CloudJump III: Optimizing Cloud Databases for Tiered Storage

Zongzhi Chen
Alibaba Cloud
Hangzhou, China
zongzhi.czz@alibaba-inc.com

Mo Sha
Alibaba Cloud
Singapore
shamo.sm@alibaba-inc.com

Feifei Li
Alibaba Cloud
Hangzhou, China
lifeifei@alibaba-inc.com

Sheng Wang
Alibaba Cloud
Singapore
sh.wang@alibaba-inc.com

Baolin Huang
Alibaba Cloud
Hangzhou, China
baolin.hbl@alibaba-inc.com

Guoqing Ma
Alibaba Cloud
Hangzhou, China
guoqing.mgq@alibaba-inc.com

Huaxiong Song
Alibaba Cloud
Hangzhou, China
songhuaxiong.shx@alibaba-inc.com

Ke Yu
Alibaba Cloud
Hangzhou, China
zhongbei.yk@alibaba-inc.com

Xizhe Zhang
Alibaba Cloud
Beijing, China
zhangxizhe.zxz@alibaba-inc.com

Yuan Wang
Alibaba Cloud
Hangzhou, China
jingxuan.wy@alibaba-inc.com

Abstract

We present **CloudJump III**, the third generation of Alibaba Cloud's CloudJump framework, which advances compute-storage disaggregation by integrating page-level, engine-integrated tiering into the database kernel. Modern cloud databases span heterogeneous storage layers, including local NVMe SSDs, remote high-performance block storage, and low-cost object storage, forming a natural hierarchy for hot and cold data separation. However, existing block-level or filesystem-level tiering lacks visibility into database semantics and often leads to suboptimal placement under dynamic OLTP workloads. Building on earlier generations that optimized I/O and version management, **CloudJump III** introduces an eviction-centric, engine-aware design that determines placement at buffer-manager control points including eviction and flush. It uses engine-visible metadata to balance performance and cost, while unifying data flow across tiers and coordinating with recovery and snapshot protocols to ensure crash-consistent, zero-downtime operation. Deployed in Alibaba Cloud's production MySQL-compatible service, **CloudJump III** achieves near-local throughput, reduces the fast-tier footprint, and maintains stable tail latency. These results demonstrate that engine-integrated tiering enables predictable performance and cost efficiency at production scale.

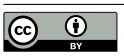
CCS Concepts

• **Information systems** → **Hierarchical storage management**; *Database management system engines.*

Keywords

cloud database, database storage management, tiered storage

ACM Reference Format:
Zongzhi Chen, Mo Sha, Feifei Li, Sheng Wang, Baolin Huang, Guoqing Ma, Huaxiong Song, Ke Yu, Xizhe Zhang, and Yuan Wang. 2026. **CloudJump III**: Optimizing Cloud Databases for Tiered Storage. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3788853.3803084



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803084

1 Introduction

Cloud databases increasingly separate compute from storage to achieve elasticity and cost efficiency [15, 32, 56]. This architectural shift broadens the range of available storage tiers [66, 87]. Beyond directly attached NVMe SSDs, providers now offer high performance remote SSD volumes connected through RDMA class networks [33, 45, 74] and virtually unlimited object storage accessible over Ethernet. Each tier occupies a specific point on the cost, latency, and throughput frontier [13]. Tiers that are closer to compute deliver lower latency and higher throughput but incur higher cost. As a result, storage expenditure, including DRAM, has become a dominant contributor to the total cost of ownership for database services [36, 49, 53]. The essential question is no longer whether to introduce tiering but how to place data across heterogeneous storage tiers so that performance and cost remain stable under dynamic OLTP workloads [9, 42, 43, 63, 86].

Effective tiering keeps hot data on faster media and cold data on lower-cost storage [9]. The primary challenge lies in robust operationalization and bounded tail latency at cloud scale. Cache misses on the fast tier incur substantial latency penalties from synchronous remote fetches, inflating latency and inducing significant jitter [37, 50]. Manual schemes (e.g., time-based partitioning or table-level reorganization) are coarse-grained and reactive, requiring observation and reconfiguration that lag behind minute-scale workload shifts and lead to uneven capacity utilization [41, 73, 76, 78, 83, 97]. Automated lifecycle management remains challenging [8, 12, 70]. Moving files or tables is costly and disruptive; fine-grained background migration introduces I/O contention, lock complexity, and correctness risks during crashes or backups [7, 85]. Operators need a tiering mechanism that preserves multi-tier performance and cost benefits, bounds miss-induced latency, and avoids additional operational complexity or correctness exposure.

Meeting these requirements is challenging. Most tiering solutions operate outside the database kernel, at block, filesystem, or

gateway layer [67, 99]. They observe I/O patterns but lack visibility into engine internal state [28], so hot pages in DRAM appear cold in trace-based analysis [55] while structurally important index or metadata pages are evicted prematurely. When decoupled from the kernel's recovery and backup logic, data movement cannot respect LSN-based invariants or snapshot protocols [62], undermining zero-downtime correctness. These limitations motivate an engine-integrated approach to tiering.

CloudJump is Alibaba Cloud's long-term initiative to rearchitect the storage substrate of its MySQL-compatible, cloud-native database engine. Over several years of continuous development and production deployment, the **CloudJump** series has advanced the boundary between compute and storage through kernel-level redesign. **CloudJump I** rebuilt the I/O path for remote, distributed cloud block storage, decoupling latency- and bandwidth-bound flows and mitigating shared-storage contention to restore predictable performance [19]. **CloudJump II** addressed inter-node consistency and recovery bottlenecks in shared-storage architectures by introducing Multi-Version Data (MVD), enabling page-level multiversion access on standard cloud storage across RW and RO nodes for high consistency, availability, and performance [20]. This paper presents **CloudJump III**, the third milestone in this line of innovation. **CloudJump III** moves tiering into the storage engine and takes placement decisions at eviction and flush using InnoDB-visible metadata, while coordinating with recovery and backup.

CloudJump III employs an eviction-centric, engine-aware design. Placement is decided at buffer-manager control points using signals such as page type and age, table identity with per-table quotas, temporary-table detection, and buffer-pool residency; these decisions govern admission and write routing across NVMe SSD, ESSD, and OSS tiers. The architecture forms a four-layer hierarchy reflecting the cost-latency gradient: at eviction, pages are retained or demoted based on engine-visible metadata; at flush, pages are routed to suitable tiers with intermediate buffering to merge writes and limit amplification. A snapshot-version protocol synchronizes placement with ongoing backups, ensuring crash-consistent evolution of tiered state. The mechanism reduces miss frequency and tail latency, maintains correctness by keeping placement independent of MVCC and WAL while coordinating with snapshot-based backup, and improves operability by reusing existing control paths instead of adding migration logic.

The design is shaped by two production constraints. First, correctness and operability: tier placement remains independent of transactional semantics, backup state, and crash recovery. Second, cloud economics: the system must deliver stable performance per unit of SSD capacity and low operator effort across multi-tenant fleets. Kernel-level integration keeps placement separate from LSN-based versions; eviction-centric control removes the need for migration threads and reduces locking cost; per-table quotas with deterministic IOPS control ensure fairness on fast tiers.

We integrated **CloudJump III** into Alibaba Cloud's production MySQL-compatible engine through targeted changes to the buffer manager, I/O subsystem, and metadata catalog. On production-grade hardware, engine-aware tiering sustains high throughput and stable latency with limited fast-tier capacity, approaching all-SSD performance at lower cost. Across workloads, **CloudJump III** maintains near-baseline throughput and bounded tail latency by decoupling remote writes from foreground transactions. Ghost-based admission, temporary-table exclusion, and DDL/BLOB bypass reduce remote I/O pressure and stabilize writes. The snapshot-version protocol ensures crash-consistent backups and predictable recovery without migration or downtime.

Table 1: Cloud Storage Tiers.

Tier	Latency	Typical size	Price (GB) per month
DRAM	80–100 ns	4–1024 GB	$1.6
Instance-attached NVMe	10–100 μs	1–32 TB	$0.05
Network block volumes	200–500 μs	1–128 TB	$0.152
Managed file systems	5–10 ms	10–100 TB	$0.03
Object storage	40–200 ms	unlimited	$0.016
Archive object tiers	mins/hours	unlimited	$0.003

The contributions of this paper are summarized as follows:

- We address a central challenge in cloud-native databases. We make tiered storage operationally viable on heterogeneous cloud media by moving placement into the storage engine and coordinating it with recovery and backup. The result is a balanced design that delivers predictable performance and cost while preserving availability and correctness under dynamic workloads.
- **CloudJump III** combines eviction-centric control with engine-aware placement and routing at eviction and flush; pairs this with adaptive write strategies and second-level write combining; and uses a snapshot-version protocol on standard object storage for zero-downtime backup with low write amplification.
- **CloudJump III** has been deployed in Alibaba Cloud's production MySQL-compatible service. Deployment and experiments validate improved throughput and tail latency at lower fast-tier footprint, robust crash recovery and zero downtime backup, and sustained stability at multi-tenant scale.

2 Preliminaries

2.1 Cloud Database Storage Hierarchy

Modern cloud databases operate on a layered storage hierarchy that includes DRAM, instance-attached ephemeral NVMe, networked block volumes, managed file systems, and object or archival stores. Although providers such as Alibaba Cloud, AWS, Google Cloud, and Microsoft Azure use different names for their tiers, their functional roles are consistent [2, 14, 84]. Table 1 summarizes typical latency, capacity, and cost ranges across these layers. DRAM provides nanosecond-level latency and the highest bandwidth but is volatile and limited to a single virtual machine, making it suitable for the buffer pool's hottest data. Instance-attached NVMe offers low latency and high IOPS without persistence or cross-node redundancy, fitting transient caches and temporary workspace rather than durable storage [59]. Network block volumes are backed by provider-managed systems with multi-replica redundancy and RDMA-class connectivity. They support configurable IOPS and throughput with improved fault isolation but are subject to per-volume quotas and rate shaping [2, 84]. Managed file systems allow shared POSIX, NFS, or SMB access with elastic capacity, though metadata operations and path traversal introduce additional overhead [14, 88]. Object storage provides high durability through

HTTP-based interfaces, with higher request latency that can be mitigated by batching or prefetching [5, 11, 27, 80]. Archive tiers achieve minimal cost but incur long retrieval times, serving long-term backup and compliance needs [65]. This cross-cloud comparison highlights a persistent tension in system design, where database engines must balance low-latency access to hot data with the need for large, durable capacity for cold data. As workloads evolve and tenant datasets grow, fixed allocations of DRAM and instance-attached SSDs become increasingly inadequate.

2.2 Current Practices and Their Limitations

Common deployments separate hot and cold data using mechanisms that span the block layer, filesystems, storage arrays, and cloud services. Typical patterns include host-level NVMe caches in front of slower volumes [60, 81] (dm-cache/bcache/LVM Cache), instance-attached SSDs used as transparent read caches for remote or shared storage (sometimes with selective write-back), Information Lifecycle Management/Hierarchical Storage Management (ILM/HSM) pipelines that migrate colder files to object tiers and recall on demand [4], array/cloud auto-tiering with performance classes and IOPS/throughput reconfiguration, file system-level tiering with front-end buffering, and object-centric designs that keep only the hottest blocks on SSDs while relegating the rest to object storage [79]. Intent is to keep the working set on fast media, batch or defer writes, and use cheaper tiers to reduce fleet cost. Engine-integrated approaches place cache management within the database engine rather than at the block layer, triggering writes to flash on buffer pool eviction [48]; they make page retention/demotion decisions within the buffer manager and engine path, emphasizing fine-grained throttling and background write coalescing to stabilize tail latency under high concurrency [54]; and use lightweight admission filtering in DRAM to batch sequential writes to SSD, reducing pollution, write amplification, and recall interference [35].

These mechanisms share several limitations. Cache effectiveness drops when the working set shifts quickly or scans dominate [6, 61]; pollution and thrashing inflate tail latency [16]; and cascading misses to slower tiers induce significant latency variance (jitter). Write-back trades foreground latency for higher write amplification and long recovery after unclean shutdowns [57, 58, 75], while cache warmup after reboot, resize, failover, or autoscaling can be prolonged [98]. ILM/HSM operates at coarse granularity: small reads trigger whole-file or large-extent recalls [5], and periodic scans or bulk moves induce bursty migration that competes with foreground I/O and can trigger bursts of recall traffic [52]. At the service level, exhaustion of I/O burst credits precipitates sharp throughput degradation; per-volume caps cause head-of-line blocking under spikes [10, 68, 96]; and cross-AZ paths add variance that appears as jitter even when medians are healthy [51]. Object-centric tiers add per-request overhead and higher access latency [80], and small random reads suffer from minimum part sizes and weak locality; lifecycle transitions, listing or metadata charges, and throttling exacerbate tail latency during promotions.

In sum, while hot–cold data separation is achievable, maintaining predictability at minute time scales remains challenging. Miss-induced penalties, cache pollution, credit or IOPS ceilings, cache warmup costs, and migration interference all undermine stability, making it challenging to sustain both performance and cost efficiency. An engine-integrated, eviction-centric design can mitigate and bound miss penalties to keep tail latency within targets, schedule background movement to prevent contention with foreground I/O [81], avoid coarse-grained recalls, operate within provider shaping limits, and decouple physical placement from transactional semantics and recovery or backup protocols [17, 84]. Compared with I/O-only external mechanisms, it achieves finer scheduling precision through engine-visible metadata and control at eviction and flush [29, 30, 48, 75], while coordinating with recovery and backup processes to lessen the effects of inevitable misses.

2.3 Operational Objectives and Scope

Engine-integrated tiering must deliver predictable performance and operability under dynamic OLTP workloads without coupling data placement with transactional semantics. The design objectives are to: (i) minimize harmful fast-tier misses and bound their tail-latency impact; (ii) constrain background data movement to avoid contention with foreground I/O; (iii) reduce write amplification through coordinated write combination and routing; (iv) preserve zero-downtime backup and crash-consistent recovery by maintaining placement orthogonal to MVCC and WAL while cooperating with snapshot protocols; and (v) improve multi-tenant cost efficiency by sustaining target throughput with a smaller fast-tier footprint. Multi-tenant fairness is also a primary goal: when local SSDs serve as shared caches, bandwidth allocation must prevent tenants from monopolizing fast-tier I/O. Placement decisions should respect per-table quotas and avoid complex, tenant-specific migration controllers that increase operational overhead.

The scope of this work is the storage engine. Decisions rely on engine-visible metadata and runtime signals such as page type and age, table identity and quotas, temporary-table detection, page cleanliness, and buffer-pool residency. The approach does not depend on query semantics or optimizer changes, requires no application hints, and targets transactional (OLTP) workloads on standard cloud storage tiers. The role of engine integration is to choose admissions and evictions at natural control points (eviction/flush), batch and rate-limit writes, and coordinate with recovery/backup so that the frequency and cost of misses are reduced and their effects contained, while maintaining correctness and operability at scale.

3 System Architecture

CloudJump III adopts an engine-integrated, tiered storage architecture with multi-level caching that aligns data placement with access locality and lifecycle stage, as shown in Figure 1. The design defines a clear boundary between the Cache Tier (Volatile) and the Storage Tier (Durable), each with coordinated semantics and explicit control points. Pages in the volatile tier are transient—loss of DRAM or SSD content does not compromise durability—while the persistent tier holds recoverable data across failures. The volatile tier, comprising the InnoDB Buffer Pool (DRAM) and the Buffer Pool Extension (BPE) on direct-attached SSDs, provides transient, low-latency caching for frequently accessed pages. The durable tier, consisting of the OSS Buffer on ESSD volumes (network-attached block storage via high-speed links), the remote OSS object store, and

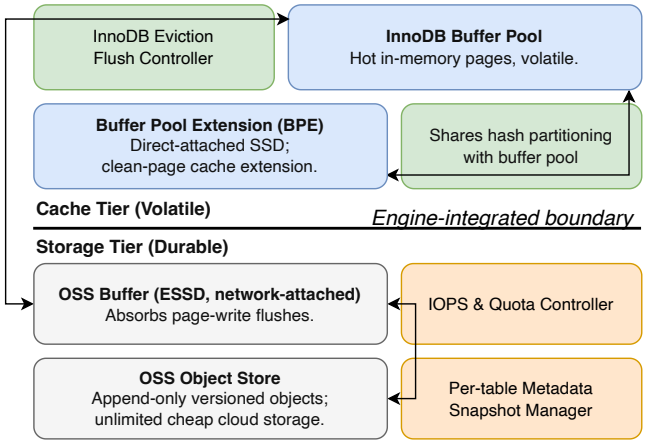


Figure 1: CloudJump III system architecture. The BPE is populated on buffer pool eviction (reuse-based admission); the diagram omits this input for clarity.

the metadata and snapshot subsystems, ensures persistence, versioning, and crash-consistent recovery. Dirty pages in the volatile tier become durable only when they are flushed to the OSS Buffer or to OSS. Until then, durability is guaranteed by the engine's write-ahead log (WAL): if the instance fails before a flush completes, recovery replays the WAL to restore consistency; once data resides in the durable tier, the corresponding WAL can be trimmed. The storage tier is thus decoupled from the WAL as placement and flush order are independent of the redo log, but the WAL remains responsible for durability until data reaches that tier. The architecture comprises four integrated components with well-defined roles:

InnoDB Buffer Pool serves as the traditional in-memory cache, holding hot pages and triggering eviction and flush operations at natural control points; although every tier can serve page data when appropriate, their primary roles differ.

Buffer Pool Extension (BPE) extends caching capacity on direct-attached SSDs and acts primarily as a read-mostly extension of the buffer pool; it caches clean pages and remains volatile. The BPE is populated exclusively from the buffer pool eviction path: when a page is evicted from the in-memory buffer pool, the engine decides whether to write it to the BPE or only to record its identity in a lightweight *ghost* list (no page data). Only pages that show reuse potential—e.g., re-accessed in DRAM or already present in the ghost list from a prior eviction—are written to the BPE; one-time pages are discarded from the SSD cache and only their identifiers enter the ghost list. Thus the data flow into the BPE is: Buffer Pool (eviction) → reuse check → BPE (if reuse) or ghost list only. When a dirty page is flushed from the buffer pool and that page resides in the BPE, the engine propagates the flush to the BPE, updating or invalidating the BPE copy according to the active write policy so that the BPE does not serve stale data. On a subsequent read, a BPE hit promotes the page back to the buffer pool.

OSS Buffer resides on ESSD volumes (network-attached block storage) and serves as the fast, redundant, durable block layer that absorbs page-write flushes; it aggregates dirty pages *within each 2 MB block* (one block corresponds to one OSS object; there is no cross-block aggregation) and makes them persistent before remote offload, reducing write amplification and shielding the object store from fine-grained updates.

OSS Object Store provides low-cost, durable remote storage for cold data, accessed via cloud-native APIs, and forms the final persistence layer; data is written as 2 MB versioned objects when OSS Buffer blocks are flushed.

The Metadata Subsystem and the Snapshot Protocol act as side control modules. Two concepts are central to the write path and backup. **Per-table metadata:** each OSS-backed table has a persistent metadata file (e.g., `.meta`) that, for every 2 MB object block, records a modification flag (whether the block was modified since the last backup) and the **current object version ID**. This metadata is updated when a block is flushed to OSS and is used during backup and recovery to determine which blocks to copy and which version to read. **Object versions:** each 2 MB OSS object can have multiple immutable versions, identified by monotonically increasing version IDs; a new write creates a new version, and older versions are reclaimed after backup. Together with a global snapshot version used at backup start, this enables point-in-time consistent snapshots and crash-consistent recovery. Further protocol details are in Section 4.3. In contrast to external tiering frameworks that operate outside the database kernel, **CloudJump III** introduces kernel-coordinated placement primitives integrated with recovery, snapshotting, and transactional semantics. This design eliminates dedicated migration controllers and user-tuned partitioning policies, reducing operational overhead in multi-tenant environments while preserving correctness and high concurrency.

3.1 Storage Tier Modules and Coordination

The storage tier decouples policy enforcement from data movement through engine facing boundaries. Two controllers, the IOPS and quota controller and the per table metadata and snapshot manager, coordinate admission, versioning, and background transfers with the OSS Buffer and OSS. This separation enables scalable movement across media while preserving correctness, consistency, and tail latency guarantees. BPE extends the in memory cache with directly attached SSD capacity, but remains volatile and does not contribute to durability. The storage tier modules are therefore presented in terms of their coordination with the cache.

The OSS Buffer sits at the entry of the storage tier on ESSD volumes (network-attached block storage). Placing it on network-attached rather than instance-attached (local) storage is necessary for durability: if the OSS Buffer resided on instance storage, its contents would be lost on node failure and could not be recovered without full replay from OSS. It buffers and coalesces updates before writing to remote object storage. For OSS backed tables, **CloudJump III** partitions each table file into 2 MB units. Each unit is stored as one *OSS object* in OSS and is mirrored by one *block* in the OSS Buffer. Here, an "object" denotes this 2 MB physical unit, not a logical table or partition. For example, a 2 GB table yields 1024 objects, each containing 128 pages of 16 KB. The OSS Buffer maintains one block per object in a block directory and caches hot pages and recent updates for that object. Objects never mix pages from different tables or discontiguous regions. On flush, the system writes exactly one 2 MB object, formed either by merging dirty pages with a prior OSS version or by writing the full block.

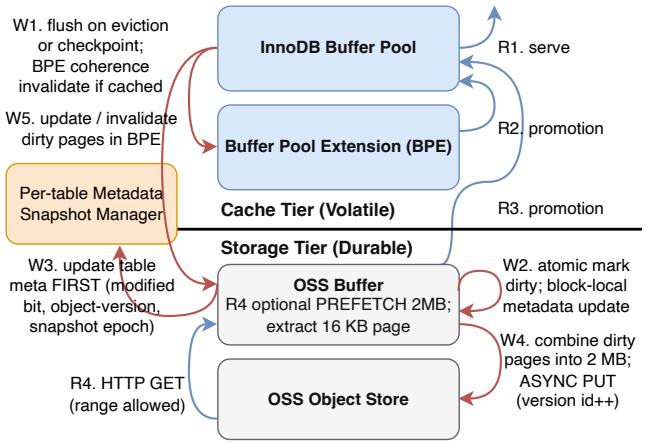


Figure 2: CloudJump III read/write path.

Each OSS Buffer block maintains metadata including its block ID (aligned with the object index), the covered page ID range, a monotonically increasing version counter, and per page dirty and residency state. Internally, the OSS Buffer mirrors InnoDB page management primitives, including a page to block hash for lookup, an LRU list for replacement, and a flush list for pending writes.

When a Buffer Pool page is evicted or checkpoint flushed, **CloudJump III** redirects it to the corresponding OSS Buffer block. The page is marked dirty atomically and scheduled for background flush with crash consistent metadata updates. Pages persisted to ESSD are durable at this tier because ESSD provides committed block storage with atomic visibility at block granularity, via atomic writes or pointer indirection as defined by the ESSD protocol. After persistence, the page can be recovered without WAL replay for that page. Remote durability is provided by the OSS object store, which stores versioned 2 MB objects through strongly consistent HTTP PUT and GET operations. Writes follow an append only versioning protocol with monotonically increasing identifiers, and obsolete versions are reclaimed asynchronously.

To manage write ordering and versioning, **CloudJump III** maintains a per table metadata file adjacent to each OSS backed table. For every 2 MB object, the metadata records a modification flag and the current object version that is visible to readers. During an OSS Buffer flush, **CloudJump III** first updates the corresponding metadata entry to record the new version and modification state, enabling backup and recovery to select the correct objects and versions. For backup consistency, the system maintains a global snapshot version number (`next_snapshot_version`). At backup start, this version is incremented and attached to subsequent OSS writes, distinguishing new writes from pre backup data. After completion, older versions are marked obsolete and their metadata entries become eligible for reclamation. Backup readers therefore observe a point in time consistent snapshot.

Together, these modules define a coherent data and control path from page creation to persistence and snapshotting. Each tier is tailored to its medium and tightly coordinated to preserve correctness and reduce tail latency. The resulting engine integrated architecture provides fine grained control over data residency and lifecycle across DRAM, SSD, and cloud object storage.

3.2 Read Path

CloudJump III adopts a hierarchical read pipeline that prioritizes fast near-end media and progressively falls back to lower tiers only when necessary. Figure 2 visualizes this hierarchy with upward blue arrows (R1–R4) denoting the data promotion flow, and small badges marking direct hits at each level. The lookup sequence proceeds from DRAM → direct-attached SSD (BPE) → OSS Buffer (ESSD) → remote OSS, yielding four distinct read paths.

R1. Buffer Pool hit. If the requested page resides in the in-memory Buffer Pool, it is served immediately with nanosecond-level latency. This case corresponds to the topmost blue arrow (R1) in Figure 2, indicating a local DRAM hit.

R2. BPE hit. On a Buffer Pool miss, the system probes the SSD-based BPE. If the page is present, it is promoted to DRAM and returned to the executor, using SSD latency while keeping cache coherence intact.

R3. OSS Buffer hit. If both DRAM and BPE miss, the request advances to the OSS Buffer on ESSD (network-attached). When the corresponding block is cached—either from recent writes or prefetching—the 16 KB page is extracted and promoted upward, as marked by the R3 arrow in Figure 2. This layer provides durable, block-level caching that bridges the gap between volatile and remote storage.

R4. Remote OSS read. If none of the local layers contain the page, the system performs a remote read from the OSS object store. The 16 KB page is retrieved through an HTTP GET request, optionally using range reads. To exploit spatial locality, the entire 2 MB object may be prefetched into the OSS Buffer.

Pages retrieved from non-DRAM tiers are reinserted into the Buffer Pool to accelerate access. Because the BPE and the OSS Buffer operate at page or block granularity, these tier transitions remain transparent to upper layers. The hierarchical read path thus maximizes hit probability on fast media while ensuring colder data remain efficiently accessible from lower tiers.

3.3 Write Path

Compared with single layer designs, **CloudJump III** uses a staged write path that balances performance, durability, and consistency. The pipeline spans multiple tiers, defers remote writes, and preserves correctness through metadata coordination and version control, as illustrated in Figure 2. Two write scenarios dominate.

(1) Buffer Pool Flush: When dirty pages in the Buffer Pool are evicted (e.g., due to updates, checkpoints, or memory pressure), they are directed to the storage tier. Instead of writing directly to remote OSS, **CloudJump III** first persists them to the corresponding OSS Buffer block on ESSD (W1: flush on eviction/checkpoint). On receipt, the OSS Buffer atomically marks the page dirty and updates block-local metadata (W2). The block is then durable on ESSD, but *no remote PUT is issued yet*: the OSS Buffer absorbs further updates to the same block and defers remote persistence. The per-table metadata update (W3) and remote persistence of the 2 MB object under a new object version (W4), as defined above, are triggered only when that OSS Buffer block is later evicted or flushed; during backups, the new version is tagged with the current snapshot version. When the flushed page resides in the BPE, the engine also

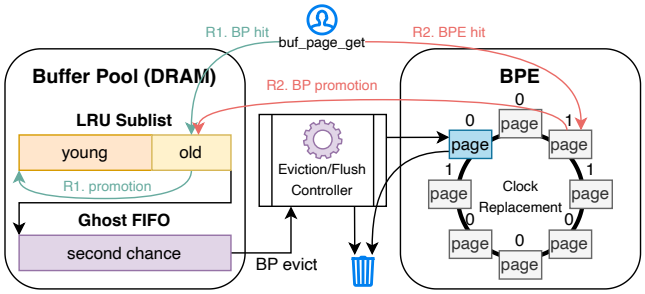


Figure 3: CloudJump III BPE Internal Structures and Flow.

propagates the flush to the BPE (W5): the BPE copy is updated with the new page content or invalidated according to the active write policy, so that the BPE does not serve stale data. This stage thus provides near-end persistence without issuing remote writes, enabling durable recovery while deferring high-latency operations and reducing write amplification.

(2) OSS Buffer Flush to OSS:. The final step asynchronously flushes OSS Buffer blocks to OSS when a block is evicted or explicitly flushed, rather than on every dirty page arrival. The system writes the entire 2 MB block as a unit, persisting the full object under a new version rather than only the dirty pages, using append only object versions (immutable writes with monotonically increasing version IDs). Flushes may be initiated by background threads (for example, during idle periods or scheduled batches) or synchronously when no clean OSS Buffer blocks remain available. Writes require strict coordination to preserve correctness across tiers because the same logical page may be present in memory, BPE, and the OSS Buffer. To prevent read after write inconsistencies, **CloudJump III** first updates the per-table metadata (the modification flag and object version ID for that block, as defined in Section 3) to record the new object version identifier and mark the block as modified. It then issues an HTTP PUT to persist the 2 MB object to OSS under that version. After the PUT completes, the system marks the OSS Buffer block clean and eligible for reuse.

This protocol ensures crash-consistent, monotonic version visibility: a block appears either at its old version or fully at the new version, even under failures or backup races. Figure 2 illustrates the write pipeline, showing W1–W2 (Buffer Pool to OSS Buffer), W3–W4 (OSS Buffer to OSS), and W5 (Buffer Pool to BPE when the page resides there). The ordered steps are annotated W1–W5 (top → down), with snapshot tagging marked where applicable. Overall, **CloudJump III** follows a staged durability model (Buffer Pool → OSS Buffer → OSS) with strong consistency guarantees and minimal remote write overhead. By leveraging local persistence and deferring high-cost operations, the system achieves operational efficiency and correctness under high-throughput OLTP workloads.

4 Implementation Details

4.1 Buffer Pool Extension (BPE)

As outlined in Section 3, the BPE is populated on the buffer pool eviction path under a reuse aware admission policy. Figure 3 summarizes the architecture and the resulting data and control flows of the Buffer Pool Extension (BPE) integrated with the InnoDB buffer pool. We build on two established ideas. First, InnoDB partitions the buffer pool into *young* (frequently accessed) and *old* (newly loaded) regions so that single pass scans do not displace the hot set. Second, the BPE on SSD uses a CLOCK style replacement policy (a circular list with a per page second chance bit) rather than strict LRU, reducing synchronization overhead and producing an eviction order that better matches SSD access patterns. The left side shows the DRAM buffer pool with three coordinated sublists: *young*, *old*, and *ghost*. The *young* and *old* sublists track age and recency, whereas the *ghost* sublist retains identifiers of recently evicted pages without their contents. This ghost layer complements the young/old design by capturing short lived evictions and enabling delayed, reuse aware readmission. The right side shows the BPE on SSD, centered on a hash indexed CLOCK structure that maintains coherence with DRAM. Overall, the design instantiates the selective admission and second chance principles of S3 FIFO [91], favoring rapid demotion of transient entries and promotion only after confirmed reuse.

4.1.1 Cache Coordination and Write Policy. As illustrated in Figure 3, each read first probes the buffer pool. On a hit, the page is served from DRAM and no list insertion occurs. On a miss, the request probes the BPE. A *BPE hit* promotes the page into the buffer pool *old* sublist rather than *young*. A *BPE miss* loads the page from the OSS Buffer or OSS and inserts it into the *old* sublist. A subsequent access in DRAM promotes the page to the *young* sublist, restoring it to the hot set. Accordingly, any page entering the buffer pool from outside DRAM first lands in *old*. It has not yet been re accessed in DRAM and may be a one time reference, so the design avoids treating it as hot prematurely. When a page is evicted from *old*, the system either admits it to the BPE or records only its identifier in the ghost list. Pages with reuse evidence, such as multiple accesses in the buffer pool or presence in the ghost list from a prior eviction, are written to the BPE. One hit pages absent from the ghost list bypass the BPE and only their identifiers are appended to the *ghost* list as metadata (no page data). The ghost list is a FIFO of evicted page identifiers, bounded by the BPE capacity. When a one-hit page is evicted from the buffer pool, it is written to the BPE only if its identifier already appears in the ghost list. This mechanism implements a reuse aware filter analogous to the S3 FIFO ghost queue, preventing SSD pollution by one hit pages while preserving accurate detection of high value reuse.

The Eviction/Flush Controller (gear icon) coordinates data and control flows between DRAM and the BPE. On a dirty flush, it propagates the flush to the BPE, invalidating or updating the BPE copy according to the active write policy. It also incorporates feedback on cache pressure and tenant quota to balance DRAM and SSD utilization. The controller enforces coherence across tiers while enabling fine grained adaptation without user intervention.

4.1.2 Dirty Page Management and Flow Control. For persistent tables, BPE retains only clean replicas of durable content. Dirty updates originate in DRAM and are synchronized during flush according to one of three policies:

- **Clean-Write:** The BPE retains only clean pages. When a page is flushed from the buffer pool, any cached copy in the BPE is invalidated to preserve consistency. This policy reduces SSD write amplification and is well suited for read-intensive workloads.

- **Dual-Write:** Both the storage layer and BPE are updated during flush. The freshly written clean page remains in BPE to improve reuse probability. This policy is the default for persistent tables.
- **Single-Write:** The page is written only to the BPE rather than to persistent storage. This mode applies to temporary or short-lived tables that can tolerate data loss, using the BPE as a transient write-back buffer to absorb write bursts.

The engine automatically selects the appropriate policy based on table type and checkpoint state. Decision rule: for persistent tables, Clean-Write is used under normal load (invalidate BPE copy on flush) to reduce SSD wear; Dual-Write is used when persisting dirty pages (update BPE in parallel) to maintain both durability and reuse; Single-Write (write only to BPE) applies to temporary data and is excluded from recovery. Operationally, each flush thus either invalidates the BPE copy, updates it in parallel with the storage layer, or writes only to the BPE, ensuring all BPE entries remain consistent with the persistent layer and avoiding read-after-write anomalies. **CloudJump III** maintains this consistency for durable data without manual tuning.

4.1.3 Eviction Policy and Replacement Algorithm. The right portion of Figure 3 also reflects BPE's internal organization and replacement logic. BPE adopts a modified CLOCK algorithm [61] that replaces LRU's linked lists with a circular pointer and a second-chance bit per page. Recently accessed pages are spared once; pages not re-accessed are evicted. This design lowers synchronization overhead and aligns scan order with SSD layout, enabling sequential writes and reduced wear. Several engineering optimizations enhance BPE's scalability and operational robustness:

- **Selective Admission:** Admits pages to the BPE only when reuse is indicated: on eviction from the *old* list, pages that were accessed more than once in the buffer pool or are already in the ghost list are written to the BPE; one-hit pages not in the ghost list are only recorded in the ghost list (no BPE write) and are admitted to the BPE only when a later access reloads them. This reuse-based admission eliminates transient pollution and aligns with the second-chance policy of S3-FIFO.
- **Metadata Compression:** Optimizes memory efficiency by compacting per-page control data and eliminating per-page locking structures. The metadata footprint scales linearly with capacity: a 64 GB BPE instance requires only about 184 MB of control metadata, enabling large-scale deployments without increasing memory pressure on the engine.
- **Chunk-based Resizing:** Divides the BPE space into fixed-size 64 MB chunks that can be added or released online within seconds. This design allows dynamic resizing in response to tenant load changes, enabling hot instances to temporarily expand cache space and later release it without restart, ensuring elasticity and predictable I/O performance across tenants.

Together, these mechanisms form the workflow illustrated in Figure 3: requests first probe DRAM, then BPE; BPE hits trigger page promotion to DRAM; evictions apply selective admission guided by the ghost sublist; and background flushes maintain BPE cleanliness and coherence. The result is a self-adaptive SSD cache that enhances read locality, reduces backend I/O amplification, and preserves full consistency with the buffer pool.

4.2 OSS Buffer

Figure 4 shows the OSS Buffer's internal data structures and asynchronous flush protocol. The left panel shows the in-memory management layer comprising a hash directory and per-block state (`Dirty` → `Clean` → `Free`), mirroring persistent metadata on network-attached ESSD. The right panel shows the three stages of the asynchronous flush pipeline: metadata update, data write, and cleanup.

4.2.1 Block Management and Metadata Structures. The OSS Buffer provides a durable staging cache between the engine and remote object storage. It absorbs small updates and coalesces them into block sized writes, reducing write amplification and shielding OSS from small random I/O.

For OSS backed tables, **CloudJump III** partitions each table into 2 MB objects aligned with InnoDB 16 KB pages. For each object, the OSS Buffer allocates a corresponding block on ESSD. A hash directory indexes blocks by mapping block IDs, aligned with OSS object indices, to control entries. Each entry stores persistent metadata: the block ID, the covered page ID range, the latest snapshot version, status flags (`dirty`, `valid`, `evictable`, `inflight`), and per page bits (dirty markers, modification timestamps, and validity).

In memory structures mirror the on disk state and include a page to block hash table, an LRU list for reclamation, and a flush list for pending I/O. The design parallels the InnoDB buffer pool, but operates at block granularity to match object storage semantics. On restart, the OSS Buffer rebuilds state by scanning table metadata and reading block descriptors, restoring the last committed view.

4.2.2 Write Path: Ingesting Page Updates. Figure 4 illustrates how page updates flow through the management pipeline. The OSS Buffer manages pages at 16 KB granularity via a page to block hash and per page metadata. It retains only pages written from the buffer pool or fetched by prefetch, rather than materializing all 128 pages of a 2 MB block in memory. When a dirty page is flushed from the buffer pool, the system resolves its target block through the hash directory. If the block is absent, the system allocates one from the LRU list and flushes it if dirty before reuse. Prefetch may fetch the OSS object, using linear or random reads, to supply unmodified pages when beneficial. Otherwise, such pages are not tracked in the OSS Buffer and are read from OSS during flush.

Metadata-first Update. Before writing page data, the block's metadata is updated to mark dirty status, assign a new version if necessary, and set validity flags. This meta→data ordering guarantees that every persisted block reflects a well-defined version boundary.

Table-level Coordination. When a block is modified under a new snapshot epoch, the table-level metadata file records its modified state and version identifier. This enables multi-version snapshots and ensures asynchronous flushes are version-aware.

After these steps, the page resides durably in the OSS Buffer's ESSD block with its version and dirtiness recorded, but no remote PUT is yet issued—preserving locality and enabling batching.

4.2.3 Crash Consistency and Write Ordering. To guarantee recoverability, the OSS Buffer enforces complementary ordering across local and remote writes: on local writes, metadata is updated before data to record intent; on remote flushes, data is transmitted before metadata is marked clean. This reversed ordering ensures idempotent replay. If metadata is updated but data missing, recovery

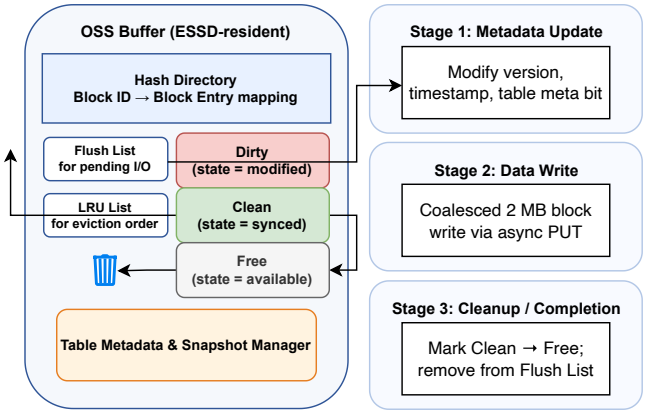


Figure 4: CloudJump III OSS Buffer block management.

replays redo logs. If data is present but metadata stale, the block is re-flushed. Only when both persist is the state set to `clean` (Stage 3 in Figure 4). This invariant guarantees each block is either fully durable or safely recoverable.

4.2.4 Asynchronous Flush and Eviction Strategy. Dirty blocks follow the three flush stages shown in Figure 4:

Stage 1: Metadata Update. The flush thread updates block metadata, records the current version, and marks the block as "in flight." It then notifies the per table metadata manager to record the updated snapshot epoch.

Stage 2: Data Write. The system writes a full 2 MB object to OSS. When pages are absent from the OSS Buffer, it fetches the current OSS object, merges the modified pages, and issues a single asynchronous PUT of the complete 2 MB object under the new version. Each flush carries a versioned object identifier to preserve object lineage. Dedicated I/O threads upload objects, decoupled from foreground transactions.

Stage 3: Cleanup and Completion. After the upload succeeds, the block transitions from `Dirty` to `Clean` and eventually to `Free`. It is removed from the flush list and the hash entry is updated. If a crash occurs after the data write but before metadata acknowledgment, recovery treats the block as dirty and replays the flush.

Adaptive flush scheduling combines batching, temporal deferral, watermark monitoring, and rate limiting. Batching aggregates dirty pages within each block (no cross-block aggregation) to improve write efficiency; temporal deferral avoids repeated writes on hot blocks; watermark control ensures background flushing before saturation; and rate limiting regulates throughput under OSS QPS and tenant-fairness constraints. Clean blocks become eviction candidates only after OSS synchronization, whereas dirty blocks must complete flushing before reuse; this policy ensures space is reclaimed only after persistence is guaranteed.

4.2.5 Consistency, Versioning, and Recovery. Consistency is maintained through metadata exchange with the table-level manager, which records object-version identifiers and snapshot epochs. This linkage enables crash-consistent versioning: during recovery, any partially flushed block is detected via a version mismatch and re-synchronized. As a result, **CloudJump III** ensures that every object visible to OSS is fully consistent with its recorded metadata, and recovery deterministically reapplies any pending flushes.

Through this pipeline, the OSS Buffer delivers durable, version-aware writeback with minimal disruption to foreground I/O. It merges database semantics with asynchronous cloud storage protocols, reducing remote write amplification while maintaining strong consistency and fast recovery semantics.

4.3 Snapshot Versioning and Backup Protocol

We specify the protocol for the per-table metadata and object versioning model introduced in Section 3. Figure 5 illustrates the end-to-end timeline of snapshot versioning and recovery. The horizontal axis traces five stages: normal operation, snapshot initiation, asynchronous OSS flush, crash/restart, and redo-based recovery. Three parallel lanes represent the coordination among the engine (OSS Buffer on ESSD), the table-level metadata manager, and the storage backends (EBS/ESSD snapshot and remote OSS). The evolution of `next_snapshot_version` and `invalid_snapshot_version`, together with the complementary ordering of metadata and data updates, ensures crash-safe backup and deterministic recovery.

4.3.1 Version Number Management. **CloudJump III** maintains a global counter `next_snapshot_version` that identifies the next snapshot epoch. When a backup begins, the system acquires a global OSS lock to block concurrent DDL and OSS writes, increments this counter, and tags the new value as the snapshot ID. From that point, all OSS-bound writes carry the current snapshot version.

For each OSS object block written to OSS (whether by flush or DDL), the write is tagged with the current snapshot version. If the snapshot version has not advanced since that object was last written, the write overwrites the existing object in place; if the global snapshot version has advanced (e.g., a new backup epoch has started), a new versioned object (e.g., `file_0002_v2`) is written and the prior version is retained for backup. During foreground execution, **CloudJump III** always reads the latest visible version of each object. The backup process, in contrast, explicitly reads the versions corresponding to its snapshot epoch, ensuring point-in-time consistency. Version tags across all OSS objects are coordinated through per-table metadata files.

4.3.2 Table Metadata Files. Each OSS-backed table maintains a persistent `.meta` file (the per-table metadata introduced in Section 3) that, for each 2 MB object block, records a dirty bit indicating whether the block was modified since the last backup and a version number capturing the most recently written snapshot epoch. If a block is updated without a snapshot version change, only the dirty bit is set. If written after a version increment, both the dirty bit and version counter are updated. This metadata enables precise change detection and determines which object version to copy. At backup completion, dirty bits are cleared and the `invalid_snapshot_version` watermark is advanced to mark older versions as safely reclaimable.

4.3.3 Backup Workflow. The backup procedure proceeds as follows and corresponds to the central timeline in Figure 5:

(1) Preparation and Version Bump. Backup initiation acquires the OSS lock and increments `next_snapshot_version`, starting a new snapshot epoch; all subsequent OSS writes carry this version.

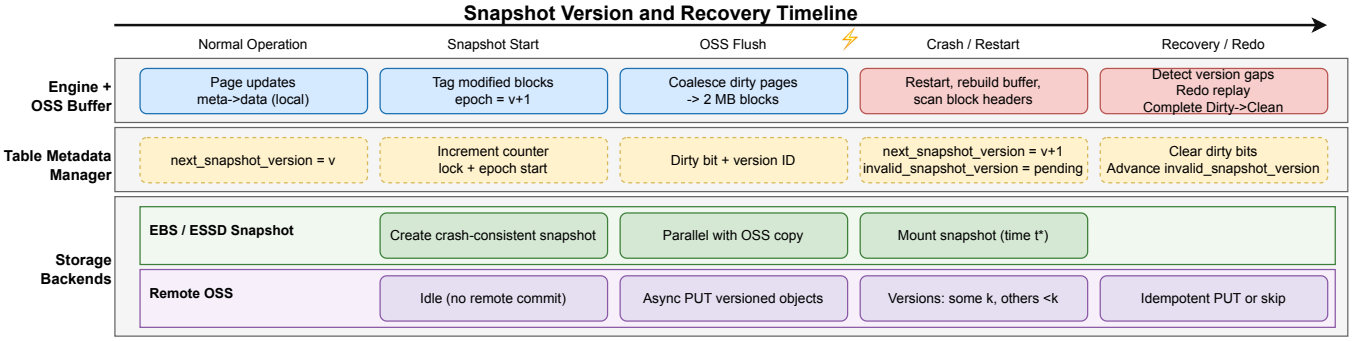


Figure 5: Snapshot Version and Recovery Timeline.

(2) Volume Snapshot. While the lock is held, **CloudJump III** captures an ESSD snapshot of the attached volumes containing the OSS Buffer (and optionally BPE), ensuring that unflushed dirty pages are preserved in the green EBS/ESSD snapshot track of Figure 5. During this period the remote OSS lane remains idle—the purple Idle (no remote commit) segment in Figure 5 highlights that no remote objects are published until the snapshot set is finalized.

(3) Change Detection. The engine scans table metadata to identify blocks with dirty bits and records their corresponding version numbers for remote copy. Concurrently, it coalesces dirty pages into 2 MB blocks before shipment, matching the blue Coalesce dirty pages → 2 MB block element in the engine lane.

(4) Lock Release. Once the snapshot set is finalized, the OSS lock is released. Foreground writes resume under the new snapshot epoch, isolated from the snapshot view.

(5) Remote Object Copy. For each marked block, **CloudJump III** issues an intra-bucket shadow copy via the OSS fast-copy API and schedules asynchronous migration to the backup destination. The green Parallel with OSS copy band indicates the crash-consistent volume snapshot remains available throughout this phase, while remote PUT operations, shown as purple Async PUT versioned objects, progress in the storage backend. Fast-copy duplicates object metadata without moving data, avoiding downtime.

(6) Metadata Finalization. After data movement is committed, dirty bits are cleared and `invalid_snapshot_version` is updated, signaling that older versions can be reclaimed.

This protocol reduces OSS write stalls to a few seconds and guarantees that backup readers observe a transactionally consistent snapshot. Block-level dirty tracking and versioned objects enable incremental backup with minimal data transfer. Administrative inspection commands (e.g., `show_tablespace_meta`) expose block states and version progress for operational diagnostics.

Zero-Downtime Snapshot Optimization. Snapshot creation follows a lock–version–copy sequence: the engine locks table metadata to stabilize the version view, then increments the snapshot counter, and initiates an intra-bucket fast-copy. Because the fast-copy duplicates object metadata rather than payload data, snapshot creation completes within seconds even for multi-terabyte tables.

Incremental Copy and Dirty-Bit Tracking. Subsequent snapshots apply incremental copy guided by the OSS Buffer's dirty-bit map—only objects updated since the prior version are duplicated. This differential replication minimizes I/O and bandwidth consumption while maintaining full recoverability. Combined, these mechanisms provide zero-downtime, crash-consistent backups that run concurrently with foreground traffic.

4.3.4 Recovery Workflow. The right segment of Figure 5 depicts the recovery process, which reuses the same metadata and version semantics to restore crash-consistent state:

(1) **Attach ESSD Snapshot.** Mount the previously captured ESSD snapshot to expose the consistent state of the OSS Buffer and BPE at backup time.

(2) **Metadata Scan.** Read table metadata to identify which OSS blocks belong to the restored snapshot and what version of each object must be fetched.

(3) **OSS Object Fetch.** For each listed block, retrieve its versioned OSS object into the active bucket. Version mismatches arise because recovery restores the pre-crash state, where block versions are not uniform: some blocks have reached version k, while others remain at earlier versions $< k$.

(4) **Database Restart.** Restart **CloudJump III**. The OSS Buffer reconstructs its control structures from recovered metadata, then InnoDB performs crash recovery.

(5) **Redo Application.** Replay redo logs for pages whose metadata advanced beyond data persistence. If data exists but metadata is stale, the flush is reissued. Both paths are idempotent.

During both normal operation and recovery, **CloudJump III** enforces complementary ordering: local persistence updates metadata before data, whereas remote flush completion writes data before marking metadata clean. This cross-ordered design guarantees that each block is either fully durable or safely replayable, preventing partial writes or version gaps.

Once recovery finishes, the system resumes normal operation on a crash-consistent state. Together, the snapshot versioning and recovery pipeline provide transactional backup and deterministic restoration with minimal service impact.

5 Experimental Evaluation

5.1 Experimental Setup

We evaluate **CloudJump III** under two representative configurations. The small-scale setup uses an 8-core CPU, a 1 GB buffer pool, and a 50 GB dataset with 16 client threads. The large-scale setup uses a 64-core CPU, a 100 GB buffer pool, and a 5 TB dataset

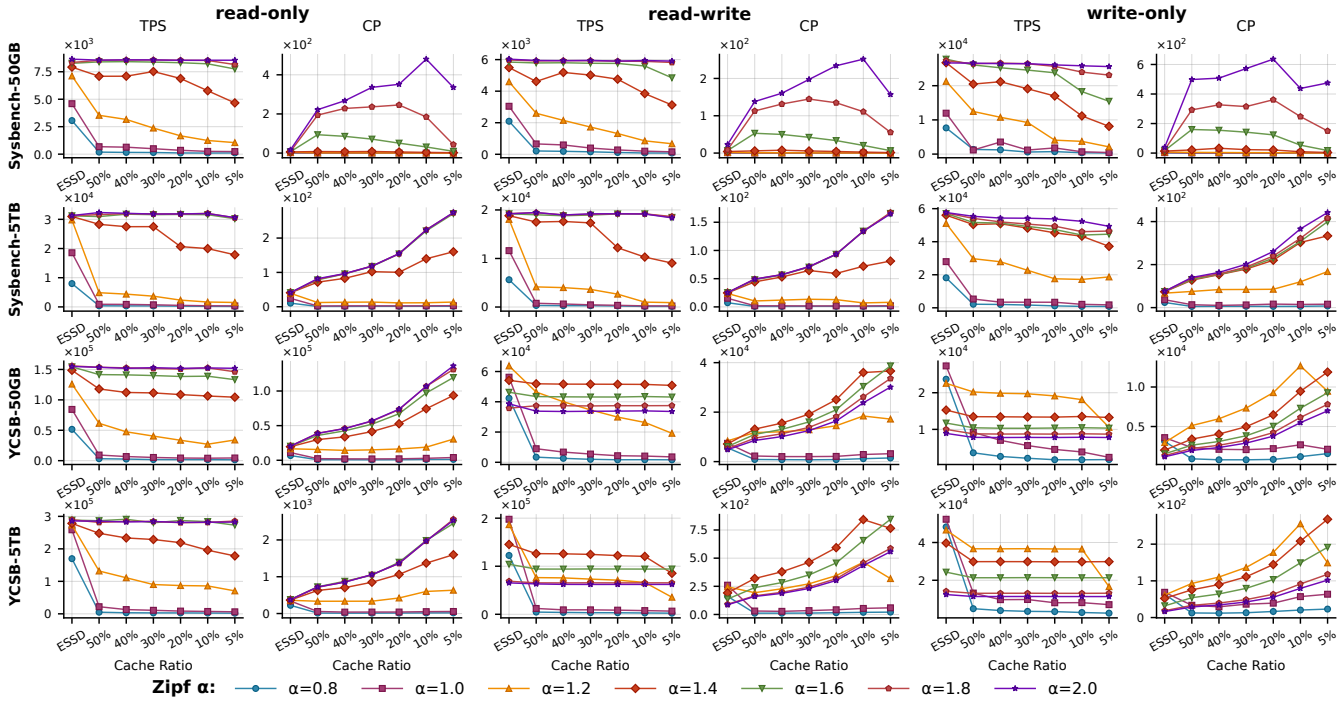


Figure 6: Performance and cost-effectiveness comparison across different workloads and cache ratios.

with 128 client threads. Each dataset is preloaded before execution, and the system is warmed up until throughput and cache hit ratios stabilize. All tests follow the **CloudJump III** hierarchy of DRAM buffer pool, SSD-based Buffer Pool Extension (BPE), ESSD-backed OSS Buffer, and remote OSS object storage. The fast tier covers 5%–50% of the dataset, with the remaining data stored on OSS. We select four representative workloads spanning read/write mix, access skew, object size, burstiness, and scale to exercise distinct **CloudJump III** mechanisms:

- **Sysbench** [93]: an OLTP-style microbenchmark with configurable read/write ratios and deterministic access patterns, used to assess concurrency and I/O scheduling.
- **YCSB** [22]: a mixed key-value workload with Zipfian distribution ($\alpha \in [0.8, 2.0]$); higher α yields stronger access skew (more concentrated popularity). Used to evaluate admission effectiveness and hotspot capture under skewed random accesses.
- **Voter** [31]: a transactional workload with short-lived sessions and tight latency targets, highlighting tail-latency stability and the OSS Buffer's ability to absorb transient bursts.
- **Game**: a production workload from Alibaba Cloud's online gaming service, characterized by large BLOB updates and bursty access patterns. It exercises bandwidth-intensive write paths under realistic production pressure.

5.2 Benchmark Performance

Figure 6 presents end-to-end results across two dataset scales (50 GB and 5 TB) and two representative workloads (Sysbench and YCSB). Each workload is evaluated under three operation mixes—read-only, read-write, and write-only—with two columns per dataset: throughput (TPS) and cost-performance (CP, i.e., throughput per storage cost). In addition to varying cache ratios, the results include a pure ESSD baseline representing a non-tiered all-SSD setup.

Two regularities emerge. First, TPS is monotone in both cache ratio and skew, with a knee once the hot set begins to fit the fast tier. On YCSB-50GB (KV access), when $\alpha \geq 1.4$ the 20–50% curves converge near the ESSD upper bound; under $\alpha \leq 1.0$ and small caches ($\leq$10%), the gap persists because popularity concentration is insufficient to sustain reuse. Second, CP improves with stronger skew and with moderate caches (typically 20–30%), reflecting larger marginal throughput per GB once eviction captures reusable keys and the OSS Buffer coalesces background writes. These effects are consistent with **CloudJump III**'s eviction-centric, engine-integrated design: admission-aware caching increases reuse on near-end media while object-granular write combining decouples foreground paths from remote `GET`/`PUT` costs.

Read-write follows the same structure but with milder slopes due to write amplification. At $\alpha \geq 1.4$, 30–50% cache ratios already deliver near-ESSD throughput and visibly higher CP; in contrast, low-skew, tiny-cache regimes are intentionally unfavorable because the BPE admits fewer reusable pages before the OSS Buffer amortizes remote writes. Write-only shows smooth CP gains as cache grows; by $\alpha \geq 1.6$ the 10–50% curves are tightly clustered, indicating bounded remote `PUT` pressure under rate-limited, merged writeback. Between benchmarks, Sysbench and YCSB exhibit qualitatively consistent behavior: KV-style fine-grained access makes YCSB slightly more sensitive at small caches under low skew, but both converge once realistic skew emerges.

Scale does not change the qualitative picture. The 5 TB curves align with 50 GB, with knees modestly right-shifted to reflect a

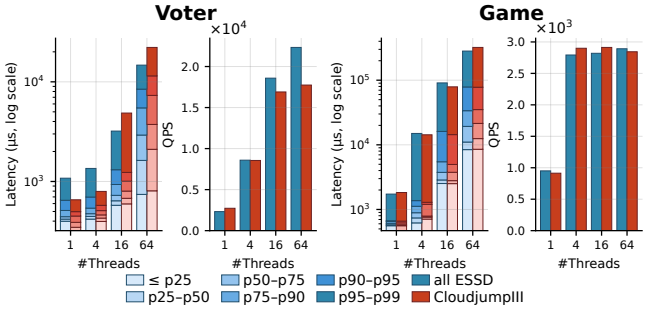


Figure 7: Latency and throughput across thread counts.

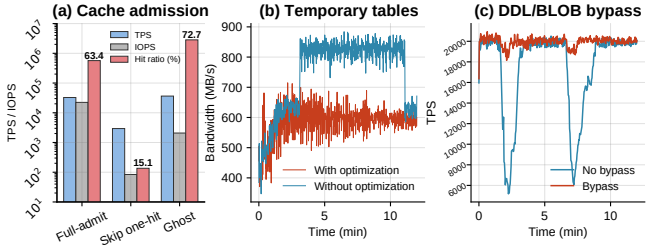


Figure 8: Ablation study results.

larger hot set; CP improvements are often more pronounced at scale because each GB of fast tier displaces more remote traffic. In practice, two regimes are salient: (i) with low skew and tight fast-tier budgets, **CloudJump III** remains functional but CP is modest; (ii) with moderate skew or a 20–30% cache, **CloudJump III** approaches local-disk throughput while delivering substantially better performance per GB than traditional storage. Overall, integrating tiering semantics inside the engine—coordinating admission, eviction, and OSS Buffer writeback—enables **CloudJump III** to recover near-ESSD performance at moderate cache ratios and to yield stable throughput and improved CP from limited capacity.

5.3 Application Scenarios

We evaluate **CloudJump III** under representative application workloads to verify that its architectural advantages translate into end-to-end stability and efficiency. Two scenarios are examined: (i) the Voter workload representing transactional business logic, and (ii) an online game workload that mixes large-object updates with bursty inserts. Both experiments run on a 50 GB data scale.

Figure 7 compares end-to-end latency and throughput between the all-ESSD baseline (blue) and the **CloudJump III** tiered configuration (red) as the number of *client* threads increases from 1 to 64. For Voter, **CloudJump III** matches or exceeds the baseline at low concurrency and stays close at moderate concurrency, while exhibiting lower medians when the working set is well captured by the BPE. At 1 thread, **CloudJump III** delivers higher throughput (+17%; 2727 vs. 2321 RPS) with a shorter median (p50 347 µs vs. 419 µs); at 4 threads, throughput is at parity (8554 vs. 8589) with a slightly lower median (429 µs vs. 446 µs). As concurrency increases, throughput remains within ~10% at 16 threads (16902 vs. 18583), and tails widen moderately due to paced remote commits (p99 4861 µs vs. 3205 µs, ~1.5×). At 64 threads, **CloudJump III** sustains 79–80% of the all-ESSD throughput (17742 vs. 22328) with bounded upper percentiles (p99 22145 µs vs. 14714 µs, ~1.5×), indicating that background persistence rather than foreground stalls becomes the limiting factor while queueing remains controlled.

For the Game workload—characterized by large-object updates and bursty inserts—**CloudJump III** maintains throughput within ±5% of the all-ESSD baseline across all thread counts, and occasionally exceeds it at moderate concurrency (e.g., +3–4% at 4–16 threads; 2899 vs. 2793 and 2913 vs. 2820 RPS). Latency profiles are comparable: medians track closely, and tails remain stable without spike-like outliers. Notably, p99 is lower at 16 threads (78.9 ms vs. 90.8 ms) and modestly higher at 64 threads (321 ms vs. 282 ms), reflecting the tradeoff from paced batching. This stability is enabled by engine-integrated mechanisms in which a bypass routes large writes directly to ESSD to avoid contention on shared flush queues, while the OSS Buffer coalesces small updates into 2 MB versioned objects before remote commit to smooth bandwidth demand.

These results show that engine-integrated tiering shifts variability off the foreground path into a controlled background pipeline. Admission-aware BPE improves hit rates by prioritizing reuse-predictive pages, and the OSS Buffer coalesces and paces remote writes to prevent burst amplification. Snapshot-synchronized metadata preserves ordering without halting transactions. The remaining tail widening at high concurrency is limited and stems from remote persistence variance, while throughput stays near-local for both latency-sensitive and mixed workloads.

5.4 Ablation Studies

To quantify each component's impact in **CloudJump III**, we perform three ablation studies using Sysbench on the 50 GB dataset. Figure 8a compares three BPE admission policies: **one-hit discard** (admit no page to the BPE unless it was re-accessed in the buffer pool; all one-hit pages are discarded), **full-admit** (admit every page evicted from the buffer pool to the BPE with no filtering), and **ghost two-chance** (the **CloudJump III** policy: reuse-validated, delayed admission via the ghost list). Under identical conditions, Ghost applies reuse-validated, delayed admission so that single-reference pages are filtered at the ghost layer rather than admitted. This avoids one-hit pollution, concentrates SSD space on reusable pages, and reaches 36422 TPS (+12% vs. full-admit) with a higher hit ratio (72.7% vs. 63.4%/15.1%). By blocking transient admits, Ghost also reduces backend load: BPE IOPS falls from 22369 to 2082 (~91% lower), reflecting fewer fine-grained writes and steadier writeback. In contrast, full-admit admits all pages into BPE, causing high SSD IOPS and weakened cache efficiency due to indiscriminate admission, whereas one-hit discard yields the lowest throughput (2945 TPS) and hit ratio (15.1%) because most pages are discarded before reuse opportunities, preventing them from entering BPE.

Figure 8b isolates temporary-table exclusion from the OSS Buffer. After a three-minute warmup, we issue complex queries that generate substantial temporary-table traffic. Without this feature, short analytic bursts spill into remote storage, producing sharp bandwidth spikes and high temporal variance. With exclusion enabled, temporary pages stay local to the BPE, which lowers peak bandwidth while stabilizing overall efficiency. Recovery is unchanged since temporary tables need no persistence.

Table 2: Backup and recovery results (50GB dataset).

Metric	ESSD only	OSS only	CloudJump III
Snapshot creation	0.52	53.73	0.64
Backup duration	49.58	53.73	51.22
Recovery time	1.15	57.64	52.26

Figure 8c analyzes the bypass path for BLOB or DDL writes, which sends both directly to OSS and removes them from the shared log-flush queue. Under sustained transactional load with periodic DDLs and concurrent BLOB traffic, the *No bypass* curve shows repeated and deep TPS drops aligned with each DDL phase. In contrast, the *Bypass* curve stays near steady-state TPS with only minor variation. These results show that bypassing BLOB and DDL writes shields transactional I/O from their background cost and prevents DDL-driven stalls from throttling foreground throughput.

Together, these results indicate that effective tiering requires engine integration. With engine-visible signals, the engine applies accurate placement and stable writeback. This filters one-hit noise, confines transient tables, and isolates non-transactional writes, yielding steady throughput and better cost efficiency from limited fast-tier capacity.

5.5 Backup and Recovery

As shown in Table 2, snapshot creation in **CloudJump III** (0.64 s) stays close to the pure ESSD setup (0.52 s) and remains far faster than the OSS-only path (53.73 s). The lock–version–copy sequence fixes the boundary and issues an intra-bucket fast-copy of metadata, yielding sub-second snapshots with remote durability. Backup and recovery also remain stable. **CloudJump III** completes backup in 51.22 s and recovery in 52.26 s, matching the OSS baseline. These times are dominated by remote bandwidth because most data resides in OSS, and **CloudJump III** adds no extra delay despite the extra tiers. Two choices keep the path efficient. Coalescing objects into 2 MB units and using table-level dirty-bit incremental copy avoid full scans and reduce API load, and ordered local–remote write handling ensures deterministic, idempotent replay. Overall, **CloudJump III** preserves OSS-level backup and recovery time while reducing cost through engine-guided tiering.

6 Related Work

Cloud-Native Database Architectures. Separating compute from storage defines modern cloud databases for elasticity and availability. Amazon Aurora [84] pioneered a log-structured shared-storage design for OLTP; Microsoft Socrates [2] and Azure SQL Hyperscale [24] add remote page servers and caching. For analytics, Snowflake [27] and Redshift [3] use object storage as a common data substrate for independent scaling; SAP IQ has been extended with cloud-native storage and instance-storage caching for object stores [1]. Spanner [23], AlloyDB [21], and F1 Lightning [90] combine consensus replication with distributed logging for cross-region consistency. In Alibaba Cloud, production systems generalize these patterns at scale for mixed workloads and elastic compute [15, 94]. Recent surveys provide a broader taxonomy and challenges for cloud-native databases [32]. Prior work studies storage disaggregation with RDMA storage, log structured layouts, shared buffer coordination, and multi tier management [33, 44, 72, 89], as well as atomic commit under disaggregation [39]. The CloudJump line follows this trajectory. CloudJump I [19] optimized the compute to object store path, CloudJump II [20] introduced multiversion shared storage, and **CloudJump III** internalizes tiering to align placement with workload dynamics while preserving recovery semantics.

Tiered and Hierarchical Storage Management. Hierarchical tiering spans ILM and HSM systems (e.g., GPFS ILM [40] and Oracle HSM [26]), OS and device caches (e.g., dm-cache [82], bcache [64], LVM [69], ZFS ARC/L2ARC [61]), and distributed object stores (Facebook f4 [80], Microsoft Pelican [4], Ceph [88]) that move data between hot and cold tiers. These mechanisms are mature but typically operate at block or file granularity and lack database semantics, index-data separation, and transactional consistency. PrismDB [70] integrates tiering within the storage engine by migrating objects across NVMe tiers, yet remains object-granular. Learning-based storage frameworks (e.g., OctopusFS [47]) migrate files using access prediction [42] but are unaware of DBMS semantics such as index-data separation and transactional eviction control. Cache libraries such as CacheLib provide general abstractions [6]. In contrast, **CloudJump III** internalizes tiering in the engine and routes pages at eviction and flush using access patterns and table heat, avoiding coarse-grained migrations and recall-induced jitter.

Hot and Cold Data Management in Databases. Within database engines, hot–cold placement draws on buffer replacement and tiering policies (LRU, CLOCK, 2Q, LIRS, ARC [18]), multi-temperature and lifecycle management in production systems (DB2 [71], Oracle ADO [25], SAP HANA [38]), and tiered writes in LSM-based stores; recent work addresses LSM layout [95] and heterogeneous memory management [77]. Admission filtering to curb cache pollution is well studied—e.g., CLOCK-Pro [46], TinyLFU/W-TinyLFU [34], CacheSack [92]. Many approaches, however, remain coarse-grained or decoupled from recovery, undermining stability under workload variation. **CloudJump III** unifies page- and table-level signals in an engine-integrated model: admission at eviction, write routing at flush, and placement orthogonal to multiversion recovery.

7 Conclusion

CloudJump III advances tiered storage by integrating placement and routing into the engine's buffer-manager path without external migration layers. This shifts tiering from reactive background movement to a predictable, eviction-driven process aligned with transactional correctness. The design unifies caching (DRAM and SSD-based buffer pool extension), durable buffering on network-attached storage, and snapshot-based persistence on object storage within one control framework that preserves crash consistency and zero-downtime backup. Eviction-centric admission and per-table quotas keep fast-tier usage bounded and fair in multi-tenant settings, while the snapshot-version protocol keeps placement consistent with recovery and backup. Deployed in Alibaba Cloud's MySQL-compatible engine, **CloudJump III** delivers near-local throughput and stable latency with a small fraction of fast-tier capacity, yielding steady and cost-efficient performance. These results show that predictable performance in disaggregated databases follows from placing tiering in the same control loop as caching and recovery, offering a practical foundation for scalable, resilient cloud database design.

References

[1] Mohammed Abouzour, Güneş Aluç, Ivan T. Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, and John C. Smirnios. 2021. Bringing Cloud-Native Storage to SAP IQ. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2410–2422. doi:10.1145/3448016.3457563

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1743–1756.

[3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2205–2217.

[4] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony I. T. Rowstron. 2014. Pelican: A Building Block for Exascale Cold Data Storage. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 351–365.

[5] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 47–60.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 753–768.

[7] Suparna Bhattacharya, C. Mohan, Karen Brannon, Inderpal Narang, Hui-I Hsiao, and Mahadevan Subramanian. 2002. Coordinating backup/recovery and data consistency between database and file systems. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*. ACM, 500–511.

[8] Souvik Bhattacherjee, Gang Liao, Michael Hicks, and Daniel J. Abadi. 2021. BullFrog: Online Schema Evolution via Lazy Evaluation. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 194–206.

[9] Renata Borovica-Gajic, Raja Appuswamy, and Anastasia Ailamaki. 2016. Cheap Data Analytics using Cold Storage Devices. *Proc. VLDB Endow.* 9, 12 (2016), 1029–1040.

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC 2013, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 49–60.

[11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. ACM, 143–157.

[12] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent N. Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. 2017. Curator: Self-Managing Storage for Enterprise Clusters. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 51–66.

[13] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2859–2872.

[14] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862.

[15] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2477–2489.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 209–223.

[17] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 275–290.

[18] Amit S Chavan, Kartik R Nayak, Keval D Vora, Manish D Purohit, and Pramila M Chawan. 2011. A comparison of page replacement algorithms. *International Journal of Engineering and Technology* 3, 2 (2011), 171.

[19] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. 2022. CloudJump: Optimizing Cloud Databases for Cloud Storages. *Proc. VLDB Endow.* 15, 12 (2022), 3432–3444.

[20] Zongzhi Chen, Xinjun Yang, Mo Sha, Feifei Li, Kang Wang, Zheyu Miao, Jie Xu, Jianfeng Wang, and Sheng Wang. 2025. CloudJump II: Optimizing Cloud Databases for Shared Storage. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 336–349.

[21] Google Cloud. 2025. AlloyDB for PostgreSQL: Overview. https://cloud.google.com/alloydb/docs/overview. Accessed: 2025-11-11.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. ACM, 143–154.

[23] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 251–264.

[24] Microsoft Corporation. 2018. Announcing Azure SQL Database Hyperscale public preview. https://azure.microsoft.com/en-us/blog/announcing-azure-sql-database-hyperscale-public-preview/. Accessed: 2025-11-11.

[25] Oracle Corporation. 2017. *Application Development with Oracle Database 12c*. Accessed: 2025-11-11.

[26] Oracle Corporation. 2019. *Oracle Hierarchical Storage Manager and StorageTek QFS Software: Installation and Configuration Guide, Release 6.1.4*. Oracle Corporation. https://docs.oracle.com/cd/E71197_01/SAMIC/E78138-07.pdf Accessed: 2025-11-11.

[27] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 215–226.

[28] Yifan Dai, Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2024. Symbiosis: The Art of Application and Kernel Cache Cooperation. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA, February 27-29, 2024*. USENIX Association, 51–69.

[29] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 79–94.

[30] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.

[31] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[32] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-Native Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 36, 12 (2024), 7772–7791.

[33] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*. USENIX Association, 401–414.

[34] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4 (2017), 35:1–35:31.

[35] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 65–78.

[36] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 42:1–42:13.

[37] Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan J. LoVerso, Michael J. Cahill, and Alexander Gorrod. 2022. Writes hurt: lessons in cache design for optane NVRAM. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*. ACM, 110–125.

[38] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[39] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation. *Proc. VLDB Endow.* 16, 2 (2022), 379–392.

[40] Nils Haustein. 2020. *IBM Spectrum Scale Information Lifecycle Management Policies v10.2*. Technical Report WP102642. IBM Corporation. Accessed: 2025-11-11.

[41] Xiaolong He, Peng Cai, Xuan Zhou, and Aoying Zhou. 2021. Continuously Bulk Loading over Range Partitioned Tables for Large Scale Historical Data. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 960–971.

[42] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *Proc. VLDB Endow.* 13, 1 (2019), 43–56.

[43] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 651–665.

[44] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. 2024. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org. https://vldb.org/workshops/2024/proceedings/ADMS/ADMS24_01.pdf

[45] Danlin Jia, Yiming Xie, Li Wang, Xiaoqian Zhang, Allen Yang, Xuebin Yao, Mahsa Bayati, Pradeep Subedi, Bo Sheng, and Ningfang Mi. 2023. SRC: Mitigate I/O Throughput Degradation in Network Congestion Control of Disaggregated Storage Systems. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 268–278.

[46] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 323–336.

[47] Elena Kakoulli and Herodotos Herodotou. 2017. OctopusFS: A Distributed File System with Tiered Storage Management. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 65–78.

[48] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2012. Flash-based Extended Cache for Higher Throughput and Faster Recovery. *Proc. VLDB Endow.* 5, 11 (2012), 1615–1626.

[49] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G. Dreslinski. 2022. Power-optimized Deployment of Key-value Stores Using Storage Class Memory. *ACM Trans. Storage* 18, 2 (2022), 13:1–13:26.

[50] Hyojun Kim, Ioannis Koltsidas, Nikolas Ioannou, Sangeetha Seshadri, Paul Muench, Clement L. Dickey, and Lawrence Chiu. 2014. How Could a Flash Cache Degrade Database Performance Rather Than Improve It? Lessons to be Learnt from Multi-Tiered Storage. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW '14, Broomfield, CO, USA, October 5, 2014*. USENIX Association.

[51] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 427–444.

[52] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 460–477.

[53] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2022. Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. *Proc. VLDB Endow.* 15, 11 (2022), 2867–2880.

[54] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196.

[55] Justin J. Levandoski, Per Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 26–37.

[56] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud Databases: New Techniques, Challenges, and Opportunities. *Proc. VLDB Endow.* 15, 12 (2022), 3758–3761.

[57] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. USENIX Association, 133–148.

[58] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[59] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 243–262.

[60] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2022. Kangaroo: Theory and Practice of Caching Billions of Tiny Objects on Flash. *ACM Trans. Storage* 18, 3 (2022), 21:1–21:33.

[61] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX.

[62] C. Mohan and Frank E. Levine. 1992. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*. ACM Press, 371–380.

[63] Koyel Mukherjee, Raunak Shah, Shiv Kumar Saini, Karanpreet Singh, Khushi, Harsh Kesarwani, Kavya Barnwal, and Ayush Chauhan. 2023. Towards Optimizing Storage Costs on the Cloud. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2919–2932.

[64] Kent Overstreet. 2010. bcache: A Linux block layer cache for SSDs. https://bcache.evilpiepirate.org/ Linux kernel module.

[65] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. USENIX Association, 217–231.

[66] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3 (2024), 180.

[67] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 1326–1339.

[68] Hojin Park, Gregory R. Ganger, and George Amvrosiadis. 2020. More IOPS for Less: Exploiting Burstable Storage in Public Clouds. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association.

[69] The LVM2 Project. 2024. *LVM2 Cache: Device Mapper Based SSD Caching for Logical Volumes*. Red Hat and Linux Kernel Community. Accessed: 2025-11-11.

[70] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 179–193.

[71] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer,

David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091.

[72] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-latency, Geo-replicated Transactions. *Proc. VLDB Endow.* 12, 11 (2019), 1747–1761.

[73] Kexin Rong, Paul Liu, Sarah Ashok Sonje, and Moses Charikar. 2024. Dynamic Data Layout Optimization with Worst-Case Guarantees. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4288–4301.

[74] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent Memory Disaggregation for Cloud-Native Relational Databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 498–512.

[75] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: a lightweight, consistent and durable storage cache. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. ACM, 267–280.

[76] Daniel Schall and Theo Härder. 2015. Dynamic physiological partitioning on a shared-nothing database cluster. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 1095–1106.

[77] Mo Sha, Yifan Cai, Sheng Wang, Linh Thi Xuan Phan, Feifei Li, and Kian-Lee Tan. 2024. Object-oriented Unified Encrypted Memory Management for Heterogeneous Memory Architectures. *Proc. ACM Manag. Data* 2, 3 (2024), 155. doi:10.1145/3654958

[78] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge-Arnulfo Quiané-Ruiz, and Aaron J. Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 229–241.

[79] Sagar Shedge, Nishant Sharma, Anant Agarwal, Mohammed Abouzour, and Gunes Aluc. 2022. An Extended SSD-Based Cache for Efficient Object Store Access in SAP IQ. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1861–1873. doi:10.1109/ICDE53745.2022.00185

[80] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 383–398.

[81] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. USENIX Association, 373–386.

[82] Joe Thornber, Heinz Mauelshagen, and Mike Snitzer. 2013. dm-cache: Linux Device Mapper Cache Target. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt Linux Kernel Documentation.

[83] Pinar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. 2013. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. *VLDB J.* 22, 2 (2013), 151–175.

[84] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1041–1052.

[85] Donghui Wang, Yuxing Chen, Chengyao Jiang, Anqun Pan, Wei Jiang, Songli Wang, Hailin Lei, Chong Zhu, Lixiong Zheng, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2025. TXSQL: Lock Optimizations Towards High Contented Workloads. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 675–688.

[86] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, Chengjun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2 (2023), 199:1–199:25.

[87] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data, SIGMOD-/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM, 37–44.

[88] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. USENIX Association, 307–320.

[89] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 401–412.

[90] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeffrey F. Naughton, and John Cieslewicz. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (2020), 3313–3325.

[91] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 130–149.

[92] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, Homer Wolfmeister, and Junaid Khalid. 2023. CacheSack: Theory and Experience of Google's Admission Optimization for Datacenter Flash Caches. *ACM Trans. Storage* 19, 2 (2023), 13:1–13:24.

[93] Peter Zaitsev. 2004. Sysbench: a modular, cross-platform and multi-threaded benchmark tool. https://github.com/akopytov/sysbench. Originally developed by Peter Zaitsev in 2004; maintained by Alexey Kopytov and contributors.

[94] Jiaoyi Zhang, Liqiang Peng, Mo Sha, Weiran Liu, Xiang Li, Sheng Wang, Feifei Li, Mingyu Gao, and Huanchen Zhang. 2025. Femur: A Flexible Framework for Fast and Secure Querying from Public Key-Value Store. *Proc. ACM Manag. Data* 3, 3 (2025), 162:1–162:29. doi:10.1145/3725299

[95] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM : Optimize Data Layout for LSM-tree Based Storage using Survival Analysis. *Proc. VLDB Endow.* 15, 10 (2022), 2161–2174.

[96] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. 2024. What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA, February 27-29, 2024*. USENIX Association, 277–291.

[97] Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. 2026. Why Database Manuals Are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents. *Proc. VLDB Endow.* 19, 6 (2026), 1358–1371.

[98] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*. USENIX, 59–72.

[99] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David E. Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2195–2207.