Enhancing Database Write Performance with the Write Operation Pushdown Framework

Hao Wu*
liujun.lj@bytedance.com
ByteDance Inc.

Lutong Zhang*
lutong.zhang@bytedance.com
ByteDance Inc.

Xinying Zheng
xz112@illinois.edu
University of Illinois Urbana-Champaign

Cheng Zhu
cheng.zhu@bytedance.com
ByteDance Inc.

Bo Shang
liubojun.7@bytedance.com
ByteDance Inc.

Lixun Cao
caolixun@bytedance.com
ByteDance Inc.

Xian Liu
xian.liu1@bytedance.com
ByteDance Inc.

Lei Zhang
zhanglei.michael@bytedance.com
ByteDance Inc.

Cheng Chen†
chencheng.sg@bytedance.com
ByteDance Inc.

ABSTRACT

While ByteDance's database infrastructure continues to serve bursty user traffic, emerging large-scale interactive applications generate increasingly write-intensive workloads spanning all stages of the data lifecycle. In production environments, we observe that these workloads are consistently bottlenecked by a common write-path inefficiency rooted in cache-miss-induced remote page fetches. In compute-storage disaggregated architectures, such remote page accesses introduce substantial I/O latency that dominates write performance. To address this bottleneck, we propose Write Operation Pushdown (WOP), a framework that eliminates Fetch-Before-Write (FBW) from the write path by pushing eligible writes to the storage layer for asynchronous execution. WOP leverages the redo-flush path to push down write operations and enhances the storage layer for semantic-aware execution. WOP supports high pushdown capacity with minimal memory footprint through Lightweight Metadata Tracking. WOP preserves correctness and consistency in distributed deployments via carefully designed mechanisms. To prevent read stalls introduced by asynchronous execution, WOP integrates a Fetch-After-Write (FAW) optimization with localized, on-demand replay. Across production workloads, WOP achieves up to 3.33× throughput improvement. On synthetic benchmarks, WOP improves throughput by up to 6.7× and reduces average latency by up to 85.2%. These results show that WOP significantly enhances write efficiency in industrial-scale workloads while preserving correctness in disaggregated deployments.

PVLDB Reference Format:
Hao Wu, Lutong Zhang, Xinying Zheng, Cheng Zhu, Bo Shang, Lixun Cao, Xian Liu, Lei Zhang, and Cheng Chen. Enhancing Database Write Performance with the Write Operation Pushdown Framework. PVLDB, 19(12): 3847 - 3859, 2026.
doi:10.14778/3827998.3828000

*Both authors contributed equally to this research.
†Corresponding Author.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828000

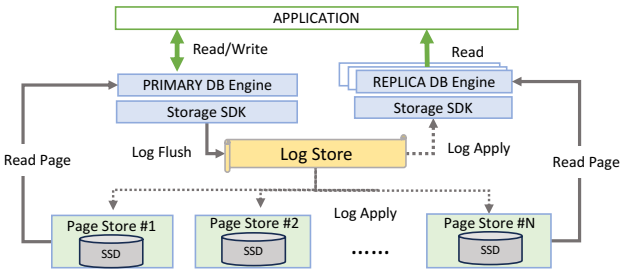

Figure 1: Architecture of ByteNDB

1 INTRODUCTION

ByteDance operates a large ecosystem of applications—including short-video platforms (e.g., Douyin, the Chinese version of TikTok), news apps (e.g., Toutiao), advertising systems, e-commerce services, and financial platforms—that require storing and accessing massive volumes of structured data. These applications exhibit several distinctive workload characteristics: (1) rapid data growth at petabyte-to-exabyte scale, (2) highly bursty access patterns driven by social events and promotional campaigns (e.g., Double 11, Chinese New Year), where traffic can surge to 10×−100× of normal volume. Such workloads impose stringent requirements on scalability, elasticity, and cost efficiency for the underlying Online Transactional Processing (OLTP) database infrastructure. Similar to prior systems such as Aurora[36], Socrates[4], and Taurus[12], we developed ByteNDB[6][34], a shared-storage, compute-storage disaggregated OLTP database designed to accommodate the above characteristics. As illustrated in Figure 1, ByteNDB separates compute nodes and storage nodes. Compute nodes process queries while storage nodes are responsible for data persistence. The two tiers communicate over the network. This architecture effectively accommodates explosive data growth and traffic bursts by elastically provisioning

compute nodes, while maintaining durable and scalable storage at the backend. We will introduce ByteNDB's architecture design in detail in §2.2.

In recent years, the rapid expansion of emerging businesses such as e-commerce and live streaming[17][11] has further intensified the demands placed on our database infrastructure. These applications generate massive volumes of fine-grained transactional writes (e.g., order placements, payment confirmations, and gifting events). As these services scale, such write traffic constitutes an increasing share of the workload, making write-path efficiency a critical requirement for the database infrastructure. However, under ByteNDB's compute-storage disaggregated architecture, we observe that these writes suffer from severe **write-path inefficiencies**. The root cause lies in the traditional Fetch-Modify-Write (FMW) paradigm used by B^+ Tree-based storage engines. Specifically, upon cache miss, pages must be fetched into the buffer pool before updates can be applied. This Fetch-Before-Write (FBW) is particularly costly in compute-storage disaggregated architectures, where even a small write requires synchronously fetching the entire page from remote storage over the network. Consequently, it introduces substantial network round-trip time (RTT) and I/O bandwidth consumption.

This inefficiency manifests across all stages of the data lifecycle (§3.2). In the write-heavy ingestion and archiving stage, the high volume of writes forces the system to fetch a large number of remote pages, amplifying network overhead. In the online serving stage, the problem is exacerbated by schemas with many secondary indexes, where a single logical transaction may trigger multiple independent, non-contiguous page fetches. To further validate the pervasiveness of this inefficiency, we conducted an extensive analysis of production workloads that exhibit performance degradation and found the same FBW amplification pattern. There are a few existing approaches that attempt to mitigate FBW overhead in the monolithic databases[24]; however, they exhibit inefficiency under ByteDance's extremely high-QPS workload scenario and struggle to guarantee consistency in distributed deployment (§3.3, §3.4).

Our Approach. To solve the write-path inefficiency, we propose a Write Operation Pushdown (WOP) framework for shared-storage, compute-storage disaggregated databases. WOP redesigns the write path by pushing eligible writes to the storage layer and executing them asynchronously, thereby removing FBW from the critical path of write operations. WOP reuses ByteNDB's redo-flush path to push down write operations and enhances the storage layer for semantic-aware execution. To enable effective pushdown, we introduce a Lightweight Metadata Tracker (LMT) that maintains only the minimal metadata required for eligibility checks. To guarantee data consistency, we design a set of consistency-preserving mechanisms in both single-node and distributed settings. Finally, to avoid introducing read stalls from asynchronous execution, we integrate a Fetch-After-Write (FAW) optimization mechanism. Even in adversarial scenarios where a page is accessed right after a pushed-down write, FAW optimization preserves the performance of that access.

Results. We evaluate WOP using both production and synthetic workloads, reporting end-to-end throughput and latency. On production workloads, WOP achieves up to 3.33× throughput improvement. On four representative synthetic write workloads, WOP improves throughput by up to 6.7× and reduces average latency by up to 85.2%. Our results further show that LMT enhances pushdown effectiveness by enabling more eligible writes to be pushed down. WOP remains robust across varying memory configurations and index counts, effectively mitigates Fetch-After-Write stalls, and preserves—often slightly improves—concurrent read performance.

Contribution. This work makes the following contributions:

- We identify the pervasive write-path inefficiencies under the compute-storage disaggregated architecture, and quantify their impact on end-to-end performance. (§3.2)
- **Write Operation Pushdown (WOP) Framework.** We redesign the write path in disaggregated databases and build the first end-to-end WOP framework (§4.1, §4.2).
- **Lightweight Metadata Tracking.** We design a lightweight metadata tracker that records only the minimal metadata needed for eligibility checks. (§4.3)
- **Consistency Safeguards for Asynchronous Pushdown.** We design a set of consistency-preserving mechanisms for WOP in both single-node and distributed settings. (§4.4)
- **Fetch-After-Write (FAW) Optimization.** We introduce an FAW optimization that targets the worst-case Fetch-After-Write scenarios, which significantly reduces read stalls. (§4.5)
- We conduct comprehensive experiments on both synthetic and production workloads and demonstrate up to 6.7× throughput improvement and 85.2% average latency reduction on write-intensive workloads. (§5)

2 BACKGROUND

2.1 Shared-nothing and shared-storage architectures

In the landscape of distributed database systems, two primary architectural paradigms have emerged: shared-nothing and shared-storage architectures. In **Shared-nothing** databases (e.g., Google Spanner[10], TiDB[14], CockroachDB[35]), data is horizontally partitioned (sharded) across nodes using hash or range strategies, with each node maintaining full authority over its respective partitions. Their performance is highly sensitive to the partitioning scheme; suboptimal sharding can trigger expensive cross-partition operations and complex distributed transactions. **Shared-storage** databases (e.g., Amazon Aurora[36], ByteNDB[6][34], PolarDB[8]), conversely, share data storage across compute nodes. The storage tier remains largely semantic-oblivious, acting as a persistent provider of physical pages. Under this architecture, compute nodes must fetch required pages from remote storage into a local buffer pool for computation. Consequently, network I/O becomes the primary bottleneck, particularly under write-heavy or random-access workloads where frequent page transfers significantly limit system throughput. In this paper, both the problem we study and the solution we propose are based on a shared-storage, compute-storage disaggregated database architecture.

2.2 ByteNDB architecture

ByteNDB is a high-performance, disaggregated relational database developed by ByteDance, specifically optimized for cloud-native

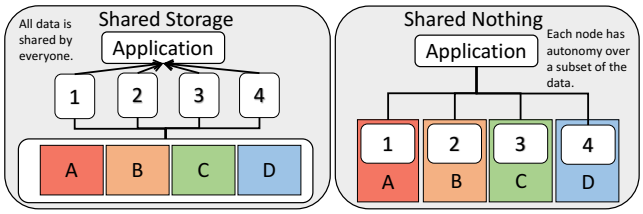

Figure 2: Shared storage vs Shared nothing architecture

environments and OLTP workloads. Following the 'Log is the Database' paradigm[36], ByteNDB employs a shared-storage architecture (Figure 1) where only the redo log—representing physical data changes—is materialized. This design minimizes the need for frequent full-page flushes and significantly reduces network I/O load.

Compute Layer. The compute layer consists of database engine instances responsible for SQL execution, transaction management, and buffer pool maintenance. ByteNDB distinguishes between Read-Write (RW) and Read-Only (RO) compute nodes. The RW node handles all write transactions; when all redo logs of a transaction have been persisted to the Log Store, the transaction can be successfully committed. RO nodes maintain local consistency by subscribing to the redo log stream (either from the shared storage or directly from the RW node) and replaying logs to refresh their local buffer pool state. To guarantee reliability, ByteNDB utilizes a hybrid replication scheme: intra-datacenter nodes that share a single storage volume utilize redo-based replication (physical replication), while inter-datacenter standbys utilize logical binlog replication to provide cross-region high availability.

Storage Layer. To address the diverging I/O characteristics of log records and data pages, ByteNDB physically decouples its storage layer into two specialized components: (1) Log Store: A high-throughput, distributed append-only system built on blob storage. It serves as a durable source of truth for all system changes, ensuring low-latency persistence for incoming redo logs. (2) Page Store: A versioned page management system that continuously accepts redo logs from the Log Store. It performs 'log-to-page' materialization, applying incremental changes to maintain up-to-date versions of data pages. The Page Store is optimized for random page access, supporting efficient reads for compute nodes.

3 MOTIVATION

3.1 Limitations of conventional write path for B+ tree-based systems

Conventional write path in MySQL-like systems typically follows a **Fetch-Modify-Write (FMW) pattern**: to update a data page that does not reside in the buffer pool, the system first fetches the full page into the buffer pool (Step ① in Figure 3), applies in-place changes and generates a redo log (Step ②), and subsequently writes the redo log back to persistent storage (Step ③). When the storage acknowledges the write, the transaction is committed (Step ④). While effective in monolithic databases, this pattern incurs high overhead in disaggregated architectures due to remote page accesses. Key limitations include:

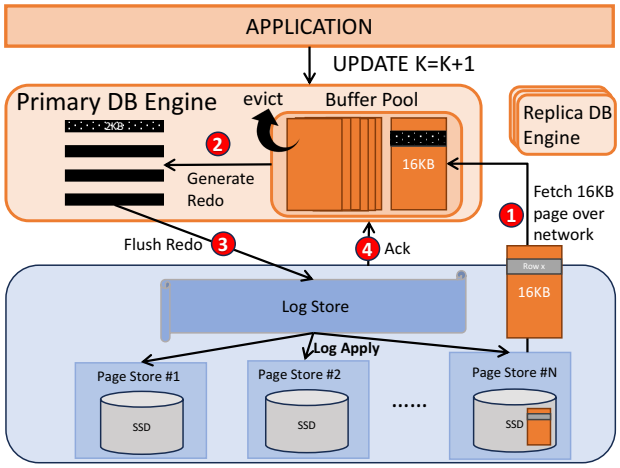

Figure 3: Conventional write path in MySQL-like systems

(1) Firstly, remote page reads introduce high latency. As Figure 3 shows, every write that targets a page that does not reside in the buffer pool requires the full page (16KB for MySQL[26], 8KB for PostgreSQL[29]) to be fetched over the network. This remote access latency can potentially become a dominant performance bottleneck, especially under write-intensive workloads.

(2) Secondly, full-page reads are often unnecessary. Many workloads—such as fine-grained Data Manipulation Language (DML) operations or metadata updates (e.g., transaction commit number updates)—modify only a small fraction of a page. Nevertheless, the system must fetch the entire page. We quantify this mismatch by comparing the redo log (which carries the modifications) bandwidth to the write throughput. For example, when redo traffic sustains 10 MB/s under 5K write operations per second, each write generates approximately 2KB modifications on average. Given a 16KB page size, this corresponds to updating only about 12.5% of a page per operation. Across different workload stages (Ingestion, Serving, Archiving, §3.2), the average modification ratio remains similarly low, approximately 29% during ingestion, 25% during serving, and 31% during archiving, indicating that a substantial portion of fetched page data is never modified. Furthermore, these full-page fetches are particularly wasteful when modified pages exhibit low temporal locality, as commonly observed in archival workloads.

(3) The page loaded by the Fetch-Before-Write (FBW) pollutes the buffer pool and may trigger repeated I/O. Under the FMW paradigm, each write operation brings its target page into the buffer pool; such pages pollute the buffer pool and displace other useful pages that would otherwise benefit from caching. When these evicted pages are accessed, they must be re-fetched from the storage layer, triggering additional remote I/O.

(4) Finally, compute nodes waste substantial memory, CPU cycles, and I/O bandwidth waiting for pages to be fetched, instead of executing user queries, causing significant resource inefficiency.

3.2 Data Lifecycle

In modern Cloud-Native Database Management Systems (DBMS), data typically undergoes a lifecycle that consists of three stages: **Ingestion stage**: Large volumes of data are ingested into the DBMS from heterogeneous external sources. This phase requires massive batch writes to establish the initial state. **Serving stage**: Data is actively accessed by different user applications. To meet stringent latency requirements for downstream queries, a space-for-time strategy is usually adopted by maintaining numerous secondary indexes for user queries[28]. **Archiving stage**: To prevent unbounded storage growth and optimize storage costs, we employ a data archival strategy in which obsolete data are periodically permanently deleted or offloaded from high-performance online storage to low-cost, high-latency media (e.g., object storage). This archival process generates deletion-heavy workloads that trigger extensive page updates. All three stages are affected by the write inefficiency rooted in the FMW pattern.

Remote page reads dominate ingestion latency. During the initial data ingestion phase, the system must establish a large persistent state through sustained batch insertions. In our experiments, importing 1 billion records into a table with 2 secondary indexes required several days, leading to a poor user experience. As shown in Table 1, each INSERT statement (3,180 records, average record size 0.3KB) exhibits a latency of 290–790 ms. A detailed breakdown reveals that only 120–140 ms is spent on computation, whereas 170–650 ms is attributed to page I/O, indicating that ingestion is predominantly network-I/O bound. Further analysis of network I/O shows that each INSERT triggers 500–1500 page reads. The base table accounts for only about 60 page reads, while secondary index maintenance contributes 450–1400 page reads, dominating the I/O footprint. These results indicate that remote page reads—particularly those caused by scattered secondary index pages—constitute the primary bottleneck during ingestion.

These excessive page reads arise from the FMW paradigm (§3.1). During each insert, the engine must fetch each target page into the compute node before applying modifications. In a compute–storage disaggregated architecture, this requirement translates into a large number of remote page fetches over the network. The effect is amplified for secondary indexes, whose entries are physically scattered across the key space, causing a single INSERT to touch many independent pages, thus bottlenecking the performance.

Table 1: Breakdown of execution of one INSERT statement.

Metric	Value
Number of Records	3,180
Average record size	0.3 KB
Secondary index entry size (2 indexes)	0.05 KB
End-to-end latency	290 ms – 790 ms
Non-I/O latency	120 ms – 140 ms
Total I/O latency	170 ms – 650 ms
Single page read latency	300 μs – 500 μs
Total page read count	500 – 1,500
Base table page reads	$\approx$ 60
Secondary index page reads	450 – 1,400

Secondary index multiplicity makes writes network-I/O bound during serving. User-facing workloads usually impose strict latency requirements for the serving phase. To meet these requirements, systems aggressively deploy multiple covering secondary indexes to eliminate base-table lookups, thus accelerating critical query paths. For example, an advertising delivery system may need to support more than 10 multidimensional queries over campaign data, including filtering by time range, advertiser, campaign status, marketing objective, or associated product ID. Supporting these access patterns efficiently often requires maintaining several specialized secondary indexes. While effective for reads, this design substantially increases write-path complexity. Each update now must propagate to all relevant secondary indexes and thus touch multiple independent pages across the storage layer. Consequently, the performance of write operations scales with index multiplicity and becomes increasingly dominated by network I/O.

Archival deletes trigger a surge of network I/O and increase primary-standby replication lag. During the archiving phase, the periodic cleanup of obsolete data triggers bulk delete operations on the primary node. Because archived records are typically cold, their corresponding pages are rarely resident in the buffer pool. As a result, delete operations must first fetch both the primary index pages and numerous scattered secondary index pages before applying the deletions. This access pattern sharply increases storage-layer page reads, triggering a surge in network I/O. Standby nodes are particularly vulnerable to this effect. During binlog replay, standbys must preserve transactional order and provide limited replay parallelism, which constrains their ability to handle the additional page read pressure. Consequently, deletion-induced I/O amplification can translate into pronounced primary-standby replication lag.

Production Analysis. To further validate the pervasiveness of the write-inefficiency, we conducted an extensive analysis of production incidents involving performance degradation. Across 38 reported instances, we find that 27 exhibit symptoms consistent with FMW amplification: the onset of performance degradation aligns with a near-synchronous rise in both write QPS and storage-side read-page QPS. Therefore, all those instances can benefit from write pushdown. Among those 27 pushdown-applicable cases, 11 arise from data ingestion dominated by sustained inserts, 14 occur during serving, and 2 are triggered by archiving workflows with deletion-heavy traffic. This evidence confirms that the same write-path inefficiency manifests across all stages of the data lifecycle.

3.3 Existing Solution: Change Buffer

The most prominent industry solution for mitigating Fetch-Before-Write (FBW) is the Change Buffer[24] (originally called the Insert Buffer in InnoDB).

3.3.1 Change Buffer overview. In traditional single-node MySQL deployments, the Change Buffer optimizes write performance for non-unique secondary indexes. Its core insight is to defer the modification of secondary index pages that are not currently resident in the buffer pool, thereby avoiding synchronous disk I/O in the write path. Instead of fetching the target page, the engine caches the modification as `ibuf_entry` in a specialized system-wide B^+ tree

Table 2: SMO-Safety Checks for different types of write operations

Operation	SMO Risk	Safety Condition (Without Page Fetch)
Insert	May trigger page split if insufficient free space.	DB engine uses a 2-bit bitmap to estimate page free space (S_{free}). Insert is buffered-safe only if: $\sum \text{Size(buffered writes)} + \text{Size(current insert)} < S_{\text{free}}$. Otherwise, fall back to conventional page fetch to avoid split.
Delete-Mark	No immediate SMO risk (logical mark-delete).	Always considered buffered-safe since deletion is implemented as a mark operation and does not physically reorganize the page.
Delete	May create empty page after physical record removal.	Delete operation is buffered-safe only if $N_{(\text{buffered inserts})} - N_{(\text{buffered deletes})} > 1$ Otherwise, fall back to conventional page fetch to avoid merge .

known as the `ibuf_tree`. These changes are subsequently merged when the page is loaded into the buffer pool for a read request or when processed by a background ibuf merge thread. By transforming random I/Os into deferred, potentially batched operations, the Change Buffer significantly improves throughput on high-latency storage media (e.g., HDDs) and write-intensive workloads.

3.3.2 How MySQL decides whether to apply or buffer the change. To determine whether a write operation can be safely buffered or must trigger a page fetch, Change Buffer relies on a set of conservative heuristics and per-page metadata. This logic prevents B^+ tree Structural Modification Operations (SMO) when the page is not resident in memory; buffering an SMO can redirect subsequent writes to stale page locations, causing page-access inconsistencies. Currently, the change buffer supports 3 types of write: Insert, Delete-Mark, and Delete. Table 2 summarizes the safety checks. Specifically, it maintains a 2-bit bitmap to approximate each page's free space. The bitmap categorizes free space into four coarse-grained levels: (i) 00, indicating no free space (page nearly full); (ii) 01, approximately 512 bytes; (iii) 10, approximately 1024 bytes; and (iv) 11, at least 2048 bytes of free space. Delete-Mark operations are always considered safe for buffering because they are implemented as logical mark deletions that do not alter page structure. In contrast, insert and delete operations physically modify page contents and may trigger SMOs; therefore, their buffering eligibility must be carefully determined. Once the data page is fetched into the buffer pool, its free-space information in the bitmap is updated based on the page's actual state, and all `ibuf_entries` for this page will be merged into the target page and removed from the `ibuf_tree`.

3.3.3 Status of Change Buffer. Change Buffer was effective in the HDD era, where random I/O dominated latency. With the adoption of SSDs, the gap between random and sequential I/O narrowed, reducing the benefit of buffering. At the same time, Change Buffer adds implementation complexity and has been associated with correctness and stability issues[20]. As a result, recent standalone RDBMS versions disable (e.g., MySQL 8.4[25]) or remove it by default (e.g., recent MariaDB versions[20]). Change Buffer is potentially more beneficial in compute-storage disaggregated architectures. However, few systems have revisited or redesigned the Change Buffer for this context. Amazon Aurora does not employ Change Buffer and instead requires all secondary index updates to be fully written through[2]. As a consequence, performance bottlenecks have been observed in workloads with heavy secondary index usage[1]. In contrast, Alibaba presents a design to integrate Change Buffer with physical replication; but it requires a complex protocol to guarantee consistency, which is a performance bottleneck[23].

3.4 Limitations of Change Buffer techniques

While Change Buffer is effective in monolithic architectures, its design principles face significant consistency and efficiency challenges in a large-scale distributed deployment.

3.4.1 Inconsistency in Distributed Environments. When the primary buffers a write in the `ibuf_tree`, replicas cannot guarantee read freshness. If the target page is already in a replica's buffer pool, reads return a stale cached version because change-buffer merging is triggered only on page fetch and therefore is not invoked. If the page is absent, the replica fetches it from storage, but as a read-only node, it cannot perform Change Buffer merging locally, so the fetched page is still stale. In both cases, reads can miss committed updates, violating data consistency.

3.4.2 Buffering Overhead. As illustrated in Figure 4(a), `ibuf_entry` is a structured record composed of metadata fields (e.g., space identifier, page number, offset counters, flags) followed by the full secondary index payload (`IBUF_REC_FIELD_USER`). Buffering the complete payload is necessary because the `ibuf_tree` is the only place where the modification is stored before the merge. Consequently, the storage cost of each buffered entry scales with the size of the secondary index record itself. This design fundamentally constrains buffering capacity. A large index payload directly leads to more buffer space consumption and reduces the number of operations that can be buffered, thereby limiting the performance benefits. By default, the `ibuf_tree` can occupy up to 25% of the buffer pool. In our experiments, increasing this allocation from 25% to 50% improved throughput from 4,000 to 6,000 QPS, demonstrating the strong sensitivity of performance to buffering capacity. Moreover, as the `ibuf_tree` approaches its capacity, the engine must trigger merge processes that fetch the corresponding pages and apply all accumulated buffered operations. Under sustained write intensity, the merge occurs frequently, diminishing the intended advantage of Change Buffer.

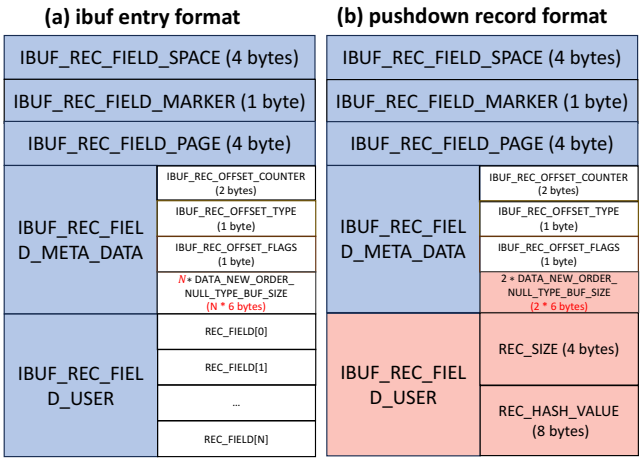

Figure 4: Metadata format

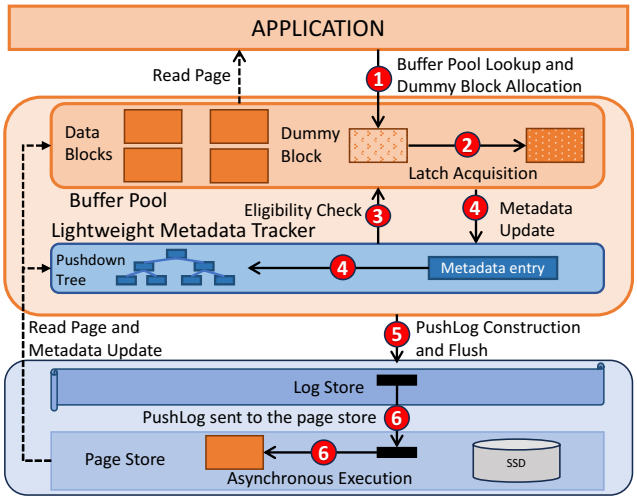

Figure 5: Architecture of Write Pushdown Framework

4 WRITE OPERATION PUSHDOWN FRAMEWORK

The core philosophy of the Write Operation Pushdown (WOP) framework is to eliminate the prohibitive overhead of the Fetch-Modify-Write (FMW) pattern by encapsulating the modification as a logical redo log and pushing it directly to the storage layer for asynchronous execution.

4.1 Overview

WOP consists of four key designs:

PushLog-Based Execution WOP encapsulates eligible write operations into a specialized log record format, termed PushLog, flushed directly to the storage layer for asynchronous execution.

Lightweight Metadata Tracking This module determines the *pushdown-eligibility* of a write operation. The performance of the WOP framework is highly sensitive to how many eligible writes can be pushed down to the storage layer. We design a lightweight metadata tracker to maintain only minimal metadata required to evaluate the eligibility of subsequent writes targeting the same page, so that the WOP framework allows pushing down more writes while minimizing buffer pool footprint at the compute layer.

Consistency Safeguards Maintaining data consistency is paramount in both standalone and distributed database deployments. The framework implements a suite of consistency safeguards to ensure that WOP does not violate transaction isolation levels. These safeguards manage the visibility of "in-flight" pushdown operations and ensure that the distributed state remains consistent, even in the event of partial failures or crashes.

Fetch-After-Write (FAW) Optimization In *fetch-immediately-after-write* scenarios, reads can be blocked while waiting for the storage layer to complete the execution of the pushed-down write. To address this performance degradation, we introduce a specialized FAW optimization. The framework retains the "stale" version of a page before its eviction and attaches the corresponding redo logs to its dummy block. Upon a subsequent read request, the system performs on-demand log replay to reconstruct the most recent version of the page.

4.2 PushLog-Based Execution

The Write Operation Pushdown (WOP) framework circumvents the physical page-load requirement by encapsulating write operations into specialized Redo logs, termed PushLogs. These logs are dispatched to the storage layer for asynchronous execution, effectively decoupling the compute node from the I/O latency associated with traditional page fetching.

4.2.1 Pushdown Write Path. As shown in Figure 5, the detailed workflow of the pushdown write path is:

(1) Buffer Pool Lookup and Dummy Block (§4.4.1) Allocation: The system searches for the target (space_id, page_no). If a cache hit occurs, the system defaults to the conventional write path (modify, write-back). Upon a cache miss, the system allocates a dummy block that does not contain any actual data in the buffer pool.

(2) Latch Acquisition: The system acquires an exclusive latch on the dummy block to ensure mutual exclusion for concurrent writes to the same logical page.

(3) Eligibility Check: The lightweight metadata tracker performs the eligibility check to confirm the operation is pushdown-safe based on the metadata of the same page.

(4) Metadata Update: The system registers the write's metadata to the lightweight metadata tracker.

(5) PushLog Construction and Flush: The system constructs a PushLog representing the operation and flushes it to the storage layer.

(6) Asynchronous Execution: The storage layer receives the operation and executes the physical modification locally.

When the read request targets the page that has pushed-down writes, the page will be fetched, and the metadata tracker will update the page map and delete all pushdown entries in the pushdown tree to reflect the actual state of the page. In this way, the metadata tracker always reflects the correct states of the page.

4.2.2 PushLog Format. InnoDB employs a technique known as *physiological logging* to construct its redo log. Under this model, each log record captures a logical modification and the physical location it targets within a specific page. Concretely, a redo record consists of *Space ID*, *Page ID* to locate the page, *Record Offset* to locate the target record within the page, enabling direct in-place modification without a full page search. *Record Data* contains the actual modification data. To enable write pushdown, we introduce a new logical Redo type, MLOG_PUSHDOWN_INSERT. This format omits precise physical offsets, containing only the operation type and record payload (Space ID, Page ID, Op Type, Record Data), where the Op_Type can be INSERT, DELETE_MARK, or DELETE.

4.2.3 On-Storage Execution. Unlike physiological redo records, which specify precise physical offsets for direct in-place modification, PushLogs describe updates at the logical level. To support this model, we extend the storage layer to interpret and apply logical PushLogs. Firstly, we equip the storage layer with schema and page-format awareness, including key encoding rules, record layouts, and page organization metadata to parse the page structure. Secondly, we implement page-level lookup logic (e.g., binary search in B^+ Tree pages) to determine the correct modification position. Importantly, this additional logic does not make storage-side CPU a bottleneck: the binary search incurs only lightweight overhead and mainly exploits the relatively abundant, often underutilized CPU cycles at the storage layer.

4.3 Lightweight Metadata Tracker

Lightweight Metadata Tracker (LMT) is responsible for the eligibility check of the write operation and keeps track of the metadata of the data page.

4.3.1 Eligibility Criteria. A two-tier validation process governs the eligibility of a write operation for pushdown: Content-Dependency Analysis and SMO Prediction. First, the framework performs a content-dependency check to ensure the operation does not require immediate, synchronous validation—for example, uniqueness constraint checking for primary key or unique secondary index. This is determined via schema-level metadata. Second, the system must guarantee that the modification will not trigger an SMO on the target B^+ tree. Our SMO prediction logic builds upon the foundational principles of the vanilla InnoDB Change Buffer (Table 2), but introduces a specialized optimization to overcome the memory-density limitations of traditional implementations.

4.3.2 The Lightweight Metadata Tracker. Our framework transforms the role of the `ibuf_tree` from a data cache into a metadata tracker (termed `pushdown_tree`). For each pushed-down write, it stores a `pushdown_entry` that records only the metadata needed to track pending changes per destination page (e.g., page ID, record size, and operation type). Because the full operation payload is encapsulated in `PushLog` and flushed to the storage layer, the compute node no longer needs to retain the index record itself. We redesign `pushdown_entry` into a fixed-length 39-byte format (Figure 4(b)). This format discards the payload and retains only compact fields, including a 4-byte record size and an 8-byte hash for verification. By tracking only minimal metadata, the `pushdown_tree` avoids the capacity bottleneck of the native `ibuf_entry` format, where buffering full index records reduces entry density and limits throughput.

Case Study: Space Efficiency. To illustrate the efficiency of this compacted format, consider the (c, pad) index on a standard sysbench table, as defined in Listing 1. A vanilla `ibuf_entry` for this index typically requires approximately 215 bytes (comprising headers, field metadata, and the 184-byte payload containing the 3 fields c, pad, id). In contrast, our framework reduces this to a fixed 39 bytes, representing an 81.8% reduction in memory overhead per entry. This optimization allows the system to buffer significantly more write operations within the same memory budget.

4.4 Consistency Safeguards

The Write Operation Pushdown (WOP) framework implements consistency guarantees in both single-node and distributed settings to ensure that pushdown writes to the storage layer do not violate the transactional or physical integrity of the database.

4.4.1 Single-Node Consistency.

Physical Data Consistency. Traditional physiological redo is generated based on the page's current physical layout—record offsets must be determined before logging the update. Without coordination, concurrent operations may compute offsets based on inconsistent page versions, leading to structural corruption during asynchronous execution. For example, consider two concurrent transactions inserting a_1 and a_2 into the same page. Suppose a_1 is pushed down and generates a PushLog record, while a_2 follows the traditional path, generating a physiological redo record. If a_1's PushLog has not yet been persisted when a_2 computes its physical offset, a_2's redo log is generated based on a stale page version ($Page_0$). If the storage layer later replays a_1 followed by redo of a_2, the page state will be corrupted. To eliminate this hazard, we introduce the Dummy Block as a metadata-only placeholder in the buffer pool. For each page with pending pushdown operations, all subsequent writes—whether pushed down or executed conventionally—must acquire an exclusive latch (X-Latch) on the corresponding Dummy Block. In this way, we preserve the mutual-exclusion semantics of the traditional write path.

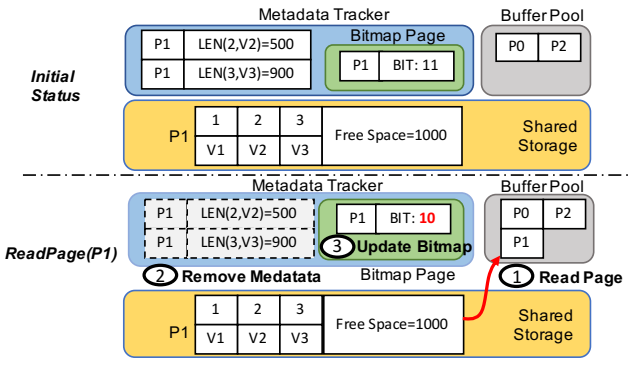

Figure 6: Crash Recovery and Atomicity

Crash Recovery and Atomicity. Assume that data page P1 initially had 2,400 bytes of free space (represented as 11 in the bitmap page). Subsequently, two records, (key1, value1) and (key2, value2), of sizes 500 bytes and 900 bytes respectively, have been pushed down and successfully applied to P1 within the storage layer. When P1 is loaded into the buffer pool at ①, as shown in Figure 6, the corresponding metadata entry must be removed from the Metadata Tracker at ② and free space size of this page in the bitmap page must be updated at ③, changing the value from 11 to 10 (indicating that the remaining free space on the page is now 1,000 bytes). The final two steps are referred to as *post-I/O metadata update*. If a server crash occurs just before point ③, the free space recorded in the bitmap page for this page deviates from the actual free space of the page. To address this issue, we implement a *Bitmap-based Marking* strategy. Before the post-I/O metadata update begins, the system clears the corresponding bit (sets to 0) in the target page's free space bitmap. This proactive marking ensures the correctness of subsequent operations after the crash recovery:

- For writes that go through the traditional write path, the system will load the page from the storage layer and update the metadata tracker to reflect the actual free space of the page.
- For pushdown-safe inserts, the system sees the bitmap bit is 0 and goes through the traditional write path.
- For pushdown-safe delete-mark operations and pushdown-safe deletes, their correctness is not affected by the potential inconsistency of the free space information.

4.4.2 *Distributed Consistency.*

Physical Replication Compatibility. Change Buffer mechanisms often struggle with physical replication, potentially leading to divergence between the primary and the replica. As discussed in (§3.4.1), the root cause is that read-only replicas do not perform Change Buffer merge into data pages, so reads may observe stale state. WOP avoids this pitfall. In our design, 'merging' does not involve modifying data pages; it only updates compute-side metadata in LMT. Replicas always obtain a consistent page state either by (i) applying the pushdown logs propagated from the primary when the page is already resident in the buffer pool, or (ii) fetching the latest page directly from the storage layer, which always holds the ground-truth page contents.

Failover Safety. Eligibility checks can fail during failover. On the primary node, LMT is updated on page fetch and accurately records each page's free-space metadata. Moreover, `pushdown_entries` exist only for pages not resident in the buffer pool: when a page is fetched, any corresponding pushdown entries are cleared, so buffer-resident pages and pushdown entries never coexist. This invariant can be violated when a read-only replica is promoted to primary during failover. Replicas do not execute any writes; they primarily replay redo logs from the primary and do not maintain pushdown metadata in the same way as the primary does. As a result, after promotion, the new primary may have pages resident in the buffer pool while stale pushdown entries for those pages still remain. If a write targets such a cached page, it updates the page and its free-space bitmap, but does not clear the stale pushdown entry. The bitmap now reflects the latest free space, whereas the pushdown tree still has leftover entries. Subsequent pushdown-eligibility checks may therefore compute an incorrect free-space estimate and make wrong pushdown decisions.

To solve the problem, we introduce a `pushdown_ready` state. Upon promotion, the new primary starts in a restricted mode and immediately launches a background reconciliation task that reclaims all pushdown entries and updates the bits in the bitmap for corresponding data pages. Only after this process completes does the system enter `pushdown_ready` and start to serve write requests.

4.5 Fetch-After-Write (FAW) Optimization

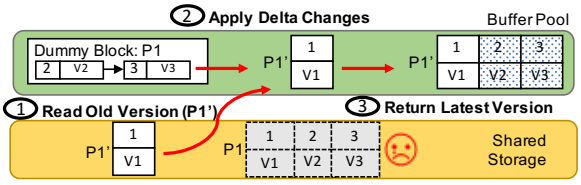

Figure 7: Fetch-After-Write (FAW) Optimization Workflow

A key challenge in pushdown architectures is the occurrence of Fetch-After-Write (FAW) stalls when a page fetch operation immediately follows pushed writes to the same page. The fetch may block while waiting for the page to be ready. We address this issue through an FAW optimization that leverages a short-lived local buffer at the compute layer to hold and replay pending updates without waiting for storage-side replay.

4.5.1 Motivation Scenarios. In write-intensive workloads, such as the oltp_update_index benchmark, systems frequently encounter patterns where a record is read immediately following a write. For example, an UPDATE operation is often decomposed into a delete-mark followed by an insert of the same key. While the delete-mark is typically eligible for pushdown, the subsequent insert may necessitate a remote page fetch from the storage layer if the target page lacks sufficient free space or if the local pushdown tree reaches its capacity threshold.

This creates a high-latency FAW stall. Subsequent fetch requests often enter a spin-wait loop (while-loop) while waiting for the Page Store to reconcile the page version, significantly increasing I/O tail latency. Given that page-full conditions and buffer thresholds are frequent and unpredictable in high-concurrency environments, this stall represents a critical performance bottleneck.

4.5.2 Solution. To eliminate the latency associated with storage-side asynchronous execution, we implement a localized, on-demand replay mechanism as shown in Figure 7.

- Read-Point Persistence: During the initialization of a Dummy Block, the WOP framework records the current page version (LSN; before eviction, LSN will be recorded in memory). This version number serves as a reference point for retrieving the 'stale' but physically consistent version of the page from storage.
- PushLog Attachment: Every time a PushLog is generated, the framework attaches this PushLog directly to the associated Dummy Block. This ensures that all 'in-flight' modifications

are accessible to the compute node before they are physically merged in the storage layer.

- Version-Aware Fallback Reading: When an I/O request detects that the required page version (LSN) exceeds the log-shipping LSN (the latest LSN acknowledged by the Page Store), the system performs a fallback read using the LSN recorded in the Dummy Block to fetch the base page version.
- In-Memory Local Replay: Upon retrieving the base page from the storage layer, the compute node immediately applies all Redo records attached to the Dummy Block in a single batch. This localized materialization allows the system to reconstruct the most recent visible version of the data page without waiting for the Page Store's asynchronous execution, thereby drastically reducing FAW stalls.

For safety, we bound the memory used by PushLogs attached to Dummy Blocks. Specifically, we limit their total size to 1/32 of the buffer pool, which is sufficient to eliminate FAW stalls in our evaluation (in practice, less than 1/1000 of the buffer pool sufficed). PushLogs are reclaimed when the corresponding data page is loaded into the buffer pool or when the Dummy Block is evicted.

5 EXPERIMENT

In this section, we evaluate the performance of the Write Operation Pushdown Framework.

5.1 Experiment Setup

Workloads. Table 3 summarizes the workloads used in our evaluation. We include four write-intensive workloads from sysbench[16], configured with 6 secondary indexes on the table (Listing 1 shows our table schema) to systematically stress different writing patterns in a controlled setting. Every workload is run with 64 concurrent threads. In addition, we also evaluate WOP on three large-scale production workloads from ByteDance spanning recommendation, financial processing, and advertising services.

Environment. Our experimental cluster consists of disaggregated compute and storage nodes, including 1 compute node and 8 storage nodes (4 for log store and 4 for page store). Each compute node is equipped with a 16-core Intel® Xeon® CPU, 64 GB memory, and a 270 GB SSD. Each storage node is provisioned with a 96-core Intel® Xeon® CPU, 1007 GB of memory, and 55 TB of SSD storage.

Methodology. We evaluate the performance of WOP using the following metrics: throughput (QPS, queries per second; TPS, transactions per second), throughput over time, average latency, tail latency (99^{th} percentile), and read page QPS (the number of pages that are fetched from Page Store per second). Higher throughput and lower latency reflect improved system scalability and efficiency. Throughput over time captures performance stability under sustained execution. For read-page QPS, lower values indicate reduced remote page fetches and improved I/O efficiency. We compare WOP with the baseline system without write operation pushdown, where each write goes through the traditional fetch-modify-write path. Each experiment runs for 300 seconds under sustained load to report the throughput over time behavior.

We list the system names that appear in our experimental plots in Table 4.

Table 3: Workload Description

Type	Workload	Datasets / Throughput	Description
Synthetic	OLTP Insert	8 tables with 352 GB data in total	Insert new rows with random data
	OLTP Delete		Delete rows by primary key
	Update Index		Update indexed columns
	Write-Only		Indexed updates + delete/insert pairs
Production	Douyin Content Distribution	14 TB, ~4K TPS	Ingests large volumes of data
	Financial Transaction	110 TB, ~1.5K TPS	Serves latency-sensitive user transactions
	Ad Delivery System	15 TB, ~150 TPS	Performs daily archival operations

```
CREATE TABLE sbtest (
  id  INT PRIMARY KEY AUTO_INCREMENT, k INT NOT NULL,
  c CHAR(120) NOT NULL, pad CHAR(60) NOT NULL,
  KEY (k), (c), (pad), (k,c), (k,pad), (c,pad)
) ENGINE=InnoDB;
```

Listing 1: Schema of the default table

Table 4: Names used in the experimental plots

Name	Description
Baseline	ByteNDB with traditional write path
WOP	ByteNDB enhanced with WOP framework
WOP+LMT	WOP with the Lightweight Metadata Tracker
WOP+Old	WOP using the original full-record format
QPS	Queries per second
TPS	Transactions per second
Read-Page QPS	The number of pages that are fetched from Page Store per second
FAW	Fetch-After-Write Operation
FAWO	Fetch-After-Write Optimization

5.2 Experiment Outline

We evaluate the WOP framework in the following aspects:

(1) **End-to-End Performance.** We compare WOP against the baseline under synthetic and production workloads, measuring throughput, latency, and page-fetch activity to quantify write-path improvements.
(2) **Metadata Efficiency.** We assess the impact of our Lightweight Metadata Tracker by comparing it with full-record buffering. We measure throughput, latency, and the number of pushed writes to characterize how the metadata footprint affects pushdown capacity and the overall system's performance.
(3) **FAW Optimization.** We evaluate WOP under controlled fetch-after-write intensity to examine its ability to mitigate write-induced stalls.
(4) **Sensitivity Analysis.** We study the impact of buffer pool size and secondary index count to understand the robustness of WOP under varying system configurations.
(5) **Impact on Read Performance.** We measure read throughput and latency under workloads with a mixed read-write ratio to determine how write pushdown affects concurrent reads.

5.3 End-to-End Performance

5.3.1 ***Production Workloads.*** First, we analyze the performance gain of three representative production workloads within ByteDance that span all stages of the data lifecycle (§3.2). The Douyin Content Distribution system ingests large volumes of data, the Financial Transaction database serves latency-sensitive user

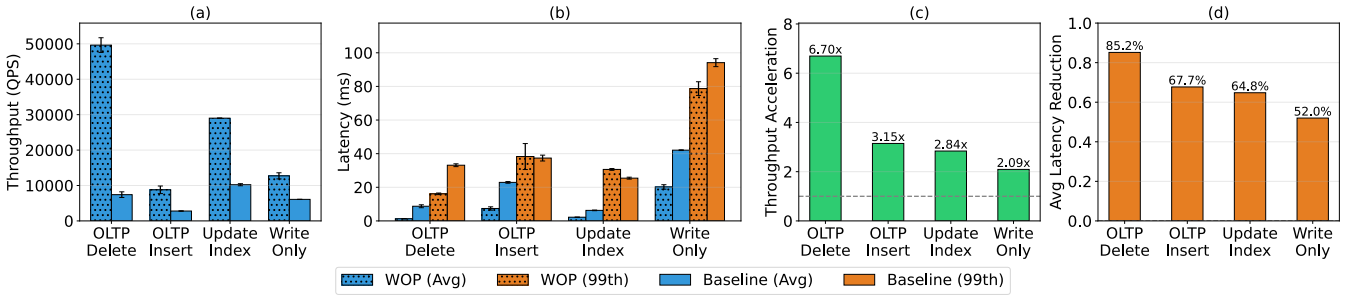

Figure 8: End-to-End performance comparison under different write workloads

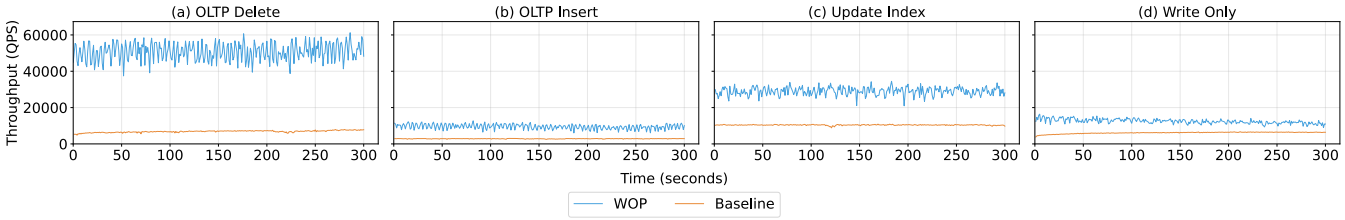

Figure 9: Throughput over time under different write workloads

Table 5: Production Performance Gains

Workload	Lifecycle Stage	TPS		Read Page QPS	
		Baseline	WOP	Baseline	WOP
Douyin Content Distribution	Ingestion	4K	12K	10K	5K
Financial Transaction DB	Serving	1.5K	3.1K	1K	500
Ad Delivery	Archiving	150	500	20K	10K

transactions, and the Ad Delivery workload performs daily archival operations. The results are reported in Table 5. WOP achieves 3.33× (Ad Delivery System), 2.07× (Financial Transaction DB), and 3.0× (Douyin Content Distribution) throughput improvements across the three workloads. Page read QPS decreases by 50% for all three workloads, which means WOP effectively reduces the number of pages read from Page Store. Larger gains are observed in the Ad Delivery System and Douyin Content Distribution workloads, which incur higher page-read intensity per transaction. Prior to WOP, each transaction reads approximately 134 pages in the Ad Delivery System, 2.5 pages in Douyin Content Distribution, and 0.7 pages in the Financial Transaction workload. Since WOP eliminates write-induced page reads, workloads with more page reads per transaction benefit more substantially from pushdown.

5.3.2 ***Synthetic Workloads.*** This section evaluates the efficiency and stability of WOP across four sysbench write workloads. For end-to-end throughput and latency comparison, we conduct three independent runs and report the average and standard deviation of the measured metrics. Overall, WOP achieves 2.09× to 6.70× throughput boost (Figure 8(c)) and reduces average latency by 52.0% to 85.2% (Figure 8(d)). It also maintains stable performance over time (Figure 9).

The largest end-to-end improvement is observed in the OLTP Delete workload. This workload benefits most because I/O latency dominates delete-mark's execution time. Delete-mark operations are page-internally simpler: they only mark records as deleted, whereas insert and delete require record placement and additional page-level metadata updates. Therefore, a large portion of the latency comes from I/O. Eliminating fetch-before-write I/O time, therefore, yields larger gains. To verify it, we conducted detailed monitoring of OLTP Delete and Write-Only workload and found that they show similar read-page latency, but delete operations exhibit substantially lower query latency. This indicates that I/O-intensive operations benefit more from WOP. Additionally, we observe a slight increase in tail latency—for example, in the Update Index workload, tail latency rises from 25.44 ms to 30.63 ms. This is because, in the worst case, when a fetch stall introduced by WOP cannot be mitigated by the FAW optimization (§4.5) once the memory threshold is reached, the operation must wait until the page becomes available in the Page Store, resulting in increased latency.

5.4 Metadata Efficiency

One of the key optimizations of our framework is the *Lightweight Metadata Tracker*. To quantify its impact, we compare two configurations: (1) WOP+LMT: WOP with the Lightweight Metadata Tracker, (2) WOP+Old: WOP using the original full-record format. The results (Figure 10) demonstrate that LMT consistently improves write efficiency across various write workloads.

The improvement arises from increased buffering capacity at the compute layer. By storing only minimal metadata rather than a full record, LMT allows substantially more eligible writes to be pushed down. During the same interval, LMT sustains 6,840,740 buffered pushdown entries, compared to 2,614,475 under the original format.

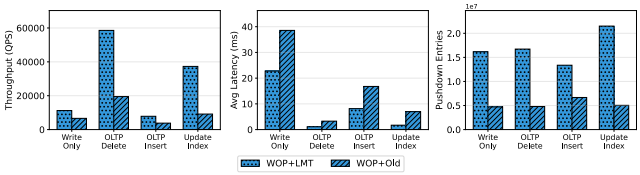

Figure 10: Metadata efficiency comparison

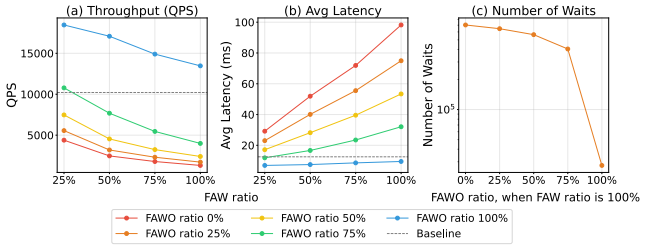

Figure 11: Performance comparison under varying proportions of Fetch-After-Write (FAW) operations.

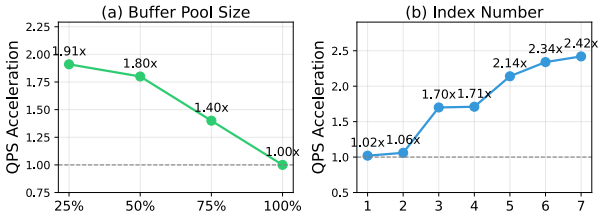

Figure 12: Sensitivity analysis.

5.5 Fetch-After-Write (FAW) Optimization

This section evaluates the performance of the Fetch-After-Write (FAW) optimization. We evaluate this mechanism using the oltp_update_index sysbench workload, where each update consists of a delete of key k followed by an insert of k+1; both operations are highly likely to target the same index page (except for the case when k is the last key in the page). We introduce two knobs to systematically evaluate the effectiveness of FAW optimization under different read-after-write intensities:

(1) FAW ratio: the ratio of FAW stalls in the write workload.

(2) FAWO ratio: what percentage of FAW is optimized.

We control the FAW ratio by always pushing down the delete k operation and only pushing down the insert k+1 operation with a certain probability (0%, 25%, 50%, 75%, 100%). In this way, when the insert k+1 operation is not pushed down, it will trigger an immediate page fetch, which leads to a FAW stall. FAWO ratio is controlled by probabilistically determining whether FAWs are resolved using the FAW optimization or by synchronously fetching the latest page from storage. In the latter case, the read will be blocked until all PushLogs associated with this page are merged at the Page Store. Figure 11 reports the results. The wait events are computed as $wait\_log\_flush + wait\_page\_ready$, representing the total number of read requests that are blocked while waiting for PushLogs to either be shipped to the Page Store or be merged into the latest version of the page in the Page Store.

These results show two clear trends. First, performance degrades monotonically as the proportion of FAW operations increases, and when the FAW is not fully covered by the FAWO optimization, the system's QPS is worse than the baseline, confirming that poorly handled FAW could significantly degrade WOP's effectiveness. Second, performance improves monotonically as the FAW optimization ratio increases, demonstrating that resolving FAW locally is highly effective. FAW optimization improves the system's performance by reducing the number of read requests that are blocked while waiting for the page to be ready. We report the number of wait events when the FAWO is fixed at 100% in Figure 11(c). The number of wait events decreases from 704,195 (0% FAW optimized) to 27,542 (100% FAW optimized), which is a 96.1% reduction.

5.6 Sensitivity Analysis

We evaluate the robustness of WOP under varying buffer pool sizes and numbers of secondary indexes. These parameters directly affect I/O intensity and write amplification, thus providing insight into how the framework behaves under different memory and network pressures. The workload we use here is `oltp_write_only`.

5.6.1 Impact of Buffer Pool Size (Figure 12(a)). We vary the buffer pool size from 25% to 100% of the dataset size. Our first observation is that even when memory pressure is reduced (75% dataset can be cached in buffer pool), WOP still achieves good performance improvement (1.40×). Overall, the performance gains become more pronounced under higher buffer pool contention. When memory is constrained, page misses increase, and the baseline incurs more synchronous fetch-before-write stalls. By eliminating these remote page fetches from the critical path, WOP provides larger benefits.

5.6.2 Impact of Number of Secondary Indexes (Figure 12(b)). We next vary the number of secondary indexes from 1 to 7. Increasing the number of secondary indexes amplifies write cost, as each logical update must propagate to more independent index structures, thus leading to more page updates. Overall, as the index count increases, acceleration rises steadily, reaching 2.42× at 7 indexes. This monotonic increase in acceleration demonstrates that WOP becomes increasingly advantageous as write amplification grows.

Across both dimensions, WOP remains consistently advantageous. Its performance benefit does not rely on specific memory sizing or indexing, but instead scales with increasing system load.

5.7 Impact on Concurrent Read Performance

This section studies the impact of WOP on concurrent read performance. We run a mixed read-write workload by fixing 32 write threads (Write-Only) and varying read threads from 8 to 64. Table 6 reports read performance and read-page QPS from read operations, and Figure 13 shows the acceleration.

WOP leads to a modest but consistent improvement in read throughput and latency across all configurations. This is because WOP saves I/O bandwidth for concurrent read operations, as confirmed by an increase in read-page QPS from read operations in Table 6. We further observe that the relative acceleration decreases

Table 6: Read Performance Comparison

Threads	Throughput (QPS)		Avg Latency (ms)		Read Pages from Read Operations	
	Baseline	WOP	Baseline	WOP	Baseline	WOP
8r	6,792.2	9,942.7	16.49	11.26	**15,247,465**	**20,930,699**
16r	16,152.1	23,135.7	13.87	9.68	**35,720,895**	**48,566,586**
32r	42,756.3	56,645.2	10.48	7.91	**91,543,770**	**118,198,673**
48r	62,525.4	73,837.4	10.74	9.10	**130,406,717**	**153,743,918**
64r	73,664.6	80,367.4	12.16	11.15	**152,287,632**	**167,200,904**

as the number of read threads increases. This behavior follows from the fact that the I/O savings introduced by WOP are shared across all read threads; as concurrency increases, the marginal benefit per thread correspondingly decreases.

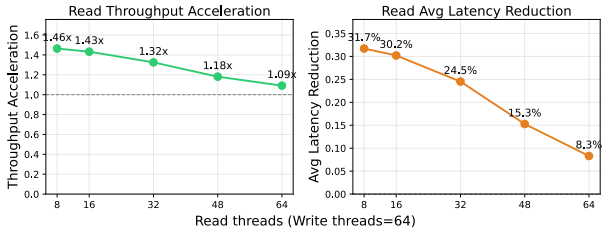

Figure 13: Impact of WOP on concurrent read performance

6 RELATED WORK

Write-Optimized Disaggregated Databases. Many disaggregated OLTP databases adopt the *log is the database* paradigm, persisting only the redo log as the source of truth and materializing data pages lazily at the storage tier. This avoids frequent full-page writes and the associated network amplification. Amazon Aurora [36], Socrates [4], and Taurus [12] adopt variants of this design, as does ByteNDB itself (§2.2). This architecture lightens the write path by changing *what* is persisted and *where* pages are constructed, but a fine-grained in-place update still requires the target page to be loaded into the compute-tier buffer pool before the modification can be applied. WOP builds on this architecture and pushes it further: it removes the read-side page fetch from the write path altogether.

Computation Pushdown and Write Offloading Compute-storage disaggregation has motivated pushdown techniques that move computation to the storage layer[40]. However, existing work primarily focuses on read pushdown or coarse-grained write offloading. Systems such as Amazon Aurora (via parallel query execution)[3], PolarDB (with computing pushdown)[7], and Taurus (through near-data processing)[19] increasingly offload operations such as filtering, projection, and aggregation to the storage layer, enabling more efficient data processing closer to where data resides. For write offloading, Socrates offloads bulk write operations, such as index creation and database reorganization, to page servers to reduce compute-layer load. Disaggregated LSM-based systems offload compaction tasks to smart storage devices or dedicated storage servers to avoid transferring large SSTables across the network[13][33][22]. In contrast, WOP differs by enabling fine-grained, more frequent and latency-sensitive write pushdown, eliminating remote page fetches from the critical path without compute-layer buffering.

Deferred-Write Data Structures Deferring the in-place application of an update—recording the modification now and applying it to its target location later—is a long-standing idea dating back to differential files, which keep recent changes in a side structure merged into the main database later [32]. The B^+-tree family has pursued these ideas extensively: InnoDB's Change Buffer (§3.3) defers updates to the non-resident secondary-index pages [20, 24, 25]; other write-optimized B^+-tree variants, such as the B$^\epsilon$-tree, reserve part of each node to buffer pending updates and flush them lazily down the tree [5]; and the BW-Tree [18] eliminates in-place updates through delta-record chains over a mapping table, as adopted in Cosmos DB [37]. Outside the B^+-tree family, Log-Structured Merge (LSM) trees[27] or their variants[31][39][30] are out-of-place by construction, which inherently bypasses the cost of FBW. MyRocks[21], for example, adopts an LSM-tree architecture to store both user data and indexes, appending updates to memory-resident structures (memtable) and performing compaction asynchronously. Out-of-place designs are inherently write-optimized and reduce sensitivity to remote I/O, making them well-suited for disaggregated deployments. In practice, the two families increasingly coexist—MySQL's pluggable engines run InnoDB and MyRocks side by side, and recent work transitions between the layouts online [15] or stacks a B^+-tree over an LSM base [38]. However, ByteNDB remains based on a B^+ tree architecture. First, B^+-trees offer superior read latency and range-scan efficiency for OLTP workloads dominated by point lookups and ordered access[9]. Second, transitioning to an LSM-style engine would require fundamental changes to indexing, concurrency control, and recovery, posing significant engineering complexity for an existing production system. Our WOP framework aims to achieve the write efficiency typically associated with out-of-place updates while preserving the structural and performance benefits of a B^+ tree-based engine.

7 CONCLUSION

We study a set of production workloads and identify a fundamental inefficiency of the traditional Fetch–Modify–Write (FMW) paradigm in compute–storage disaggregated architectures, where cache-miss page fetches incur remote I/O. We propose Write Operation Pushdown (WOP), which redesigns the write path to eliminate fetch-before-write via four key components: a PushLog-based execution model, a Lightweight Metadata Tracker that increases pushdown capacity with low memory overhead, consistency-preserving mechanisms for distributed deployments, and a localized on-demand replay mechanism that mitigates Fetch-After-Write stalls. Across synthetic and production workloads, WOP delivers substantial throughput improvements and latency reductions, demonstrating that fine-grained write pushdown can significantly improve write scalability while preserving correctness in disaggregated databases.

REFERENCES

[1] Amazon Web Services. 2018. Planning and Optimizing Amazon Aurora with MySQL Compatibility. https://aws.amazon.com/blogs/database/planning-and-optimizing-amazon-aurora-with-mysql-compatibility-for-consolidated-workloads/. Accessed: 2026-02-28.

[2] Amazon Web Services. 2019. Best Practices for Amazon Aurora MySQL Database Configuration. https://aws.amazon.com/blogs/database/best-practices-for-amazon-aurora-mysql-database-configuration/. Accessed: 2026-02-28.

[3] Amazon Web Services, Inc. 2025. *Parallel query for Amazon Aurora MySQL.* Amazon Web Services. https://docs.aws.amazon.com/AmazonRDS/latest/

AuroraUserGuide/aurora-mysql-parallel-query.html Accessed: 2026-03-02.

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[5] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 546–554.

[6] ByteDance. [n.d.]. veDB for MySQL. https://www.volcengine.com/product/vedb-mysql. Accessed on 2026-02-28.

[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.

[8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[9] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[11] Deep Focus Finance. 2026. *Where Will the Live-Streaming E-commerce Industry Head in 2026 as It Enters a More Competitive Phase?* 36Kr Global. https://eu.36kr.com/en/p/3639239270960515 Discusses the transformation of live-streaming e-commerce from traffic dependence to value competition, focusing on product, technology, and compliance. Accessed on 2026-03-03.

[12] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.

[13] Chen Ding, Jian Zhou, Kai Lu, Sicen Li, Yiqin Xiong, Jiguang Wan, and Ling Zhan. 2024. D2Comp: Efficient Offload of LSM-tree Compaction with Data Processing Units on Disaggregated Storage. *ACM Transactions on Architecture and Code Optimization* 21, 3 (2024), 1–22. https://doi.org/10.1145/3656584

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

[15] Varun Jain, James Lennon, and Harshita Gupta. 2019. LSM-Trees and B-Trees: The Best of Both Worlds. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1829–1831. https://doi.org/10.1145/3299869.3300097

[16] Alexey Kopytov. [n.d.]. SysBench Manual. https://github.com/akopytov/sysbench. Official GitHub repository, accessed on 2026-02-28.

[17] Frank L. 2026. *Livestream Shopping Key Statistics & Growth Trends 2026*. Getstream.io. https://getstream.io/blog/livestream-shopping-statistics/ Accessed: 2026-03-02.

[18] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

[19] Shu Lin, Arunprasad P Marathe, Per-Åke Larson, Chong Chen, Calvin Sun, Paul Lee, Weidong Yu, Jianwei Li, Juncai Meng, Roulin Lin, et al. 2022. Near Data Processing in Taurus Database. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1662–1674.

[20] MariaDB Foundation. [n.d.]. InnoDB Change Buffering. https://mariadb.com/kb/en/innodb-change-buffering/. MariaDB Knowledge Base, accessed on 2026-02-28.

[21] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[22] Sungho Moon, Hera Koo, Hyeongjun Jeon, and Beomseok Nam. 2022. LSM Tree Compaction Offloading Using NVMe-oF. *Journal of KIISE* 49, 7 (2022), 569–577. https://doi.org/10.5626/JOK.2022.49.7.569

[23] MySQL Monthly. 2016. MySQL Monthly. Online: http://mysql.taobao.org/monthly/2016/05/01/. Accessed: 2026-03-03.

[24] Oracle. [n.d.]. MySQL 8.0 Reference Manual: 15.5.2 Change Buffer. https://dev.mysql.com/doc/refman/8.0/en/innodb-change-buffer.html. Accessed on 2026-02-28.

[25] Oracle Corporation. [n.d.]. MySQL 8.4 Reference Manual: 17.5.2 Change Buffer. https://dev.mysql.com/doc/refman/8.4/en/innodb-change-buffer.html Accessed: 2026-03-03.

[26] Oracle Corporation. [n.d.]. *MySQL Reference Manual: InnoDB Startup Configuration*. Oracle. https://dev.mysql.com/doc/refman/8.0/en/innodb-init-startup-configuration.html Default `innodb_page_size` is 16384 bytes (16 KB). Accessed: 2026-03-03.

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

[28] PingCAP. 2024. *Mastering Indexing in TiDB for Optimal Query Performance*. PingCAP. https://www.pingcap.com/article/mastering-indexing-in-tidb-for-optimal-query-performance/ Section 'Key Features of Indexing in TiDB' discusses secondary and covering indexes for query optimization. Accessed on 2026-03-03.

[29] PostgreSQL Global Development Group. [n.d.]. *PostgreSQL Documentation: Database Page Layout*. https://www.postgresql.org/docs/current/storage-page-layout.html Database pages are fixed-size, typically 8 KB. Accessed: 2026-03-03.

[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[31] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[32] Dennis G. Severance and Guy M. Lohman. 1976. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems* 1, 3 (1976), 256–267. https://doi.org/10.1145/320473.320484

[33] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. 2019. Collaborative Compaction Optimization System using Near-Data Processing for LSM-tree-based Key-Value Stores. *J. Parallel and Distrib. Comput.* 133 (2019), 29–43. https://doi.org/10.1016/j.jpdc.2019.06.002

[34] Jason Sun, Haoxiang Ma, Li Zhang, Huicong Liu, Haiyang Shi, Shangyu Luo, Kai Wu, Kevin Bruhwiler, Cheng Zhu, Yuanyuan Nie, et al. 2023. Accelerating cloud-native databases with distributed pmem stores. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3043–3057.

[35] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[36] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[37] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, 473–488. https://doi.org/10.1145/3183713.3196895

[38] Suzhen Wu, Zuocheng Wang, Maoxin Ye, Shengzhe Wang, Jiahong Chen, Chunfeng Du, Jie Zhang, Ke Zhou, and Bo Mao. 2025. BL-Tree: The Best of Both Worlds by Combining B^+-Tree on Top and LSM-Tree on Bottom. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 1910–1923. https://doi.org/10.1109/ICDE65448.2025.00146

[39] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[40] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (2024), 1643–1670. https://doi.org/10.1007/s00778-024-00867-8