PALF: Replicated Write-Ahead Logging for Distributed Databases

Fusheng Han
OceanBase

Hao Liu
OceanBase

Bin Chen
OceanBase

Debin Jia
OceanBase

Jianfeng Zhou
OceanBase

Xuwang Teng
OceanBase

Chuanhui Yang
OceanBase

Huafeng Xi
OceanBase

Wei Tian
OceanBase

Shuning Tao
OceanBase

Sen Wang
OceanBase

Quanqing Xu*
OceanBase, Ant Group

Zhenkun Yang
OceanBase

OceanBaseLabs@service.oceanbase.com

ABSTRACT

Distributed databases have been widely researched and developed in recent years due to their scalability, availability, and consistency guarantees. The write-ahead logging (WAL) system is one of the most vital components in a database. It is still a non-trivial problem to design a replicated logging system as the foundation of a distributed database with the power of ACID transactions. This paper proposes PALF, a Paxos-backed Append-only Log File System, to address these challenges. The basic idea behind PALF is to co-design the logging system with the entire database for supporting database-specific functions and to abstract the functions as PALF primitives to power other distributed systems. Many database functions, including transaction processing, database restore, and physical standby databases, have been built based on PALF primitives. Evaluation shows that PALF greatly outperforms well-known implementations of consensus protocols and is fully competent for distributed database workloads. PALF has been deployed as a component of the OceanBase 4.0 database and has been made open-source along with it.

PVLDB Reference Format:
Fusheng Han, Hao Liu, Bin Chen, Debin Jia, Jianfeng Zhou, Xuwang Teng, Chuanhui Yang, Huafeng Xi, Wei Tian, Shuning Tao, Sen Wang, Quanqing Xu, and Zhenkun Yang. PALF: Replicated Write-Ahead Logging for Distributed Databases. PVLDB, 17(12): 3745 - 3758, 2024.
doi:10.14778/3685800.3685803

PVLDB Artifact Availability:
https://github.com/oceanbase/oceanbase/tree/develop/src/logservice/palf.

1 INTRODUCTION

The write-ahead logging (WAL) system was originally introduced to recover databases to their previous state after a failure. Beyond this initial purpose, more requirements have been gradually emerging from distributed databases. The logging system should be capable of replicating logs to multiple replicas for durability and failure tolerance. Several important database features rely on the design of the logging system, such as transaction processing [9, 50], redo log archiving [35], database backup/restore [37], and physical standby databases [36].

To keep consistent states of multiple database replicas, consensus protocols have been widely used to replicate logs in distributed databases [9, 42, 48]. Most of these databases were abstracted into replicated state machines (RSM) for integrating with consensus protocols. In the typical RSM model, the client first handles all intended operations and generates logs, these logs are then replicated to all replicas by consensus protocols. After operation logs have been persisted by majority of replicas, each replica applies them to its state machine.

The RSM model has been working well for operations that modify small datasets (e.g., setting a key-value to the Key-Value store). However, it may be unsuitable for operations that involves a large amount of data, an example is transactions in distributed databases. First, databases are usually required to equip additional buffer to cache temporary data from clients for log generation, therefore, it is difficult for databases to handle large transactions with data volume greater than its cache. A compromised approach is to limit the size of transactions and break up large transactions into small operating units[8], but at the cost of losing the atomicity of users' original transactions. Second, reads in a transaction possibly cannot see previous writes in the transaction, because the writes may have not been applied to the database[9]. Reading from the cache is a possible approach; but this will introduce overhead of merging data from the storage engine and the cache, resulting in a decrease in read performance.

To address above problems, our design choice is to integrate consensus protocols into the write-ahead logging model. In the WAL model, a database writes logs using local file system interfaces, the order of logging and applying operations can be reversed compared to RSM model. Writes are applied to the storage engine of database (in-memory state machine) directly, and then redo logs (operations) are generated and flushed. Therefore, the upper limit of transaction size is expanded and read requests just need to access the storage engine. However, designing such a replicated logging system which provides guarantees like local file system to support the WAL model still faces the following challenges:

Leader Election. In practical deployments, the database leader is usually co-located with the logging system leader to reduce latency [9, 46, 48]. The requirements of the database should be considered when electing the leader of the logging system, for example, a replica located in the same region/IDC as the application system

*Quanqing Xu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685803

should be elected as the leader preferentially. However, whether a replica could be elected as the leader traditionally depends on the consensus protocol itself. A leadership transfer extension has been proposed in Raft [33], but it relies on an external coordinator to actively transfer leadership to the designated replica, which harms the availability of the distributed databases.

Uncertain Replication Results. In WAL model, whether a transaction should be committed or aborted depends on whether its commit record has been persisted. Data may become inconsistent if the logging system returns an incorrect result to the transaction model due to exceptions (e.g., leadership transfer). Local file system indeed returns explicit write results. However, most consensus protocol implementations do not return explicit replication results when exceptions occur [2, 15]. For example, a previous leader had been transformed to a follower due to temporary network error. If the previous leader had not received acknowledgements for some in-flight logs before its retirement, it is not able to perceive whether the logs have been committed by majority. Therefore, transaction processing may get stuck because the transaction engine can not determine whether its commit record has been persisted.

Data Change Synchronization. The log is the database, physical log synchronization is one of the most common approaches to export data changes from the database to downstream systems. For example, physical standby databases (e.g., Oracle Data Guard[36]) provide identical copies of the primary database by transporting and applying redo logs to standby databases. Unlike copying log files directly, log replication in distributed databases poses challenges in synchronizing logs from one replication group in the primary database to a downstream group in a standby database, moreover, these groups should be independently available. Some replication protocols [2, 15] embed cluster-specific information (e.g., membership) into logs, which breaks the continuity of data changes and makes the downstream replication groups unable to reconfigure the cluster independently.

Performance. For many log replication systems, throughput of a single replication group is limited. As a result, they resort to multiple groups to improve overall throughput by parallel writing [13, 15, 31]. However, numerous replication groups may incur additional overheads. A data partition in a database is usually bound with a replication group [9, 42, 46]; more replication groups imply smaller data partitions. This will result in more distributed transactions and degrade performance of the entire database [41].

This paper presents PALF, a Paxos-backed Append-only Log File System. PALF has been co-designed with the OceanBase database to support its WAL model. It provides typical append-only logging interfaces, as a result, the database can interact with PALF much as it interacts with local files. PALF further abstracted database-specific features into primitives, such a clear boundary between the log and the database brings benefits in maintainability for a practical database system, and makes PALF become a building block to construct higher-level distributed systems. These design choices led us to address above challenges by balancing the particularity of databases and the generality of logging systems.

First, PALF decouples leader election from the consensus protocol to support database-related election priorities. For instance, a database replica that closer to upper applications could be elected as the leader by configuring its election priorities. As a result of independent election, a log reconfirmation stage is introduced to PALF for correctness.

Second, PALF returns explicit replication results to the log writer (database) unless its leader crashes, which makes PALF act like a local file. The log writer (database) will be notified of whether logs have been committed by PALF, even if the previous leader has lost its leadership. To achieve this, a novel role transition stage *pending follower* has therefore been introduced into the consensus protocol to determine the status of pending logs; the role of the previous leader will not be switched to follower until it receives logs from the new leader. After that, the state of transactions can be advanced. For example, the previous leader will roll back a transaction if its commit record has not been persisted by the new leader.

Moreover, to synchronize data changes between distributed databases, a downstream Paxos group has been abstracted as a mirror of the primary Paxos group. It only accepts logs from the primary group and can be reconfigured independently. This feature has been used to synchronize redo logs from the primary database to standby databases in OceanBase. To the best of the authors' knowledge, this is the first Paxos implementation that supports synchronizing proposals from one Paxos group to another group.

Finally, to reduce the overhead incurred by distributed transactions, we limit the number of log replication groups to the number of servers in a cluster. Fewer replication groups require higher throughput for a single group because it handles logs from multiple partitions. We maximize write performance with systematic optimizations such as pipeline replication, adaptive group replication, and lock-free write path.

To summarize, the contributions of this paper are:

- PALF is proposed as the replicated write-ahead logging system of OceanBase. Its high availability, excellent performance, and file-like interfaces are suitable for distributed databases (§3).
- We abstract database-specific demands as PALF primitives, such as explicit replication results and change sequence number, which benefits OceanBase database greatly (§4).
- A novel method has been proposed to synchronize logs from a Paxos group to others, which powers functions such as physical standby databases (§5).
- We describe designs for building a high-performance consensus protocol in §6, discuss PALF's design considerations in §7. Evaluations under both closed-loop clients and database workloads show excellent performance (§8).

2 BACKGROUND

This section briefly describes the architecture of the OceanBase database to provide context for how PALF is designed.

2.1 OceanBase Database

OceanBase [46] is a distributed relational database system built on a shared-nothing architecture. The main design goals of OceanBase include compatibility with classical RDBMS, scalability, and fault tolerance. OceanBase supports ACID transactions, redo log archiving, backup and restore, physical standby databases, and many other functions. For efficient data writing, a storage engine based on log-structured merge tree (LSM-tree)[38] has been built from

the ground up and co-designed with the transaction engine. The transaction engine ensures ACID properties by using a combination of pessimistic record-level locks[45] and multi-version concurrency control; it is also highly optimized for the shared-nothing architecture. For example, the commit latency of distributed transactions has been reduced to almost only one round of interaction by an improved two-phase commit procedure [46]. OceanBase relies on a Paxos-based write-ahead logging system to tolerate failures. This brings the benefits of distributed systems, but incurs log replication overhead at the same time.

2.2 Redesigned Architecture

In the previous version of OceanBase [46] (1.0-3.0), the basic unit of transaction processing, logging, and data storage was the table *partition*. As an increasing number of applications have adopted OceanBase, we found that the previous architecture is not as well-suited to medium and small enterprises as to large-scale clusters of large companies. One of the problems is the overhead of log replication. OceanBase enables users to create tens of thousands of *partitions* in each server. This number of Paxos groups consume significant resources for no real purpose, therefore raising the bar for deployments and operations. Another challenge is the huge transaction problem. One such transaction probably spans tens of thousands of *partitions*, which means that there are tens of thousands of participants in the two-phase commit protocol, which will destabilize the system and sacrifice performance.

To address these challenges, the internal architecture of version 4.0 of the OceanBase database was redesigned [47]. A new component, *Stream*, has been proposed, which consists of several data partitions, a replicated write-ahead logging system, and a transaction engine. The key insight of the *Stream* is that tables in a database are still partitioned, but the basic unit of transaction and logging is a set of partitions in a *Stream*, rather than a single partition. A table partition simply represents a piece of data stored in the storage engine. The transaction engine generates redo logs for recording modifications of multiple partitions within a *Stream* and stores logs in the WAL of the *Stream*. Multiple replicas of a *Stream* are created on different servers. Only one of them will be elected as the leader and serve data writing requests. The number of replication groups in a cluster can be reduced to the number of servers to eliminate the overhead incurred by massive replication groups.

With the abstraction of *Stream*, version 4.0 of the OceanBase database achieves comparable performance to centralized databases in stand-alone mode (for medium and small enterprises) and can be easily scaled out to distributed clusters by adding machines (for large enterprises). In default settings, each server has one *Stream* whose leader is elected just at the server, therefore, leaders of different *Stream* in multiple servers can process queries simultaneously as an active-active architecture. Besides strong consistency service provided by the leader, other replicas of OceanBase database can serve read requests with eventual consistency guarantee (§4.2).

3 DESIGN OF PALF

The design purpose of PALF is to provide a replicated write-ahead logging system, which should be capable of serving the OceanBase database and be general enough for building other distributed systems. This purpose of PALF drove its design: a hierarchical architecture for balancing particularity of database and generality of the logging system. Database-specific requirements have been abstracted as PALF primitives and integrated in different layers.

This section first describes PALF as the replicated WAL system of the OceanBase database, and then introduces the interfaces provided by PALF. Finally, the implementation of the consensus protocol is described in detail.

3.1 Replicated WAL Model

In OceanBase database, the replicated logging system is abstracted as an append-only log file, transaction engine interacts with PALF much as it interacts with local files. As depicted in Fig. 1, transactions modify data in the in-memory storage engine directly (step 2-3). Therefore, the upper limit of a transaction's size is greatly expanded and is bounded only by the capacity of the storage engine. Log records are then generated and appended to PALF (step 4). The transaction engine of the leader treats PALF as a local log file system, and it is only concerned with whether log records have been flushed. The responsibility of PALF is to replicate modifications performed in the leader to followers (step 5). If a log has been committed by PALF, the leader will inform the transaction engine of the results (step 6), and followers will replay modifications that the leader has performed to itself (step 7-8).

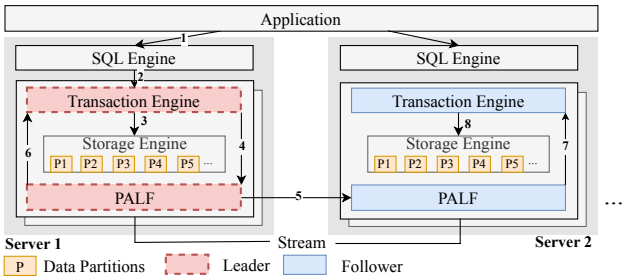


Figure 1: Replicated write-ahead logging model

3.2 PALF Architecture

As depicted in Fig. 2, PALF is a replicated logging system consisting of multiple replication groups called PALF groups. In each PALF group, multiple PALF replicas are placed on different servers for fault tolerance. The transaction engine can append logs to a PALF group and read logs from it, just like a normal append-only file. PALF consists of three main layers: the interface layer, the PALF replicas layer, and the PALF runtime environment. The lower two layers take charge of log replication, reconfiguration, and log storage; the upper one provides user interfaces and coordinates the states of PALF and the transaction engine.

For each PALF group, records generated by the transaction engine are first appended to the leader. The log sequencer will assign a monotonically increasing log sequence number (LSN) to each record, which uniquely identifies a log entry within the PALF group. Records will be encapsulated as log entries and replicated to and persisted by other PALF replicas (followers) in the order of LSN through the Paxos protocol. A log entry is committed only

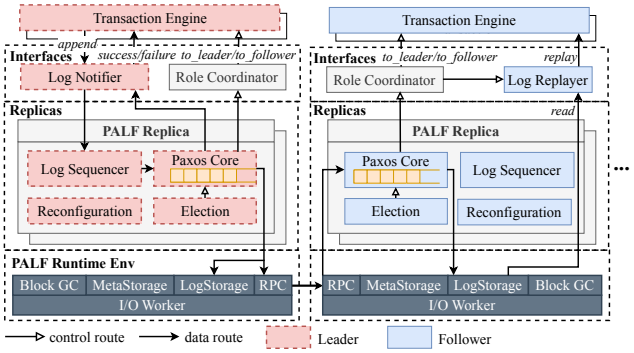


Figure 2: PALF architecture.

```
int append(Record r, CSN refcsn, AppendCb *cb, LSN &lsn, CSN &csn);
int mirror(LSN lsn, LogEntry l);
int read(LSN lsn, Record &r);
int locate(CSN csn, LSN &lsn);

int monitor_tail(TailCb *cb);
int monitor_role(RoleCb *cb);
int trim(LSN lsn);

class AppendCb
{ virtual int success() = 0;
  virtual int failure() = 0; }
class TailCb
{ virtual int tail(LSN lsn) = 0; }
class RoleCb
{ virtual int to_leader() = 0;
  virtual int to_follower() = 0; }
```

Figure 3: PALF interfaces.

when a majority of PALF replicas have persisted it. Unlike some Paxos variants that bind the leader election and log replication together[6, 24, 32, 34], the leader candidate in PALF is elected by an independent election module. The reconfiguration module manages membership of the PALF group (§5.3).

On each server, a PALF runtime environment called *PALFEnv* is activated to provide remote procedure call (RPC) interfaces and manage disk resources for PALF replicas. Specifically, all log entries in a PALF replica are stored as several constant-size blocks in a unique directory in LogStorage. MetaStorage stores meta-information such as membership of all PALF replicas. BlockGC is responsible for trimming log blocks when they are no longer required. All I/O requests issued by PALF replicas are processed by a uniform I/O worker pool in *PALFEnv*.

We generalize the interaction between PALF and the transaction engine into the interface layer. This isolates the impact of database features on PALF and improves PALF's generality. The log notifier in the leader informs the transaction engine of whether logs are committed. The log replayer in followers replays mutations recorded in log entries to the transaction engine. If the role of a PALF replica has been switched (i.e., leader to follower or follower to leader), it will throw a role-changing signal to the role coordinator, the role coordinator forwards signals to transform the role of the transaction engine.

3.3 System Interfaces

Figure 3 shows a set of data-related APIs, omitting the interfaces for system management such as bootstrapping and reconfiguration. PALF offers two methods for writing logs: *append* and *mirror*. The *append* method submits a record *r* to the leader of a PALF group, it returns an LSN to identify the log entry. LSNs of log entries are monotonically increasing, it represents the physical offset of the log entry on the log block. The change sequence number (CSN) is another log entry identifier and will be discussed in §4.2. To achieve high throughput, the *append* method is asynchronous. When it returns, the transaction engine is simply guaranteed that the log entry has been assigned a unique LSN and submitted to the buffer of the leader. The transaction engine will be informed whether the log entry has been committed through the callback function *AppendCb*. Specifically, the *success* method will be invoked when the log entry is committed, meaning that it has been persisted by a majority of replicas and must not be lost; otherwise, the *failure* method will be called. PALF guarantees that the callback function of a log entry will be called at most once.

The *mirror* method offers another approach for writing logs to PALF, it is designed for mirrors of the PALF group and only accepts log entries committed by another PALF group. Only one of these two methods can write logs to a given PALF group at the same time (See §5.2).

The *read* method enables random access to log entries by a given LSN. A *locate* method is provided to map the change sequence number to a log sequence number. To facilitate real-time log consumption in all replicas (e.g., log replayer), the *monitor_tail* method is provided to register a callback function *TailCb* to monitor the tail of the PALF group. When new logs are committed, PALF replicas invoke the *tail* method to notify log consumers of current tail of logs. The *RoleCb* function is used to coordinate the role of PALF replicas and the transaction engine. When the role of a PALF replica switches from leader to follower (follower to leader), the *to_follower* (*to_leader*) method will be invoked. Finally, the *trim* method is designed to indicate useless log entries before the given LSN. BlockGC is responsible for reclaiming these logs.

3.4 Implementation of Consensus

The Paxos protocol and its variants [6, 24, 26, 27, 32, 34] are widely recognized for resolving consensus in distributed systems [5, 9, 14, 23, 42, 48]. Raft[33] is a typical implementation of Paxos, which offers good understandability and builds a solid foundation for practical systems. PALF implements Paxos with a strong leader approach, it keeps the log replication of Raft for simplicity. Different from Raft, PALF decouples leader election from the consensus protocol to manipulate the location of the database leader without sacrificing availability. More differences are summarized in §7.

Demands for Leader Election. In distributed databases, the location of the leader affects almost all functions, such as failure recovery, maintenance operations, and application preference. For example, in cross-region deployment, users tend to make the upper application and the database leader in the same region to reduce latency. Raft has provided a leadership transfer extension to manipulate the location of the leader[33]. However, the leadership transfer extension only works when both the previous leader and the desired leader are alive. If the previous leader crashed, whether a replica can be elected as the leader is completely restricted by the logs that it stores, rather than users' desires. If an undesired replica

is elected as the new leader, this relies on an external coordinator to detect failure and execute leadership transfer operation. This approach may incur availability risks to databases if the coordinator crashes.

To address the problem, PALF decouples leader election from the consensus protocol. Users own the flexibility to specify the priorities of replicas elected as the leader. If the previous leader crashed, replicas with second-highest priority will be preferentially elected as the new leader without any external operations. If the previous leader recovers from failure and its priority is still higher than current leader's, leadership can be automatically transferred back to the recovered replica.

Role Transition. At any given time, each replica is in one of four roles: *leader*, *follower*, *candidate*, or *pending follower*. Figure 4 shows these roles and the transitions among them. The role of a replica is initiated to be *follower* when it starts up. Each replica periodically polls the election algorithm about whether the candidate is itself. If a follower finds that itself has become the candidate, it switches to *candidate* role and performs log reconfirmation, before taking over as a normal *leader*. We will introduce why the log reconfirmation is needed and its procedure in the following paragraph. If the leader finds that the candidate is not itself anymore or the leader receives messages from a new leader, it will revoke its leadership and switch to *pending follower*. The reason for switching to *pending follower* rather than *follower* is that the transaction engine may have appended some logs to the leader before the leadership transfer. To determine the replication results of these pending logs, the previous leader must enter the *pending follower* role and wait for logs from the new leader (§4.1). After the status of all pending logs is clear, the replica will switch to *follower*.

Leader Election. The candidate is elected by a lease-based election algorithm, which ensures that no two or more replicas will be elected as candidates at the same time. Replicas can be assigned different election priorities by users, the election algorithm guarantees that the replica with the highest priority in a majority of replicas will be elected as a candidate. The election algorithm is essentially a variant of Basic Paxos, it reaches a consensus about which replica owns the highest priority in a majority. To ensure that, each replica tries to broadcast its priority to all replicas before proposing a new round of election; A replica will respond to a request which has the highest priority among all requests the replica has received in a certain duration. Besides that, the election algorithm utilizes monotonic timing in each replica to guarantee that a candidate will be elected within a certain time if any majority of replicas survives. This paper focuses the design of the replicated logging system, therefore, we leave implementation details of the election algorithm for another paper.

Log Reconfirmation. Due to flexible leader election, the candidate may have fewer logs than other replicas. Before taking over as a leader, the candidate should re-confirm the logs appended by the previous leader to guarantee that its logs are not fewer than any replica in a majority. The log reconfirmation (Alg.1) is essentially a complete instance of Basic Paxos [27]. Specifically, the candidate broadcasts the advanced ProposalID(identifier for the leader's term) $pid + 1$ to all replicas with Paxos *Prepare* messages (line 2). Each replica will store the *pid* of the *Prepare* message, and respond to the candidate with its logs, only if the *pid* in the *Prepare* message is

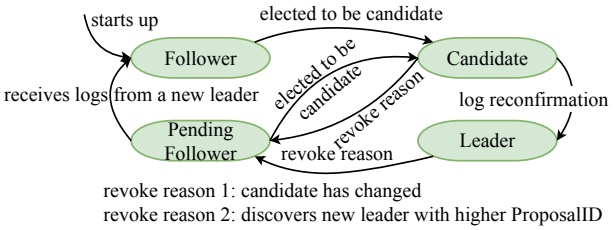


Figure 4: Role transition of PALF replicas.

Algorithm 1 Log Reconfirmation

```
1:  function RECONFIRM
2:      broadcast (Prepare<pid + 1>)
3:      if majority(ACK_prepare) then
4:          (member, LSN_max) = max(ACK_prepare)
5:          getlog (member, LSN_max)
6:          for all log ∈ (LSN_committed, LSN_max) do
7:              broadcast (Accept<pid, log>)
8:          broadcast (SW<pid, membership>)
9:  upon receive Prepare<pid> from r:
10:     if pid > pid_max then
11:         pid_max ← pid
12:         Send(r, ACK_prepare<self, LSN_accepted>)
```

larger than the maximum ProposalID pid_{max} the replica has seen (line 9). To avoid transporting useless logs, the acknowledgement of *Prepare* message only contains the tail LSN of logs.

Once the candidate receives votes from any majority of replicas (line 3), it starts the Paxos *accept* phase: selecting the replica with the longest logs (line 4), getting logs from it (line 5), and replicating these logs to all replicas (line 6). Finally, the candidate replicates a *StartWorking* log to all replicas (line 8). Note that the *StartWorking* log is a special reconfiguration log (§5.3), it is used to roll back the possible uncommitted membership of the previous leader. The candidate will serve as the leader as long as the *StartWorking* log reaches majority.

Log Replication. Once a leader takes over successfully, it takes the responsibility for replicating and committing logs. Log replication in PALF resembles that in Raft [34]. Briefly, log entries are appended to the leader, replicated by the leader, acknowledged by followers, and committed by the leader in the order of LSN. When records are appended to the leader, a log sequence number (LSN) will be assigned to each log entry by the log sequencer. The LSN indicates the physical offset at which the log entry is stored in log blocks. As shown in Fig. 5, log blocks in LogStorage continuously store log entries, the LSN of the next log entry is equal to the sum of the current log's LSN and the log size. The way of identifying log entries with LSN makes clients operate PALF just like a normal file and facilitates redo log consumption in databases.

When a log entry arrives at a follower, the follower will not accept it until all preceding logs have been accepted. If existing logs conflict with new logs with higher proposal number, PALF will truncate the conflicting logs in the same way as Raft. This mechanism establishes the *log matching* property [34].

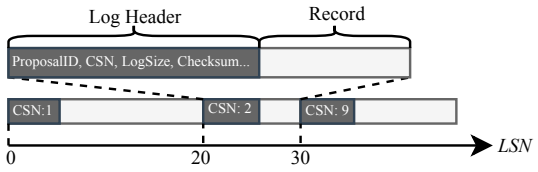


Figure 5: Log structure in PALF.

Correctness. Benefiting from the safety argument in Raft, we simply need to validate whether PALF ensures the following properties: *election safety*, *leader append-only*, *log matching*, *leader completeness*, and *state machine safety*. The *leader append-only* and *log matching* are provided by PALF naturally because it borrows the same log replication scheme from Raft. The key difference between Raft and PALF is the log reconfirmation. The candidate performs an instance of Basic Paxos to learn missing logs from the replica that accepted the most logs in a majority. If a log entry submitted by the previous leader has been accepted by a majority, it must be seen and learned by the candidate, and therefore *election safety* and *leader completeness* are ensured. We leave the proof of the *state machine safety* property in §4.1, because it is related to the way of applying logs to the transaction engine.

4 INTERACTION WITH TRANSACTION ENGINE

This section introduces the features of PALF designed for the transaction engine of OceanBase database based on the implementation of consensus protocol.

4.1 Explicit Replication Results

If the leadership has been transferred away due to a network hiccup, the previous leader may be unclear about whether appended log entries are committed, these logs are referred to as *pending logs*. Pending logs may incur complexities for the transaction engine. For example, the transaction engine generates a commit record of a transaction and appends it to PALF. If the leader loses its leadership unexpectedly, the transaction engine must decide whether to commit or rollback the transaction according to whether the commit record has been persisted.

PALF guarantees that the transaction engine will be explicitly notified of replication results unless the leader crashes or the network is interrupted permanently. The leader is responsible for committing logs and notifying of results. If the leadership has been transferred to another replica, the previous leader switches itself to *pending follower*. When the *pending follower* receives logs from the new leader, the replication results of pending logs become explicit (committed or truncated). Replication results of committed logs will be notified by calling the *success* function, truncated logs will be notified as failed replications by invoking the *failure* function. This is why the previous leader must switch to *pending follower* and wait for logs from the new leader before it becomes a *follower*. For each record, only one of the callback functions will be invoked, and it will be called at most once.

As depicted in Fig. 6a, the previous leader A committed log1, log2 and log3 have been appended to replica A, but have not been replicated. Then A has been partitioned off from B and C temporarily, C was elected as the leader and reconfirmed log2 from B (Fig. 6b). After the network recovers (Fig. 6c), replica A has been transformed to *pending follower* because its election lease has expired. It receives logs from the new leader C, log2 in A will be accepted because it has been committed by the new leader. As a result, replica A informs the database of replication results about log2 by calling *success* function. In contrast, log3 in A will be truncated because another log3 from C contains larger ProposalID. The log3 replicated by the previous leader A fails to reach consensus, therefore replica A invokes *failure* function to notify the database.

Note that if the leader A crashes in Fig. 6a, PALF does not need to notify the database of replication results directly, because all states in memory will be lost. When it starts to recover, the transaction engine switches to *follower*, receives logs from the new leader, and replays local logs to recover transactions. The previous leader can not provide replication results if its network is interrupted, because the new leader can not reach it. If the network recovers, the previous leader can receive logs and determine whether in-flight logs have been committed.

The *state machine safety* property of the consensus protocol is ensured by the explicit replication results. If a log has been committed, it must have been applied to the leader and will be replayed to the follower. If PALF fails to reach consensus on the log, state machine of the previous leader will be rolled back by calling the *failure* function.

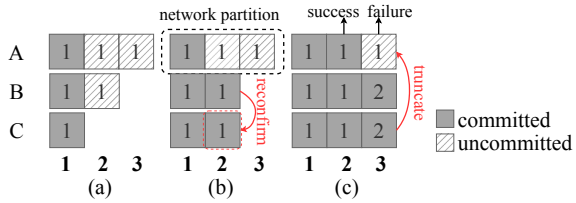


Figure 6: Previous leader A returns explicit replication results to the database. The number in each box is the ProposalID of the leader when the log is appended.

4.2 Change Sequence Number

Data synchronization tools (such as Change Data Capture) usually consume transactions in the order of logs; however, the LSN is incompetent for tracking the order of transactions because it is locally ordered within single PALF group. For scalability, data partitions are usually distributed among multiple *Streams*. If different transactions modify data partitions in different *Streams*, the LSNs of their logs are incomparable. To track the order of transactions with logs across PALF groups, a natural approach is to record commit versions of transactions in the payloads of log entries, as some systems have done [18, 42]. This approach does work, but it has disadvantages. For instance, commit versions may not strictly increase with LSN due to parallel executions of transactions (i.e., assigning smaller LSN to logs of transactions with greater commit

version). This approach requires log consumers to parse payloads of logs, which incurs additional overhead.

PALF provides another log entry identifier, Change Sequence Number (CSN), which maintains consistency with LSN and reflects the order of transactions across multiple PALF groups. CSN is a 64-bit integer stored in the header of each log entry. PALF does not infer the meaning of CSN, meaning that the overhead of maintaining and recognizing CSN is very small. The log sequencer in the leader allocates LSN and CSN to logs when the *append* method is invoked. PALF maintains an invariant on CSN: monotonic increasing within a PALF group. Within each PALF group, for any two logs α and β, if the LSN of α is greater or equal to the LSN of β, then the CSN of α must be greater or equal to the CSN of β. Formally,

$$LSN_\alpha \geq LSN_\beta \rightarrow CSN_\alpha \geq CSN_\beta, \quad \forall \alpha, \beta.$$

The invariant guarantees that CSN increases monotonically within PALF group and thereby keeps consistent with the order of LSN. Figure 5 shows a visualization of the relationship between LSN and CSN of logs. The *CSN* is persisted along with the log entry. As a result, the above invariant is always valid even if the leader crashes.

PALF provides another key guarantee: for the *append* method, its input argument $RefCSN$ and its output argument *CSN* always follow $CSN >= RefCSN$. $RefCSN$ serves as a causal reference; the guarantee indicates that the log entry must be appended after the event happened at $RefCSN$. Therefore, the *CSN* of a log can reflect the system-wide order if the $RefCSN$ is a globally-meaningful value.

The OceanBase database provides globally meaningful commit versions to transactions using CSN. When a transaction is going to be committed, the transaction engine fetches a timestamp from a global timestamp oracle and appends the commit record with the timestamp as $RefCSN$. The *CSN* returned by the *append* method tracks the order indicated by $RefCSN$ and acts as the commit version of the transaction. Note that the *CSN* is not generated by the global timestamp oracle, which may have a value greater than the current global timestamp. As a result, the transaction may be invisible to a future read request that fetches a smaller readable version from the global timestamp oracle. To avoid this, the transaction engine will not respond to the client until the global timestamp is greater than the *CSN*. Through cooperation between the transaction engine and PALF, CSN successfully tracks the order of transactions across PALF groups.

Another database feature that benefits from CSN is follower reads. Follower reads enable follower replicas to serve read-only requests with an eventual consistency guarantee to reduce latency. Read requests with T that can read complete data in followers require that all logs with CSN less than T have been replayed in the follower and commit versions of any future write should be greater than T. The monotonic increasing property of CSN provides this guarantee naturally. Logs are replicated and replayed to followers in the order of LSN, which is consistent with the order of CSN. If the log with CSN T has been replayed in the follower, CSN of the following logs must be greater than T. Compared to other distributed databases that advance readable timestamps in followers by broadcasting special commands periodically (e.g., closed timestamps [43]), the follower read feature of the OceanBase database does not require additional mechanisms.

5 DATA CHANGE SYNCHRONIZATION

Besides serving transactions, distributed databases also act as the source of data flow. Downstream applications can be deployed to provide various services by synchronizing data changes recorded in physical logs. This section introduces two typical physical log synchronization scenarios in OceanBase, describes what challenges they bring to PALF, and depicts how to address these challenges by utilizing features of PALF.

5.1 Overview

When clients write data to databases, records of modifications are appended to the leader of the PALF group and replicated to followers. Besides replicating logs within the database, data changes can be synchronized out of the database for richer functions. There are two typical scenarios in the OceanBase database: physical standby databases and database restore.

As shown in Figure 7, the physical standby database is an independent database in which the data are identical to the primary database. It could serve part of read requests to relieve pressure on the primary database. Compared to traditional primary-backup architecture, it offers higher availability because each database cluster can tolerate failures. One of the most important features of the physical standby database is database-level data protection and disaster recovery, a physical standby database can be switched to be the primary database by a failover operation if the original primary database becomes unavailable, which distinguish it from replica-level protection such as Paxos learners [7]. In production databases, database restore is a core component of the high-reliability feature. If data have been lost due to storage media damage or human errors, archived logs stored in offline storage (such as NFS or Cloud Object Stores) could be used to restore an identical database.

The basic idea behind physical standby databases or database restore is similar: synchronizing data changes recorded in physical logs from the primary database or external storage to the standby database or the restoring database. For the OceanBase database, one of the challenges in implementing these features is synchronizing logs from one PALF group (or external storage) to another PALF group. In addition, these PALF groups should be independently available. A naïve solution is to read log entries from the PALF group in the primary database and append the logs to the corresponding PALF group in the standby database as *records* (Fig. 3). However, the consensus protocol will attach a log header to the appended record for replication, which results in inconsistency between the log format of the primary database and that of the standby databases.

5.2 PALF Group Mirror

We abstracted the requirements of synchronizing data changes across PALF groups as a primitive: the PALF group mirror, which is an independent PALF group that performs the same consensus protocol as described in §3.4. It maintains a mirror of the data change prefixes, which are stored in the primary PALF group or external storage. The PALF group mirror can be reconfigured independently and be switched to a primary group as needed.

One of the most important differences between the primary PALF group and PALF group mirrors is the pattern of writing log records. In primary PALF group, a log record is *appended* to the

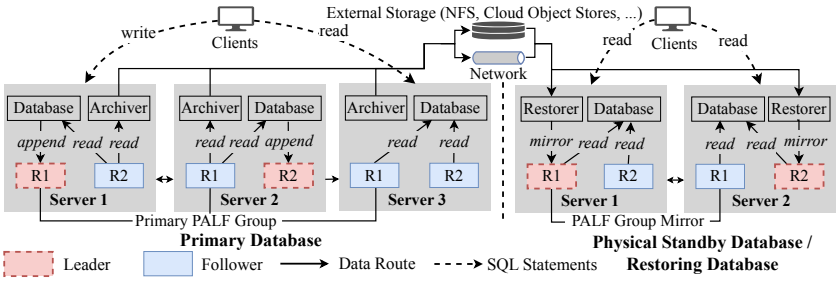


Figure 7: Data change synchronization in OceanBase.

PALF group, attached with a log header, and replicated to replicas by the consensus protocol. As for the PALF group mirror, it only accepts logs committed by a primary PALF group. When a committed log is *mirrored* to the leader, some fields of the log header (e.g., ProposalID) will be replaced with the leader's own values. The leader reuses the LSN and CSN of the original log entries, stores logs to the LSN, and replicates logs to followers.

Two access modes *Primary* and *Mirror* were proposed to differentiate the primary PALF group from its mirrors. The access mode of a PALF group can be switched by failover or switchover operations. The problem is how to broadcast the new access mode to all replicas atomically. Obviously, the atomic broadcast of access mode is equivalent to the consensus problem [12]. Hence, a basic Paxos was implemented to switch the access mode of PALF groups and store each replica's access mode to MetaStorage.

With the PALF group mirror primitive, constructing a data change synchronization architecture for OceanBase database becomes natural. As shown in Figure 7, all PALF groups in the physical standby database are mirrors of PALF groups in the primary database, and all PALF groups in the restoring database are mirrors of data changes stored in external storage. After transaction logs have been committed by the primary PALF group, log archivers read logs from each PALF group and then store them in external storage or transport them to the standby database. In the physical standby database, restorers receive logs from the archivers (fetch logs from external storage) and *mirror* logs to the leader of PALF group mirrors. After logs have been committed by the leader, they will be replayed to the transaction engine in all replicas (including the leader). As a result, transactions that execute in the primary database will be synchronized to the physical standby database; database restore performs a similar procedure.

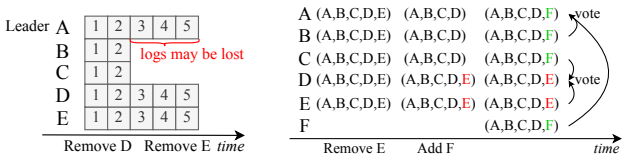
It is worth noting that the interaction between the transaction engine and the PALF group in standby databases is different from that in primary databases. In standby databases, the transaction engines perform the standard RSM model, and all replicas simply read log records from PALF replicas and replay data changes to data partitions. Therefore, the role of a transaction engine is always follower, even though the role of the PALF replica may be leader.

5.3 Independent Reconfiguration

Many implementations of consensus protocols [2, 15] store and replicate reconfiguration commands as normal log entries; however, this embedded approach may be harmful to the usability of PALF group mirrors. First, the transaction engine must filter useless reconfiguration commands because they only concern the data changes that they wrote. Second, the membership of a PALF group mirror is different from that of the primary PALF group, and it should be capable of being reconfigured independently. However, a PALF group mirror can only accept logs from its primary group. As a result, physical standby databases cannot be reconfigured by writing reconfiguration commands as common log records.

The reconfiguration of PALF resembles the single-server approach described in Raft [33], only one replica can be added or removed at a time. The leader replicates a reconfiguration log which records new membership and commits it with the acknowledgements of new membership. Each replica updates its own membership upon receiving a newer reconfiguration log. To make the PALF group mirror can be reconfigured independently, PALF stores the reconfiguration log in MetaStorage, which is separated from LogStorage.

Independent meta-storage is not as simple as it seems, it may harm the safety of the consensus protocol, following examples demonstrate some thorny problems caused by independent meta-storage. As shown in Figure 8a, all logs with LSN less than or equal to 5 has been committed by leader A, where A, D, and E have the latest logs, but B and C are behind. Leader A wants to remove replicas D and E from the group by replicating new membership to replicas that are in new membership; a safety risk that committed logs are lost may occur. Specifically, some committed logs (3, 4, 5) have not been flushed by any majority of new membership (A, B, C). If replica A crashes after D and E have been removed, these logs will be lost. Figure 8b illustrates a possible split-brain[11] situation when reconfiguring clusters successively. After replacing replica E with replica F by removing replica E and adding replica F, replica B and F could vote for A, but replica C and E vote for D. As a result, two leaders, A and D, will be elected.



(a) Loss of committed logs **(b) Split-brain situation**

Figure 8: Anomalies caused by independent reconfiguration.

PALF introduces constraints on reconfiguration and leader election to address these anomalies. For reconfiguration, the leader issues a new configuration along with a log barrier to followers. The barrier is the tail of logs in the leader when it issued the configuration. Each follower refuses to accept a configuration until its flushed logs before the barrier. Therefore, replica B or C will not accept a new configuration until it accepts all logs (3, 4, and 5) before the barrier (Fig. 8a). For leader election, PALF maintains *config version* to indicate the version of membership; reconfiguration operation will increment it. The *config version* acts as the chief election priority, therefore replica D will not be elected as the leader because its *config version* is lower than that of replica C (Fig. 8b).

It is argued here that even though independent meta-storage incurs extra complexities to consensus protocol, it is advantageous because it makes meta-information invisible to log consumers and enables PALF group mirrors to reconfigure clusters independently.

6 PERFORMANCE OPTIMIZATION

As described in §2.2, the transaction engine imposes massive redo logs of multiple data partitions on a single PALF group, which may make PALF become a bottleneck. This section introduces how PALF is designed systematically for maximizing the performance of PALF.

Pipelining Replication. To improve throughput, PALF processes and replicates logs concurrently by exploiting modern multicore processors. The consensus-related states of multiple logs are cached in an in-memory sliding window to avoid CPU cache miss. Therefore, replication stages of multiple logs can be overlapped.

Adaptive Group Replication The consensus protocol incurs additional overhead to database, which contains at least two network messages (log replication and acknowledgement) and some CPU cycles. Batching multiple log entries in one instance is a common way to dilute the overhead incurred by consensus. At the heart of batching logs lies how to determine an appropriate batch size. Batching logs periodically or batching logs immediately when the I/O worker is idle (feedback) are two common approaches. However, the former may incur additional latency under low concurrency. The latter may harm throughput because a massive number of small requests may overwhelm I/O devices under high concurrency.

PALF replicates logs with adaptive group size to balance latency and throughput. The leader caches appended logs within a group buffer. The *freeze* operation will pack cached logs into a group log entry and then replicates it to followers. The number of log entries within a group log entry (group factor) depends on how often the freeze operation is performed. The key idea of adaptive group replication is to *freeze* logs periodically under low concurrency and *freeze* logs according to I/O worker feedback under high concurrency, but the concurrency of clients is difficult to measure directly. Suppose that n clients are appending records r to PALF concurrently. If logs of all clients are appended, but have not been committed, these cached logs should be frozen to a group log gl immediately, without waiting for a constant interval. Therefore, the degree of concurrency (n) correlates with the group factor (gf, the number of cached logs), which is easy to measure by counting logs. The relation can be formalized as:

$$gf = \frac{size(gl)}{size(r)} \propto \frac{n * size(r)}{size(r)} = n \propto CPUCores.$$

Experimentally, the degree of concurrency is proportional to the number of CPU cores occupied by PALF, and hence the group factor threshold could be determined by hardware resources rather than manually tuning a batch size. If the group factor is smaller than the threshold, this means that concurrency is low, and PALF will *freeze* logs if the I/O worker is idle, otherwise, PALF will freeze logs at a constant interval (1 ms by default). Compared to existing batching algorithms [20], adaptive group replication is simple and predictable enough, and is suitable for production deployment.

Lock-Free Write Path Instead of improving throughput, a high level of concurrency may degrade performance if severe contention occurs. A lock-free write path has therefore been designed for PALF to avoid contention among threads. The main components in the write path are the log sequencer and the group buffer. The log sequencer assigns LSNs to log entries sequentially. We have implemented a lock-free log sequencer to avoid it becoming a bottleneck. When a thread appends a log to PALF, it loads the value of the LSN tail atomically to a temporary variable, updates the temporary value, and stores it to the LSN tail with an atomic compare-and-swap operation. If the compare-and-swap operation fails due to concurrent appending, then the thread reloads the LSN tail and loops around to acquire LSN again [21].

After acquiring the LSN, multiple threads fill log entries into the group buffer concurrently. The LSN not only acts as a log entry's address on disk, but also serves as the offset of the log entry in the group buffer, which means that the reserved buffer for a log entry never overlaps with another log entry. As a result, the group buffer is not a competing resource, and multiple threads can fill log entries to different offsets concurrently without any lock overheads.

7 DESIGN CHOICES AND DISCUSSIONS

Raft vs. PALF. Raft and PALF are essentially implementations of the Paxos protocol[27]. PALF adopted the log replication of Raft for simplicity, here are some differences worth discussing.

- State Machine Model. PALF adopt the replicated WAL model for serving OceanBase database (§3.1), Raft is built at the setting of RSM model.
- Leader Election. In Raft, leader's logs must be at least as up-to-date as logs of any replica in a majority. In PALF, election priorities manipulate which replica can be elected as the leader, the *config version* acts as the chief election priority (§5.3). For correctness, the log reconfirmation is introduced (§3.4) to ensure that the candidate will hold the longest logs in a majority before it takes over as a leader.
- Pending Follower. PALF adds a new stage to the transition from *leader* to *follower* for determining the replication results of logs (§3.4) when some failures occur.
- Reconfiguration. A reconfiguration command of Raft is a normal log entry, but PALF decouples it as a meta entry for independent PALF group mirror (§5.3).
- Log Index. Raft adopts a continuous numeric log index for log replication. PALF adopts two log entry identifiers, LSN and CSN. The continuous LSN is used to replicate and store logs, the CSN is used to track the order of operations across multiple PALF groups.

PALF Group Mirror vs. Paxos Learners. An intuitive question may be why we choose to construct physical standby databases by streaming between PALF groups, rather than synchronizing logs to physical standby databases as Paxos learners. In practice, two main reasons motivates the design of the PALF group mirror.

The first reason is failover. If the primary database crashed, the standby database should be able to be switched as new primary database and start to serve user requests. However, it is complicated for Paxos learners to be elected as the leader of new primary database because learners are not the part of the Paxos membership. In contrast, PALF group mirror is an independent Paxos group, it can be easily switched to a primary PALF group by changing its access mode. The second reason is maintainability. Most reconfiguration algorithms in consensus protocols are executed by the leader. If the standby databases depend on Paxos learners, reconfiguring a standby database must require it to contact with the primary database, it is unacceptable when the network between them is broken. For PALF group mirror, independent reconfiguration enables administrators to reconfigure the standby database even if the primary database crashes.

8 EVALUATION

In this section, we evaluate PALF performance experimentally. More specifically, we seek to answer the following questions:

- What level of performance could PALF achieve?
- How does the optimization in PALF impact its performance?
- Does log reconfirmation impact failure recovery?
- Is PALF competent as the WAL of OceanBase database?

8.1 Overall Performance

Testbed. All experiments (except §8.4) were performed on a cluster of three commodity servers. Each server is equipped with a 32-core 2.5GHz Intel Xeon CPU, 256 GB memory, and four SSD disks, all connected via 10 Gigabit Ethernet with 0.2 *ms* average latency. Three replicas were placed on different servers.

Baselines. We compared PALF with etcd-raft[15] and braft[2], which are two open-source implementations of Raft[33], upon which some industrial distributed systems[14, 42] have been built. The main differences in system design are multicore scalability and group replication. For instance, in etcd-raft, clients propose logs to raft node through a Go channel, which facilitates its usage but limits throughput under high concurrency. Even though both etcd-raft and braft implement some batching optimizations, such as batching disk I/O requests and reducing the number of network packets, they still process log entries one by one in the consensus protocol, which limits throughput.

Client Model. We built closed-loop clients for PALF, etcd-raft, and braft. Each client does not append new logs to the leader until its previous appended log has been committed. To emulate common use cases of the write-ahead logging system (as an inner component of the distributed database), clients are co-located with the leader and append logs to the leader directly.

Throughput. PALF scales throughput greatly as the number of clients increases. As illustrated in Fig. 9a, PALF handles 478K *append* requests per second with 1500 clients and 1480K requests per second with 8000 clients. These throughput numbers are much higher than the baselines. The speedup ratio of PALF is 5.98 when the number of clients increases from 500 to 8000, whereas the speedup ratios of etcd-raft and braft are 1.386 and 1.8 respectively. This means that PALF can make full use of modern multicore hardware. There are several reasons for this. First, PALF's lock-free write path minimizes the overhead caused by lock contentions. Second, the group buffer could be filled concurrently, and a fleet of cached logs could be replicated within one consensus instance.

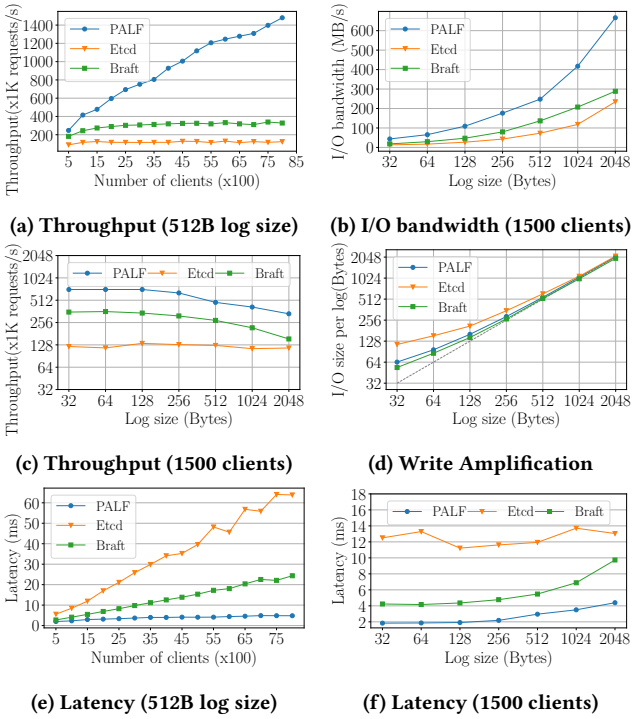
We further evaluated the impact of log size on I/O bandwidth and throughput. As shown in Fig. 9b, etcd-raft achieves comparable I/O bandwidth to braft and PALF, especially for small log sizes. This seems to be inconsistent with the result depicted in Fig. 9a. To discover the reason, we further plotted the relationship between throughput and log size in Fig. 9c. The result shows that the valid throughput of braft is much higher than that of etcd-raft, but that their I/O bandwidths are of the same order of magnitude. This implies that the real I/O size for each log may be amplified by etcd-raft. Write amplification depicted in Fig. 9d validates the previous reasoning. It was also found that the real I/O size of each log in PALF is larger than that of each log in braft when log size is smaller than 128 bytes. The reason for this is group replication. Besides the log header (Fig. 5), an additional group log header will be attached to each group log, which amplifies the real I/O size of logs.

Latency. As illustrated in Fig. 9e, the latency of PALF is about 2 *ms* when the number of clients is less than 1500. Although PALF has been evaluated to be friendly to high concurrency, latency still increases slightly with the number of clients, up to 4.8 *ms* under 8000 clients. In the write path of PALF, LSN and CSN allocation is a procedure that must execute sequentially. If a thread fails to allocate LSN due to concurrent requests, the retry operation will incur additional latency. We performed an evaluation to determine whether the log sequencer is the bottleneck under high concurrency. Our evaluations showed that the log sequencer can handle 5.88 million requests per second under 1000 threads. Therefore, the LSN allocator is far from being a bottleneck.

The replication latency consists of memory copy, disk flushing, and network transmission; these overheads are all correlated to the log size. Therefore, it is reasonable that larger logs incur higher latency (Fig. 9f).

8.2 Adaptive Group Replication

We evaluated the effect of adaptive group replication under different concurrency levels. The threshold of the group factor was set to 10 because 10 workers were handling clients' requests. In Figure 10a, fewer consensus instances mean less overhead. PALF proposes more consensus instances when group replication is disabled; it must consume more computing resources than group replication. As the number of clients increases, adaptive group replication sharply reduces the number of consensus instances. The performance improvement is visually depicted in Fig. 10b. When throughput is low, the adaptive group replication tracks the low latency property of the feedback group replication. As the degree of concurrency increases, PALF switches to periodic group replication and achieves significant throughput. In conclusion, the adaptive group replication combines the advantages of two aggregation strategies, which allows PALF to perform well under different concurrency levels.

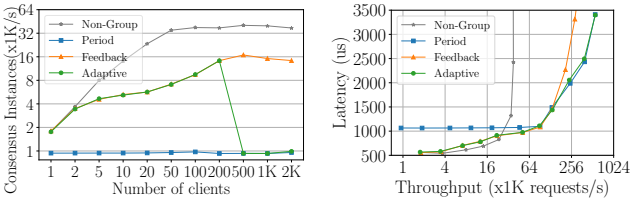


(a) Throughput (512B log size) (b) I/O bandwidth (1500 clients)

(c) Throughput (1500 clients) (d) Write Amplification

(e) Latency (512B log size) (f) Latency (1500 clients)

Figure 9: Performance with different clients and log size.

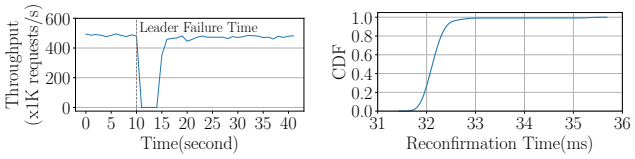


(a) Consensus Instances (b) Latency vs. Throughput

Figure 10: Log Aggregation Evaluation.

8.3 Failure Recovery

Besides peak performance, we also measured the performance trace and recovery time when the leader crashes, which is critical for database availability. Figure 11a shows that PALF recovers from leader failure within a short time, and the new leader achieves equivalent throughput to the previous leader. PALF recovery consists of two stages: leader election and log reconfirmation. The duration of leader election is mainly subject to lease (4 s as default); log reconfirmation takes up extra time compared to Raft. Figure 11b illustrates the cumulative distribution function (CDF) of the reconfirmation time. The median and 90-th percentile were 32.2 ms and 32.5 ms respectively. Even the reconfirmation time is related to the gap of log size between the new leader and the replicas that own the most logs, election restriction (in §5.3) guarantees that the logs of the new leader will not fall far behind. Moreover, the efficient write path of PALF further shortens the recovery time. These optimizations make the overhead of log reconfirmation negligible.



(a) Throughput (Leader failure) (b) CDF of Reconfirmation Time

Figure 11: Failure recovery of PALF.

8.4 Evaluating PALF within OceanBase

To validate how log replication affects database performance, we measured OceanBase 4.0 under both benchmarking tools and production environments. To exert enough pressure on PALF, we chose two Sysbench [25] cases (*insert* and *update*) and TPC-C[10] (200 warehouses) as workloads, evaluated PALF with key-value mode of OceanBase under YCSB *batch insert* workloads, and measured PALF performance in a production cluster. The benchmarking experiment was performed on a cluster of three commodity servers, which are equipped with a 96-core 2.5GHz Intel Xeon CPU and the same hardware as in the earlier experiments; The production cluster consists of 46 Aliyun ECS r5.16xlarge instances.

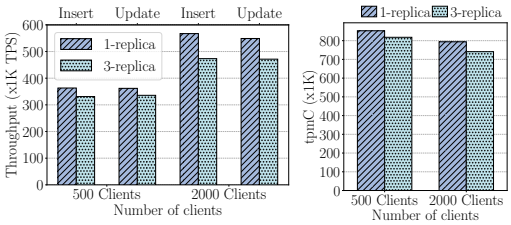
The results of transactional performance are depicted in Figure 12a and Figure 12b. Compared to stand-alone performance, log replication slightly reduces the performance of three-replica OceanBase by 8.8% on average (For fairness, only one replica of the database can process transactions). This result proves the efficiency of PALF. Note that the performance of OceanBase scales with the number of clients under Sysbench, but experiences a little decrease under TPC-C. The reason for this is the proportion of write transactions. The logging system greatly impacts the performance of write transactions, but have minimal impact on read transactions. Therefore, the performance of write-intensive workloads in Sysbench scales well due to the scalability of PALF. In comparison, the performance on hybrid workloads in TPC-C is limited to the database itself. This result shows that log replication is not a bottleneck for OceanBase.

The profiling of PALF within OceanBase during the benchmarking experiments (three-replica) and the production workloads are exhibited in Table 1, Figure 12c, and Figure 12d. The results are much lower than PALF's peak performance described in §8.1, which can achieve 1.4 million append operations per second with 512-byte payloads. Therefore, the performance of PALF is more than sufficient for OceanBase to serve intensive writes.

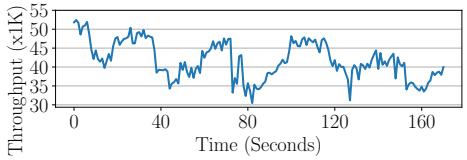
9 RELATED WORK

In this section, we discuss other contributions and how they relate to the features of PALF.

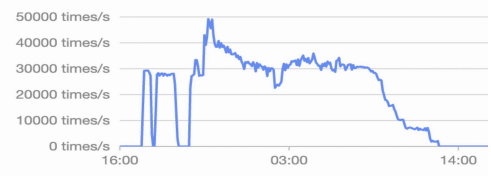
WAL in Distributed Databases. Many distributed databases have been built on top of replicated logging for fault tolerance and availability. The Raft protocol [33, 34] is widely used to synchronize logs among replicas, such as CockRoachDB [42] and YugabyteDB [48]. Spanner [9] implements a replicated state machine with Paxos on top of each tablet. Their transaction engines interact with the logging system through the replicated state machine model [39]. The choice made by PALF is to provide file-like interfaces, which makes the integration between consensus protocol and the typical



(a) Sysbench (b) TPC-C (WH=200)



(c) PALF throughput under YCSB workloads



(d) PALF throughput under production workloads

Figure 12: Performance under different workloads.

WAL model possible. Aurora[44] is a shared-storage database, the database engine takes charge of log replication and offloads the log processing to log storage, by contrast, PALF fits in a shared-nothing architecture, the database appends logs to PALF by file-like APIs and is unaware of consensus protocol.

These distributed databases adopt the approach of bundling log replication with data partition (tablet in Spanner, range in CockRoachDB, etc.), which achieves high throughput by parallelizing multiple partitions, but incurs more distributed transactions. A single replication group of PALF provides excellent performance, it can replicate logs of multiple data partitions concurrently, this way indirectly reduces the number of distributed transactions.

Socrates [1] uses an unbundled log service XLOG built on the Azure storage service to support an upper database tier. FoundationDB [50] adopts another unbundled architecture in which log servers are decoupled from transaction processing and logs are replicated across log servers in the charge of proxies in the transaction system. PALF provides file-like interfaces for bundled architectures.

In bundled architectures, even if additional performance gains could be attained by coupling WAL and the database tightly (e.g. making WAL transaction-aware to commit buffered logs[29], distributed transaction optimizations [17, 19]), PALF still abstracts database-specific functions as filesystem-like APIs to keep a clear boundary between the log and the database, this will bring maintainability and stability benefits to a practical database system.

xCluster[49] of YugabyteDB supports asynchronous replication between databases by transmitting logical logs with the change data capture tool, but imposing many constraints; for example, it does not support synchronizing DDL statements. CockroachDB[42] only supports replica-level asynchronous replication by non-voting replicas. These schemes are not suitable for physical standby databases, which provide an identical copy of the primary database and should be independently available. PALF offers the PALF group mirror to construct the standby database naturally.

Table 1: Profiling of PALF within OceanBase

Workloads	LogSize	Throughput	RT(μs)	I/OPS
Insert	499 B	472181	3312	1125
Update	433 B	484061	3111	1111
TPC-C	3386 B	27975	3878	1929
YCSB	1993 B	41481	2024	1090
Production	1060 B	23253	4752	1129

Replicated Logging Systems. Replicated logging systems are widely used to persist and order updates in distributed systems. Many replicated logging protocols sequence records by a distinguished leader [9, 18, 22, 30, 34]. A shared log abstraction has been proposed to funnel all updates through a global ordering layer, such as CORFU [4] and Scalog[13]. For distributed databases, the order of logs is essentially determined by the transaction engine rather than the logging system. Some logging systems [18, 42] pack transaction identifiers into log records, which may incur inconsistency between the transaction order and the logging order. PALF provides a change sequence number primitive that tracks the transaction order with the logging order.

As for independent reconfiguration, another promising solution is separating configuration management from log replication, as in Vertical Paxos[28] and Delos[3]. Most of these studies give the responsibility for reconfiguring clusters to another service; the approach may introduce additional availability risk to systems [16]. These systems are often internal services of big companies rather than a common software product like the OceanBase database. The PALF approach achieves both independence and generality of reconfiguration. MongoRaftReconfig[40] also stores reconfiguration logs separately like PALF, but its motivation is to recover a consensus group from majority failures. In that scenario, safety properties of Raft reconfiguration may not be ensured; MongoRaftReconfig proposed an extended reconfiguration protocol and proved its safety.

10 CONCLUSION

This paper has presented PALF, which acts as the replicated write-ahead logging system of OceanBase and is expected to serve as a building block for many other distributed systems. The key idea is to abstract database-specific requirements to PALF primitives. Specifically, PALF provides typical file system interfaces and explicit replication results at a consensus level, which facilitate the integration between the transactional system and WAL. The CSN primitive helps to track the order of transactions by the order of logs. Data change synchronization has been abstracted as PALF group mirrors; downstream applications would benefit from this. Under typical OLTP workloads, PALF serves the OceanBase database in a comfortable manner and offers space for future advances.

ACKNOWLEDGMENTS

This is supported by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. TD2019001).

REFERENCES

[1] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1743–1756, New York, NY, USA, 2019. ACM.

[2] Baidu. Braft: An industrial-grade c++ implementation of raft consensus algorithm based on brpc., Sep 2015. URL: https://github.com/baidu/braft.

[3] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632, Virtual Event / Portland, OR, USA, 2020. USENIX Association.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan.

[5] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, USA, 2006. USENIX Association.

[6] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 2007 annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[7] Cockroach. Replication layer of crdb., Oct 2022. URL: https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html#non-voting-replicas.

[8] Cockroach. Transactions of crdb., Oct 2022. URL: https://www.cockroachlabs.com/docs/stable/transactions.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[10] T. P. P. Council. Tpc benchmark, 2010. URL: http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf.

[11] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, sep 1985.

[12] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, dec 2004.

[13] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 325–338, USA, 2020. USENIX Association.

[14] Etcd-Io. Etcd: Distributed reliable key-value store for the most critical data of a distributed system, 2013. URL: https://github.com/etcd-io/etcd.

[15] Etcd-Io. Etcd raft library, Jun 2013. URL: https://pkg.go.dev/go.etcd.io/etcd/raft/v3.

[16] Pedro Fouto, Nuno Preguiça, and Joao Leitão. High throughput replication with integrated membership management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 575–592, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/fouto.

[17] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, mar 2006. doi:10.1145/1132863.1132867.

[18] Sijie Guo, Robin Dhamankar, and Leigh Stewart. Distributedlog: A high performance replicated log service. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1183–1194, 2017.

[19] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. Cornus: atomic commit for a cloud dbms with storage disaggregation. *Proc. VLDB Endow.*, 16(2):379–392, oct 2022. doi:10.14778/3565816.3565837.

[20] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *International Workshop on High Performance Transaction Systems*, pages 301–329. Springer, 1987.

[21] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

[22] Heidi Howard. *Distributed consensus revised*. PhD thesis, University of Cambridge, 2019.

[23] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[24] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[25] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench.sourceforge.net/*, 2004.

[26] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[27] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[28] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[29] Jonathan Lewis. *Oracle Core: Essential Internals for DBAs and Developers*. Apress, 2012.

[30] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[31] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained replicated state machines for a cluster storage system. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 305–323, Santa Clara, CA, February 2020. USENIX Association.

[32] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, 2010.

[33] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, Stanford, CA, USA, 2014. AAI28121474.

[34] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association.

[35] Oracle. Archived redo logs in oracle, Jul 2013. URL: https://docs.oracle.com/cd/B19306_01/server.102/b14231/archredo.htm.

[36] Oracle. Oracle data guard, Jul 2013. URL: https://www.oracle.com/database/dataguard/.

[37] Oracle. Oracle recovery manager, Jul 2013. URL: https://www.oracle.com/database/technologies/high-availability/rman.html.

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

[39] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[40] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2021.26, doi:10.4230/LIPIcs.OPODIS.2021.26.

[41] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. Database system concepts. 2011.

[42] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, 2020. ACM.

[43] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 2312–2325, New York, NY, USA, 2022. ACM.

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3056101.

[45] Zhenkun Yang, Chen Qian, Xuwang Teng, Fanyu Kong, Fusheng Han, and Quanqing Xu. LCL: A Lock Chain Length-based Distributed Algorithm for Deadlock Detection and Resolution. In *Proceeding of the 39th IEEE International Conference on Data Engineering (ICDE)*, 2023.

[46] 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. Oceanbase:

A 707 million tpmc distributed relational database system. *Proc. VLDB Endow.*, 15(12):3385–3397, sep 2022.

[47] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. Oceanbase paetica: A hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster. *Proc. VLDB Endow.*, 15(12):3385–3397, sep 2023.

[48] Yugabyte. Yugabyte-db: The cloud native distributed sql database for mission-critical applications., Jan 2016. URL: https://github.com/yugabyte/yugabyte-db.

[49] Yugabyte. xcluster replication., Oct 2022. URL: https://docs.yugabyte.com/preview/architecture/docdb-replication/async-replication/.

[50] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2653–2666, New York, NY, USA, 2021. ACM.