OceanBase Bacchus: A High-Performance and Scalable Cloud-Native Shared Storage Architecture for Multi-Cloud

Quanqing Xu* OceanBase, Ant Group
Mingqiang Zhuang* OceanBase, Ant Group
Chuanhui Yang† OceanBase, Ant Group
Quanwei Wan OceanBase, Ant Group
Fusheng Han OceanBase, Ant Group
Fanyu Kong OceanBase, Ant Group
Hao Liu OceanBase, Ant Group
Hu Xu OceanBase, Ant Group
Junyu Ye OceanBase, Ant Group
OceanBaseLabs@service.oceanbase.com

ABSTRACT

Although more databases embrace shared-storage architectures, current storage-disaggregated systems lack an optimal balance between cost and performance. B+-tree–based shared storage struggles with frequent in-place updates under high concurrency; disaggregated designs still require a scalable shared-log path for multi-writer OLTP. This paper presents OceanBase Bacchus, a *production-grade* disaggregated shared-storage architecture for OceanBase. We foreground two technical elements: **(i) service-oriented PALF logging** with **sharded log streams** and multiple Read-Write (RW) leaders, where partition-local ordering is composed with distributed **2PC across stream leaders**, breaking the single-writer bottleneck of classic shared-log designs while preserving ACID, and **(ii) a Shared Block Cache Service** that pools macro-blocks within an availability zone, a less common alternative to purely per-node caches and motivated by **stateless compute** scaling. Cloud object storage backs the LSM-tree capacity tier for cost, but we do not treat "LSM + object storage" as the headline novelty (precedents exist). Bacchus's value is integrating and hardening these mechanisms in a shipping system, demonstrated by credible OLTP/OLAP performance and substantial storage-cost reductions. The architecture decouples modules and enables elastic scaling of compute, cache, and storage independently. Experimental results show that OceanBase Bacchus matches or exceeds HBase and PolarDB in OLTP and significantly outperforms StarRocks in OLAP. With multi-cloud support, the system retains high availability and competitive performance while cutting storage costs by 59% (OLTP) and 89% (OLAP).

PVLDB Reference Format:
Quanqing Xu, Mingqiang Zhuang, Chuanhui Yang, Quanwei Wan, Fusheng Han, Fanyu Kong, Hao Liu, Hu Xu, and Junyu Ye. OceanBase Bacchus: A High-Performance and Scalable Cloud-Native Shared Storage Architecture for Multi-Cloud. PVLDB, 19(12): 4089 - 4102, 2026.
doi:10.14778/3827998.3828018

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/oceanbase/oceanbase/tree/4.4.x.

*Quanqing Xu and Mingqiang Zhuang contributed equally to this work.
†Chuanhui Yang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828018

1 INTRODUCTION

The database industry is rapidly embracing cloud-native architectures; "Public Cloud First" is the prevailing trend. Shared storage architectures enable cloud-native databases by eliminating data redundancy [51], simplifying consistency management [7], and enabling elastic scaling [18]. However, existing shared storage solutions still face deep challenges for high-concurrency OLTP [26].

The Challenges of B+-tree– and LSM-tree–based Shared Storage. B+-tree–based shared storage struggles under high-concurrency writes. Random I/O and in-place updates create conflicts when multiple compute nodes access shared storage; frequent page splits and rebalancing degrade write-heavy workloads, making B+-tree designs ill-suited for cloud applications that need sustained write throughput. By contrast, LSM-trees excel at writes, yet LSM-tree–backed disaggregated storage lacks efficient cross-node shared logs. Persisting logs directly to durable storage incurs high latency, while log-replication approaches struggle with consistency and scale. That gap has slowed LSM adoption in shared storage despite LSM outputs matching object storage's append-only pattern.

The Need for Object Storage Integration. Cloud vendors provide low-cost object storage (e.g., AWS S3, Alibaba Cloud OSS (Object Storage Service)) that is 85% cheaper than traditional cloud disks. However, object storage's high latency and limited IOPS (I/O operations per second) challenge OLTP workloads requiring millisecond response times. Existing systems either avoid object storage or use it only for cold data, missing substantial cost savings.

The Motivation for OceanBase Bacchus. We need shared storage that: (1) pairs LSM write efficiency with coordinated shared logs, (2) exploits low-cost object storage without sacrificing OLTP responsiveness, (3) disaggregates storage and compute using stateless nodes, and (4) supports multi-cloud deployment. This paper targets **OLTP** first, with hot/cold access patterns (historical DBs, time-series, multi-model) where shared storage gains are largest. The architecture also supports **online analytical processing (OLAP)** and **HTAP** [53]; we evaluate both OLTP and OLAP workloads. Figure 1 illustrates hot/cold data storage.

To address these challenges, we propose OceanBase Bacchus[1], built around a **service-oriented PALF shared log** and a **Shared Block Cache Service** on OceanBase's LSM-tree [40] engine with cloud object storage as the durable capacity tier. We position Bacchus as **production synthesis** within OceanBase, not an isolated primitive; the emphasis is on the log and shared-cache layers rather than "LSM + object storage" alone.

[1]Bacchus was considered a deity associated with the sharing spirit in ancient Roman mythology, so it is used to name our shared storage database, i.e., OceanBase 4.4.x.

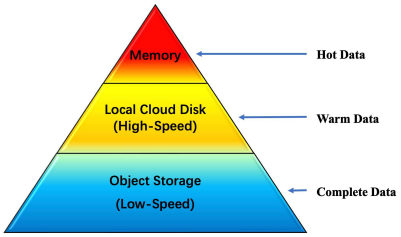


Figure 1: An illustration of hot/cold data storage

First, shared-log coordination runs as an independent PALF-backed logging service [25]. **Log streams** shard data; each leader appends CLog to PALF so RW nodes scale writes, while **distributed 2PC across stream leaders** preserves ACID across partitions. Against Aurora/Socrates [7, 51] (single-writer shared-log), this removes the primary-only append bottleneck at the cost of cross-stream coordination. Unlike shared-nothing [50], Bacchus avoids redundant per-node WAL; unlike direct object-store logging [63], the log service keeps commits off the object-store critical path.

Second, multi-layer caching (local persistent + distributed) closes the gap: hot data on low-latency local tiers; cold data on low-cost object storage. The LSM-tree engine matches append-only object I/O, avoiding in-place updates and write conflicts. We tune I/O aggregation, async writes, and concurrency for throughput.

Third, shared services enable cloud elasticity: stateless compute via shared cache and async background work (compaction, DDL, backup, recovery). Cache scales with hot data, balancing cost and performance. Multi-cloud deployment (AWS, Alibaba Cloud, etc.) avoids vendor lock-in.

In summary, OceanBase Bacchus targets shared storage through: (1) service-oriented shared-log coordination, (2) multi-layer caching between object storage and OLTP, and (3) cloud-native elasticity and storage–compute disaggregation. It delivers stateless compute, lower storage costs (59% OLTP, 89% OLAP), and high performance. LSM-trees on remote or object-backed tiers, replicated shared logs, and distributed caches are precedented [28, 45, 48]; we claim no architectural first. We document how **PALF**-based shared logging, SSLog metadata aggregation, the Shared Block Cache Service, and OceanBase-specific dumping/compaction co-design compose and run *together* in a shipping system to yield no loss, elasticity, and the reported multi-cloud relational cost/performance.

This paper is organized as follows. §2 describes the new design of the Bacchus architecture. §3 provides an intuitive overview of the implementation of Bacchus shared storage. §4 presents the increment and compaction strategies. §5 describes the multi-layer caching and preheating mechanisms. §6 details garbage collection mechanisms. We present the experiments in §7 to validate high availability and competitive performance of the shared-storage architecture and its shared services, including tail latency under adverse caching (§7.4). We discuss lessons learned in §8, and review the related work in §9. Finally, we conclude in §10.

2 SYSTEM ARCHITECTURE

2.1 Layered Design

Figure 2 splits the stack into the Database Layer and the Shared Storage Layer. The Database Layer has **Memory Cache**, holding the hottest micro-blocks and index data, so key transactions stay millisecond-scale; and **Local Persistent Cache**, improving access and acting as the first persistence tier for hot, dumped, warmed-up, and metadata—mitigating random-read pressure from object storage. Its roles: **(1) Data Caching:** Cache reads from object storage locally. **(2) Data Prefetching:** Prefetch imminent reads (e.g., full-table scans) to improve latency. **(3) Data Warming:** Load hot data before version merges; brief outages still leave cache effective, limiting jitter. Access order syncs to replicas in real time for consistency; replication migrates cached data for performance. **(4) Write Cache:** Exploit low-latency local storage to persist essential state (e.g., metadata files, latest change logs).

The Shared Storage Layer consists of the Shared Service Layer and Object Storage:

• **Shared Service Layer** coordinates cross-cluster resources via three components: **(1) Distributed Cache:** BlockServers implement the Shared Block Cache Service, storing and serving macro-blocks in a three-tier stack for hot data and warming. **(2) Shared Log (Service-Oriented Logging)** runs on LogServers with three replicas (typically one per availability zone (AZ) in a region), supporting parallel operation across clusters. Metadata and data logs use PALF consensus; commits wait for majority persistence of the matching CLog entries (Section 3). **(3) Background Tasks** (e.g., baseline compaction, DDL, garbage collection) run transparently without impacting foreground performance.

• **Object Storage** stores infrequently accessed data at scale. S3-compatible [6], it supports multi-cloud object stores (AWS S3, Alibaba Cloud OSS [5], Blob [35]) and local MinIO [36]. Bacchus offers flexible alternatives for object storage services and deployment.

The architecture enables efficient, reliable data retrieval. Unlike PolarDB [55] and GaussDB [31], which are tightly coupled to specific clouds, Bacchus supports flexible multi-cloud deployment, avoiding vendor lock-in.

2.2 Interaction between Nodes

The database layer writes data and metadata to shared storage. Concurrent nodes partition data into log streams; each stream has one leader writing to the PALF log service. The sharded model prevents overlapping write sets across streams, avoiding conflicts and enabling horizontal scale. Cross-partition transactions use distributed 2PC (two-phase commit). RW and Read-Only (RO) nodes interact through the following workflow: **(1)** During transactions, RW ships WAL (Write-Ahead Logging) to the log service. **(2)** RO pulls WAL from the log service and replays it into local storage. **(3)** RW writes data to shared storage as required for sharing. **(4)** RW posts metadata updates as journals to the log service when required. **(5)** RO pulls journals, replays them locally, and updates cached metadata. **(6)** RO fetches the newly generated SSTable (Sorted String Table) list [16] from shared storage under updated metadata. **(7)** RW materializes metadata from the journal to bring up new nodes for recovery or scaling.

2.3 Architecture Features

Disaggregating storage and compute lets tiers scale independently with load. CLog (Commit Log) entries use local cloud disk for low-latency updates; older CLog migrates to shared storage under policy,

Figure 2: System architecture

balancing cost and speed. Durable data stays in shared storage with hot sets cached locally. The local disk tier acts as Local Persistent Cache—persisting across restarts to mitigate cache avalanche and cold-start latency. Object storage holds durable data, cached locally; failures do not forfeit commits, and caches refill from object storage. Hot-data identification tunes caching, trimming redundant footprint while preserving concurrent read/write throughput.

The design is serverless-style: compaction and DDL run in separate processes, isolating foreground traffic and improving CPU utilization. Preheating, smooth major-version upgrades, replication migration, and primary/backup DR harden operations; optional Warm Backup Cluster in another zone replays logs for fast failover. These keep workloads stable through upgrades and faults, bound user-visible disruption, and balance availability with performance.

2.4 Architecture Advantages

LSM-trees pair naturally with cloud object storage (append-only SSTables), as disaggregated LSM research and engines show [28, 45, 48]. Bacchus is not novel *solely* for that pairing or as isolated mechanisms; its value is **end-to-end engineering synthesis**: **service-oriented PALF logging** (multi-writer sharded streams, 2PC across partitions), the **Shared Block Cache Service**, and workload-specific dumping, compaction, and caching, yielding RPO=0 and the production-scale cost/performance trade-offs we evaluate. RPO (Recovery Point Objective) caps acceptable data-loss duration and the recovery point after failure; RPO=0 admits no loss and demands near-real-time replication. Relative to other distributed database designs, Bacchus offers the following.

• **Coordination between LSM-tree and Object Storage**. Compared to B+-tree–based [17] database systems, B+ trees excel at reads ($O(logN)$) but incur random I/O, page splits, and fragmentation under high-concurrency writes. LSM-trees use tiered storage (MemTable + multi-level SSTables), offering superior write performance through sequential writes and tiered compaction. That aligns with object storage's append-only nature and reduces storage cost.

• **Separation of Functional Modules**. We leverage LSM tiering to offload long-running tasks (e.g., major compaction) to underutilized resources, realizing storage-compute disaggregation and separating foreground from background work. Shared distributed caching isolates caches from compute nodes for elastic scaling; a serverless layout sends background tasks to dedicated processes, eliminating over-provisioning and reducing foreground interference.

• **Optimization of Dumping Data**. Unlike Socrates [7] (B-Tree + Shared-disk) and SingleStore [43] (LSM-tree + Blob Store with high synchronization latency), Bacchus achieves RPO=0 through PALF-backed logging, stateless compute, and fast incremental persistence—not merely object-backed LSM outputs. We optimize dumping with minute-level and continuous strategies and introduce *micro compaction* (§4) for OLTP workloads.

• **High Availability and Scalability.** The Log Service achieves high availability through triple replication; OceanBase servers target RTO via two replicas per tenant, e.g., `F1@AZ1` and `F1@AZ2` (full-function replicas in AZ1 and AZ2). On AZ1 failure, the leader fails over to AZ2 for continuity. Multi-replication theoretically guarantees Bacchus's availability. Scalability follows replica migration inherited from OceanBase's Shared-Nothing architecture [59, 60]; Bacchus avoids copying shared data during migration, significantly accelerating transfers.

• **Reduction of Storage Costs**. In Bacchus, reducing replicas from three to one and exploiting elastic object storage optimizes overall resources versus triple-replicated stacks. Compared with traditional cloud disks (PL1), OSS costs only 15% per unit, requires no pre-purchased capacity, and can be expanded on demand. Cost savings are captured by:

$$Save = \frac{1 \times N}{(0.15 + P \times 1 \times N) \times S} \tag{1}$$

where S denotes spatial utilization, P the hot-data ratio, and N the replica count. With $S = 80\%$ and $N = 3$: 1) At $P = 10\%$, storage cost drops 8.33×. 2) At $P = 20\%$, savings remain near 5×, significantly

outperforming traditional solutions. 3) At $P = 50\%$, the ratio falls to 2.27× still far exceeds conventional approaches.

This design dynamically adapts to business workloads, balancing cost efficiency with data reliability while reducing redundant storage overhead. CLog cloud-disk allocation shrank from 3× memory to 1:1 with memory, freeing capacity. Combined with inexpensive OSS, these optimizations preserve low-latency hot-data access under high concurrency and balance storage cost against performance through flexible elastic expansion.

3 SYSTEM IMPLEMENTATION

3.1 Overview

Beyond standard database components, we distill five key modules in OceanBase Bacchus: **Shared Log**, **Metadata Management**, **Cache Warming**, **Baseline Compaction**, and **Garbage Collection**. They integrate cleanly with clear responsibilities, trimming storage while improving stability, consistency, and availability. We describe them as a *production* integration path: each technique has precedents in isolation, but Bacchus foregrounds **service-oriented PALF logging** (sharded streams, 2PC across leaders) and the **Shared Block Cache Service** as the primary technical surfaces, with LSM-on-object storage as the cost-oriented capacity substrate; the contribution is how these pieces are wired together under OceanBase's transactional, metadata, and operational constraints. Figure 3 illustrates the information flow between these modules.

Each module serves a distinct purpose: **1) Shared Log** manages consistency, updates, and recovery. **2) Metadata Management** holds foundational metadata for all hierarchical modules. **3) Cache Warming** accelerates access to hot data and frequently used metadata. **4) Baseline Compaction** periodically consolidates and persists large data volumes. **5) Garbage Collection** reclaims obsolete version files and idle system resources.

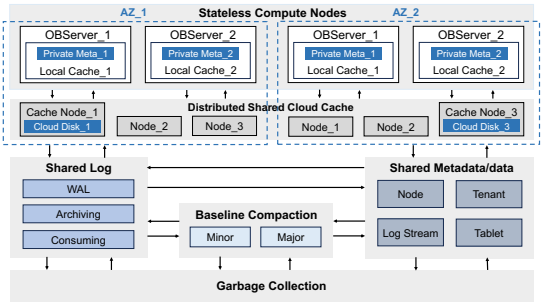


Figure 3: The information interaction among main modules

3.2 Shared Service-Oriented Logging

3.2.1 Shared Log. Inspired by log and data separation [14], we adopt a design where multiple partitions share one log stream, achieving efficient log sharing across clusters. We implement service-oriented logging based on PALF [25]: logs persist and synchronize via PALF on LogServers in the shared storage layer, while ECS cloud disks provide durable and nonvolatile local caches for vital data. Figure 4 illustrates the overall process.

Write Coordination and Concurrency Control. Bacchus employs a sharded write model rather than true multi-writer: we partition data into log streams mapped to logical partitions (e.g., tablets). Within each stream, one leader appends CLog to PALF on the LogServer—single-writer-per-stream avoids conflicts at the log. The database layer handles leader election after failures. Writes to the same partition serialize through its leader without heavy distributed locking; RW nodes scale horizontally by leading disjoint streams while preserving per-partition write order.

For foreground transactions, CLog entries land in local cache first for low latency. Replicas follow leader/followers analogous to *Coordinator* and *Participant* [24]; leaders relocate historical CLog to the log service. For OLTP, near-real-time archiving supports PITR (Point-in-Time Recovery) [19]: it aggregates cloud-disk writes, uploads incrementally via *Append* and *MultiUpload*, and supports active flush for faster snapshots.

PALF implements Paxos [30] for strong consistency and durability. Operations emit CLogs replicated to standbys; consistency holds once a majority persist the log, and minority failures neither lose data nor interrupt service. PALF optimizes Multi-Paxos via batching and pipelining—batching cuts round-trips and raises throughput, while leaders pipeline proposals as earlier entries complete, overlapping hot-path consensus. *Pipelining is not unique to Paxos:* Raft and leader-based stacks also pipeline `AppendEntries` while prior entries replicate [39]. We do not claim inherent latency benefits of "Paxos pipelining vs. Raft"; batching and pipelining in the LogServer service matter because log persistence sits on the commit critical path in our shared-storage deployment.

Commit Barrier vs. Asynchronous Archival (RPO=0 vs. PITR). Bacchus explicitly separates two distinct durability paths. *(1) Commit path (RPO=0):* a transaction commit is acknowledged to the client only after the commit CLog is replicated by PALF and **persisted by a Paxos majority (quorum)** of LogServers for that stream. PALF batching/pipelining improve throughput and overlap rounds but do *not* acknowledge commits before quorum persistence. *(2) Archival path (PITR):* near-real-time archiving uploads cloud-disk logs to object storage asynchronously for PITR and snapshots; archival progress (and transient lag) does not gate commits and is not required for RPO=0.

Replica Topology and Failure Model. In our default deployment, each PALF stream replicates to **three** LogServers across **three AZs** in-region (2-of-3 commits). RPO=0 holds while a quorum survives (e.g., one LogServer or single-AZ loss). The scenario that can break RPO=0 is **simultaneous loss of a majority** of LogServers for the stream before corresponding records reach object storage.

Multi-cloud Clarification. Bacchus supports multi-cloud deployment across providers. By default, PALF on the commit path is synchronous *within a region* (across AZs) rather than synchronously across clouds/regions, since cross-cloud latency would dominate commits. Cross-cloud DR can use asynchronous archival to object storage and replay on a warm backup cluster.

Each node consumes logs through a unified iterator: local cloud-disk logs are consumed first; if unavailable, shared service logs apply. After CLog relocation, replicas reclaim local files. GC reclaims shared and local CLogs based on the minimum replay position and relocation progress.

3.2.2 Shared Storage Log. Shared Storage Log (SSLog) is a special CLog serving as WAL for metadata. Frequent metadata updates

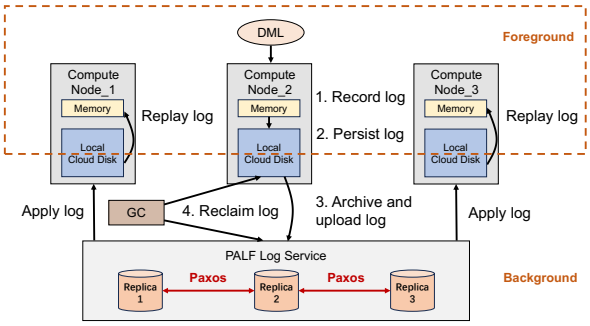


Figure 4: The workflow of Bacchus shared log

risk excessive I/O; we introduce region-level SSLog in the log service, with SSLog for metadata and object storage for data. SSLog stores KV tables, turning costly shared-storage I/O into cheaper log-service I/O. Like Iceberg [10] and Delta Lake [11], we aggregate metadata in SSLog before flush to cut overhead. Each tenant has one SSLog stream with a single leader responsible for writes (§ 3.3).

SSLog benefits scenarios with non-data modifications (ID changes, marker files) and partitions with only metadata changes (empty MemTable dumps, compaction without data, heartbeats). We write metadata to SSLog and confirm via the SSLog tablet. RW nodes write to SSLog instead of sharing metadata; RO nodes poll SSLog to update local metadata.

3.3 Metadata Management

Metadata is hierarchically organized (node, tenant, log stream, tablet) as small files. Bacchus distinguishes the Sys Tenant Cluster (system metadata and control plane) from User Tenant Clusters (user data and application workloads). Each level has node-private and shared types. A metadata service manages metadata, with SSLog stored in the log service. Self-contained metadata with independent lifecycles reduces reliance on global indexes, lowering complexity. We promote block index micro-blocks to memory/cache, reducing random I/O reads. Different metadata levels use different write strategies. Log stream level and above use write-through (promptly updated to shared storage) because there are no multiple versions and lower frequency.

Tablet level and below use write-back (asynchronous persistence) for multiple versions and real-time performance. Since metadata files are separate, we adjust OceanBase 2PC [59, 60]: prepare phase generates metadata files; commit phase updates parent-level files, ensuring consistency. For shared metadata, concurrent updates by different nodes are managed by a Shared Storage Writer (SSWriter) node per region. Shared tablet-metadata modifications go through SSWriter. SSWriter broadcasts changes to other nodes, which trigger local updates upon detection.

Table-level Metadata Changes. Table-level metadata changes (e.g., schema modifications, partition splits, table drops) require coordinating database and storage layers. The database layer writes to SSLog, then updates metadata-service files. SSLog replay propagates changes: RO nodes poll SSLog and apply locally; RW nodes read SSLog when needed. Atomic table operations (e.g., table drop) use prepare (intent in SSLog) and commit (parent-level metadata), preserving visibility despite transient offline nodes. During table-level operations, the database layer maintains transaction isolation by tracking active transactions referencing the table, ensuring ongoing queries complete before metadata changes take effect.

Versioning. When a new SSTable is generated for each partition, tablet metadata is created simultaneously. Tablet metadata is multi-versioned; each version lists all SSTables it depends on. Several versions are retained and obsolete ones are garbage-collected. SSTables exclusively referenced by reclaimed metadata are recycled, purging corresponding cache entries locally.

3.4 Replication and Migration

The replication and migration process of OceanBase Bacchus has been adjusted to leverage shared storage. The primary change is that baseline data is directly shared from shared storage, while incremental data and warm data are shared from the distributed cache. Only a small amount of data needs to be copied from the source node to the target node, such as the hottest data in local cache and node-private data/metadata.

In Bacchus, node replication migration proceeds as follows. First, a new log stream is created at the target without initiating replay (step 1). Next, a suitable source end is selected (step 2). The stream goes offline; target metadata updates from PALF and source-stream information (steps 3–4). Empty tablet shells at the target embed source metadata only. Step 5 copies necessary private information from the source server. Step 6 brings the stream online, preparing log filtering and replay according to the checkpoint SCN (System Change Number) in tablet metadata. In steps 7–8, tablets parallel-copy local cache data and obtain baseline from object storage plus dumped distributed-cache data; the target replays logs until the position advances far enough for freshness. Finally, after tablets complete data reference, the member list is updated and migration status is cleaned up, updated, and reported (steps 9–10).

4 INCREMENT AND COMPACTION

The compaction process in OceanBase Bacchus is categorized into two types: Dumping (Micro/Mini Compaction) and Merging (Minor/Major Compaction [16]).

4.1 Shared Incremental Data

Incremental data in local cache must reach OSS promptly; Bacchus adopts fast dumping for shared CLog and incremental data (Figure 5). User data is written into MemTable; transaction persistence is via CLog. When MemTable crosses a memory threshold, mini compaction freezes it into a mini SSTable, freeing memory (a logging "checkpoint"). Crash recovery and replica addition replay MemTable data, so we introduce micro compaction (micro SSTables): emitting micro SSTables before freezes advances checkpoints, trims elastic scaling cost, and speeds crash recovery and replica loading. Nodes independently schedule mini/micro SSTables to release memory quickly without hurting OLTP. Micro and mini SSTables reside on local disk caches until a replica uploads them in the background to OSS. Minor compaction [16] merges multiple micro/mini/minor SSTables in shared storage into one minor SSTable.

During minor compaction, macro-block reuse bounds write amplification; afterward GC can delete input SSTables to save space,

yielding a faster dump-and-persist cycle. Concurrent replicas contend for shared writes while the object storage API lacks mutual exclusion, so each log-stream leader elects a lower-load replica as Shared Storage Writer (SSWriter); during its lease only that replica may execute object-storage writes for all tablets under the stream.

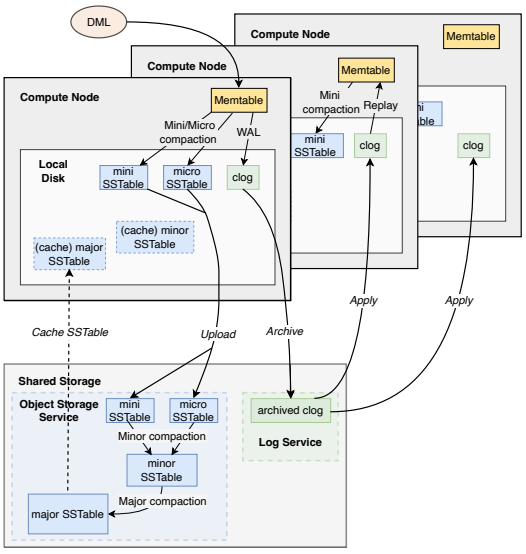


Figure 5: The workflow of Bacchus compaction

4.2 Major Compaction

We design Major Compaction (MC) for object storage. When mini compactions (dumps) exceed a threshold or during daily off-peak, the system merges the baseline (Major) SSTable with incremental Mini/Minor SSTables into a new Major SSTable in seven phases. Daily compaction separates frontend and backend: it runs chiefly in the shared storage layer, independent of the database layer, with compute nodes unaware of the background work.

Algorithm 1 and 2 illustrate the process: 1) The Root Service (RS) initiates daily MC; a User Tenant Cluster compute node detects the pending merge task. 2) The leader schedules tablet merging, distributes compaction tasks, and records them in the shared storage metadata service. 3) The shared storage node executing MC runs major compaction, merges minor/major SSTables into one Major SSTable, and stores it in object storage. 4) The execution node updates tablet metadata and marks the merge task completed. 5) User Tenant compute nodes detect completion via SSLog replay, reference and warm the new data, and report the replica's CRC [41] checksum to the internal table. 6) The RS monitors warm-up via the internal table, verifies checksums between primary and index tables and cross-region replica consistency, ensuring data consistency. 7) After successful RS verification, daily compaction completes.

4.3 Compaction Offloading

We have offloaded minor/major compaction from the database layer to the shared storage layer, where it runs on shared-storage machines and isolates foreground from background work. Because major compaction is compute-intensive, the layer can also push

Algorithm 1 Major Compaction Algorithm in Database Layer

Require: Root service RS, metadata service MS, leader l, replicas $F = \{f_1, \ldots, f_n\}$; each f_i holds cached SSTable(s) D_i; checksum $\mathbf{crc} : D \to \mathbb{R}$ for SSTable D.

Ensure: Major Compaction (MC) that is adapted to shared storage.

1: RS launches MC. l receives MC.
2: l schedules tablets and $MS \leftarrow PrepareMC(l)$
3: Call Algorithm 2 and get Major SSTable D_{baseline}
4: $checksum_{\text{exec}} \leftarrow \mathbf{crc}(D_{\text{baseline}})$
5: **for** $f_i \in F$ **do**
6: $D_i \leftarrow PreheatCache(f_i, D_{\text{baseline}})$, $checksum_i \leftarrow \mathbf{crc}(D_i)$
7: **if** $checksum_i \neq checksum_{\text{exec}}$ **then**
8: Retry
9: **end if**
10: **end for**
11: RS checks $\mathbf{crc}(PrimaryTable) = \mathbf{crc}(IndexTable)$

Algorithm 2 Major Compaction Algorithm in Shared Storage Layer

Require: Metadata service MS and shared storage node N executing MC.

1: $D_{\text{increment}} \leftarrow MinorSSTable, D_{\text{baseline}} \leftarrow MajorSSTable$
2: $N \leftarrow SelectMCNode(MS)$
3: $D_{\text{baseline}} \leftarrow MC(N, D_{\text{baseline}}, D_{\text{increment}})$
4: $MS \leftarrow SendMCMessage(N)$
5: **return** A major SSTable D_{baseline} in shared storage.

work to compute nodes, further separating storage and compute while exploiting idle resources.

With shared storage supporting metadata, transaction tables, and incremental data, we offload compaction compute to idle machines that consume the shared data. The process is as follows: (1) choose an offload machine; (2) create a log stream carrying tablet, data, and transaction information for compaction; (3) the log-stream leader designates SSWriter—during its lease only SSWriter may perform writes on shared storage; (4) load tablets for SSTable lists, run compaction, and update metadata; (5) preload new data into RW/RO local caches. After checksum verification, the machine is released to the underutilized resource pool.

5 CACHING AND PREHEATING

5.1 General Cache and Cache Warming

The local cache splits into tablet metadata, dump and temporary files, and micro-blocks; the first two are small and typically fully cached. The micro-block cache employs ARC (Adaptive Replacement Cache) [34], which uses access history to tune replacement and combines LRU with LFU as a mainstream LRU alternative. Ghost lists track recent evictions so admission stays workload-aware. Bacchus keeps LRU and LFU blocks in local cache; ghost cache blocks live in object storage.

To ensure stable throughput and latency, we provide multiple preheating methods for different scenarios:

- **Baseline Switching Preheating**. Bacchus shares baselines with shared logs. During a major version switch, we proactively warm

the new baseline by loading hot macro-blocks into local cache to minimize performance fluctuations.

• **Leader/Follower Preheating**. When switching leader/follower, the leader derives access order from the log stream, periodically syncs it to followers, and followers warm micro-block caches so leader and follower caches stay aligned.

• **Replication Migration Preheating**. During migration, incrementals come from the Shared Block Cache Service and baselines from shared storage; hot macro-blocks copy source-to-target and populate the destination micro-block cache.

• **Cloud Disk Scaling Preheating**. When local cache disks scale up or down, ARC LRU/LFU lists exchange entries with ghosts: scale-up grows lists and recalls ghosts; scale-down moves entries to ghosts and trims ghost lists as needed.

Cache warming involves read-triggered and proactive inter-node communication. Read-triggered warming periodically scans the in-memory KV cache for micro-blocks touched within a time window, persists those incremental micro-blocks locally, then synchronizes them to other nodes. Proactive warming loads hot data from user rules or Bacchus heat-detection policies.

5.2 Distributed Shared Cache

Object storage has high latency (100ms+) and limited bandwidth; local caching alone (ms+) is insufficient for core OLTP. Frequently modified data needs a distributed cache to alleviate OSS bandwidth and shield bursty writers. Bacchus follows AWS S3 Express One Zone [6] (macro-block cache when cross-DC is unnecessary) and Socrates's two-layer cache [7] (sparse hottest-page layer plus dense layer). We develop Shared Block Cache Service (Figure 6) on BlockServers that store and serve macro-blocks. Memory, local compute cache, and distributed cache hold hottest, second-hottest, and warm data; granularity rises from micro-blocks locally to macro-blocks in the distributed tier, and tiers preheat in sequence.

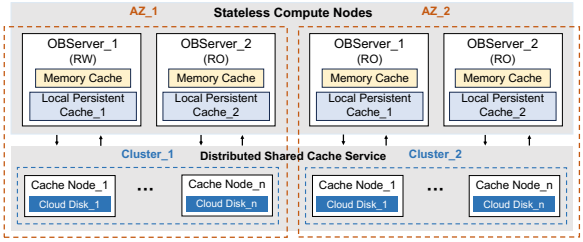


Figure 6: Three-level cache hierarchy

The distributed cache is read-only and cluster-independent. Each AZ deploys a set shared by RO/RW peers, eliminating redundant copies and saving resources. BlockServers cache on durable local cloud disks for crash-safe restart. Shared cache minimizes data copying during scaling/migration (mostly metadata) and isolates cache scaling from compute nodes. ECS-heavy cache nodes remain I/O-intensive; Bacchus elasticity still lowers aggregate compute. Compared with third-party caching services, OceanBase's self-developed shared caching framework is tailored to the database and its access model, achieving better performance and stability.

5.3 Concurrency Control in Multi-level Caching

The three-level cache architecture (memory, local cache, distributed cache) requires careful concurrency control to maintain consistency. When data is modified in the database layer, the system applies write-through to the memory cache and write-back to local and distributed caches. SSLog coordinates invalidation: when metadata changes (e.g., SSTable updates after compaction), the change is logged and all nodes invalidate matching entries upon replay. For the distributed cache we use version-based invalidation: entries carry version tags, writers increment versions and record updates in SSLog, and other nodes detect mismatches when reading cached data and refresh from shared storage. Thus stale cached data is never served, even with concurrent updates across nodes.

6 GARBAGE COLLECTION

Garbage collection in Bacchus must coordinate with the database layer to ensure correctness while reclaiming obsolete data. The system tracks data versions using SCN and maintains visibility information to determine when data can be safely reclaimed.

6.1 Global Garbage Collection Coordination

Global GC reclaims shared CLog files and obsolete SSTables across compute nodes, coordinating deletions to prevent concurrent conflicts and preserve consistency. We employ lease-based coordination akin to SSWriter but specialized for GC: leader election installs a *GC Coordinator* per log stream to orchestrate GC for that stream. The coordinator holds a time-bound lease (typically 30–60 seconds) from the metadata service, recorded in SSLog for visibility. During the lease only that coordinator may delete shared storage objects (CLog files and SSTables) for its stream, avoiding races where multiple nodes delete the same file.

The procedure has four steps: (1) *Lease Acquisition*: The GC Coordinator renews by writing lease records to SSLog with timestamps and expirations; if renewal fails the lease expires and a new coordinator is elected. (2) *Safe Reclamation Point Calculation*: Before deletion it queries the metadata service for global min_read_scn and the minimum log replay position across nodes, only marking data that predates both. (3) *Atomic Deletion*: It logs a deletion intent listing files, waits briefly so peers observe the intent, then deletes; partial failures recover via the intent and avoid orphaned references. (4) *Metadata Synchronization*: After successful deletion it strips metadata references to removed objects, and SSLog replay propagates updates so every node shares a consistent view.

Renewals use exponential backoff: if the coordinator cannot renew (e.g., network partition) it stops and releases its lease so others can elect a successor. A two-phase protocol marks files "pending deletion" during prepare and commits deletion only after confirming no active transactions reference them. Together this keeps GC safe, consistent, and highly available.

6.2 Cross-Partition Transaction Coordination

Although each log stream has a single leader writer, transactions may span multiple streams and need partition coordination. Bacchus runs the OceanBase 2PC protocol [60, 62]: a coordinator node collects prepare votes from participating leaders, each appending prepare entries to its PALF log stream; after unanimous prepare it

writes commit and every participant appends commit logs. This ensures atomicity across partitions: either all log streams commit the transaction or all abort. The PALF service's consensus mechanism guarantees that commit decisions are durable and consistent across all replicas. Because each stream keeps leader-ordered writes while cross-partition work stays at the transaction layer, Bacchus achieves partition-level parallelism with global transaction consistency. Distributed deadlock detection and resolution in this setting use LCL [57] and LCL+ [58] algorithms.

6.3 Handling Long-running Transactions

Long-running transactions [21] challenge GC by pinning historical versions that would otherwise be reclaimable. Bacchus relies on multi-version visibility: each active transaction receives a read SCN at start that bounds accessible versions. The database layer maintains global *min_read_scn*—the minimum active read SCN across nodes representing the oldest live read point, and GC reclaims only versions older than *min_read_scn*, ensuring that no active transaction loses access to required data. When a transaction exceeds a configurable timeout threshold, the database layer either aborts it or promotes its read SCN to prevent indefinite data retention. This coordination mechanism is managed through the metadata service: each compute node periodically reports its local minimum read SCN, and the metadata service aggregates these values to compute the global *min_read_scn*, which is then broadcast to all nodes for cluster-wide GC decisions.

7 PERFORMANCE EVALUATION

7.1 Experimental Configuration

We evaluate Bacchus (OceanBase 4.4.x), Paetica (OceanBase 4.0) [59], HBase [9], and StarRocks [49]. Experiments cover **OLTP** (SysBench [2] and write/read benchmarks vs. HBase, Aurora, TDSQL, PolarDB) and **OLAP** (TPC-H [4], TPC-DS [3], ClickBench [1] vs. StarRocks). §7.4 complements average-case traces with *adverse-cache* customer cases—cold starts and mixed hot/cold access, quantifying OLTP tail latency when object storage is exercised.

§7.2.1 uses ecs.r7.2xlarge in Alibaba Cloud with ecs.g7.xlarge for log storage. §7.2.2 deploys OceanBase and PolarDB in Alibaba Cloud (32 cores). §7.2.3 uses db.r5.xlarge on AWS. §7.2.5 uses two configurations: (1) 32-core/256GB and 48-core/384GB Intel Xeon Platinum 8575C for TPC-H 100GB and 1TB; (2) 16-core/64GB Intel Xeon Platinum 8369B for TPC-DS and ClickBench. §7.5 deploys OceanBase in Alibaba Cloud (32-core 8575C).

Unless stated otherwise, experiments adopt vendor default tunings on public clouds so comparisons reflect typical managed service deployments. Workload-specific overrides are called out next to the relevant figures and tables whenever they apply.

7.2 Performance Comparison

7.2.1 OceanBase 4.4.x vs. HBase 1.4.7. With the dataset fixed at 500 GB, we compared sustained write throughput between Bacchus and HBase. Figure 7 shows similar averages (3,770.86 vs. 3,993.79 ops/s), yet HBase intermittently falls to zero when foreground writes exceed flush/dump capacity, triggering blocking; raising flush/compaction frequency reduces stalls but cuts peak write rate. Bacchus posts a lower peak than HBase but avoids stalls through fast dumps and delivers comparable overall throughput.

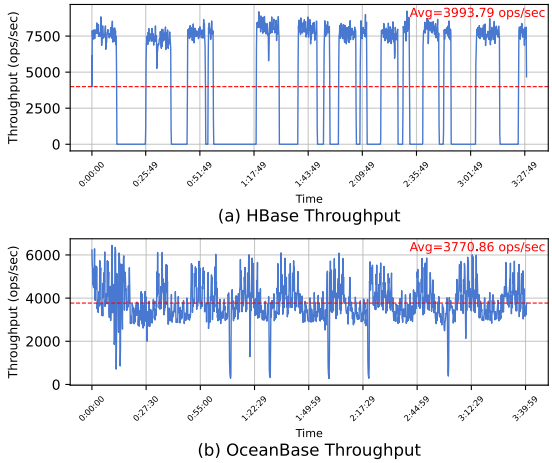


Figure 7: Write throughput in OceanBase and HBase

We further compared Bacchus against Apache HBase derivatives: Alibaba Lindorm HBase 2.0, Tencent Cloud HBase (EMR Serverless), and OceanBase 4.2.5. All clusters used two nodes except Tencent (vendor minimum three nodes), uniform hardware (16 cores, 32 GB RAM, >1 TB storage), 200 clients, an empty-table PUT stress test, and GET over 500M rows.

On PUT, Bacchus delivers the highest aggregate and per-node TPS (51,612 / 25,806) with substantially better mean and percentile latency (mean 41.8% of legacy OceanBase). On GET, Bacchus slightly trails three-node Tencent in total QPS but leads per-node QPS (28,208) and response time. Details appear in Table 1 and Figure 8.

Table 1: TPS/QPS in OceanBase and HBase

Product	Node	TPS/QPS per node	
		PUT	GET
OB v4.2.5	2	22,666	24,400
OB v4.4.x (Bacchus)	2	**25,806**	**28,208**
Lindorm HBase	2	22,014	8,000
Tencent HBase	3	13,590	20,200

7.2.2 OceanBase 4.4.x vs. PolarDB MySQL 8.0.1. On 32 vCPUs and 128 GB RAM we loaded 30 tables with 1M rows each, benchmarked *Point Select*, *Read-Only*, *Write-Only*, and *Read–Write*, and reported aggregate average TPS. Figure 9 shows OceanBase ahead on throughput with only modest latency overhead; throughput stays high as concurrency rises. At 1,600 threads PolarDB fails whereas OceanBase continues under load—highlighting both higher peak capacity and operational resilience in these scenarios.

7.2.3 OceanBase 4.4.x vs. Aurora 3.08. Figure 10 summarizes 32C-128GB experiments on 30 non-partitioned tables (500K rows). Across *Point Select*, *Write-Only*, *Insert*, and *Update*, OceanBase achieves much higher TPS under high concurrency (400–1,500 threads), exceeding Aurora by 2× on demanding *Insert*/*Update*, yet *Read-Only* latency spikes into hundreds of milliseconds at 1,500 threads while Aurora stays calmer on read-heavy mixes. Under the same data

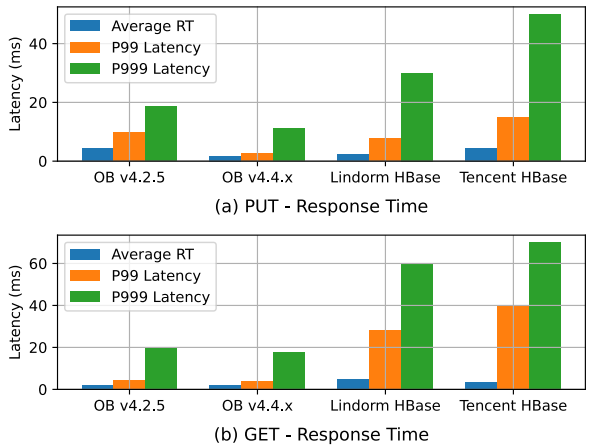


Figure 8: Response time in OceanBase and HBase

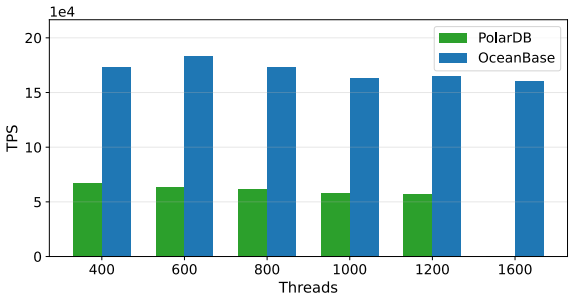


Figure 9: Average Throughput in OceanBase and PolarDB

scale and defaults, OceanBase favors write throughput; Aurora balances read latency better.

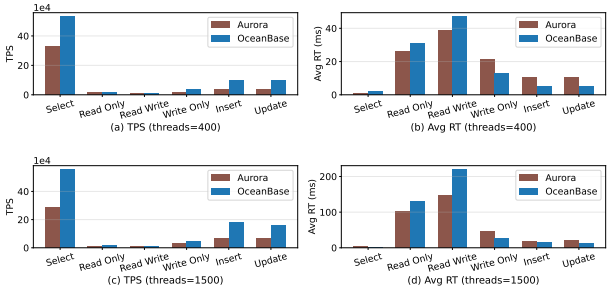


Figure 10: Throughput and Latency in OceanBase and Aurora

7.2.4 OceanBase 4.4.x vs. TDSQL MySQL 8.0. Figure 11 summarizes our 32C-128GB experiments on 30 non-partitioned tables with ~500,000 rows. OceanBase and TDSQL [32] highlight complementary strengths. On read-heavy workloads (*Select*, *Read-Only*), OceanBase leads on TPS at 400 and 1,500 threads, more than doubling TDSQL on *Select*. As concurrency rises across mixed and write-intensive scenarios (*Read-Write*, *Write-Only*, *Insert*, *Update*), OceanBase generally achieves higher peak throughput (catching up by 1,500 threads on *Insert*/*Update*). Overall, mean RT advantages flip by scenario, yet averaged across all six workloads TDSQL is slightly slower than OceanBase. In sum, OceanBase skews toward sustained throughput, while TDSQL occasionally delivers tighter latency control under contention.

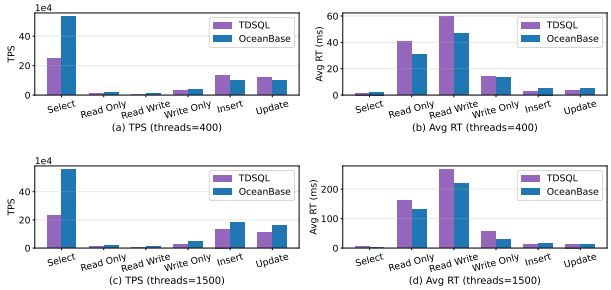


Figure 11: Throughput and Latency in OceanBase and TDSQL

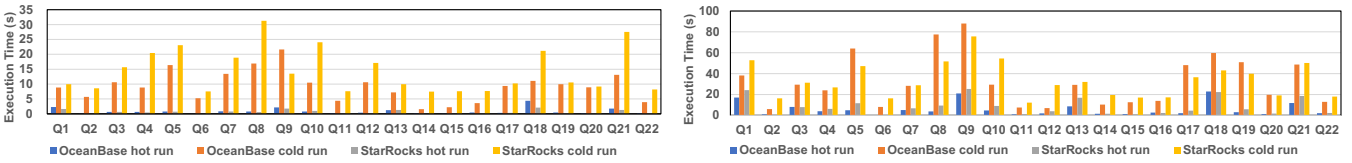
7.2.5 OceanBase 4.4.x vs. StarRocks 3.3.13. Although the introduction stresses OLTP, Bacchus also targets HTAP/OLAP; those techniques largely apply to OLAP without trading away OLAP headroom for OLTP wins. We evaluate **analytical query performance** using TPC-H [4], TPC-DS [3], and ClickBench [1], comparing against StarRocks [49]—a leading engine and our prior production best—to show the architecture still preserves OLAP. Each system (OceanBase 4.4.x and StarRocks 3.3.13) runs its **own** optimizer and plans; tests are end-to-end with identical queries, scale, and cold/hot methodology (hot runs preload data; cold runs do not). Bacchus retains OceanBase's planner while changing only **storage** (shared/object stores plus multi-layer caching), so differences primarily reflect the storage layer. Figure 12 presents TPC-H 100GB and 1TB; TPC-DS 100GB (99 queries) and ClickBench (43 queries) are detailed in [54].

For TPC-H 100GB (Figure 12(a)), Bacchus is ~48.5% faster than StarRocks, with about one-third of queries approaching 2× speedups (>90%). Q14 gains the most (327.93%); Q9 regresses the most (35.71%) on aggregated profit by nation/year (grouping, sorting, aggregates, subqueries), likely stressing the optimizer. For 1TB (Figure 12(b)), cold runs improve ~40% whereas hot runs move little (Q9 again weakest at 3.0%). These results show that shared-storage optimizations do not regress query performance and that adaptive object-storage tuning keeps Bacchus comparable to or faster than a leading analytical engine on these workloads.

7.3 Multi-Layer Cache Hit Ratio in Production

To validate Bacchus's multi-layer caching (§5) under production workloads, we collected cache hit ratios from two deployments over one week. We report three metrics: (1) overall hit ratio, (2) shared macro-block hit ratio, and (3) private macro-block hit ratio. Deployment 1 runs **OLTP**; deployment 2 runs **HTAP**, mixing OLTP and OLAP. OLAP queries tolerate looser latency than OLTP because they tend to issue sequential, large-block scans suited to OSS bandwidth and because analytics SLAs generally exceed OLTP targets; consequently some HTAP reads may intentionally hit OSS to balance cost instead of caching everything.

Figure 13 plots deployment 1 (OLTP): private macro-blocks remain at 100%; shared macro-blocks and overall ratios stay near 100% with brief dips during workload shifts. Higher tiers absorb



(a) TPC-H 100GB

(b) TPC-H 1TB

Figure 12: Query performance of OceanBase and StarRocks with different benchmarks

micro misses, keeping aggregate hits high and OSS reads minimal—demonstrating resilient OLTP behavior across varying conditions. Because misses are absorbed before they reach object storage, operators observe millisecond-scale reads for hot tenants even as compute scales elastically.

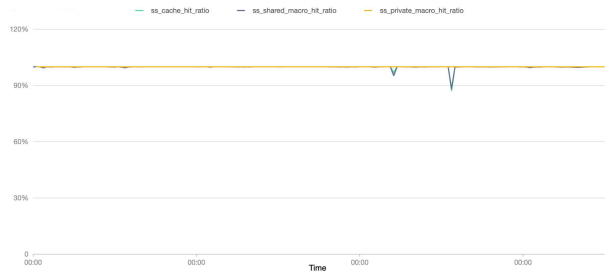


Figure 13: Cache hit ratios for the first production workload (OLTP) over the observation period.

Figure 14 shows deployment 2 (HTAP), mixing OLTP and OLAP like typical HTAP applications. Private macro cache remains the steadiest (often 100%), while shared macro-block and overall ratios track each other with shallow dips that rebound quickly—many coincide with OLAP scans or cold reads where OSS latency is acceptable. Together with three-tier caching and warming (§5), Bacchus keeps latency-sensitive OLTP paths hot while admitting OSS reads for colder analytics to balance performance and cost.

Figure 14: Cache hit ratios for the second production workload (HTAP) over the observation period.

Tail latency and caching. As in other designs that place a high-latency, capacity-priced object tier behind caches, Bacchus reads are *cache-sensitive*: requests that miss past Memory Cache, Local Persistent Cache, and the Shared Block Cache Service pay OSS latency and widen percentiles, while hits stay on millisecond local or zone-local paths (§2, §5). Shared Block Cache is therefore a *latency hedge* for shared macro-blocks on elastic compute, not a stand-alone fix—it works with higher tiers to keep aggregate hit ratios high (Figures 13–14). Mitigations include BlockServer scaling, cache warming, hot-data identification, and I/O aggregation for large sequential reads (§8). Cheaper OSS still offloads cold reads from fast media; meeting strict OLTP tail SLOs requires tuning and monitoring cache coverage like other disaggregated stacks. §7.4 reports numeric cold-cache and partial-miss behavior.

7.4 Tail Latency Under Adverse Conditions

Production traces above capture typical hit ratios. To study OLTP *tails* when caching fails or is cold—central for object-storage-backed SQL—we summarize two **customer cases** sized like real tenants on public clouds; configurations mirror production quotas rather than toy clusters.

Case A (time-series telemetry, AWS). Three-zone Bacchus stores ~3.7 TB SST on S3 with ~20% local cache relative to data (~8% read-cache fraction). After flushing memory KV and local SSD caches (*cold start*), replay begins at mean 255 ms and P99 8.32 s while micro-blocks refill from OSS (50–100 ms single-read latency); means drop <10 ms after ~20 min, <5 ms after ~67 min, and <3 ms after ~165 min, reaching steady 86–99% local hits and ~96% block-cache hits. Warm SS vs. SN averages are 2.51 ms vs. 1.37 ms but SS P99 stays wider (82.7 ms vs. 57.3 ms), illustrating tail inflation when rare paths hit OSS or disk stragglers—P99 tracks caching health more than averages. Table 2 summarizes phases.

Table 2: Cold-start read replay (Case A): end-to-end SQL latency after clearing memory KV and local SSD caches.

Phase	Mean (ms)	P99 (ms)
Immediately after cache flush	255	8,320
~20 min replay	<10	—
~67 min replay	<5	—
~165 min (stable)	<3	—
Stable hits: local cache 86–99% (per node); block cache ~96%.		

Case B (order history; point, range, insert). We stress hared-storage (SS) (OceanBase 4.4.x with time partitions and global indexes) vs. Shared-nothing (SN) such that ~90% of queries hit a *hot* window and ~10% fall outside—about one OSS read per ten requests, deliberately worse than many deployments. Table 3 lists mean, P90, and P99 (ms). Under this skew, SS `SELECT` and `INSERT` P99 climb to ~94 ms and ~91 ms compared with SN and steady hot-cache SS, which remain aligned through median and mid-percentiles; partial misses therefore dominate tail SLOs while frequent paths stay fast.

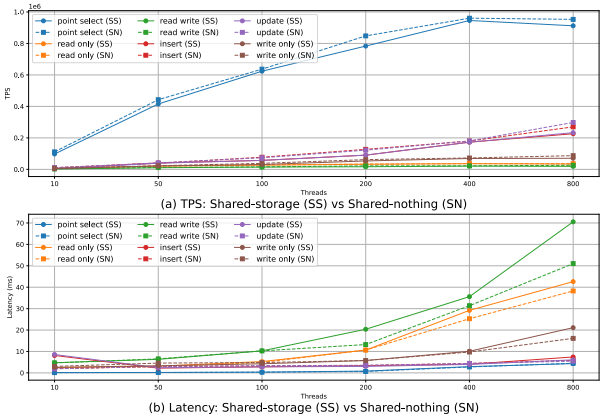
7.5 OB Shared-storage vs. Shared-nothing

In Alibaba Cloud we compared OceanBase under Shared-storage and Shared-nothing. The SysBench workload spans 30 tables with 500,000 rows each, and we benchmarked six scenarios spanning read/write mixes plus insert and update operations. Figure 15 shows

Table 3: Stress test (Case B): latency (ms) for SN vs. SS steady vs. SS under ~10% object-store read mix (adverse).

Scenario / stmt	Mean	P90	P99
SN SELECT	2.83	8.97	9.91
SS SELECT (steady, hot)	2.41	9.17	9.99
SS SELECT (~10% OSS)	5.16	40.82	94.19
SN INSERT	1.51	9.11	9.92
SS INSERT (steady)	1.82	9.13	9.95
SS INSERT (~10% OSS)	4.02	12.11	91.22

comparable TPS between SS and SN as concurrency scales. Although calling into external shared storage can add latency, Bacchus largely hides that overhead, leaving overall throughput aligned between architectures.



(a) TPS: Shared-storage (SS) vs Shared-nothing (SN)

(b) Latency: Shared-storage (SS) vs Shared-nothing (SN)

Figure 15: OceanBase in Alibaba Cloud

7.6 Storage Cost Comparison

We compared storage costs between Bacchus and OceanBase (Paetica): Paetica uses tightly coupled compute with EBS (GP2) on AWS, whereas Bacchus disaggregates persistence onto S3 and retains EBS for cache disks. Table 4 contrasts them at 100 TB.

Why Bacchus lowers replica counts. Shared-nothing deployments maintain multiple full replicas (e.g., three) for fault tolerance and consensus, so billable bytes scale with replica fan-out. Bacchus stores user data once in shared object storage; durability and availability leverage cloud replication plus PALF WAL replication, while stateless compute reads that single copy. Data replicas can therefore drop to one (e.g., User Tenant Single Replica Cluster) while logs still replicate for RPO=0. Consolidating cold bytes in OSS replaces triple-replicated attached disks for user data yet keeps PALF's independent redo replication for control-plane safety. Inexpensive, durable OSS drives the 59% (OLTP) and 89% (OLAP) savings in Table 4 while preserving the evaluated performance. "Data cache ratio" denotes business-selected hot-data fractions (e.g., 10%).

8 LESSONS IN PRACTICE

We summarize the lessons learned from the design and practice of shared storage in OceanBase Bacchus.

Lesson 1. Object storage has limited IOPS and bandwidth, along with high latency. It is crucial to avoid frequent small read/write operations and to aggregate small objects before storage, rather than storing them individually.

Lesson 2. In production, we configure a separate bucket for each cluster or tenant to leverage the I/O isolation and billing capabilities of object storage buckets.

Lesson 3. Multiple separation strategies are critical: storage and computation separation, read and write separation, log and data separation, hierarchical caching and data separation, and front-end and back-end separation. Service-oriented and pooled resources are equally important.

Lesson 4. The log service is crucial in our architecture, because it ensures statelessness of compute nodes and high performance of OLTP, with RPO=0.

Lesson 5. Combining local micro-block with shared macro-block caching is effective. The former optimizes OLTP point queries; the latter enables stateless compute nodes and better scaling.

Lesson 6. Various methods are employed to improve cache hit rates, including cache warming, automatic identification of hot data, elastic scaling of cache, and user-defined rules for hot data.

Lesson 7. Hierarchical, log-service-based metadata management achieves aggregation and enables efficient transactions and queries. It simplifies system implementation, especially for snapshots, and improves metadata access performance.

Lesson 8. In practice, the unified multi-cloud native shared storage architecture offers cost-effective and elastic scaling capabilities for OLTP, OLAP, and KV workloads alike.

Lesson 9. Read tails worsen when misses reach object storage; size elastic cache (including BlockServers), use warming, and monitor hit ratios—notably after cold starts or working-set shifts—to meet OLTP SLOs. Tiered LSM-on-OSS trades cost for tail sensitivity; the multi-layer hierarchy and these practices mitigate it.

9 RELATED WORK

9.1 Shared-storage Databases

Aurora [51] is a quorum-based [22] cloud-native relational database with "the log is the database": redo logs go to the storage layer, which asynchronously replays them into pages, cutting network overhead without specialized hardware. Compute nodes push log streams to a durable shared volume; storage applies logs to pages and handles GC, backup, and recovery, reducing compute burden. In the "single-writer, multi-reader + shared storage" model, PolarDB [55] combines a hierarchical modification tracker, Lamport timestamps, and RDMA so read-only nodes obtain strong-consistent reads without waiting for log replay. On PolarFS [15], a POSIX-like distributed file system for cloud databases, it emphasizes low latency and availability. PolarDB-MP [56] extends to multi-primary mode with disaggregated shared memory and storage, supporting concurrent read-write nodes with strong performance under high conflict. Databricks Lakehouse [61] pairs open formats (e.g., Parquet) with object storage and transactional metadata (ACID, versioning, management), mitigating staleness, reliability, cost, and lock-in relative to traditional warehouses.

Unlike Aurora/PolarDB (B+-tree-based, platform-coupled), Bacchus pairs LSM with OSS for multi-cloud [52]. Durable LSM-on-cloud stacks exist [28, 45, 48]; for RPO=0 and cross-node coordination it foregrounds **service-oriented PALF shared logging**, **Shared Block Cache Service**, multi-layer caching, and RW/RO

Table 4: Comparison of storage costs between old and new versions of OceanBase in OLTP and OLAP in 100 TB storage

OLTP				
Storage Architecture	**Storage Type**	**Monthly Cost per GB**	**Replica Count**	**Monthly Total Cost**
OceanBase (Shared-nothing)	EBS (GP2) Cloud Disk	$ 0.10	x3	$ 30,000
OceanBase (Shared-storage)	EBS (GP2)	$ 0.10	x1	$ 10,000
	S3 Standard	$ 0.023		$ 2,300
Storage Cost Savings				***59.0%***
OLAP				
Storage Architecture	**Storage Type**	**Monthly Cost per GB**	**Replica Count**	**Monthly Total Cost**
OceanBase (Shared-nothing)	EBS (GP2) Cloud Disk	$ 0.10	x3	$ 30,000
OceanBase (Shared-storage)	EBS (GP2) (e.g., data cache ratio 10%)	$ 0.10	x1	$ 1,000
	S3 Standard (pay-as-you-go)	$ 0.023		$ 2,300
Storage Cost Savings				***89.0%***

pipelines. Lakehouse targets analytics; Bacchus targets OLTP via hot/cold placement and caching to offset OSS latency.

9.2 LSM-tree–based vs. B+-tree–based Databases

The LSM-tree [40] stages writes in MemTable/SSTables: sequential MemTable/WAL ingest flushes immutable SSTables with sequential I/O, avoiding random writes. Examples include LevelDB [23], RocksDB [20], Cassandra [8], and TiDB [27]. The B+ tree [17] stores fixed-size pages (e.g., 4KB) with in-place updates (splits/merges); reads are $O(\log N)$ with efficient ranges [47]. Examples include MySQL (InnoDB) [38], WiredTiger [37], and PostgreSQL [42].

LSM trees excel at writes but amplify reads/writes; compaction burns CPU, I/O, and space without tuning [33]. B+ trees predict reads for OLTP yet suffer concurrent-write penalties (random I/O, splits, fragmentation). Hybrids narrow the LSM/B+ gap [29, 44]. Many LSM engines target local disks or shared-nothing replication (TiDB [27], CockroachDB [50]). OSS-backed stacks include Nova-LSM [28], RocksDB-Cloud [45], and SlateDB [48]. Bacchus is shared-storage *relational* infrastructure distinguished by **PALF**-centric logging (multi-writer streams, SSLog aggregation, 2PC across streams) and the **Shared Block Cache Service**. It also integrates OceanBase dump, compaction, and cache—not LSM-on-OSS alone.

Comparison with Distributed LSM-tree Databases. TiDB [27] and CockroachDB [50] replicate schemas, partitions, and manifests via Raft per node. Bacchus centralizes metadata in SSLog so RO nodes poll without joining consensus groups. Transactions route through PALF Multi-Paxos (batching, pipelining) on cross-AZ LogServers. Raft and Multi-Paxos both expose linearizable replicated logs; the contrast is *architectural*—central WAL plus SSLog versus replicated stores with per-node metadata—not a "Paxos vs. Raft" split on consistency or pipelining. SSLog aggregates metadata asynchronously, trimming compaction/schema I/O for elastic, stateless compute where the shared log anchors consistency instead of per-node state machines.

9.3 Shared Log Stream

LFS (Log-Structured File System) [46] treats disk as a circular log appending data and metadata; though single-machine, it influenced shared-storage designs. Corfu [12] builds a cluster-wide shared log for flash clusters, separating log storage from compute so many clients append; *Projection* targets performance. Tango [13] layers HA metadata with strong consistency atop Corfu.

SingleStore [43] disaggregates storage and compute, storing logs in blob storage [35], but relies on replication rather than a unified shared log. Socrates [7] (from Aurora) places redo in shared *XLOG*: one writer appends for throughput while replicas tail asynchronously. Unlike Corfu/Tango (general logs) or SingleStore/Socrates (B+-tree-based), Bacchus joins PALF logging with LSM-on-OSS for cross-cluster sharing without redundant data+log at compute, RPO=0, and multi-cloud deployment. Shared-log and cache differentiate the stack; LSM-on-OSS [28, 45, 48] is precedented, integrated with OceanBase's engine and operations.

Multi-Writer Support and Correctness. Socrates stays single-writer (the primary appends to XLOG), capping write scalability. Bacchus shards the WAL into streams whose leaders append CLog to PALF, letting RW nodes lead streams and scale writes. Correctness rests on: (1) *Partition-level serialization*: each leader serializes its stream; (2) *Cross-partition coordination*: OceanBase 2PC across leaders for distributed transactions; (3) *PALF consensus*: Paxos per stream with majority quorum for durability; (4) *Metadata consistency*: SSLog exposes one unified metadata log to all nodes. That improves write throughput over single-writer designs while preserving ACID through ordering and 2PC; correctness is reflected in cross-partition writes, cross-stream atomicity, and SSLog-driven metadata propagation.

10 CONCLUSION

This paper introduces OceanBase Bacchus, a *production-grade* disaggregated shared-storage architecture built on OceanBase. We foreground **service-oriented PALF logging** with **sharded multi-writer log streams** and **2PC across stream leaders**, eliminating classic shared-log cloud SQL's single-writer append bottleneck while preserving ACID, plus the **Shared Block Cache Service**, which pools macro-blocks across elastic compute in an availability zone—less common than purely per-node caches and suited to **stateless compute**. An LSM-tree on object storage supplies low-cost durable capacity; ingredients still have precedents, and Bacchus's value is **engineering synthesis at scale** with our OLTP/OLAP measurements and storage-cost results. Shared PALF logging persists and replicates the WAL off the object-store commit path; BlockServers improve cache utilization; asynchronous background tasks protect foreground throughput. Collectively, these designs endow OceanBase Bacchus with low cost, elastic resource scaling, and superior resource utilization. As a cloud-native architecture balancing performance and cost, Bacchus advances public cloud databases.

REFERENCES

[1] 2025. ClickBench — a Benchmark For Analytical DBMS. https://benchmark.clickhouse.com

[2] 2025. SysBench. https://github.com/akopytov/sysbench

[3] 2025. TPC-DS. https://www.tpc.org/tpcds/

[4] 2025. TPC-H. https://www.tpc.org/tpch/

[5] Alibaba Cloud. 2025. *Object Storage Service (OSS).* https://www.alibabacloud.com/en/product/object-storage-service

[6] Amazon Web Services. 2025. *Amazon S3.* https://aws.amazon.com/s3/

[7] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, et al. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756.

[8] Apache Cassandra. 2025. *Apache Cassandra.* https://cassandra.apache.org

[9] Apache HBase. 2025. *Apache HBase.* https://hbase.apache.org

[10] Apache Iceberg. 2025. *Apache Iceberg.* https://iceberg.apache.org

[11] Michael Armbrust, Tathagata Das, Liwen Sun, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3411–3424.

[12] Mahesh Balakrishnan, Dahlia Malkhi, et al. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al. 2013. Tango: distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340.

[14] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 251–264.

[15] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, et al. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1849–1862.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.

[17] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[18] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, et al. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226.

[19] Jianjun Deng, Jianan Lu, Hua Fan, et al. 2024. TimeCloth: Fast Point-in-Time Database Recovery in The Cloud. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 214–226.

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (Oct. 2021), 32 pages.

[21] Chenguang Fang, Chen Qian, Qi Yang, Zeyu Wang, Zhenkun Yang, Fanyu Kong, Quanqing Xu, Hui Cao, Fusheng Han, and Chuanhui Yang. 2025. MaLT: A Framework for Managing Large Transactions in OceanBase. In *Companion of the 2025 International Conference on Management of Data*. 378–390.

[22] David K. Gifford. 1979. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '79)*. Association for Computing Machinery, New York, NY, USA, 150–162. https://doi.org/10.1145/800215.806583

[23] Google. 2025. *LevelDB.* https://github.com/google/leveldb

[24] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, Berlin, Heidelberg, 393–481.

[25] Fusheng Han, Hao Liu, Bin Chen, et al. 2024. PALF: Replicated Write-Ahead Logging for Distributed Databases. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3745–3758.

[26] Daokun Hu, Quanqing Xu, and Chuanhui Yang. 2025. OLTP Engines on Modern Storage Architectures. In *Companion of the 2025 International Conference on Management of Data (SIGMOD '25)*. ACM, 8. https://doi.org/10.1145/3722212.3725633

[27] Dongxu Huang, Qi Liu, Qiu Cui, et al. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084.

[28] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 749–763.

[29] Donguk Kim, Jongsung Lee, Keun Soo Lim, Jun Heo, Tae Jun Ham, and Jae W. Lee. 2024. An LSM Tree Augmented with B+ Tree on Nonvolatile Memory. *ACM Trans. Storage* 20, 1, Article 4 (Jan. 2024), 24 pages.

[30] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4* (December 2001), 51–58.

[31] Guoliang Li, Wengang Tian, Jinyu Zhang, et al. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3786–3798.

[32] Wei Lu, Zhanhao Zhao, Xiaoyu Wang, et al. 2019. A lightweight and efficient temporal database management system in TDSQL. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2035–2046.

[33] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (July 2019), 393–418.

[34] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: a self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (San Francisco, CA) *(FAST'03)*. USENIX Association, USA, 9.

[35] Microsoft. 2025. *Azure Blob Storage.* https://azure.microsoft.com/en-us/products/storage/blobs/

[36] MinIO. 2025. *MinIO AIStor.* https://min.io

[37] MongoDB. 2025. *WiredTiger.* https://source.wiredtiger.com

[38] MySQL AB. 2001. *MySQL.* https://www.mysql.com

[39] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 305–319.

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

[41] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. *Proceedings of the IRE* 49, 1 (1961), 228–235.

[42] PostgreSQL. 2025. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* https://www.postgresql.org

[43] Adam Prout, Szu-Po Wang, Joseph Victor, et al. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2340–2352.

[44] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, et al. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, Dean Hildebrand and Donald E. Porter (Eds.). USENIX Association, 69–82.

[45] Rockset Inc. 2019. RocksDB-Cloud: A Key-Value Store for Cloud Applications. https://github.com/rockset/rocksdb-cloud. Open-source RocksDB extension; durable cloud-backed files (e.g., S3).

[46] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[47] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2011. *Database System Concepts* (6th ed.). McGraw-Hill.

[48] SlateDB contributors. 2024. SlateDB: An Embedded Storage Engine Built for Object Storage. https://github.com/slatedb/slatedb. Open-source embedded LSM-tree on object storage.

[49] StarRocks. 2025. *StarRocks.* https://www.starrocks.io

[50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, et al. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509.

[51] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052.

[52] Quanqing Xu, Wei Sun, Chuanhui Yang, Jinlong Liu, Ziyun Wei, Fusheng Han, Liang Wang, and Xiaowei Zhai. 2025. OceanBase Unitization: Building the Next Generation of Online Map Applications. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 4183–4196.

[53] Quanqing Xu, Chuanhui Yang, Ruijie Li, Dongdong Xie, Hui Cao, Yi Xiao, Junquan Chen, Yanzuo Wang, Saitong Zhao, and Fusheng Han. 2026. OceanBase Mercury: Building a Distributed Real-time Analytical Processing Database System. In *2026 IEEE 42nd International Conference on Data Engineering (ICDE)*. IEEE.

[54] Quanqing Xu, Mingqiang Zhuang, Chuanhui Yang, Quanwei Wan, Fusheng Han, Fanyu Kong, Hao Liu, Hu Xu, and Junyu Ye. 2026. OceanBase Bacchus: a High-Performance and Scalable Cloud-Native Shared Storage Architecture for Multi-Cloud. *arXiv preprint arXiv:arXiv:2602.23571* (2026).

[55] Xinjun Yang, Yingqiang Zhang, Hao Chen, et al. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3754–3767.

[56] Xinjun Yang, Yingqiang Zhang, Hao Chen, et al. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 295–308.

[57] Zhenkun Yang, Chen Qian, Xuwang Teng, Fanyu Kong, Fusheng Han, and Quanqing Xu. 2023. LCL: A lock chain length-based distributed algorithm for deadlock detection and resolution. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 151–163.

[58] Zhenkun Yang, Chen Qian, Xuwang Teng, Fanyu Kong, Fusheng Han, Quanqing Xu, and Daokun Hu. 2025. LCL+: a Lock Chain Length-based Distributed Deadlock Detection and Resolution Service Built for OceanBase. *ACM Transactions on Computer Systems* 43, 4 (2025), 1–33.

[59] Zhifeng Yang, Quanqing Xu, Shanyan Gao, et al. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3728–3740.

[60] Zhenkun Yang, Chuanhui Yang, Fusheng Han, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3385–3397.

[61] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org.

[62] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient Distributed Transaction Processing in Heterogeneous Networks. *Proc. VLDB Endow.* 16, 6 (2023), 1372–1385. https://doi.org/10.14778/3583140.3583153

[63] Tao Zhu, Zhuoyue Zhao, Feifei Li, et al. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2, Article 11 (June 2019), 26 pages.