-tree into the base bitmap to reduce

merge cost, but it must address two fundamental challenges.

1. **Durability of purged entries.** Deletion marks cannot be directly discarded from the delta-tree during promotion, as they constitute durable data. Any purged entries must be preserved elsewhere to allow the system to recover from a consistent checkpoint.
2. **Visibility consistency under concurrency.** When deletion marks are removed from the delta-tree and merged into the base bitmap in place, concurrent readers may access the base bitmap mid-transition. If entries are removed from the delta-tree before being reflected in the base bitmap, a reader may observe an inconsistent state in which committed deletions appear to be missing.

As illustrated in Figure 6, the key insight addressing the first challenge is to partition the delta-tree into a *runtime-tree* and a *checkpoint-tree*. Both are LSM-tree structures. The runtime-tree absorbs all foreground deletions and is discarded upon restart; the checkpoint-tree is maintained solely for crash recovery. The runtime-tree promotes suitable deletion marks into the base bitmap. The checkpoint-tree periodically scans and persists delta deletion marks from the runtime-tree to ensure a clean recovery point. Without the checkpoint-tree, CakeMask must log each write and perform a full log scan to construct the volatile base bitmap at startup, which incurs a large recovery time. Also, writing extra logs in the foreground would cause a 30%–50% write performance drop. The checkpoint-tree may introduce some write amplification, but the footprint is negligible. Across many production instances from our observation, the heap store averages one fifth of total storage, and the checkpoint-tree uses $<5\%$ of the heap store—less than 1% of total storage.

This architecture also decouples checkpoint progress from consolidation progress: because consolidation is gated on the minimal read snapshot, which can lag far behind the latest committed transaction (e.g., when blocked by a long-running query), the checkpoint-tree allows durable progress to track the commit frontier independently.

Formally, consolidation promotes the deletion mark set $C = \{(rowid, seq) \mid seq \leq seq_{min}\}$, where $seq_{min}$ is the current minimal read snapshot number, from the delta-tree into the base bitmap, while the checkpoint cycle durably records $D = \{(rowid, seq) \mid seq \leq seq_c\}$, where $seq_c$ is the most recent commit sequence at the time of the checkpoint. To ensure that every promoted entry has already been durably recorded, consolidation must not advance beyond the latest checkpoint. This invariant is enforced by setting the consolidation sequence guard to $seq_g = \min(seq_{min}, seq_c)$.

Addressing the second challenge, note that with the checkpoint-tree for durability, the base bitmap reduces to a single-version, in-memory object without a persistence requirement. It is accessed concurrently by foreground readers and the background mover. Naively requiring mutual exclusion between them would serialize reads against background

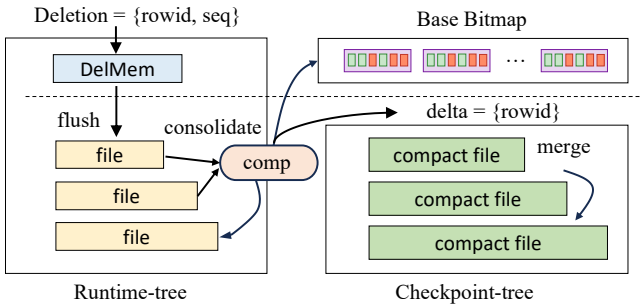



Figure 6: The workflow of dynamic consolidation.

consolidation, unacceptably degrading read throughput and increasing write latency. Beyond the race condition, two additional correctness concerns arise. First, snapshot isolation requires that the reader must not see future deletions. Second, a subtle missing-read hazard exists: if (1) a reader copies the base bitmap, then (2) the mover promotes entries into the base bitmap, and finally (3) the reader merges the delta-tree, the reader's local bitmap copy will not reflect the promoted entries, causing deletions to appear as live rows. Dynamic consolidation resolves these concerns through the following mechanisms.

**Concurrent Race for Base Bitmap.** This is a classic single-writer/multiple-reader scenario in which a concurrent reader may observe a torn write. However, since each bit-reset in the bitmap modifies exactly one machine word, the MESI cache coherence protocol guarantees that the write to a cache line is atomic at the hardware level. Readers therefore observe either the pre-update or post-update word, with no possibility of a partial read.

**Snapshot Isolation.** The mover promotes only deletion entries $\{(rowid_i, seq_i) \mid seq_i \leq seq_g\}$, and any reader $j$ holds a snapshot $seq_j \geq seq_{min}$. Since $seq_g \leq seq_{min}$, it follows that $seq_i \leq seq_j$ for all promoted entries. Therefore, every deletion reflected in the base bitmap is visible under reader $j$'s snapshot, and no future deletion can be exposed through the base bitmap.

**Missing Reads.** The missing-read hazard is eliminated by embedding the mover directly into the compaction of the runtime-tree, which is already protected by the LSM-tree's atomic meta-snapshot mechanism. A reader acquires a meta-snapshot that enumerates the exact set of files to scan, and then merges those files with the base bitmap to construct its visibility bitmap. When compaction selects a set of files, it produces a new meta-snapshot atomically, where deletion marks with $seq_i > seq_g$ are retained in the compacted output file, while the remaining marks are promoted into the base bitmap. During the compaction window, a deletion mark may transiently reside in both the runtime-tree and the base bitmap; a reader may therefore observe a deletion twice, but will never miss one. Because the bit-reset operation is idempotent, such transient duplication is harmless and self-correcting. Once the new meta-snapshot is installed, all deletion marks from

the compacted files are accounted for either in the new output file or in the base bitmap, restoring a clean invariant.

# 4 Evaluation

We begin with micro-benchmarks (§4.1 - §4.3) for each core design component presented in §3, assessing its individual performance characteristics in isolation. We then conduct a comparative study against representative deletion vectors to quantify the end-to-end benefits of our proposal (§4.4). Next, we implement CakeMask in IMCI [41], named IMCI-CM, and compare it against competing production databases under industry-standard transactional benchmarks, Sysbench and TPC-C [26], to demonstrate its effectiveness under standard workloads (§4.5 - §4.6).

**Micro-benchmarks (§4.1 - §4.4).** All micro-benchmarks are conducted on a server equipped with an Intel® Xeon® Platinum 8163 CPU (96 logical cores, 32 MB L3 cache), 512 GB DDR4 DRAM and a 4 TB Intel P4500 NVMe SSD formatted with ext4 (journaling enabled), running 64-bit CentOS with a Linux v4.9 kernel.

**Production-benchmarks (§4.5 - §4.6).** CakeMask is deployed in PolarDB-IMCI, available on Alibaba Cloud, replacing the version array of its columnar index. For the production database comparisons, all instances are provisioned from their respective cloud vendors with identical CPU core counts and memory capacity, i.e., 16 CPUs with 64 GB DRAM.

## 4.1 Evaluation of Non-blocking-scan B+ Tree

**Setup.** We benchmark the non-blocking-scan B+ tree (4 KB page size) against a lock-free concurrent SkipList [33] (from the Folly library). We evaluate two workloads. First, we assess insertion scalability by inserting 10 million deletion entries with thread counts ranging from 1 to 80, where keys are drawn from $[1, 10^7]$. Second, we fix 16 insertion threads and increase scan threads from 1 to 80, where each scan issues a seek and traverses a consecutive range of 10 to 100. All experiments are repeated under both uniform and Zipfian key distributions ($\theta = 0.99$). To isolate the benefit of our non-blocking scan design, we additionally implement a lock-based B+ tree variant as a controlled baseline, in which a scan acquires a lock on each leaf page upon access and immediately copies the page contents locally to minimize write blocking.

**Result.** Figure 7 (a) plots insertion throughput as a function of thread count. Under a uniform key distribution, our B+ tree achieves near-linear scaling from 1 to 80 threads, outperforming the SkipList by 2.6× at 80 threads. This gap stems from two structural advantages: (1) page-level memory allocation amortizes per-entry overhead across all slots, and (2) the contiguous page layout yields superior cache locality compared to the pointer-chasing traversal in SkipList nodes. Under the Zipfian distribution, both throughput and scalability degrade, as expected: skewed keys concentrate writes on a small set of hot leaf pages, serializing concurrent inserters.

Figure 7 (b) reports the mixed scan-insert workload under a uniform key distribution, with the left y-axis showing insertion throughput and the right y-axis showing seek-scan throughput. As scan threads increase, both B+ tree variants exhibit a gradual decline in insertion throughput due to cache and memory-bandwidth contention; the SkipList degrades similarly. The non-blocking-scan B+ tree sustains higher insertion throughput than the lock-based B+ tree, since it eliminates lock acquisition on the scan path entirely. For seek-scan throughput, the non-blocking-scan B+ tree plateaus at 35.3M/s at 80 scan threads, outperforming the lock-based B+ tree and the SkipList by 2.1× and 6.1×, respectively.

Figure 7 (c) presents results under the Zipfian key distribution. Up to 32 scan threads, both the B+ tree and the SkipList exhibit a gradual decline in insertion throughput, consistent with the uniform case. Beyond 32 scan threads, however, the B+ tree's insertion throughput begins to increase, while scan throughput grows superlinearly. The explanation is that under a Zipfian distribution, a small fraction of hot keys dominates both insert and scan traffic; as more scan threads access these hot keys, the corresponding leaf pages are repeatedly loaded into the L1/L2 caches and remain resident. At 80 scan threads, the non-blocking-scan B+ tree outperforms the lock-based B+ tree and the SkipList by up to 3.1× and 8.9×, respectively, since it scans pages lock-free with better cache locality.

## 4.2 Evaluation of File Merge

**Setup.** We generate 1 billion deletion pairs with sequence number zero, partitioned into $k$ fully interleaved sorted runs with $k \in [1, 32]$, each materialized as a separate on-disk file. To isolate CPU cost from I/O, we set the prefetch buffer equal to the file size so that all data bypasses the file I/O and the OS page cache during the merge. To quantify the contribution of each design decision described in §3.2, we evaluate three configurations: (1) heap-base, a naïve binary heap merge with no vectorization and no mini-batch buffer, serving as the baseline; (2) loser-single, a loser-tree merge with mini-batch size 1 and no vectorization; and (3) loser-batch, the full design with a loser tree and mini-batch size 32. All configurations are measured single-threaded to isolate merge throughput from parallelism effects, as merge performance can be easily scaled with multiple threads over immutable files.

**Result.** Figure 8 reports merge throughput as a function of the number of sorted runs. With a single run, the merge path is bypassed entirely and throughput reflects pure sequential iteration. Even in this degenerate case, loser-single outperforms heap-base by 1.53×, owing to reduced virtual dispatch overhead: each iterator step in heap-base requires three virtual calls—$valid()$, $data()$, and $next()$—whereas in loser-single, a single $next()$ call buffers one entry into the mini-batch slot, reducing the per-record hot path to two virtual calls. As the run count increases to 32, this advantage compounds to 4.92×: the heap must issue two virtual calls per pairwise comparison at every merge step, whereas the loser tree requires only one comparison per tree advancement, and the mini-batch slot

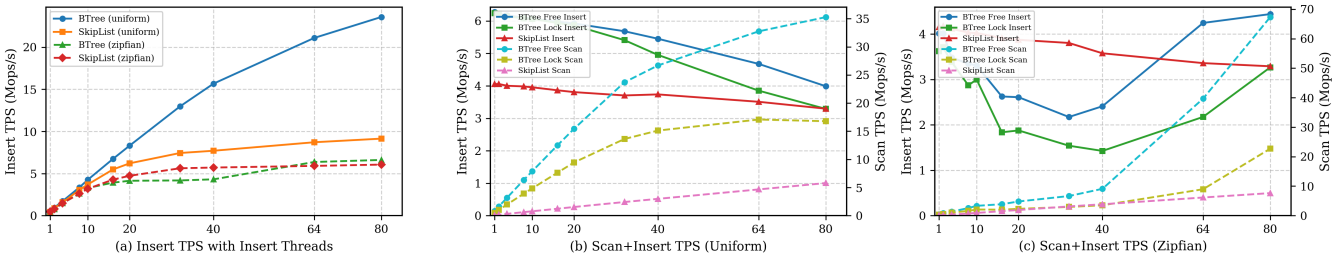



Figure 7: Evaluation of the non-blocking-scan B+ tree.

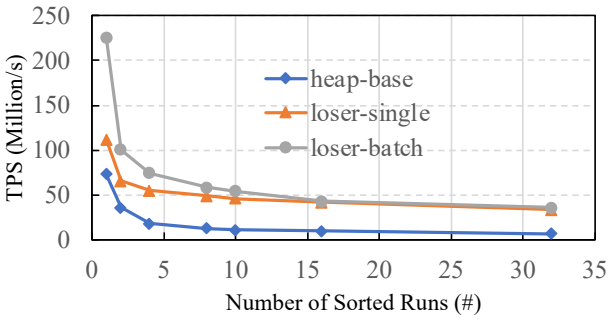



Figure 8: Evaluation result of file merge.

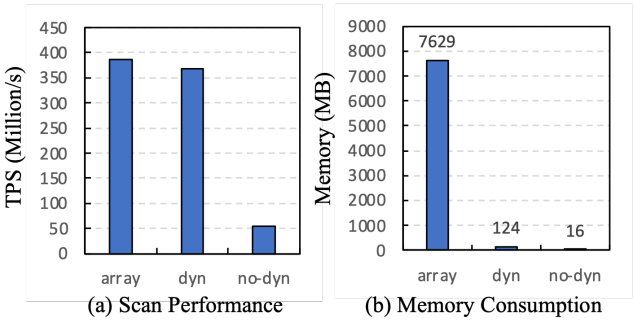



Figure 9: Evaluation result of dynamic consolidation.

eliminates the *valid*() call entirely.

The loser-batch delivers a further 2.01× improvement over loser-single at a single run, as batching 32 entries per *next*() call amortizes virtual dispatch across the entire batch and allows intra-batch comparisons to proceed in local registers without pointer indirection. However, this advantage erodes as the run count grows: at 32 runs, loser-batch converges toward loser-single in throughput, because loser-tree rebalancing dominates—consuming approximately 90% of CPU cycles—regardless of batch size. This behavior motivates the compaction policy in §3.3, which bounds the merge fan-in within 4 runs: at this shallow fan-in, the loser tree is amenable to vectorization of its comparison path, yielding over 41% additional throughput improvement.

## 4.3 Evaluation of Dynamic Consolidation

**Setup.** We run a standalone CakeMask instance and inject 1 billion uniformly random deletion pairs using 16 concurrent threads. We compare three configurations: (1) *array*, a 1-billion-entry in-memory version vector used as the upper-bound performance baseline; (2) *dyn*, CakeMask with dynamic consolidation enabled; and (3) *no_dyn*, CakeMask with dynamic consolidation disabled. The *array* is lock-free by construction, since each row can be deleted at most once and a deletion $\langle rowid, seq \rangle$ reduces to a simple write $vector[rowid] = seq$. To emulate transactional commits, each insertion is assigned a strictly monotonically increasing sequence number. Immediately upon completion of all insertions, we launch a scan thread that continuously checks the visibility of every row, from which we measure steady-state scan throughput and peak memory consumption.

**Result.** Figure 9 (a) reports scan throughput. With dynamic consolidation enabled, *dyn* achieves ∼370 M rows/s, matching the *array* baseline. This parity arises because the consolidation thread continuously promotes eligible deletion entries into the base bitmap, so the scanner encounters only a small residual delta that must be merged at query time. In contrast, *no_dyn* forces the scanner to merge all 1 billion deletion entries from on-disk delta files, suffering an 85% throughput degradation. The severity is further amplified by the experimental conditions: scans begin immediately upon completion of all insertions, at which point the delta-tree has accumulated many unflushed immutable memtables and on-disk files without any intervening compaction, driving the merge fan-in to its worst-case value and maximizing the per-scan merge cost that *no_dyn* must pay for every row visibility check.

Figure 9 (b) reports peak memory consumption. The *array* occupies 7,629 MB, as it maintains one 8-byte word per row regardless of deletion density. The *dyn* configuration stores a 1-billion-bit base bitmap (119 MB) plus a 5 MB active memtable, totaling 124 MB—only 1.6% of the *array*'s footprint while delivering equivalent scan throughput. The *no_dyn* configuration consumes just 16 MB at the measurement point, as consolidation has not yet promoted entries and most data remains in the delta-tree. However, this apparent memory economy comes at the cost of severe scan degradation. In the *dyn* configuration, the consolidation thread has already promoted the vast majority of deletion pairs into the base bitmap by the time scanning begins, leaving very few entries to flush to on-disk files; consequently, the number of delta files—and thus the merge fan-in encountered by scanners—is substantially lower in *dyn* than in *no_dyn*.

## 4.4 Combined Evaluation

**Setup.** We combine the design strategies as CakeMask to break down each design choice. We model a table with one billion rows and inject deletion pairs whose keys are drawn uniformly at random from $[0, 10^9]$, a representative table size for real workloads. To comprehensively evaluate deletion efficiency across insertion throughput, scan throughput, and memory consumption, we implement six representative mark-based deletion strategies that are typically adopted in production systems.

1. *array*: an in-memory version array indexed by rowid, used in IMCI [41], serving as the performance upper bound for both insertion and scan.
2. *bitmap_rocksdb*: a persistent monolithic bitmap backed by RocksDB [12] as the storage layer. Because the bitmap carries no versioning, all concurrent reads and writes must be fully serialized.
3. *segment_rocksdb*: a segmented bitmap (segment size $= 2^{16}$ rows) backed by RocksDB. Concurrent deletions within the same segment are serialized; however, each segment write generates a new version, so reads are not blocked by writes.
4. *rowid_rocksdb*: a straightforward row-level deletion index in RocksDB that uses the rowid as the key, with no domain-specific optimizations. RocksDB internally assigns a monotonic sequence number to each key and resolves visibility during reads.
5. *CakeMask*: our proposed design with all optimizations.
6. *no_dyn*: CakeMask without dynamic consolidation.

We evaluate three scenarios: (1) insertion scalability, by varying the number of insertion threads from 1 to 64 with no concurrent scan threads; (2) overall system behavior, by launching a single scan thread to continuously check row visibility immediately after all insertions complete; and (3) scan scalability, by fixing 16 insertion threads and varying the number of scan threads. For all RocksDB-based implementations, we use the default configuration with WAL [29] disabled to maximize write throughput. Each thread issues single-entry writes, i.e., the RocksDB write batch size is one.

**Result.** Figure 10 (a) reports insertion throughput as threads scale from 1 to 64. We exclude *bitmap_rocksdb*, which forbids concurrent writes by design, and *no_dyn*, whose insertion path is identical to *CakeMask*. The *array* baseline delivers the highest throughput, as each write is a simple in-memory operation. *segment_rocksdb* exhibits negligible improvement with additional threads, since concurrent writes to the same segment remain serialized. Both *rowid_rocksdb* and *CakeMask* scale with thread count, but *CakeMask* scales better, as pipeline locks in RocksDB's write path limit *rowid_rocksdb*'s concurrency. *CakeMask* scales linearly up to 16 threads, beyond which the internal switch barrier becomes the bottleneck. At a single thread, *rowid_rocksdb* slightly outperforms *CakeMask*, as its SkipList-based [33] memtable incurs lower per-insertion overhead than a B+ tree—an advantage erased by RocksDB's internal locking under higher concurrency. At 64 threads, *CakeMask* outperforms *rowid_rocksdb* by 3.7× and *segment_rocksdb* by 79×.

Figure 10 (b) presents scan scalability, with 16 fixed insertion threads and scan threads varying from 1 to 16. Scanning begins after 15 minutes of sustained insertions to accumulate sufficient data. *bitmap_rocksdb* is excluded as it lacks concurrent access. *array*, *segment_rocksdb*, and *CakeMask* scale near-linearly. *array* delivers the highest absolute throughput, and *CakeMask* closely tracks it, as both scan paths are dominated by in-memory operations. *segment_rocksdb* incurs modest storage access for segment loading, which is minor relative to the in-memory bitmap traversal. Even under heavy insertion pressure, *CakeMask*'s dynamic consolidation keeps most deletions promoted into the base bitmap, leaving few delta entries to merge at scan time. *no_dyn* scales linearly up to 4 scan threads, after which the syscall overhead of loading data from the kernel page cache becomes the bottleneck. *rowid_rocksdb* suffers from internal lock contention and unoptimized file merging, yielding poor scan throughput. At 16 scan threads, *no_dyn* outperforms *rowid_rocksdb* by 54×.

Figure 10 (c) reports peak memory consumption during scans. *array* consumes the most, at 7,629 MB. Both *bitmap_rocksdb* and *CakeMask* occupy approximately 130 MB: *bitmap_rocksdb* loads the entire bitmap into memory on first access, while *CakeMask* keeps only the base bitmap resident and scans it segment by segment with a reused buffer. Figure 10 (d) shows that *CakeMask*'s scan throughput closely matches *array*'s. *bitmap_rocksdb* ranks second, as subsequent scans reuse the bitmap loaded in the first pass. *segment_rocksdb* repeatedly reloads segments from storage, though each access remains efficient. Both *rowid_rocksdb* and *no_dyn* represent deletions at row granularity, yet *no_dyn* outperforms *rowid_rocksdb* by 17.9×, owing to its fine-grained, vectorized, and parallelized file merge optimizations.

## 4.5 Comparison with Production Databases

We select four representative production databases: two HTAP databases, *OceanBase* (Community v4.3) and *IMCI* (Enterprise v8.0.1 on Alibaba Cloud), and two OLAP databases, *Apache Doris* (SelectDB v4.0) and *Hologres* (v3.2), both on Alibaba Cloud. Both HTAP databases provide full ACID-compliant transactions. *OceanBase* appends delete marks to the LSM-tree, requiring scan queries to merge all primary keys. *IMCI* employs a heap table with an in-memory version array to trace deletions, eliminating primary key merging during scans. In *IMCI-CM*, we replace the version array in *IMCI* with our CakeMask. The OLAP databases lack ACID transaction support. *Doris* partitions a table into rowsets with versioned segment deletion bitmaps, serializing concurrent writes to the same segment. *Hologres* similarly uses segment deletion bitmaps but buffers scattered writes via an LSM-tree.

For all database instances, we use a uniform hardware configuration of 16 cores and 64 GB memory. We use the *Sys-*

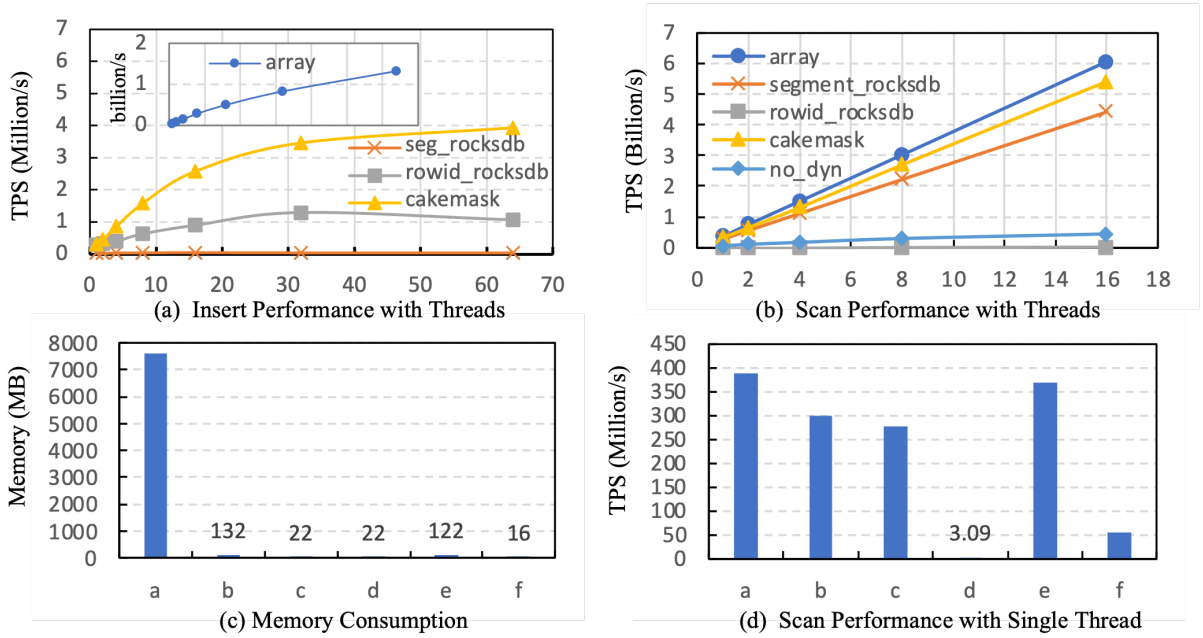



Figure 10: The evaluation result covering insertion scalability, scan scalability, and overall system behaviors. In panels (c) and (d), x-axis labels a-f denote *Array*, *bitmap_rocksdb*, *segment_rocksdb*, *rowid_rocksdb*, *CakeMask*, and *no_dyn*, respectively.

Table 1: Scan Latency (ms) of Different Deletion Strategies.

| OceanBase | Hologres | Doris | IMCI | IMCI-CM |
|---|---|---|---|---|
| 214144 | 137 | 70 | 100 | 30 |

*bench* tool to populate a table with 1 billion rows, then launch 128 concurrent clients running the `oltp_write.lua` workload for 30 minutes. Afterward, we issue an SQL query to count the live rows in the table and report the execution time. We focus exclusively on scan performance under heavy deletion rather than comparing update throughput. We do not compare raw insertion throughput, as it is dominated by the underlying storage architecture rather than the deletion strategy under evaluation. For example, comparing the write throughput of a B+ tree-based row store (IMCI) against that of a write-optimized LSM-tree (OceanBase, Hologres) would not constitute a meaningful evaluation. Moreover, OLAP systems (Doris, Hologres) trade consistency for ingestion throughput, violating OLTP correctness requirements. The central challenge for HTAP databases is handling concurrent mutations while preserving transactional consistency.

Table 1 reports the results; *IMCI-CM* achieves the best scan time. *OceanBase* exhibits poor scan performance because its LSM-tree engine must merge all primary keys even for a simple row-count query. An aggressive compaction policy could merge all data levels offline to eliminate merge-on-read overhead, but this incurs substantial write amplification. Moreover, the continuous heavy write traffic of HTAP workloads makes it impractical to compact all data levels in a timely manner. *IMCI-CM* outperforms the in-memory deletion array (*IMCI*) by 3.3×. We observe that *IMCI-CM*'s dynamic consolidation promotes deletions into the base bitmap promptly, so scan queries can directly copy the base bitmap as a local visibility bitmap and merge only a small number of delta entries, largely eliminating the costly sequence-to-bitmap conversion required by *IMCI*. *Doris* achieves scan performance comparable to *CakeMask*; however, its update throughput is three orders of magnitude lower than that of *IMCI-CM* and *OceanBase*, owing to its single-threaded write model.

## 4.6 TPC-C Benchmarks

We evaluate four representative HTAP databases—*SQL Server*, *OceanBase*, *IMCI*, and *IMCI-CM*—using the TPC-C benchmark with 100 warehouses, executed via BenchmarkSQL on instances provisioned with 16 cores and 64 GB memory under default configuration. The benchmark generates over 100 GB of data, which is much larger than the available memory. These systems were selected because they all support fully ACID-compliant transactions and share a similar production architecture, in which a secondary columnar index is attached to the primary row store. As in the previous benchmark, we do not compare absolute TPC-C numbers, but focus on the key point of how they handle data changes. Specifically, we issue a full-table row-count SQL query across all tables and measure its execution latency to evaluate only the deletion strategy. The experiment proceeds in three phases: (1) the *before* phase, measured immediately after the initial data load completes; (2) the *run* phase, measured while running TPC-C for 30 minutes; and (3) the *after* phase, measured 20

Table 2: Scan Latency (ms) of Different HTAP Databases

| Phase | SQL Server | OceanBase | IMCI | IMCI-CM |
|---|---|---|---|---|
| Before | 1617 | 35570 | 10 | 10 |
| Run | 24027 | 313460 | 60 | 20 |
| After | 6786 | 10770 | 60 | 20 |

minutes after the workload terminates. For all systems, the columnar index is enabled at table creation time.

Table 2 reports the results; *IMCI-CM* consistently outperforms the other three systems across all phases. In the *before* phase, data is loaded in near-primary-key order. *SQL Server* tracks deletions via a segment bitmap and a B+ tree of deleted primary keys resolved at query time. *OceanBase*'s merge-on-read model avoids a full all-to-all merge under ordered load. *IMCI-CM* uses the delta-tree to track deletion pairs and merges deletion marks into the base bitmap, incurring substantially lower overhead. In the *run* phase, heavy random deletions degrade *OceanBase* by 10× and *SQL Server* by 15×, as both systems must process large volumes of accumulated deleted primary keys during query execution. *IMCI-CM* incurs only a 2× slowdown: updates are converted into deletions plus insertions, and the additional insertions generate new row groups that modestly increase base bitmap accesses. In the *after* phase, *SQL Server* and *OceanBase* perform background compaction—migrating deleted keys into bitmaps and merging data levels, respectively—yielding significant latency reductions. *IMCI-CM* remains stable at 20 ms, as its compaction policy triggers only when 75% of rows in a row group are deleted; many row groups in the experiment had not reached this threshold, so the scan still accessed more base bitmap entries than in the *before* phase.

## 5 Related Work

**Merging with Delta Stores.** HTAP databases such as OceanBase [45], TiDB [18], and SAP HANA [37] employ an LSM-tree-based storage engine to buffer updates in both row and columnar formats, requiring analytical queries to merge all data levels to determine record visibility. Oracle Dual Format [21] records updates as a delta atop in-memory column units, which the column scanner merges with the base data at query time. ClickHouse [35] uses a MergeTree engine to eliminate duplicates, but deduplication is asynchronous and requires explicit SQL queries to filter out stale records. The all-to-all merge in these systems becomes a performance bottleneck under heavy random update workloads.

**Differential Updates.** Column-oriented OLAP databases such as Kudu [28], Doris [2], and GaussDB DWS [27] support differential updates for individual columns [17]. Since reconstructing an entire row from physically dispersed columns in columnar storage is prohibitively expensive, these systems log only the changed columns and merge them into the columnar store at query time. However, this remains costly owing to variable-length column types and requires an auxiliary tree structure for delta storage, which complicates data retrieval. Merging delta columns with fixed-layout base columns also requires runtime type dispatch, adding CPU overhead that compounds under wide tables. In HTAP databases, per-column delta logging is unnecessary since the updated row is already available from the row store.

**Deletion Vector.** Several databases employ deletion vectors for logical deletions to avoid full key merge. Hologres [19], AnalyticDB [48], and Doris [2] attach a deletion bitmap to each segment file, but a large bitmap causes lock contention under concurrent updates, whereas a small bitmap produces excessive small files that degrade scan performance. PolarDB [41] uses an in-memory version array with MVCC to decouple reads and writes, but at the cost of significant memory consumption. SQL Server [23, 24] and Vertica [22] combine both approaches: foreground-deleted keys are buffered in a delta store, and a background process migrates deleted keys to the deletion bitmap. The executor uses an anti-join operator to exclude deleted delta records, but this still incurs considerable merge overhead under high update rates.

**Updates in Lakehouse Systems.** Lakehouse systems are built atop commodity object storage and open formats [3, 7, 8, 36, 40], combining columnar formats (e.g., ORC, Parquet) for base data with row formats (e.g., CSV, Avro, and JSON) for incremental writes. They support either full-file rewrites or position deletions, but both strategies scale poorly under high-concurrency updates. These systems target time-insensitive workloads with large analytical queries and infrequent updates, making them ill-suited for HTAP scenarios.

## 6 Conclusion

This paper presents CakeMask, a visibility-management design for fast data synchronization from the row store to the columnar store in dual-store fresh-data HTAP databases. CakeMask achieves high read and write throughput with low memory consumption simultaneously. CakeMask maintains a delta-tree to buffer row deletion pairs alongside a compact base bitmap for fast scans; readers consult the base bitmap and merge only the relevant delta entries to determine final row visibility. To minimize merge cost, CakeMask incorporates three key techniques: (1) a non-blocking-scan in-memory B+ tree as the delta-tree memtable, enabling non-blocking scans under concurrent writes; (2) fine-grained, parallel, and vectorized file merge to improve delta-tree scan performance; and (3) dynamic consolidation, which continuously promotes deletion pairs from the delta-tree into the base bitmap to bound the data volume that scanners may touch. CakeMask has been deployed in production PolarDB-IMCI and validated across diverse production workloads for years.

## 7 Acknowledgments

We are grateful to our shepherd, Wenguang Wang, and the anonymous reviewers for their constructive comments.

## References


[1] Improve vectorization performance with intel® avx-512, December 2016.

[2] Apache doris: Open source data warehouse for real time data analytics, December 2025.

[3] Apache paimon: A lake format that enables building a realtime lakehouse architecture with flink and spark for both streaming and batch operations, December 2025.

[4] Duckdb on apsaradb rds for lightning-fast analytics, September 2025.

[5] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[6] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, 2008.

[7] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[8] Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, et al. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8, page 28, 2021.

[9] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[10] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1773–1786, 2019.

[11] E Donald et al. The art of computer programming, volume 3: Sorting and searching, 1997.

[12] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[13] G Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135, 1994. Publisher: IEEE Computer Society Series Number: 1.

[14] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: system programming guide, Part*, 2(11):0–40, 2011.

[15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[16] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[17] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 543–554, 2010.

[18] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[19] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, et al. Alibaba hologres: A cloud-native service for hybrid serving/analytical processing. *Proceedings of the VLDB Endowment*, 13(12):3272–3284, 2020.

[20] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow.*, 5(1):61–72, September 2011.

[21] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1253–1258. IEEE, 2015.

[22] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. 5(12):1790–1801, August 2012.

[23] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with sql server. *Proceedings of the VLDB Endowment*, 8(12):1740–1751, August 2015.

[24] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. Sql server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 1177–1184, Athens Greece, June 2011. ACM.

[25] Philip L Lehman and S Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[26] Scott T Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *ACM Sigmod Record*, 22(2):22–31, 1993.

[27] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation. *Proceedings of the VLDB Endowment*, 17(12):3786–3798, 2024.

[28] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: Storage for fast analytics on fast data. *Cloudera, inc*, 28:36–77, 2015.

[29] C Mohan and Frank Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. *ACM Sigmod Record*, 21(2):371–380, 1992.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta informatica*, 33(4):351–385, 1996.

[31] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, 1984.

[32] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. Cloud-native transactions and analytics in singlestore. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2340–2352, 2022.

[33] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[34] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. Two birds with one stone: Designing a hybrid cloud storage engine for htap. *Proceedings of the VLDB Endowment*, 17(11):3290–3303, 2024.

[35] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. Clickhouse-lightning fast analytics for everyone. *Proceedings of the VLDB Endowment*, 17(12):3731–3744, 2024.

[36] Prashant Wason Shiyan Xu. *Apache Hudi: The Definitive Guide: Building Robust, Open, and High-Performing Data Lakehouses*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 1st edition, 2025.

[37] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742, 2012.

[38] Michael Stonebraker. *The design of the POSTGRES storage system*, page 278–289. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[39] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[40] Alex Merced Tomer Shiran, Jason Hughes. *Apache Iceberg: The Definitive Guide: Data Lakehouse Functionality, Performance, and Scalability on the Data Lake*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 1st edition, 2024.

[41] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. Polardb-imci: A cloud-native htap database system at alibaba. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[42] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a BW-tree takes more than just BuzzWords. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (1):473–488, 2018.

[43] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[44] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, 2014.

[45] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. Oceanbase paetica: a hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster. *Proceedings of the VLDB Endowment*, 16(12):3728–3740, 2023.

[46] Matei Zaharia. Bringing the operational and analytical worlds together with lakebase. *Proc. VLDB Endow.*, 18(12):5539, August 2025.

[47] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. Htap databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 36(11):6410–6429, 2024.

[48] Fangyuan Zhang, Caihua Yin, Hua Fan, Fenghua Fang, Yineng Chen, Xuqi Wang, Mengqi Wu, Bing Chen, Tianbo Jin, Sibo Wang, et al. Analyticdb-pg: A cloud-native high-performance data warehouse in alibaba cloud. *Proceedings of the VLDB Endowment*, 18(12):5139–5152, 2025.