A Tree-Structured Two-Phase Commit Framework for OceanBase: Optimizing Scalability and Consistency

Quanqing Xu*
OceanBase, Ant Group

Chen Qian*
OceanBase, Ant Group

Chuanhui Yang†
OceanBase, Ant Group

Fanyu Kong†
OceanBase, Ant Group

Guixiang Liu
OceanBase, Ant Group

Fusheng Han
OceanBase, Ant Group

Zixiang Zhai
OceanBase, Ant Group

OceanBaseLabs@service.oceanbase.com

ABSTRACT

Distributed transactional consistency over partitions is costly: classical two-phase commit (2PC) incurs high coordination and latency, and dynamic partition transfers complicate recovery. We introduce a tree-structured 2PC framework for OceanBase (OB) using single-machine log streams in three ways. First, we use log streams as *coalesced* 2PC participants: co-located partitions that would each be separate in a partition-centric design share one log-stream participant, so a transaction spanning N such partitions interacts with one participant in commit, cutting coordination by orders of magnitude (e.g., 99% for $N = 100$). Second, a coordinator-rooted DAG recursively builds commit trees; migration contexts as leaves avoid explicit participant-list updates and circular dependencies while ensuring linearizable commits under topology change. Third, `prepare_unknown` and `trans_unknown` states prevent violations when participants lose context; they signal uncertainty during retries, avoiding erroneous aborts from "lying" participants while isolating users from ambiguity. Experiments approach single-machine transaction performance with lower latency and bandwidth consumption, supporting modern distributed databases.

PVLDB Reference Format:
Quanqing Xu, Chen Qian, Chuanhui Yang, Fanyu Kong, Guixiang Liu, Fusheng Han, and Zixiang Zhai. A Tree-Structured Two-Phase Commit Framework for OceanBase: Optimizing Scalability and Consistency. PVLDB, 19(12): 4250 - 4262, 2026.
doi:10.14778/3827998.3828030

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

1 INTRODUCTION

Large-scale distributed databases must ensure transactional consistency across geographically dispersed partitions as workloads grow. Traditional 2PC protocols [2, 11, 20, 24] remain the cornerstone for maintaining ACID properties in distributed environments. Yet scaling to thousands of partitions and millions of concurrent transactions strains the performance and scalability expected of modern cloud-native applications.

Traditional 2PC faces two limitations. First, *scalability bottlenecks* stem from partition-centric designs treating each partition as an independent participant—standard 2PC is coordinator-centric and linear in participants per phase; a 100-partition transaction names 100 participants, so prepare/commit fan-out and per-participant durable logging grow with partition width even when co-located on one machine, inflating coordination overhead—limiting throughput under high concurrency. Second, *dynamic topologies* emerge during partition transfers for elastic load balancing: approaches either require costly post-transfer list re-validation [13] or introduce circular dependencies between partitions and logs [1, 19], undermining consistency when partitions migrate during active transactions.

Recent advances in distributed transaction management have made significant strides, yet fundamental limitations persist. Systems like Spanner [10] and CockroachDB [25] still rely on per-partition coordination, inheriting the scalability bottlenecks of traditional approaches. While log-structured systems demonstrate the potential of log-centric architectures, they do not adopt log streams as coalesced 2PC participants, leaving the scalability and dynamic topology challenges unresolved.

OceanBase co-locates unit leaders on one machine sharing a log stream. Using the stream—not each partition—as the 2PC participant reduces coordination overhead while aligning transfers with the stream layout. We propose a log-stream-based 2PC framework addressing scalability and dynamic topology through three innovations. Our approach coalesces partition-mapped 2PC participants into log-stream participants (without conflating this with transactional atomicity at the data plane), introduces a dynamic tree-structured protocol that adapts to partition migrations, and provides principled handling of context loss scenarios.

The contributions of this paper are as follows:

- **Log Stream as Coalesced Participant**: We register each log stream as one 2PC participant instead of many partition-level participants implied by naive partition-centric mapping, shrinking counts by orders of magnitude. With 100 co-located partitions, one log-stream participant cuts coordination overhead by 99%, yielding single-machine-like transaction performance.

*Quanqing Xu and Chen Qian contributed equally to this work.
†Chuanhui Yang and Fanyu Kong are corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828030

• **Dynamic Tree-Structured 2PC**: Unlike classical hierarchical 2PC [21, 27], where coordinators sit in a largely *static* site hierarchy, we introduce a *dynamic* tree over *log streams* whose participant DAG evolves as partitions migrate. Recursively tracking log-stream dependencies, it constructs trees on-the-fly without explicit participant-list updates, resolving "circular transactions" [19].
• **Terminal-decision recovery with TDT and unknown-state fallback**: Without full context, participants must not guess prior decisions—doing so can wrongly affirm earlier prepares or commits. We pair TDT—retained terminal decisions serve duplicate and delayed requests—with `prepare_unknown` and `trans_unknown` states that forbid fabricating Commit or Abort when neither full context nor a retained terminal decision remains. Participants signal uncertainty; recreated coordinators may return `trans_unknown` to users, preventing forced commits and ambiguity (Section 5).

Together, the log-stream participant abstraction shows 2PC need not use one participant per partition; the dynamic tree-structured protocol handles runtime topology changes; the unknown-state mechanism ensures consistency under context loss. This framework meets two goals: (1) **high performance for static transactions**—single-machine latency (1.2ms) through log-stream aggregation; and (2) **simplicity for dynamic transactions**—partition transfers with minimal complexity. OceanBase scales linearly under heavy transfers, underscoring its importance for cloud databases.

The paper is organized as follows. §2 covers OceanBase, atomicity, and log-stream topologies under transfer. §3 presents the tree-structured 2PC protocol, latency optimizations, and state machines. §4 discusses concurrent transfers during transaction execution. §5 presents unknown-state consistency. §6 covers optimizations and the TLA+ proof; §7 reports experiments. §8 complements this evaluation with production experience, including massive-partition mobile and SaaS workloads and operational lessons on commit-tree depth under transfer. §9 reviews related work, and §10 concludes.

2 BACKGROUND

2.1 OceanBase Database

OceanBase is a high-performance distributed RDBMS engineered for scalability, cross-regional fault tolerance, and internet-scale cost-efficiency [30, 34]. Earlier generations emphasized *shared-nothing*: independent servers own shards and coordinate over the network, yielding horizontal scalability and fault isolation. OceanBase 4.0's Paetica [33] updates this positioning: the system is not purely shared-nothing at every layer but *hybridizes* shared-nothing partitioning with shared-everything mechanisms where storage and logging consolidate (e.g., shared log services), letting operators switch between standalone and distributed modes while keeping node-local execution for most queries and transactions.

Co-located partition leaders within a unit share one machine and one log stream. This supports our log-stream-based 2PC by aggregating partitions into one stream participant, reducing coordination overhead while maintaining transaction consistency.

2.2 Atomicity and Traditional 2PC

Traditional 2PC is a fundamental protocol for atomicity in distributed transactions [20]. Two phases: (1) *prepare phase*, where the coordinator sends prepares and participants vote YES (via a prepare log) or NO; (2) *commit/abort phase*, where the coordinator collects votes and broadcasts commit or abort to all participants per outcome. If all vote YES, the transaction commits; otherwise it aborts. All participants commit or abort together, preserving atomicity.

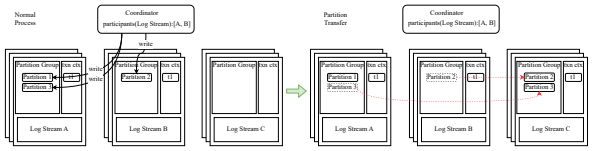


Figure 1: Participant list changes during a transaction.

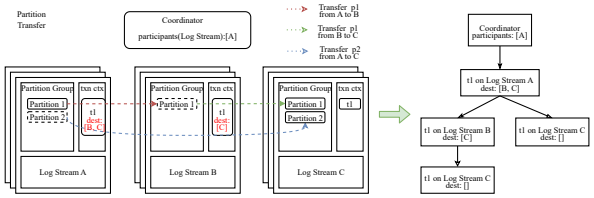


Figure 2: Log stream tree.

With single-machine log streams, partitions remain transactional read/write units and concurrency control stays partition-level. For ACID [28], atomicity requires each concurrency-control unit to pass isolation and persistence checks. If log streams act as 2PC participants, the minimum participant set must include all streams where partitions participating in concurrency control finally reside—the **minimum set requirement**. Any list containing this minimum set is valid. The challenge is obtaining all such streams during 2PC and completing commit.

2.3 Log Stream Topologies

Partition transfer is critical for load balancing and resource optimization but complicates coordination: active migrations require the participant list to cover every host log stream. Naively, coordinators list only streams from reads/writes at transaction start. Transfers before commit can violate the minimum set requirement (Section 2.2). Figure 1 shows streams A and B; a transfer introduces stream C that creates transaction context and must join concurrency control—omitting C yields an incorrect list.

Transaction contexts arise from reads/writes and transfers. R/W streams are immediate; on $L_s \rightarrow L_d$ moves, L_s records L_d, and recursive record tracing builds a *log stream tree* spanning all hosting streams. Example 2.1 illustrates Figure 2; Theorem 2.2 states the tree satisfies the minimum set requirement.

Example 2.1. Figure 2 (left): t1 uses Partition1 and Partition2 on LS A; Partition1 migrates A→B→C; Partition2, A→C. On A, t1 records B and C; on B, C. The tree (right) roots at the coordinator; leaves are final concurrency-control streams.

Theorem 2.2. *The log stream tree meets the minimum participant set requirement.*

Proof outline. Every partition's final log stream is an initial node or recorded transfer destination, so the tree contains the minimum participant set (Section 2.2); see Appendix A.1 [29].

3 TREE-STRUCTURED 2PC

3.1 Tree Structure Design

By Theorem 2.2, the resulting log stream tree satisfies the minimum participant-list requirement. Tree-structured 2PC arranges coordinators and participants in a tree: interior nodes relay prepare/commit messages toward the root, reducing fan-out versus a flat coordinator contacting every leaf [21, 27]. R*'s formulation [21] assumes a *static* site-level hierarchy throughout the protocol; nodes are *log streams* after coalescing co-located partitions; edges can shift during 2PC when partitions transfer (Section 4). Each internal node acts as coordinator to its children and participant to its parent; subtree nodes form its participant list. Leaves are participants only.

In *commit-protocol* roles, each interior log stream parallels an *interposed coordinator*: one resource to its parent, driving prepare/vote among children—the CORBA OTS *interposition* pattern [15, 22], with CORBA-based adoption [9]. Unlike nested transactions, tree-structured 2PC only relays two-phase commit messages; unlike OTS interposition, interior nodes are *log streams* after coalescing partitions; edges may appear mid-commit on migration (Section 4). However, classical hierarchical 2PC leaves room to improve response time under OceanBase's logging and dynamic topology. We combine OceanBase optimizations for traditional 2PC with tree structure for performance and correctness.

3.2 Optimization for Transaction Latency

Traditional 2PC runs prepare/commit phases; the coordinator and participants synchronously log prepares and commit/aborts. The coordinator responds only after its commit log persists (two roundtrips and three log syncs) [36]. OB eliminates separate coordinator logs; the coordinator participates, recording state in its participant log. It replies once all prepare acks arrive, without awaiting commit-log persistence. (a) Prepare logs must carry the participant list for recovery inference; (b) participants must determine status independently to release resources without coordinator commit/abort logs.

OB 2PC added a clear stage: participants synchronously write a clear log after Commit/Abort, before Tombstone; the coordinator replies after prepare OK/NO. This yields two RPCs and one I/O (participants' prepare logs) but adds clear-log overhead. Later the clear stage was dropped (Section 6.1): the coordinator writes the commit/abort log after client reply, keeping one I/O during response time while maintaining correctness.

3.3 State Machine

Following §3.2, we present OceanBase's tree-structured 2PC. The participant list spans log streams where the transaction runs and partition-transfer targets; destinations are recorded in the source context's participant list (Section 4). We abstract nodes without leaf/non-leaf distinctions. Each node maintains: 1) *status*: OK or NO, initially OK. 2) *state*: Running, Prepare, Commit, Abort, Tombstone, initially Running. 3) *participant list*: children for the current state, initially empty. 4) *collected participant list*: participants with collected messages, initially empty.

Figure 3 illustrates the state machine. We outline two phases:

• **Prepare Phase.** In Running, upon a parent Prepare or scheduler Commit request, it asynchronously writes a prepare log, sends

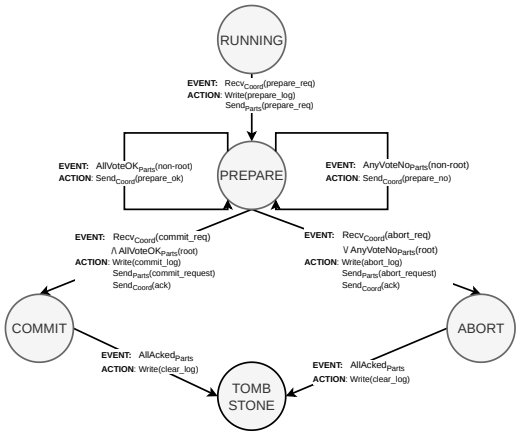


Figure 3: Tree-structured two-phase commit state machine.

Prepare to all children, and resets *collected participant list*. Once the prepare log persists, it enters Prepare. Child prepare OK/NO messages update *collected participant list*; *status* becomes NO if any prepare NO arrives.

• **Commit Phase.** When all children return OK (*collected participant list* equals the child set), it sends prepare OK to its parent if any, or Commit to all children if root. On parent Commit, a Prepare node synchronously writes a commit log, sends Commit to all children, resets *collected participant list*, ACKs its parent, and enters Commit. After all child ACKs, it asynchronously writes a clear log and enters Tombstone. Abort is analogous; Appendix B [29] gives rules consistent with TLA+ model.

3.4 Theoretical Performance Characteristics

We derive the theoretical performance characteristics of the tree-structured 2PC algorithm under normal protocol execution (successful commit). N is the average number of child participants per node and H is the average tree height. The metrics below describe one fixed tree; cluster scaling and partition-transfer behavior are not covered here (Section 7).

• **Transaction Latency.** Time from Commit request to response: $2H$ roundtrips (Prepares from coordinators/sub-coordinators; Prepare responses from participants/sub-participants) and one log sync (participants' prepare logs).

• **Row Lock Release Latency.** From lock acquisition to final release (conflict footprint): $3H$ roundtrips (Prepare and Commit requests from coordinators/sub-coordinators; Prepare responses from participants) and two durable log syncs (participants' prepare and commit logs).

• **Transaction Resource Consumption.** Per transaction: $5NH$ transmissions (Prepare, Commit, and Release from coordinators; Prepare and Commit responses from participants; see Section 6.1.1 for Release) and $2NH$ synchronously replicated prepare and commit logs at participants.

4 TRANSFER PROCESS

4.1 Transfer Principles

In OceanBase 4.0, tablets [8] are the smallest physical units; partitions are logical and may contain multiple tablets. Partitions

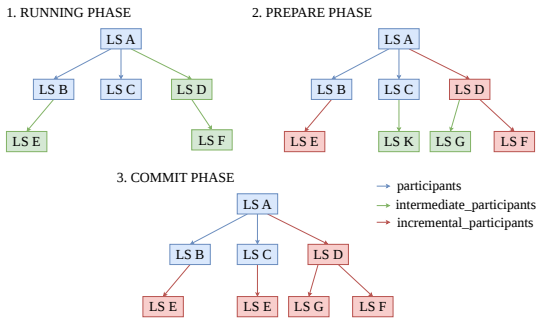


Figure 4: Transfer during tree-structured 2PC.

start symmetrically across machines; partition transfer [2] migrates them among servers as workloads evolve for load balancing. Section 3 describes a tree-structured 2PC on a fixed log-stream tree. Transfers may interrupt 2PC anytime and change participants. For ACID we give one equivalent principle (logs or messages): Prepare/Commit/Abort messages mirror prepare/commit/abort logs.

Theorem 4.1 (Transfer principle). ***(Logs.)*** *Transaction logs (prepare, commit, abort) written before the transfer-out log must be applied to the destination log stream through transfer. Logs written afterward must account for the transfer (e.g., include the destination in the participant list).* ***(Messages.)*** *Prepare/Commit/Abort messages sent before transfer completes must be applied to the destination log stream through transfer. Later messages must account for the transfer's impact. The log and message formulations are equivalent.*

Proof outline. Without pre-transfer state at dst, a partition can join concurrency control unseen by dst, violating atomicity; logs after transfer that ignore dst omit the partition's final host from the participant list. Appendix A.2 [29] formalizes the contradiction.

Transaction State Migration. Transaction logs and 2PC messages migrate during transfer to preserve state across failures. Paxos replicates each log stream: tree-structured 2PC decides Prepare/Commit/Abort; Paxos orders and majority-replicates WAL records only. The source extracts contexts for transferred partitions and writes destination WAL entries with transaction id, source-side state, participant info, and any persisted prepare/commit decision. Transfer completes only after those entries are majority-replicated on the destination WAL and applied to its transaction context. Until then the source stays authoritative; afterward the destination recovers the same state by replaying its WAL if the source fails.

4.2 Dynamic Partitioning and Transfer

To handle partition transfers during 2PC, we first assume atomic partition transfer and 2PC log synchronization to present the core protocol; Section 4.3 relaxes this assumption and further details the concurrency-control mechanisms in the real implementation. To ensure complete participant lists per Theorem 4.1, we keep prior-stage participants for the current stage and current-stage participants for the next. The transaction context maintains three participant variables: 1) *participants*: SQL-layer participants; persisted in both the log stream and transaction context. 2) *incr_parts*: Participants from transfers in the previous stage, merged into this stage's participant list; persisted in both the log stream and transaction context. 3) *interm_parts*: Participants from transfers in this stage for the next stage's list; persisted only in transaction context.

Algorithm 1 illustrates the commit process when transfers overlap Running, Prepare, and Commit (successful commit). Each transfer $trans_i$ has source $trans_i.src$ and destination $trans_i.dst$. migrateData ships transaction data and state from source to destination; commitLogs merges *interm_parts* into *incr_parts* and Paxos-synchronizes 2PC logs containing *participants* and *incr_parts*. Example 4.2 demonstrates this process.

Algorithm 1 Commit with Concurrent Partition Transfer

Require: Transaction T, scheduler s, log stream tree $tree$
`/* T enters the Running state */`
1: s finds initial tablets and corresponding log streams $log_{initial}$.
2: $tree \leftarrow log_{initial}$
`/* Transfer` $trans_1$ `occurs during Running */`
3: $trans_1.src.interm\_parts.add(trans_1.dst)$
4: migrateData($trans_1.src$, $trans_1.dst$)
`/* T enters Prepare state */`
5: commitLogs($tree$)
`/* Transfer` $trans_2$ `occurs during Prepare */`
6: $trans_2.src.interm\_parts.add(trans_2.dst)$
7: migrateData($trans_2.src$, $trans_2.dst$)
`/* T enters Commit state */`
8: commitLogs($tree$)
`/* Transfer` $trans_3$ `occurs during Commit */`
9: migrateData($trans_3.src$, $trans_3.dst$)
10: **procedure** migrateData(src, dst)
11: Migrate data and transaction state of src context to dst.
12: **end procedure**
13: **procedure** commitLogs($tree$)
14: **for** all nodes $p \in tree$ **do**
15: $p.incr\_parts.add(p.interm\_parts)$
16: Paxos($p.participants$, $p.incr\_parts$) ▷ Synchronize logs via Paxos
17: **end for**
18: **end procedure**

Example 4.2. As shown in Figure 4 (top left), Running starts with *participants* A, B, C (blue). Transfers A→D, B→E, and D→F put D in A's *interm_parts*, E in B's, F in D's (green). When the transaction enters Prepare (top right), *interm_parts* become *incr_parts* and join *participants* for the current stage. Two transfers during Prepare add K and G as *interm_parts* (green). At Commit (bottom), all nodes in the log stream tree participate in the commit.

4.3 Transfer and Log Commit Concurrency

Section 4.2 assumes atomicity, but partition transfer and 2PC log commits are not atomic. Merging *interm_parts* into *incr_parts* during Prepare can race with a concurrent transfer that commits the transfer-out log before the prepare log while recording a destination in *interm_parts*, omitting it from the prepare log (Theorem 4.1). To ensure correctness, two requirements apply: (1) Partition transfer must migrate all transaction context preceding the transfer-out log to the destination; (2) Transactions must include all previous transfer destination log streams in *incr_parts*.

The transfer mechanism handles the first requirement. For the second, the following operations must be atomic (a per-transaction lock $lock_t$ shared by Algorithms 2 and 3 for the same transaction avoids races). (1) Partition transfer broadcasts the destination log stream to all transactions and commits the transfer-out log; (2) Participants merge *interm_parts* into *incr_parts* and commit the 2PC log. Algorithms 2 and 3 present the detailed partition transfer and log commit procedures.

Algorithm 2 Partition Transfer Procedure

Require: Transfer service s, transfer-out log *log* (with tablet *tab*, destination log stream *dst*, timestamp *ts*, source log stream *src*), per-transaction lock $lock_t$, transaction contexts set tc

1: **for** transaction $t \in tc$ **do**
2: **if** *tab* is involved in t **then**
3: t acquires $lock_t$
4: Block t from writing new logs
5: Collect t's transaction context into migration set (for destination log)
6: t releases $lock_t$
7: **end if**
8: **end for**
9: s commits log on source log stream: tablet *tab*, destination *dst*, timestamp *ts*
10: s commits log on destination log stream: tablet *tab*, source *src*, transaction information set
11: **for** transaction $t \in tc$ **do**
12: **if** *tab* is involved in t **and** $ts > t.ts'$ **then**
13: t acquires $lock_t$
14: Add *dst* into $t.interm\_parts$
15: Unblock t from writing logs
16: t releases $lock_t$
17: **end if**
18: **end for**
19: Clear the information on s

Algorithm 3 Log Commit Procedure

Require: Transfer service s, per-transaction lock $lock_t$, transaction contexts set tc

1: **for** transaction $t \in tc$ **do**
2: **if** t is not stopped from writing logs **then**
3: t acquires $lock_t$
4: Merge $t.interm\_parts$ into $t.incr\_parts$
5: Commit logs containing $t.participants$ and $t.incr\_parts$ via Paxos
6: t releases $lock_t$
7: **end if**
8: **end for**

Theorem 4.3. *Algorithms 2 and 3 ensure that the two requirements above are satisfied, and thus satisfy Theorem 4.1.*

Proof outline. Algorithm 2 migrates pre–transfer-out context before recording completion and adds destinations to *interm_parts*; Algorithm 3 merges these into durable *incr_parts* at phase boundaries. Interleaving analysis under the per-transaction lock $lock_t$ shows that every transfer completing before a prepare-log commit is reflected in that log. Appendix A.3 gives the full argument.

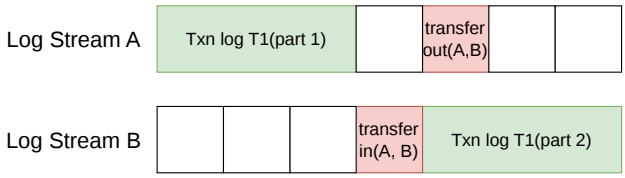


Figure 5: Transfer and log stream synchronization.

4.4 Exception Handling of Transfer

Transfer must preserve participant lists under failures. If migration succeeds but the source log stream fails, lists must remain correct. Transfer runs as a distributed transaction: exceptions roll back all transfer operations; at commit, transaction and partition state update atomically so either the full outcome is visible or none of it is. Because transfer synchronizes in-flight transaction data to the destination log stream (Figure 5), we apply the following.

- *Migration-based transfer.* Transactions migrate via transaction logs or persistent transaction tables; during persistence we update the participant list, as in master switching.
- *Atomic activation and Tree-2PC integration.* During transfer, the source log stream adds the destination to *interm_parts* while the destination creates a temporary transaction context. Committing the transfer transaction activates both updates atomically; if it aborts, both roll back. This keeps participant list consistency across log streams without explicit coordination. They integrate with Tree-2PC by updating *interm_parts* and *incr_parts* (Section 4.2) when transfer logs are applied.

4.5 Transaction Dependence on Partitions

Tree-structured 2PC relies on transfer to maintain correct participant lists (log stream trees containing the minimum set). The transfer process must update participant lists for transactions containing transferred partitions on the source side, requiring transactions to record included partitions. To minimize partition dependence, we set an upper limit on recorded partitions. When this limit is exceeded, transfers add destinations to the transaction's participant list. Since any participant list containing the minimum set is correct (Theorem 2.2), extra participants do not affect correctness.

4.6 Circular Transactions

Circular topologies are rare under typical workloads and low churn. Long transactions overlapping concurrent transfers can recreate cycles: migration may revisit a log stream along another transfer path, so one participant may receive Prepare from multiple parents. When a participant receives Prepare messages from multiple parents, it does not immediately invoke handle_2pc_prepare_request. It waits until its local prepare log has synchronized per the state machine, but not for downstream logs, then replies prepare OK upstream—downstream prepares remain coordinated in the parent's log stream, mitigating circular conflicts during linear commits. The modified state machine is handle_2pc_prepare_request(p):

- **Enable:** Running with a 2PC_PREPARE from parent p or a TXN_COMMIT from the scheduler.
- **Action:** On the first parent message, asynchronously write a prepare log, send 2PC_PREPARE to all participants (participant list and node status in logs/messages), and enter Prepare; otherwise

await prepare-log sync, send PREPARE to parent p (participant list and node state), and enter PREPARE.

Preventing cycles requires blocking or rejecting transfers that may interact with in-flight transactions, coupling load balancing to transaction duration—too restrictive for elastic storage. Pruning repeated log streams on first traversal is unsafe: transfer-induced dependencies can remain invisible to the root until every affected log stream is visited, so premature pruning may drop state needed for a consistent terminal decision. Tree compaction follows D2PC-style phase-boundary flattening in Section 6.1.2: once all known downstream participants have been exposed to the root, the root retains them as a flattened participant set for later phases, while recursion applies only to participants introduced by later transfers.

Liveness. The modified rule preserves liveness and avoids deadlock despite cycles. Each node waits for *its own* prepare log to synchronize before PREPARE_OK upstream and never waits on downstream or sibling nodes, so no node waits on another that (directly or via the tree) waits on it. Under fairness, every recipient completes local sync and sends PREPARE_OK upward; leaf-to-root progress lets the root reach COMMIT or ABORT in finite steps. The protocol terminates with either all commits or all aborts, preserving tree-structured 2PC liveness (cf. §6.2.2).

4.7 Achieving Design Goals

This section fulfills the two design goals from Section 1: high performance for transactions without transfers, and simplicity for transactions with transfers.

High performance. When logs omit embedded participant lists, 2PC proceeds without partition awareness. A height-1 log-stream tree matches optimized 2PC latency and resources at the same participant count, improved by lightweight log-stream participants that coalesce many partition-level participants. With one log stream and no participant list, execution collapses to one-phase commit and single-machine performance.

Simplicity. Tree-structured 2PC is a classic variant needing minimal code changes: intermediate nodes coordinate downstream and participate upstream. Participant-list immutability within a phase keeps recovery straightforward during commit and focuses critical logic on transfer and transaction log ordering and processing, preserving simplicity and robustness.

5 "UNKNOWN" TRANSACTION STATES

We cover lying participants (from Ant Group Alipay production), unknown-state recovery, and TDT off the common path via compact terminal decisions. For ambiguous retries, TDT returns retained terminal decisions; `prepare_unknown` and `trans_unknown` block inventing COMMIT or ABORT when neither full context nor a retained decision exists.

5.1 The Lying Participants Problem

In the tree-structured 2PC protocol of Section 3, participants (log streams) may "lie" to parent coordinators when transaction contexts are lost. This problem arises in any hierarchical 2PC structure, including our tree-structured protocol. Example 5.1 illustrates.

Example 5.1. As in Figure 6 (top), consider a tree-structured 2PC with root coordinator C and participant log streams P1 and P2. After

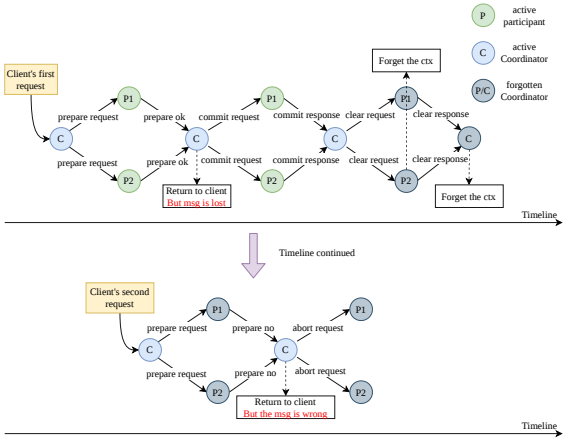


Figure 6: Lying participants in tree-structured 2PC.

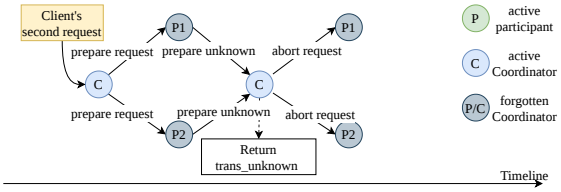


Figure 7: "Unknown" states for lying participants.

C receives a COMMIT request, the protocol correctly transitions participants (P1, P2) to the commit state and reclaims transaction logs (transaction context records log indices only). The final COMMIT response to the user is lost. The user retries by resending COMMIT to restore C's context; tree-structured 2PC restarts (Figure 6, bottom). Since participant contexts are reclaimed, they respond only with PREPARE_NO, forcing rollback and an ABORT response to the user. Participants lie in their second PREPARE responses, leaving the user in an incorrect state and violating Consistency (we treat the coordinator and user as special participants).

5.2 `prepare_unknown` and `trans_unknown`

We add `prepare_unknown` to the tree-structured 2PC. Without recoverable PREPARE context, a participant (log stream) replies `prepare_unknown` to its parent; the coordinator ABORTS all subtree participants, ending the reconstructed attempt, without waiting on irrecoverable state. This ABORT applies only to this attempt; terminal decisions from the original attempt remain unchanged.

For user responses, the logic is more nuanced. The preceding rule introduces `prepare_unknown` as an internal state-machine outcome: it lets the reconstructed coordinator safely terminate an attempt when a participant has lost recoverable context. At the user boundary, OceanBase distinguishes two cases. If the root coordinator is newly created and encounters `prepare_unknown`, the system returns ABORT; if the root coordinator is recreated, e.g., after a lost client response, and detects `prepare_unknown`, it returns `trans_unknown`. Figures 6 and 7 illustrate retries where participants cannot determine whether the original attempt committed.

Reclaiming context drops log-index metadata; after log GC or cross-log-stream transfer, a stale index may miss the final-decision record. Persisting only a context identifier with an old log index cannot guarantee durable answers; full-context retention would keep participant, transfer, and concurrency-control metadata beyond their normal lifetime. OceanBase separates terminal-decision lookup from full-context retention. `trans_unknown` is conservative, user-visible, not an atomicity violation: the retry terminates without fabricating Commit or Abort for the original attempt.

5.3 Reducing User-Visible Unknown States

Although `prepare_unknown` and `trans_unknown` are necessary for correctness, they must stay rare: they arise only when neither full context nor a terminal decision is recoverable. In Alipay, post-commit reclaim with a lost reply rebuilt coordinator state while participants had forgotten the transaction, yielding incorrect rollback (missing data), long investigation, and retained fault-injection regression. `trans_unknown` forces applications or users to confirm outcomes or accept error/timeout semantics instead of definitive Commit/Abort. We introduce TDT with context machinery below to keep unknown states off the common path.

Transaction Context Table (TCT) vs. Transaction Data Table (TDT). Following MaLT [14], the *Transaction Context Table* (TCT) stores per–log-stream *in-flight* context—2PC phase, coordinator/participant roles, participant-set fields (*participants*, *incr_parts*, *interm_parts*), and metadata (Section 4.1). Transfers read/update TCT with logs so source and destination agree as tablets move. With TCT at both endpoints, destination log-stream metadata wins (e.g., destination Commit overrides source Prepare); partition-level fields merge by max across contexts for monotonicity. *Transaction Data Table* (TDT) records only *terminal* committed/aborted outcomes for a bounded interval so participants that discarded TCT can answer duplicate or retried Prepare without "lying." TDT stays off the hot path for commit/transfer coordination and complements TCT after cleanup. Duplicate/retried Prepare without full context triggers a TDT lookup returning the terminal decision, avoiding `prepare_unknown` and user-visible `trans_unknown`.

TDT retention is configurable—30 minutes by default—covering retries and delayed messages. Within this window, production shows no `trans_unknown`. Beyond it, unresolved responses require strict app confirmation. If neither full context nor the retained TDT decision remains, unknown states stay the safety net: the system must not guess Commit or Abort; `trans_unknown` is the only safe user-visible response, preventing fabrication.

6 DISCUSSION

This section covers key aspects of the framework: optimizations (row lock release and clear log minimization), and TLA+ verification of safety and liveness.

6.1 Optimization

6.1.1 Release Messages In our tree-structured 2PC, transaction latency stays flat and resource consumption drops, but resource-release latency grows by one log operation. Row locks and logs are critical [5, 16, 23]: locks capture contention footprint; logs govern reclamation. Reclamation latency is inherently tolerable and batchable; lock contention is not—higher footprints degrade throughput.

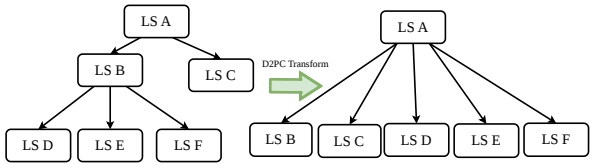


Figure 8: D2PC optimization.

To reduce row-lock release latency, the coordinator's synchronous commit log lets participants release log resources by querying transaction status when peers fail, guaranteeing no residual dependencies. Row locks need not wait on that write: once any participant confirms commit, locks can release.

We decouple lock release from reclamation with Release and Commit steps. After the commit point (all Prepare acknowledgments), the coordinator issues Release (locks) and synchronizes the commit log; Commit messages that trigger log cleanup follow successful commit-log sync. Log reclamation keeps two log synchronizations; lock release replaces one synchronization with an extra roundtrip. The following summarizes optimized tree-structured 2PC for successful commits with N participants:

- **Transaction latency**: Two roundtrips (coordinator Prepare, participants' Prepare responses) and one log sync (participants' prepare logs).
- **Row lock release latency**: Three roundtrips (coordinator Prepare and Release; participants' Prepare responses) and one log sync (participants' prepare logs).
- **Log reclamation latency**: Three roundtrips (coordinator Prepare and Commit; participants' Prepare responses) and two durable log syncs (commit log at coordinator; prepare and commit logs at participants).
- **Transaction resource consumption**: $5N$ messages (coordinator Prepare, Release, Commit; participants' Prepare and Commit responses), $2N + 1$ synchronously replicated, durable majority-acknowledged logs (coordinator's commit log and participants' prepare and commit logs), and one asynchronously replicated majority-acknowledged log (coordinator's clear log).

6.1.2 Optimizing Clear Logs via D2PC We observe that only leaf nodes skip clear logs; other nodes must write them, creating inefficiencies under our optimization framework. Prepare and Commit participant lists may differ after partition transfers, and frequent transfers inflate clear-log overhead during 2PC. We therefore adopt D2PC (Dynamic Decentralized Two-Phase Commit) [13]. As illustrated in Figure 8, only the root log stream (Log Stream A, i.e., LS A) needs to write clear logs; dynamically adjusting the participant list between phases minimizes redundant logging while preserving atomicity and consistency guarantees.

6.2 TLA+ Proof

This section proves safety and liveness of the proposed tree-structured 2PC protocol. Specified in TLA+ (module `TreeTwoPhaseCommit`; Appendix C [29]). Unlike flat TM/RM, each node is uniformly modeled as coordinator to children and participant to parent. *rmState* holds local state; *parent* and *children* encode topology. Transfer yields dynamic membership by merging an intermediate child set into the active child set only at phase boundaries. The next-state

relation defines prepare, decision, propagation, acknowledgement, internal abort, and transfer.

6.2.1 Proof of Safety For safety, all nodes must agree on the outcome, and commit or abort decisions must never change.

Theorem 6.1. *From the TLA+ specification, the safety property of the tree-structured 2PC follows directly from traditional 2PC safety.*

Proof outline. Each subtree agrees before its root votes upward, collapsing to an equivalent flat 2PC execution; Appendix A.4 [29] makes the simulation step precise.

6.2.2 Proof of Liveness For liveness, once the root initiates the protocol, every node eventually reaches a terminal commit or abort state within finitely many transitions.

Theorem 6.2. *From the TLA+ specification, the liveness of the proposed tree-structured 2PC protocol follows from traditional 2PC liveness under standard fairness assumptions.*

Proof outline. Fairness lifts decisions bottom-up along the tree like reliable flat 2PC; Appendix A.5 [29] gives the correspondence.

6.3 Trade-offs and Limitations

Using log streams as coalesced 2PC participants reduces commit fan-out when partitions co-locate. Choices shift from partition to log-stream granularity, trading commit-path scalability for placement, failure handling, and skew management.

Failure-domain granularity. OceanBase runs Paxos leader election and membership changes at log-stream granularity: co-located partitions share one leader and replication path. Machine or leader failure widens blast radius from one partition to the stream's group. Recovery and membership must preserve stream-wide state, complicating disaster recovery versus per-partition replication.

Load-balancing flexibility. Coalescing reduces load-balancing flexibility. Partition-granularity designs rebalance by moving a partition and changing replication-group membership. Hot partitions rebalance through transfers (Section 4), preserving in-flight state, updating the log-stream tree seen by 2PC—adding protocol surface and latency jitter. Section 7 shows bounded overhead on workloads; rebalancing stays more complex than placement-only moves.

Workload skew. Shared WALs amplify skew: hot co-located partitions on one stream contend for append and callback capacity. OceanBase mitigates by migrating hotspots off overloaded streams and tuning append throughput. PALF [17] reports up to ~1.48M appends/s per PALF group at 512 B log size (≈750 MB/s logical writes), using pipelined replication, adaptive group commit, and a lock-free write path. Mitigations temper skew but not the trade-off of many partitions sharing one WAL.

7 EXPERIMENTAL EVALUATION

This section evaluates our tree-structured 2PC framework in OceanBase, focusing on partition-transfer behavior.

7.1 Experimental Setup

All experiments use OceanBase 4.4.1 (community) in SN mode with TPC-C and sysbench; unless stated otherwise, we follow the configurations, workloads, and metrics below.

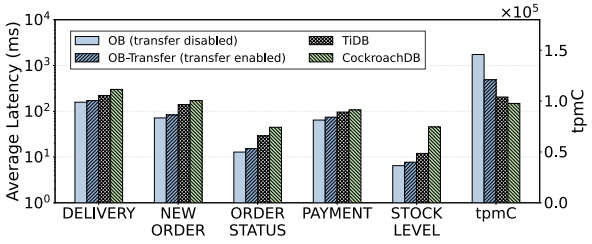


Figure 9: TPC-C latency–throughput under partition transfer.

7.1.1 System Configuration We deploy up to eight OBServer nodes and one proxy on Alibaba Cloud (Ubuntu 24.04); each OBServer has identical hardware (Intel Xeon, 32 cores, 128 GB RAM, 2 TB SSD). Log streams use two-replica Paxos on different nodes. We compare transfer-enabled and disabled settings under the same configuration unless stated otherwise.

7.1.2 Workloads TPC-C evaluates transfer effects on high-concurrency OLTP; its multi-table distributed transactions match our latency, throughput, and stability. We inject transfers during steady state to emulate load balancing. Runs use 3,000 warehouses and 500 concurrent clients; each lasts one hour (warm-up then steady state), with transfers every three minutes unless noted.

7.1.3 Metrics We measure throughput (*tpmC*) and client-side latency (mean and P99), emphasizing P99 for coordination overhead and transfer-induced stalls. We track fluctuations and P99 spikes during transfers to assess resilience and predictability under log-stream topology changes.

7.2 Impact of Transfer on Online Transactions

We evaluate partition-transfer effects on OLTP latency, throughput, and stability under high concurrency.

7.2.1 Experimental Setup and Baselines We use TPC-C with **OB (Transfer Disabled)** as the peak baseline and **OB-Transfer (Transfer Enabled)** with three-minute steady-state transfers to emulate load balancing. We compare against TiDB [18] (v8.1) and CockroachDB (v25.4.0). All systems use the same workload parameters: 3,000 warehouses, 3,000 active warehouses, and 500 concurrent connections. Transaction optimizations remain enabled on TiDB and CockroachDB. TiDB uses Async Commit and One-phase Commit when eligible; CockroachDB uses transaction pipelining and Parallel Commits. They mainly benefit small, single-Region/Range transactions. The three systems run under their default isolation levels. We configure TiDB and CockroachDB with two-replica Raft (overriding their three-replica defaults) so all systems share an identical fault-domain. All other system parameters (region/range size, PD/placement) are left at each system's deployment defaults.

7.2.2 Overall Performance under Transfer Figure 9 gives comparative TPC-C latency–throughput results under partition transfer for static and dynamic runs.

- **Performance and System Resilience.**

The right panel shows OB peaking. OB-Transfer lowers throughput moderately but keeps it stable: tree-structured 2PC lets participants evolve during execution without aborts or global barriers, bounding transfer-induced coordination. TiDB and CockroachDB

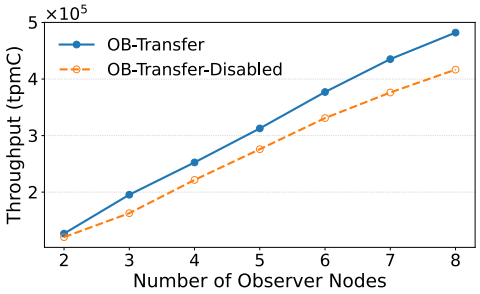


Figure 10: TPC-C throughput scalability vs. cluster size.

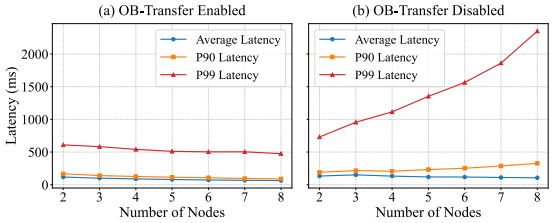


Figure 11: Transaction latency distribution across cluster sizes.

stay below OceanBase because their commit paths still pay distributed transaction and replication overheads on cross-partition transactions in this workload.

- **Transaction Latency Characteristics and Root Causes.**

The left panel reports client-side latency by transaction type (log scale). OceanBase beats TiDB and CockroachDB on most categories: Single Log Stream (SLS) consolidation cuts participants and RPC/cross-node coordination. OB and OB-Transfer nearly overlap; even for *New-Order* and *Payment*, transfer adds little delay because tree-structured 2PC confines topology impact to a small fraction of the commit path. CockroachDB matches TiDB on *tpmC* but shows higher per-transaction latency on several classes. The framework optimizes steady-state coordination and absorbs transfer-induced complexity for predictable elastic OLTP.

7.2.3 Observed Commit-Tree Geometry under TPC-C To quantify transfer-aware-path usage, we instrument Section 7.1 and record commit-tree geometry for every committed TPC-C transaction in a one-hour steady-state run (transfer every three minutes). Table 1 reports the distribution over 1,560,076 transactions. Tree height concentrates at H=2 (> 99.999%); only 17 transactions reach H=3, and none exceed it. Max width is dominated by 1–4 participant log streams (98.2% cumulative), tailing to 10. Two factors drive this skew: (i) TPC-C transactions finish in tens to hundreds of milliseconds, leaving little overlap between concurrent transfers and in-flight transactions; (ii) transfers extend a transaction's tree only when the transferred tablet was already touched, and small footprints keep overlap rare. This distribution grounds the design goal in Section 4.7: most commits follow the static path (cf. Figure 9: near-overlap of OB-Transfer and transfer-disabled latency bars), while the tail (H=3; widths up to 10) exercises the dynamic path without visible degradation.

We repeated the measurement with a 30-second transfer interval—an order of magnitude above production—and saw the same height distribution because TPC-C transactions are too short to commonly

Table 1: Commit-tree geometry for 1,560,076 TPC-C commits (1-hour steady state; transfer every 3 min).

Tree height H	Count	%
2	1,560,059	99.999
3	17	0.001

Max width W	Count	%
1	257,361	16.50
2	664,362	42.59
3	461,360	29.57
4	148,229	9.50
5	23,492	1.51
6	3,938	0.25
7–10	1,334	0.09

intersect a transfer. To stress the dynamic path, we lengthened TPC-C transactions with per-statement sleep to emulate long-running analytic queries. There, H=3 rose to 0.029% while remaining bounded below H=4. Deeper commit trees emerge only when reconfiguration frequency and transaction duration are both elevated; the static path dominates under the evaluated TPC-C workload.

7.2.4 Sensitivity to Transfer Rate Transfer rate affects the tree-structured 2PC in three aspects. First, it may affect the height and width of the commit tree. However, §7.2.3 shows that even under a higher transfer rate, i.e., with a 30-second transfer interval, the distribution of tree heights remains unchanged. This is because TPC-C transactions are sufficiently short that they rarely overlap with the transfer window; increasing the transfer rate does not push more transactions onto deeper commit paths. Second, higher transfer rate amplifies the per-transfer overhead. §7.4 shows that when the sysbench concurrency varies by 16×, both the transfer execution time and the CPU cost per transfer remain bounded; a higher transfer rate therefore only amplifies a bounded constant cost. Third, transfer may overlap with active transactions. The production experience in §8.2 reports latency spikes when a single transfer overlaps with a large number of active transactions. However, since TPC-C transactions typically complete in tens to low hundreds of milliseconds, and with pre-transfer checks and transfer concurrency control enabled, the overall impact remains limited. A dedicated rate sweep with production-representative dynamic patterns would strengthen this further; we leave it to future work.

7.3 Scalability under Partition Transfer

We evaluate partition-transfer scalability under shared-nothing cluster expansion, verifying high throughput and stable latency as OBServer count grows.

7.3.1 Experimental Methodology We vary OBServer counts while holding workload and configuration per Section 7.1, with transfer enabled and TPC-C (3,000 warehouses, 500 concurrent clients). Scaling from 2–8 nodes, we measure tpmC and client-side latency at each size to isolate expansion effects on transfer and transactions.

7.3.2 Throughput Scalability under Cluster Expansion Figure 10 shows TPC-C throughput versus OBServer count. Throughput grows *near-linearly* with and without transfers; enabled runs sit lower from migration overhead, but the gap stays fixed as scale

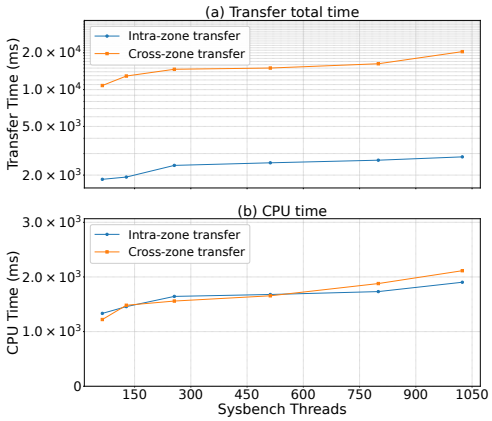


Figure 12: Partition transfer execution time and CPU overhead under varying sysbench thread counts.

increases, so overhead stays bounded without compounding. In this cluster-expansion experiment, rebalancing also redistributes data and load across newly added OBServer nodes, partially offsetting transfer overhead. Log streams plus tree-structured 2PC: co-located partitions share one participant, and transfers extend only localized commit-tree branches, so adding nodes raises capacity without proportionally increasing per-transaction coordination.

7.3.3 Latency Stability across Different Cluster Sizes Figure 11 shows Average/P90/P99 latency across cluster sizes. Latency holds steady as nodes increase; P99 does not worsen, so transfer coordination stays bounded. Logging and messages occur only on the transaction's commit path and do not grow with cluster size. The framework separates scalability from dynamic partition moves and keeps performance predictable in elastic SN deployments.

7.4 Transfer Efficiency and Overhead

We evaluate partition-transfer efficiency and overhead under high-concurrency workloads, examining whether transfers remain lightweight and predictable with concurrent transactions, and do not become performance or resource bottlenecks.

7.4.1 Transfer Efficiency Experimental Setup We conduct the experiment on an OceanBase cluster with partition transfer enabled. The workload uses sysbench with distributed read-write transactions, varying client threads from 64 to 1024 to emulate increasing transactional pressure. Each transaction includes point queries and delete–insert operations within a transactional context.

Partition transfer uses table granularity on a 30-table cluster; each run randomly selects a table and migrates it from its current log stream to another random log stream, limiting bias and reflecting realistic scheduling. We evaluate two scenarios: *intra-zone transfer*, where source and destination log streams share a zone, and *cross-zone transfer*, where they span zones.

7.4.2 Transfer Execution Time Figure 12 (a) reports total partition-transfer execution time across sysbench thread counts. As transactional pressure rises, execution time grows slowly and remains stable for all configurations; under high concurrency, transfer latency shows no abrupt spikes or instability.

Cross-zone transfers incur higher latency than intra-zone transfers due to added cross-zone communication and coordination. The gap stays consistent as transactional load increases. Nearly parallel trends show cross-zone overhead is bounded and does not scale with concurrent transaction intensity; partition-transfer execution remains largely decoupled from foreground transaction pressure and avoids interfering with critical-path transaction processing.

7.4.3 CPU Overhead of Partition Transfer Figure 12 (b) shows accumulated CPU time per transfer across sysbench thread counts. CPU time aggregates execution costs of transfer-related operations: transaction filtering, transfer log generation, metadata reconstruction, transaction migration, and the physical transfer phase.

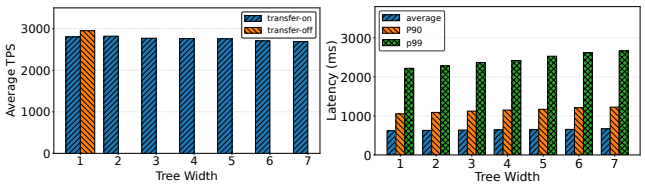
CPU consumption per transfer increases moderately with transactional pressure but remains within a narrow, predictable range. As with latency, cross-zone transfers use more CPU than intra-zone transfers; the gap stays stable across loads. Bounded overhead keeps partition transfer from becoming a CPU hotspot under high-concurrency workloads, confirming the mechanism stays lightweight alongside intensive distributed transactions.

7.5 Tree-structured Commit Performance

This subsection evaluates commit-tree geometry under partition transfer. Tree-structured 2PC organizes log streams; transfers embed streams along branches. Per Section 3.4, critical-path scales with H: response latency needs $2H$ RTTs and one log sync; lock release, $3H$ RTTs and two log syncs; fan-out widens branches without extra rounds. Per Section 7.1, we report throughput and client latency, emphasizing P99 for coordination stalls and transfer instability.

7.5.1 Methodology: Constructing Width/Depth-controlled Commit Trees We construct commit trees with controlled *width* and *depth* by manipulating transfer patterns while holding workload and resources unchanged. To increase **tree depth**, we repeatedly transfer one partition across log stream groups, extending the commit path (increases H). To increase **tree width**, we transfer multiple partitions to different log stream groups at the same level, enlarging fan-out while keeping depth shallow. We measure throughput (TPS) and transaction latency (mean, P90, P99) under `transfer-on` and `transfer-off` baselines.

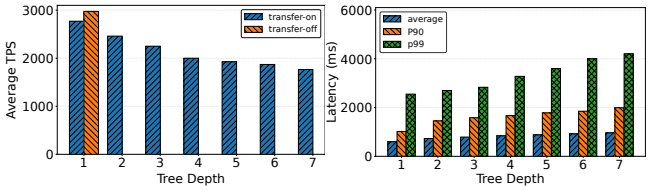
7.5.2 Impact of Tree Width Figure 13a shows throughput versus width: under `transfer-on`, TPS stays stable with jitter, so fan-out barely perturbs the critical commit path, matching Section 3.4 (parallel branches; critical-path rounds unchanged). Figure 13b: mean and P90 rise while P99 climbs—wider fan-out raises straggler risk (queuing, log flush, scheduling). P99 rises gradually, so width overhead stays local, avoiding system-wide instability.



(a) Throughput vs. tree width. **(b) Latency vs. tree width.**

Figure 13: Tree width sensitivity under transfer.

7.5.3 Impact of Tree Depth We deepen the tree to stress coordination. Figure 14a shows throughput decreasing as depth adds hops and synchronous logging on the critical path, lowering completion rate. Degradation is smooth without abrupt collapse, keeping the protocol stable under deep transfer chains.



(a) Throughput vs. tree depth. **(b) Latency vs. tree depth.**

Figure 14: Tree depth sensitivity under transfer.

Figure 14b shows latency rising with depth (P99 > mean/P90), matching Section 3.4 ($2H$ round-trips for response, $3H$ for lock release). Delays (RPC, queuing, log sync) accumulate on the critical path, amplifying tails; growth tracks depth; transfer overhead stays localized. Results validate tree-structured 2PC under transfer: *depth* dominates coordination cost and tail latency; *width* scarcely affects throughput or average latency but moderately lifts P99. Commit-tree sensitivity yields bounded, localized overhead and stable performance under partition movement.

7.6 Coordination Scalability

This subsection contrasts OceanBase 3.x/4.x coordination scalability. Section 7.5 studies tree-structured 2PC under transfer; we isolate granularity's commit effect. 3.x uses *partition granularity*: N independent partitions; 4.x uses *log-stream granularity*: one participant per log stream, lowering cardinality and redefining boundaries.

7.6.1 Motivation and Setup Partition granularity grows participants linearly, inflating fan-out, aggregation, contention, and log-sync pressure; log-stream granularity merges partitions into fewer streams, shrinking coordination overhead—explaining 4.x gains. We measure commit latency under 3.x and 4.x with matched workload and hardware without transfer, isolating 2PC coordination for transactions spanning hundreds to tens of thousands of partitions.

7.6.2 Results and Analysis Figure 15 shows commit latency versus partition count. On 3.x, latency grows nearly linearly; at large N, overhead drives aggregation delay, log-sync pressure, and contention until 3.x fails to complete commits within reasonable bounds. On 4.x, latency grows slower: log-stream grouping decouples participants from partitions (one per stream), preventing coordination-round explosion. 4.x shrinks the coordination domain rather than tuning RPC/logging—moving 2PC from partition-linear to log-stream-bounded scaling—so it stays stable where 3.x fails. This shift prevents coordination explosion, supports transfer-aware transactions, and explains stable tree-structured 2PC under dynamic transfer (Section 7.5); it scales commit latency better than partition-level 2PC in 3.x via log-stream coordination, validating our design.

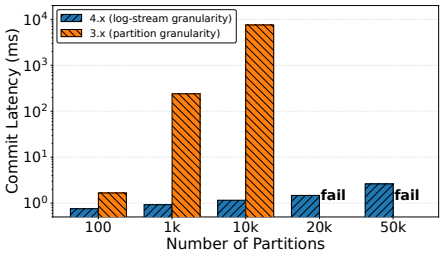


Figure 15: 2PC commit latency comparison.

8 PRODUCTION EXPERIENCE

This section complements Section 7 with OceanBase field observations: massive-partition mobile/SaaS workloads where partition-centric coordination limits scale; tree-structured 2PC under transfer at log-stream granularity.

8.1 Corroborating the Coordination Gap

The aforementioned challenges also appear in customers' core mobile production workloads and OceanBase public-cloud SaaS workloads, which can require hundreds of thousands to millions of partitions within a single database. This scale is driven by mobile core systems with thousands to tens of thousands of tables and secondary indexes, commonly partitioned by city and time, together with three- to five-year data retention.

Massive partitioning can make a single transaction involve excessive participants under partition-granularity 2PC. Production tcpdump analysis revealed transactions with more than 4,000 participants. In one business workflow, the system observed more than 13 concurrent transactions, each involving over 11.2K partition-level participants. Because each participant persists the complete participant list for recovery, participant metadata and network traffic are strongly amplified; in some cases, participant lists alone consumed more than 50% of network bandwidth. Internal stress tests confirmed this bottleneck: 2PC commit-request latency rose from 1.66 ms at 100 participants to 240 ms at 1,000 participants and 7.54 s at 10,000 participants; at 20,000 participants, it exceeded 1,000 s.

Operationally, million-partition workloads stress Paxos heartbeats, 2PC participants, and background partition-maintenance services such as load balancing and transaction checkpoints. Since OceanBase 4.0, released in 2022, OceanBase has transparently migrated from a partition-based architecture to a log-stream-based architecture while preserving application-facing partition semantics. Massive-partition transactions execute on the log-stream-based transaction path, decoupling transaction execution and high availability from individual partitions and eliminating amplification from per-partition Paxos instances and per-partition 2PC participants. Commit latency remains stable at around 60 ms in the evaluated massive-partition stress setup, showing that the architectural change addresses the production bottleneck motivating this design.

8.2 Transfer Overlap and Admission Control

Tree-structured 2PC trades participant fan-out for depth along transfer chains. As shown in Figure 14, deeper commit trees can increase tail latency, especially P99 latency. In practice, tree height is bounded by the number of machines in the transfer chain and is rarely the dominant production risk. Tail jitter is more often driven

by active transactions overlapping transfer, because transfer briefly pauses affected transactions while migrating their state.

Deep trees are uncommon in production because log-stream transfer is a background administrative operation controlled by scheduling and concurrency limits. Still, transfer can introduce jitter under high business pressure. In Ant Group's internal deployment, transfer once overlapped with more than 5,000 active transactions on a single machine, causing hundreds-of-milliseconds stalls. This shows that worst-case latency depends not only on tree depth, but also on the number of in-flight transactions overlapping transfer.

OceanBase therefore performs pre-transfer checks: if too many transactions still need migration, transfer waits until the number becomes sufficiently small. Together with transfer concurrency control and scheduling policies that avoid repeatedly transferring the same log stream within a short window, this keeps commit trees shallow and reduces transfer-induced jitter. Thus, Figure 14 is a worst-case sensitivity study rather than the common production case. Deeper trees may increase tail latency, but they do not affect correctness because transaction decisions and recovery metadata still propagate through the tree.

9 RELATED WORK

Traditional 2PC [2, 11, 20, 24] underpins ACID; per-partition coordination hurts scalability/latency; recent work targets optimization. Tree-structured 2PC [21, 27] cuts fan-out via intermediaries; classical work assumes static hierarchies at *sites*, not elastic resharding with mid-commit topology change. We review coordination, log-centric transaction processing, and transfer-aware partitioning.

9.1 Global Transaction Coordination

Spanner [10] combines hybrid logical clocks and TrueTime APIs for strongly consistent global transactions but stays tied to per-partition coordination and 2PC bottlenecks. Calvin [26] uses deterministic execution and log scheduling to cut coordination; CockroachDB [25] applies hybrid logical clocks to geo-distributed 2PC.

Spanner and fine-grained Paxos groups. Spanner [10] shards into Paxos-replicated splits, preserving placement, migration, fault isolation, and load balancing—e.g., hot ranges or moving replicas. Transactions spanning many ranges or groups pay coordination and metadata costs. OBTree2PC takes a different tack: co-located partitions coalesce into one log-stream-level 2PC participant, cutting commit participants while accepting log-stream-level failure-domain, load-balancing, and skew trade-offs.

Paxos-group designs preserve placement flexibility and fault isolation but do not exploit log-stream aggregation to reduce 2PC participants or address transfer-induced overhead. Our framework treats log streams as coalesced 2PC participants, dramatically reducing coordination when partitions share one log stream.

9.2 Log-Centric Transaction Processing

Hekaton [12] is a single-node log-centric OLTP engine: main-memory storage, latch-free structures, optimistic MVCC, and logging optimize execution and durability in one SQL Server instance. Single-node execution lets Hekaton sidestep distributed 2PC rather than shrinking participant counts; it therefore does not tackle failure-domain granularity, load-balancing flexibility, or workload-skew isolation in large-scale distributed databases, where data must be partitioned, replicated, migrated, and balanced across nodes.

Hyder [6] and Tango [4], with CORFU [3] as a shared-log substrate, explore another log-centric point. Durable global ordering means shared-log appends determine atomicity, isolation, or conflict resolution; scaling focuses on that path, and once saturated, adding partitions alone need not raise aggregate write throughput. OBTree2PC takes a different approach: it avoids both single-node execution and a global total-order shared log. WAL replication stays local per log stream, and each log stream acts as the distributed-commit participant. Co-located partitions coalesce into one 2PC participant, preserving fine-grained placement and load-balancing flexibility while cutting commit fan-out.

9.3 Dynamic Partitioning and Transfer

Existing dynamic-partitioning solutions—e.g., CockroachDB [25]—often need post-transfer participant-list re-validation or hit circular dependencies between partitions and logs [1, 19]. They need explicit participant-list updates and complex recovery when partitions migrate mid-transaction. Our tree-structured 2PC recursively tracks log-stream dependencies, avoiding explicit list updates and simplifying recovery. It also addresses "circular transactions" [19], preserving linearizable commits during live migrations via automatic tree construction. Under transfer/commit lock contention, OceanBase uses distributed deadlock detection/resolution (LCL [31], LCL+ [32]) to prevent indefinite blocking.

Unlike prior work [35], we take the *log stream* as the coalesced 2PC participant/coordination endpoint rather than per-partition 2PC participants, supporting transfers and context loss via recursive log-stream trees and unknown-state mechanisms without post-transfer participant-list re-validation or circular-dependency handling. Prior systems either (1) treat partitions as separate 2PC participants, inflating counts and coordination cost, or (2) struggle with dynamic topologies during transfers. Our work bridges these gaps by coalescing participants at log-stream granularity, reducing participant counts, and introducing an unknown-state mechanism to preserve consistency during context loss. These innovations align with the principles of modern log-structured systems [7] while addressing challenges in dynamic, cloud-native environments.

10 CONCLUSIONS

We present the tree-structured 2PC for OceanBase, improving scalability and latency of distributed coordination. By using single-machine log streams as coalesced 2PC participants in place of partition-level participants, we reduce participant counts by orders of magnitude (e.g., 99% for 100 partitions), cutting coordination overhead and bandwidth consumption. Transfers use a coordinator-rooted tree: contexts recursively embed as leaves, avoiding explicit participant list updates and circular dependencies while ensuring linearizable progress. Unknown-state mechanisms (`prepare_unknown`, `trans_unknown`) prevent consistency violations during context loss and isolate users from ambiguity. OceanBase 4.4.1 results approach single-machine transactions with lower latency and bandwidth, linear scalability, and stability during transfers. Section 8 corroborates results on mobile and SaaS workloads at millions of partitions. It bridges strong consistency with high performance for elastic cloud deployments with dynamic partitions.

REFERENCES

[1] Ghazi Alkhatib and Ronny S Labban. 2002. Transaction management in distributed database systems: the case of oracle's two-phase commit. *Journal of Information Systems Education* 13, 2, 95–104.

[2] Muhammad Atif. 2009. Analysis and verification of two-phase commit & three-phase commit protocols. In *2009 International Conference on Emerging Technologies*. IEEE, 326–331.

[3] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 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.

[5] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2325–2338.

[6] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder—A Transactional Record Manager for Shared Flash. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. 9–20.

[7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin 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*. 275–290.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.

[9] Edward E. Cobb. 1997. The Impact of Object Technology on Commercial Transaction Processing. *The VLDB Journal* 6, 3 (1997), 173–190.

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

[11] Bipin C Desai and Boutros S Boutros. 1996. Performance of a two-phase commit protocol. *Information and Software Technology* 38, 9, 581–599.

[12] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254.

[13] Paul Ezhilchelvan, Amjad Aldweesh, and Aad van Moorsel. 2018. Non-blocking two phase commit using blockchain. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. 36–41.

[14] 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.

[15] Ennio Grasso. 1997. Implementing Interposition in CORBA Object Transaction Service. In *Proceedings of the 1st International Conference on Enterprise Distributed Object Computing (EDOC '97)*. IEEE Computer Society, Los Alamitos, CA, USA, 184–189.

[16] James N Gray. 1978. Notes on data base operating systems. *Operating systems: An advanced course*, 393–481.

[17] Fusheng Han, Hao Liu, Bin Chen, Debin Jia, Jianfeng Zhou, Xuwang Teng, Chuanhui Yang, Huafeng Xi, Wei Tian, Shuning Tao, et al. 2024. Palf: Replicated write-ahead logging for distributed databases. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3745–3758.

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

[19] Heine Kolltveit and Svein-Olaf Hvasshovd. 2007. The circular two-phase commit protocol. In *International Conference on Database Systems for Advanced Applications*. Springer, 249–261.

[20] Butler Lampson and David Lomet. 1993. A new presumed commit optimization for two phase commit. In *19th International Conference on Very Large Data Bases (VLDB'93)*. 630–640.

[21] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[22] Object Management Group. 2005. *Additional Structuring Mechanisms for the Object Transaction Service, Version 1.1*. OMG Available Specification formal/05-01-01. Object Management Group. Retrieved June 29, 2026 from https://www.omg.org/spec/OTS/1.1/PDF

[23] In Kyung Ryu and Alexander Thomasian. 1990. Analysis of database performance with dynamic locking. *Journal of the ACM (JACM)* 37, 3, 491–523.

[24] George Samaras, Kathryn Britton, Andrew Citron, and C Mohan. 1995. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases* 3, 325–360.

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

[26] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[27] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining ACID and BASE in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 495–509.

[29] Quanqing Xu, Chen Qian, Chuanhui Yang, Fanyu Kong, Guixiang Liu, Fusheng Han, and Zixiang Zhai. 2026. A Tree-Structured Two-Phase Commit Framework for OceanBase: Optimizing Scalability and Consistency. *arXiv preprint arXiv:submit/7313362* (2026).

[30] 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 *IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 4183–4196.

[31] 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.

[32] 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.

[33] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-nothing/Shared-everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow.* 16, 12 (2023), 3728–3740.

[34] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.

[35] 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. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1372–1385.

[36] Hongyao Zhao, Wei Lu, Zhanhao Zhao, Yinhao Hong, Quanqing Xu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2025. An Efficient Two-Round Distributed Transaction Processing Approach over Heterogeneous Networks. *The VLDB Journal* 34, 4 (2025), 51.