Occam's Razor for Distributed Protocols: Theory and Applications in Geo-distributed Commit

Modern geo-distributed databases rely on state-machine-replication (e.g., Multi-Paxos) and atomic commit (e.g., 2PC) for high availability and consistency, but they associate with high overhead. Previous efforts often improve performance by sacrificing quality-of-service (QoS) including availability and reliability, or compromising workload generality. Instead of throwing away battle-tested protocols and inventing yet another complex one, this paper presents a general theory for optimizing any existing protocols without affecting QoS or generality. Incorporating the optimizations only requires bolt-on message adapters without touching the implementation of core protocol modules. Applying the theory to Spanner's 2PC&Paxos can improve its latency by 50% and its throughput by 1.5×. The resulting protocol can even beat state-of-the-art by having 81% lower tail latency than TAPIR, 1.3× and 1.9× higher throughput than E3 and GPAC, respectively.

1 INTRODUCTION

Distributed protocols like State-Machine-Replication (SMR) [57–59, 79, 84] and Atomic Commit (AC) [5, 6, 17, 43, 77, 95] are the cornerstones of many modern geo-distributed databases, including Spanner [29], CockroachDB [99], and PolarDB-X [21]. SMR offers high availability and replica consistency. AC provides atomicity when transactions span across service domains. In response to the dual demands for high availability and high scalability, *replicated-partitioning* is now the *de facto* setup for geo-distributed databases [41, 45, 53, 74, 76, 81, 91, 111–113, 118, 120, 122]. Figure 1 shows an example of a typical 3-way-replicated-partitioning setup, where user data is partitioned by region and each partition is replicated onto three geo-distributed data centers.

In a replicated-partitioning setup, when a distributed transaction finishes execution, atomically committing a transaction with consistent replication requires a *Replicated Atomic Commit* (RAC) protocol. Early systems like Spanner use a combination of two-phase commit (2PC) [77] and Paxos [57] as their RAC protocol. However, since a plain integration of 2PC and Paxos would incur many cross-data-center communications, the latency of committing a distributed transaction in Spanner is as high as 3 round-trip times (RTTs) [29].

Due to its importance, numerous new RAC protocols have been proposed in recent years. Most of them, however, are piecemeal solutions that optimize performance by sacrificing other important quality-of-service (QoS) attributes or limiting the class of transactions that can be supported. For example, ReplicatedCommit [74] is able to reduce the commit latency from 3 RTTs down to 1. Unfortunately, our thorough analysis (Section 2) uncovers that it actually sacrifices liveness, i.e., it blocks upon a single data-center outage. As another example, MDCC [53] and TAPIR [118] also enable committing a transaction in 1 RTT, but they actually sacrifice the system's reliability — by increasing the quorum size from $f + 1$ to $\lceil \frac{3}{2} f \rceil + 1$. Under a typical 3-way replication setup with $f = 1$ fault tolerance, that means a commit needs responses from *all* 3 data centers instead of 2, making their tail latency way more sensitive to individual data center loads and the cross-data-center traffic. Recent research on deterministic databases has achieved good throughput and latency while maintaining high availability. However, they actually trade off generality in terms of the transaction model (e.g., within a transaction, a query cannot be based on a previous query's results).

Since the release of Spanner a decade ago, the community has been questing for a RAC protocol with strong performance and strong consistency, without sacrificing other key features. Specifically, we identify five requirements essential for a strong consistent RAC protocol in modern geo-distributed environments: **(R1) high throughput**, **(R2) low (average) latency**, **(R3) low tail latency**, **(R4) high availability**, and **(R5) workload generality**. R1 and R2 represent typical performance requirements although they are less coupled in a distributed setting — R2 can generally be achieved by optimizing the *critical path* (as most existing work does) but R1 requires optimizing *every path* in the protocol to avoid lock contentions and mitigating bottlenecks (e.g., WAN bandwidth bound). R3 is always an important factor for good user experience [16, 26, 33, 51, 60, 85, 87, 90], but is also often traded for R1 and R2. In today's cloud-major market, R4 is not just an option but a requirement. R5 is a crucial functional requirement that would influence the user base and market share.

In this paper, we present *Reduction Theory*, a theory that serves as a "razor" to identify and decouple any unnecessary dependency in a distributed protocol. Decoupling unnecessary dependencies in every part of a

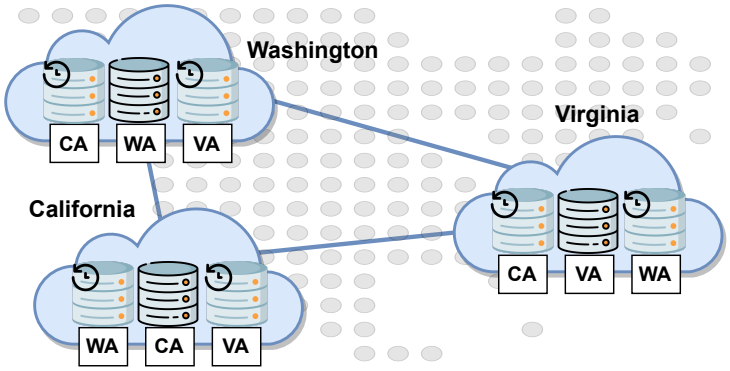


Fig. 1. A 3-way-replicated-partitioning setup. User data is partitioned by regions (CA, WA, VA) and each partition is replicated onto 3 data centers.

protocol would not only reduce the commit latency but also increase the throughput. Beyond that, the theory has three nice properties:

- **General.** It does not assume any specialized hardware (e.g., NOPaxos requires programmable switches [64]), imposes no restriction on the transaction model, nor requires any specific setups (e.g., ReplicatedCommit requires a *full replication* setup, i.e., for a dataset that has 10 partitions, each datacenter must replicate all 10 partitions).
- **No QoS Tradeoff.** When applied to a protocol, it maintains the protocol's original quality-of-service (i.e., it preserves the protocol's original availability and reliability level).
- **Low Adoption Barrier.** The community has been developing new RAC protocols, but they are often subtle and complex, leading to very high barriers to adoption and only a few successfully entering the industry. Our theory is not a new protocol but a new optimization tool. Protocol optimizations derived from the theory can be applied to existing systems without throwing away any battle-tested modules, highlighting its potential of widespread adoption.

Focusing on geo-distributed databases, we apply the theory to some state-of-the-art RAC protocols. Starting from Spanner's 2PC&Paxos protocol, our theory inductively decouples all its unnecessary dependencies and results in a strongly consistent RAC protocol that satisfies all five requirements. We name the resulting protocol, *Occam*, reflecting its way of using the reduction theory as the razor. In summary, the contributions of this paper include:

- **Reduction Theory.** A theory that uncovers optimization opportunities in any distributed protocol, improving performance while maintaining safety and liveness, without altering the implementation of the core protocol modules.
- **Occam.** A strongly consistent RAC protocol that can support all SQL workloads without QoS tradeoffs. Experiments demonstrate that Occam achieves 32–50% lower latency than Spanner, GPAC [76], CockroachDB, with up to 1.9× higher throughput. When compared to latency-optimized ones, Occam achieves up to 81% lower tail latency than TAPIR, and 1.3× higher throughput than E3 [93]. Distributed protocols are notoriously hard to get right (see a list of bugs found in [88]). Occam has a TLA+ [56] specification in [4] to ensure its correctness.
- **Case studies.** Beyond 2PC&Paxos, we apply our theory to two more recent RAC protocols: TAPIR and GPAC. Despite the fact that they have already been optimized for low latency, we demonstrate that applying our theory to them can uncover additional optimization opportunities that were unknown otherwise. We further demonstrate the generality of our theory by optimizing WormLog [93], a shared-log protocol, thereby showcasing its applicability beyond RAC protocols.

2 BACKGROUND AND PRELIMINARY

Distributed transactions typically consist of an execution phase and a commit phase. To support strong consistency, the execution phase has to use a strong isolation level (e.g., serializable) and the commit phase has

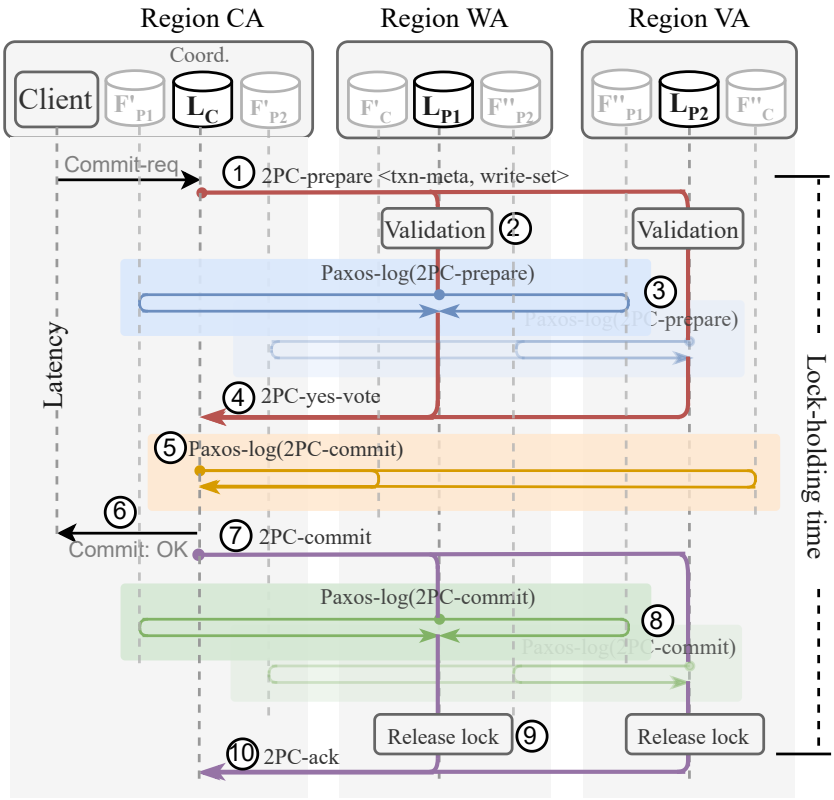


Fig. 2. 2PC&Paxos (3-way-replicated-partitioning)

to guarantee atomicity, durability, and replica consistency (e.g., sequential consistency). In this paper, we focus on the commit phase. Our discussion is orthogonal and compatible with other execution components (e.g., concurrency control).

2.1 2PC&Paxos

Paxos has many variants, but its stable-leader versions, Multi-Paxos (M-Paxos) [57] and Raft [84], are the de facto standard with numerous industrial-strength implementations. In M-Paxos and Raft, there is a designated leader (L) with the most up-to-date copy, and follower replicas (F) regard the leader as the single source of truth to catch up. Both M-Paxos and Raft employ stable leaders, meaning that a leader serves until failure, at which point a new leader is elected. The leader's role is to act as a gateway, processing external commands and replicating them to the followers. Under the replicated-partitioning setting, a data center typically hosts not only the leader replica of a partition coming from its local region but also some follower replicas of partitions from other regions. In what follows, we use the term "Paxos" to refer to M-Paxos or Raft.

Figure 2 illustrates the operations in a commit phase using 2PC and Paxos after a distributed transaction has finished an OCC-style (e.g., [9]) execution phase. The setup in Figure 2 is a typical 3-data-center geo-replicated-partitioning setting. The example transaction is ready to start 2PC using L_C as the transaction coordinator and L_{P1} and L_{P2} as participants. This scenario is based on a common setup where the client and the coordinator exhibit spatial locality and are co-located within the same region.

In the beginning ①, the coordinator L_C in the CA data center first sends a 2PC-prepare message, including transaction metadata (e.g., transaction ID, concurrency control information such as read keys and version numbers) and the transaction write-set, to the participants: L_{P1} and L_{P2} in the WA and VA data centers, respectively [1 one-way trip time (OWTT)]. Upon receiving the 2PC-prepare message ②, participants independently verify locally if they can commit the incoming write-set by cross-checking the version numbers and locking the write sets. If local validation is successful ③, they individually "log" the 2PC-prepare message. Unlike a classic distributed database, participants use "Paxos as the log" to consistently log the 2PC-prepare message across three replicas. With stable leaders, such Paxos-based logging only requires 1 RTT to replicate the log

entry, without spending RTT to elect a new leader every time. After logging, ④ participants either return a 2PC-yes-vote or a 2PC-no-vote to L_C [1 OWTT]. The coordinator L_C awaits votes from all participants ⑤ to make an abort-vs-commit decision or unilaterally decides to abort on timeout. The decision is logged using Paxos, using another 1 RTT. Finally, ⑥ the decision is returned to the client. After acknowledging the client, ⑦ the coordinator initiates an *asynchronous phase* to relay the 2PC-commit/2PC-abort decision to other participants [1 OWTT]. Participants receiving the commit/abort message ⑧ again log the decision using Paxos using 1 RTT and then apply/discard the write-sets. Regardless of the abort-or-commit decision, each participant releases all locks ⑨ acquired and replies a 2PC-ack in response ⑩.

Generally, in the absence of failures, the **average latency** of an RAC protocol is influenced by both the *RTTs in the critical path* and the *quorum size*, while the **tail latency** is primarily influenced by the *quorum size*. WAN bandwidth is limited (1Gbps compared to 10Gbps LAN), making *bandwidth consumption* the dominant factor for **throughput** in a geo-distributed system when workload contention is low. When the workload contention is high, the *lock-holding time* typically determines the system throughput [54].

In Paxos&2PC, the user receives the commit result in Step ⑥. Consequently, the user-perceived latency for the commit phase consists of 2 OWTTs (for the 2PC prepare phase), 1 RTT to log the 2PC-prepare message by the participants, and 1 RTT to log the 2PC-commit decision by the coordinator. This amounts to a total of 3 RTTs. Regarding the lock-holding time (a.k.a, the contention footprint [54]), locks are obtained from Step ② (OCC's validation) until Step ⑨. This results in a lock-holding time of 3 RTT + 1 OWTT + 1 RTT = 4.5 RTTs.

Since its logging is based on Paxos, 2PC&Paxos requires a quorum of $f + 1$ replicas. The 2PC-prepare messages account for the majority of bandwidth consumption, as their payloads contain both transaction metadata and the write-set (e.g., up to 7Kb in TPC-C). The sizes of other messages are relatively negligible, as they have no payload. Assuming an N-way replication setting and a transaction accessing data across M partitions, the coordinator sends 2PC-prepare messages of size P to M participants during the prepare phase, consuming MP bandwidth. Each participant logs (replicates) those to $(N-1)$ followers, consuming $M(N-1)P$ bandwidth. Therefore, the total bandwidth consumption is MNP.

2.2 Beyond 2PC&Paxos

Next, we provide background on other protocols beyond 2PC&Paxos. We include only those that offer strong consistency and are specifically optimized for the geo-replicated-partitioning setting. Other related work can be found in Section 6.

2.2.1 Protocols without QoS tradeoff. GPAC [76] elects the coordinator as a new leader for each transaction (unlike M-Paxos, which has a stable leader), such that 1 RTT can be saved because the coordinator itself can initiate the Paxos-based logging process without contacting a remote leader. Specifically, during each leader election, the coordinator combines the leader election message and the 2PC-prepare message to followers using 1 OWTT. Followers validate the transaction and reply by combining their ballot messages and their 2PC-yes/no-vote using another OWTT. The coordinator then makes the final commit/abort decision based on the 2PC-votes and logs (and replicates) that decision using another RTT. Consequently, GPAC requires 2 RTTs in the critical path. Electing the coordinator as the leader also saves bandwidth. The coordinator sends 2PC-prepare directly to all replicas, with M of them in the local data center. This reduces the MNP bandwidth consumption to $M(N-1)P$. Nonetheless, GPAC may increase the lock-holding time for aborted transactions. Unlike 2PC&Paxos where a leader validates a transaction and releases the lock right after finding the validation fails, since GPAC decentralizes the validation to all replicas, a successfully validated replica has to hold the lock until it gets notified by some other who fails the validation. This degrades GPAC's throughput under high contention.

2.2.2 Protocols that trade reliability. Early research like MDCC [53] and TAPIR [118] adopt the concept of *fast-path* [59] to execute Step ② (validation) in a *leaderless* mode. Specifically, MDCC and TAPIR allow the coordinator to send the 2PC-prepare (write-set) *straight* to both leader and follower replicas. These replicas then independently validate the transaction and reply with a 2PC-yes/no-vote to the coordinator like in GPAC, saving 1 RTT.

However, for fault-tolerance, the coordinator must increase the quorum size from $f+1$ to $\lceil \frac{3}{2}f \rceil + 1$. The fast-path requires a larger quorum size because there is no longer a single source of truth in the leaderless setting [59]. By forming a quorum with more replicas (i.e., $\lceil \frac{3}{2}f \rceil + 1$), the system can ensure that even if f of the quorum members have failed, the $\lceil \frac{f}{2} \rceil + 1$ survivors are still the majority of the remaining $f+1$ nodes and can recover the quorum's decision. However, when the coordinator fails to receive a quorum of matching votes from replicas, it has to fallback to a leader-based *slow-path*. The latency of both MDCC and TAPIR is 1 RTT when the fast-path is successful. But when the slow-path is required, another RTT is needed. The larger quorum size and the case of slow-path can lead to increased *tail* latency. In a 3-replica setting, all three replicas must respond, making the latency sensitive to stragglers and network jitters. Since both TAPIR and MDCC also decentralize the validation to all replicas like GPAC, their WAN bandwidth consumption is also $M(N-1)P$ and with a longer lock-holding time for aborted transactions.

2.2.3 Protocols that trade availability. ReplicatedCommit (RC) [74] can be considered a leaderless protocol, but unlike MDCC and TAPIR, it neither increases the quorum size to $\lceil \frac{3}{2}f \rceil + 1$ nor incorporates a slow path. Given $2f+1$ replicas, RC commits (or aborts) a transaction once the client receives $f+1$ commit (or abort) votes. However, as discussed in Section 2.2.2, a quorum of $f+1$ is insufficient for liveness in the fast-path. Consider an example with three replicas, where R_F fails after voting, and the two survivors have split votes, say R_1 votes for yes and R_2 votes for no. In this case, using a quorum size of f+1=2, RC would regard that a quorum has been reached but just the final decision value is unclear to the survivors. Hence, the only option left is to block until R_F recovers.

2.2.4 Protocols that trade generality. Some recent protocols optimize performance by restricting the transaction models. For instance, deterministic databases [41, 81, 91] can support very efficient replicated-partitioning by replicating (small) commands instead of (large) write-sets. This is possible because deterministic databases are designed to produce the same (serializable) output given the same input. However, these solutions often require workloads without randomness or runtime uncertainties. For example, Janus [81], OceanVista [41], SLOG [91], Detock [83] cannot support `Date()` and dependent queries (i.e. branches based on query results). Lynx [120], Carousel [111], and Natto [112] are not deterministic databases but share similar problems.

2.3 Protocol Optimizations

Compartmentalization [105] and PigPaxos [25] propose optimizing a state-machine-replication (SMR) protocol by scaling individual components with more instances. Both approaches specifically target SMR and optimize it at the *implementation* level.

PaxosCommit [43] proposes a decentralization approach to optimize atomic commit. Industrial products such as CockroachDB [99], PolarDB-X [21], and OceanBase [113] adopt this idea to reduce the critical path of 2PC&Paxos to 2 RTTs by eliminating Step ⑤ in Figure 2. This is made possible by leveraging the fact that the coordinator does not timeout in practice due to the Paxos-backed highly available participants. Therefore, the coordinator's veto right can be completely revoked such that the final commit/abort decision of a transaction can always be *cooperatively learned* [44] from the participants' logs (Step ③). Consequently, Step ⑤ becomes unnecessary and can be removed from the protocol.

WormSpace [93] is a framework for implementing distributed protocols and its capability has been demonstrated by constructing multiple protocols, one of which is an RAC protocol named E3. E3 adopts all optimizations in PaxosCommit, and it outperforms PaxosCommit by incorporating another common optimization that uses one of the partition leaders as the coordinator. However, E3 exhibits a higher WAN bandwidth consumption of $MN\mathcal{P}$ (where $\mathcal{P} > P$) because it requires each replica to broadcast all received messages, many of which may be unnecessary for certain clients.

To our knowledge, this paper is the first to present a general framework to optimize any distributed protocol at the *protocol* level. Our theory not only encompasses all protocol-level optimizations discussed above (e.g., the decentralization technique proposed by PaxosCommit), but also uncovers new optimizations that are novel to the community (e.g., *shortcutting* and *parallelization*; Section 4).

3 THE RAZOR: REDUCTION THEORY

Distributed systems like geo-distributed databases are often the powerhouse of multi-billion-dollar businesses [12, 21, 24, 29, 94, 99]. Systems of this scale often consist of thousands to millions of lines of code, with numerous well-optimized and battle-tested modules. Our theory is designed to elicit optimizations from existing protocols, allowing them to take effect by adding message adapters.

The theory has two parts: formulating a protocol P as a *transducer network* (Section 3.1), and applying a *reduction rule* (Section 3.2) to reduce certain subpaths in the network. Given a protocol P with some serial steps ($a \to b \to c \to d \dots$), the reduction rule can unveil any *weak dependency* that is implicitly parallelizable, say $b \to c$, and introduce message adapters on top so that b and c can run in parallel (i.e., $a \to \{b, c\} \to d$) without altering the protocol semantics. Each successful application of the reduction rule can reduce the latency of two sequential OWTTs to one. The theory allows the reduction rule be applied inductively. Hence, the final protocol P^* is obtained from a sequence of reductions $P \Rightarrow P' \Rightarrow P'' \Rightarrow \cdots \Rightarrow P^*$, until P^* contains no more weak dependency.

3.1 Transducer Network

Following the *relational transducer network* that formalized the CALM theorem [48], we propose a simplified version, called *transducer network*, as a specification for distributed protocols. It is important to note that our theory is agnostic to the modeling language — it works regardless of whether the protocol is specified in TLA+ [56], relational transducer network, or our proposed transducer network. Nonetheless, we find that the transducer network offers the best clarity for our context.

A transducer network $G(V, E)$ is a directed graph consisting of a set V of *transducers* as vertices and a set E of edges as *message buffers* between the transducers. A transducer is simpler than a relational transducer [48] because it concerns no relational operations. Specifically, a transducer v is an abstract state machine with a state s and a set of transitions Π. Each transducer transition $\pi = (\tau, \delta) \in \Pi$ consists of two transition functions τ and δ, which are triggered upon receiving a message set M_i containing one or more messages from incoming message buffers. The output transition function τ outputs a message set $M_o = \tau(M_i, s)$, while the state transition function δ transits the current state s to $s' = \delta(M_i, s)$. Both τ and δ are deterministic functions. In the following discussion, when the context is clear, we use the terms "message(s)" and "message set" interchangeably. Also, for easy identification of the transducer V_i of a concerned transition π, we sometimes may tag the name of a transducer to a transition like this π^{V_i} (or Π^{V_i} when plural).

In a transducer network, one transducer sends messages to another by placing the messages into the corresponding message buffer. The message buffer models an asynchronous network, accommodating out-of-order messages, duplicates, and unbounded delays. The receiving transducer can pull messages from a message buffer an unbounded number of times, and it will eventually receive all messages sent by the others. As synchrony is a special case of asynchrony, our reduction theory based on the latter also works for the former. To model a non-deterministic protocol, besides receiving messages from other transducers, each transducer regularly receives a heartbeat message h as a source of randomness. We assume a crash-recovery model [19], where a transducer halts (no transitions) after a failure but can recover with a π_r transition to restore to some previous state.

In the following, we use $\pi_a \xrightarrow{m} \pi_b$ to denote a transducer's transition π_a sends message m to another transducer to trigger its transition π_b. Similarly, $\pi_a \xrightarrow{m} \Pi_b$ means π_a broadcasts a message to trigger a set Π_b of transitions; $\Pi_a \xrightarrow{M} \pi_b$ means that transition π_b receives a set of messages M from a set Π_a of transitions; $\Pi_a \xrightarrow{M} \Pi_b$ can be viewed as a set of $\Pi_a \xrightarrow{M} \pi_b$ for each $\pi_b \in \Pi_b$.

With the above notations, we define a message path as a sequence of message passings: $\Pi_a \xrightarrow{M_a} \Pi_b \xrightarrow{M_b} \Pi_c \xrightarrow{M_c} \cdots$, where Π_a, Π_b, and Π_c are called *hops* in the path, each of which may contain one or more transitions. The message path reflects the dependencies among the transitions. For instance, $\Pi_a \xrightarrow{M_a} \Pi_b$ indicates that transitions in Π_b depend on the transitions in Π_a. A message M_x is *induced* by Π_a if there is a message path with Π_a involved before M_x is generated, i.e., $\cdots \to \Pi_a \to \cdots \to \pi_x \xrightarrow{M_x}$.

A transducer can model a process or a replicated state machine (RSM) in practice, with transition functions corresponding to event handlers in their implementations. Heartbeat messages can be generated by an external

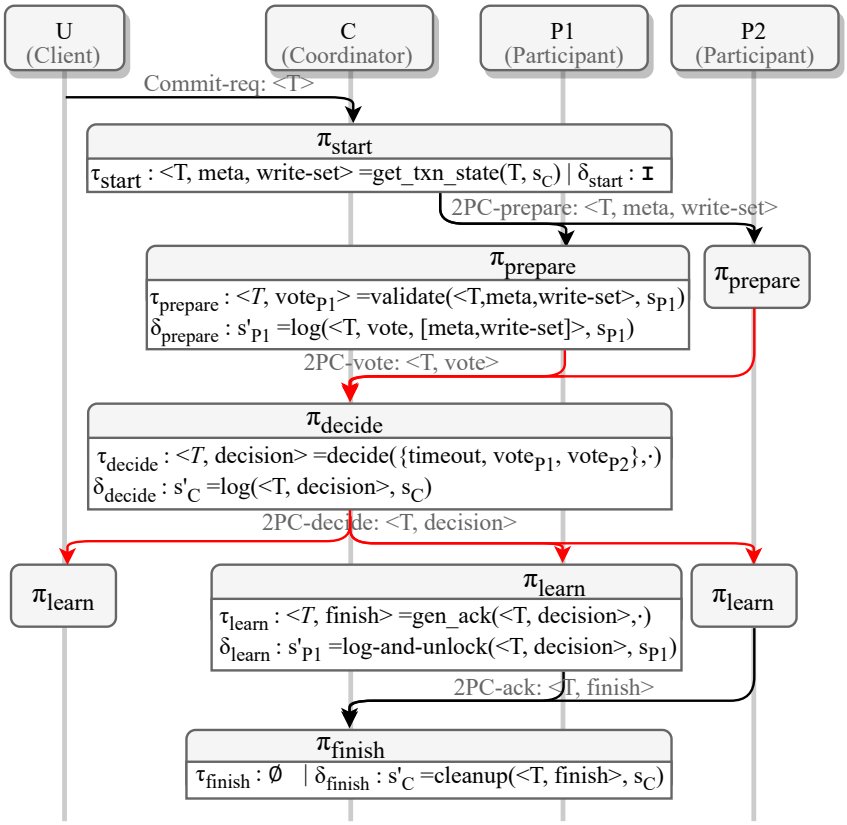


Fig. 3. Transducer network of 2PC (details of P2 are omitted for brevity.)

process, such as an operating system timer or a standalone time service. As an example, Figure 3 shows the message path of committing a transaction in the classic 2PC protocol. Note that other parts of the protocol, such as failure handling are omitted there. In this example, the client (U), coordinator (C), and participants ($P1$, $P2$) are transducers.

(1) $\pi_{\texttt{start}}$ on the coordinator is triggered by a client's transaction T as the input message. Internally, its output transition function $\tau_{\texttt{start}}$ constructs the 2PC-prepare message $\langle T, \texttt{meta}, \texttt{write-set}\rangle$. The state transition function $\delta_{\texttt{start}}$ is an identity function (i.e., it does not change state).

(2) $\pi_{\texttt{prepare}}$ on $P1$ and $P2$ react to the 2PC-prepare message from C. The output transition function $\tau_{\texttt{prepare}}$ validates the transaction, and generates 2PC-vote message $\langle T, \texttt{vote}\rangle$, where `vote = yes` if T passes validation or `vote = no` otherwise. If `vote = yes`, the state transition function $\delta_{\texttt{prepare}}$ logs the vote, metadata and write-set. Note that if optimized with *presumed abort* [78], logging in $\delta_{\texttt{prepare}}$ is not required when `vote = no`.

(3) $\pi_{\texttt{decide}}$ on the coordinator receives the 2PC-votes from the participants. Its $\tau_{\texttt{decide}}$ generates the `decision = commit` if votes are all `yes` and `timestamp` (from a heartbeat) does not exceed timeout, otherwise `decision = abort`. $\delta_{\texttt{decide}}$ logs the decision. But again, if presumed abort is enabled, it will only log when the decision is a commit.

(4) $\pi_{\texttt{learn}}$ on the participants has an output transition function $\tau_{\texttt{learn}}$ that simply generates an 2PC-ack message, and its state transition $\delta_{\texttt{learn}}$ logs the received decision and then releases the locks of T. $\pi_{\texttt{learn}}$ on the client is similar but no need to generate outputs or release locks.

(5) $\pi_{\texttt{finish}}$ on the coordinator has an empty output transition, and its state transition simply cleans up the intermediate state of T.

3.2 Reduction Rule

Given a transducer network G, the reduction rule seeks to decouple weak dependencies within the network to minimize the number of serial hops in a message path. For example, to reduce user-perceived latency, the rule could be applied to the message path from the client sending the commit command to the client learning

the commit/abort decision. To minimize the lock-holding time, the rule could be applied to the message path from lock acquisition to lock release. Shortening a message path can also help reduce bandwidth consumption. The reduction rule specifies the process of identifying a weak dependency and the appropriate method for decoupling it:

The Reduction Rule

In a transducer network G, a message path $\Pi_a \xrightarrow{M_a} \pi_b \xrightarrow{M_b} \Pi_c$ among three transducers (V_a, V_b, V_c) can be reduced to $\Pi_a \xrightarrow{M_a} \pi'_b \cup \Pi'_c$ if $\pi_b \xrightarrow{M_b} \Pi_c$ is a **weak dependency**, characterized by the following conditions:

C1: For the output transition τ_b of π_b, $\tau_b(M_a, s_b) = \tau_b(M_a, \cdot)$
(i.e., τ_b is agnostic to the state and the heartbeat)

C2: π'_b is eventually triggered if $\exists \pi'_c \in \Pi'_c$ is triggered in the **reduced transducer network G'**.

Each $\pi_c \in \Pi_c$ in G' is transformed to $\pi'_c = (\tau'_c, \delta'_c)$ with:

- $\tau'_c(M_a, s_c) = \tau_c(\tau_b(M_a, \cdot), s_c)$
- $\delta'_c(M_a, s_c) = \delta_c(\tau_b(M_a, \cdot), s_c)$ (i.e., onloading τ_b to π'_c)

For V_b in G',

- a new transition $\pi_{\texttt{proxy}}$ is added to buffer any potential message M_x induced by Π'_c until π'_b is triggered:
  - $\tau_{\texttt{proxy}} = \emptyset$
  - $\delta_{\texttt{proxy}} = \texttt{buf\_msg}(M_x, s_b)$ (i.e., buffer M_x)
- π_b in V_b is transformed to π'_b:
  - $\tau'_b(M_a, s_b) = \tau_b(M_a, s_b) \cup \texttt{get\_buf\_msgs}(s_b)$
    (i.e., deliver the buffered messages to V_b)
  - $\delta'_b = \delta_b$

Not all message paths can be reduced. Conditions C1 and C2 ensure that $\pi_b \xrightarrow{M_b} \Pi_c$ is a weak dependency such that decoupling it would not compromise the liveness and safety of the original protocol. Specifically,

- C1 checks if the output transition τ_b genuinely requires the state of its transducer to generate the output message M_b, or if the message M_b can be computed directly from M_a by any other transducer.
- C2 addresses an issue in G' where π_c could occur without π_b if the transducer of π_b fails, unlike the original network G where π_c implies that π_b must have occurred. Condition C2 rules out this scenario.

When C1 and C2 are satisfied, we can ascertain that $\pi_b \xrightarrow{M_b} \Pi_c$ is a weak dependency. In G', $\pi_{\texttt{proxy}}$ and π'_b are designed for retaining the ordering in G. Specifically, in G, M_a first triggers π_b who then triggers Π_c. Π_c may however induce a message M_x back to V_b (via other paths). In G, V_b must receive M_a before M_x. However, in G', V_b may receive M_x before M_a because Π_c could trigger before π'_b (their dependency is gone). $\pi_{\texttt{proxy}}$ is added as a guard to buffer M_x in case M_a has not arrived yet. After receiving M_a, M_x can then be delivered, essentially ensuring the original order in G. Note that the introduction of $\pi_{\texttt{proxy}}$ does not add any extra dependency because the dependency $\Pi_a \xrightarrow{M_a} \pi_b$ for waiting M_a at V_b has been in the original message path and is not our reduction target. $\pi_{\texttt{proxy}}$ is just for realizing that dependency and our focus is on decoupling the weak dependency $\pi_b \xrightarrow{M_b} \Pi_c$ in the message path.

The following theorem establishes the correctness of the reduction rule.

Theorem 1. *After applying the reduction rule to transform the transducer network G into G', G' provides the same safety and liveness guarantees as G.*

3.3 Proof

To prove Theorem 1, we first demonstrate Lemma 1, which serves as a sufficient condition that implies Theorem 1:

Lemma 1. *For every run R' in the reduced network G', there exists an equivalent run R in the original network G.*

Lemma 1 implies Theorem 1 because if G provides safety and liveness guarantees, R would eventually terminate (liveness) in a safe state. As R' is equivalent to R, each run in G' would also terminate in the same safe state, ensuring that G' provides the same safety and liveness guarantees as G.

We now formalize the concepts of a *run* and *equivalent run* and proceed to prove Lemma 1. Let S_0 represent the initial state of all transducers in G, with G starting its execution from state S_0 using an initial set of messages M_0. For this proof, we assume the sequence of all events of the execution are recorded in a log L, where an event e is a tuple $e = (M, \pi)$, capturing a transition π triggered by message M. For each transducer V, we define its local log l_V as a subsequence of L containing all events with transitions in V. We also define the state-history s_V of V as the sequence of states that V takes during the execution. G might produce different logs in distinct executions even when given the same initial states S_0 and messages M_0. This is due to the non-determinism in asynchronous networks (e.g., message reordering) and the randomness in heartbeat messages. Therefore, we use $\mathcal{L}(G, S_0, M_0)$ to represent the set of all possible logs of G for a given S_0 and M_0. Nonetheless, replaying a log $L \in \mathcal{L}(G, S_0, M_0)$ is deterministic, as transition functions are deterministic. Therefore, we define a run R of G as (S_0, M_0, L), the 3-tuple uniquely determines the state-history during a particular execution. For a protocol with liveness, every run has a finite L. For the other case, L might be infinitely long. Two runs are equivalent if their transducers' state-histories are identical.

Given a log $L \in \mathcal{L}(G, S_0, M_0)$, we can construct an equivalent run by creating a causal dependency graph C that represents the causal order of events in L. In C, the vertices are events from L, and directed edges are added between events e_a and e_b if they satisfy one of the following conditions: (a) e_a and e_b are consecutive events in the same local log l_V; (b) The transition in e_b depends on message(s) generated by the transition in e_a. As events with no causal order (i.e., lacking a path between them in C) run in parallel, their positions in L can be permuted. Consequently, for a given log L, there might be various sequences L' that conform to the topological order in C, leading to the run (S_0, M_0, L') that is equivalent to (S_0, M_0, L).

To prove Lemma 1, we begin with $R' = (S_0, M_0, L')$ in G' and demonstrate that it has an equivalent run $R = (S_0, M_0, L)$ in G. Recall that our reduction rule simplifies the message path to $\Pi_a \xrightarrow{M_a} \pi'_b \cup \Pi'_c$. We can assume $\Pi_c = \{\pi_c\}$ without loss of generality since, when Π_c contains multiple transitions $\pi_{c_1}, \pi_{c_2}, \ldots, \pi_{c_n}$, applying the reduction rule to $\Pi_a \xrightarrow{M_a} \pi_b \xrightarrow{M_b} \Pi_c$ is equivalent to applying it n times, each time on the message path $\Pi_a \xrightarrow{M_a} \pi_b \xrightarrow{M_b} \pi_{c_i}$ for $i \in \{1, \ldots, n\}$.

We examine all cases concerning events (M_a, π'_b) and (M_a, π'_c) in the reduced network's run R':

(1) Neither (M_a, π'_b) nor (M_a, π'_c) exist in L'.
(2) (M_a, π'_b) exists, but (M_a, π'_c) does not.
(3) (M_a, π'_c) exists, but (M_a, π'_b) does not.
(4) Both (M_a, π'_b) and (M_a, π'_c) exist.

Case (3) is void, as condition C2 ensures the existence of (M_a, π'_b) when (M_a, π'_c) exists. Therefore, we only need to consider (1), (2), and (4).

For Case (1), we show that the equivalent run of R in G is exactly R' itself, i.e., $R = R'$, because when (M_a, π'_b) and (M_a, π'_c) do not exist in L', the reduction rule does not manifest and affect R'.

For Case (2), we demonstrate that R has a log L identical to L', except all events (M_a, π'_b) in L' are replaced by (M_a, π_b). R is equivalent to R' because π'_b and π_b have identical state transition functions, and thus, their state-histories must be the same.

For Case (4), we show that the equivalent run R of R' in G has a log L constructed as follows:

(1) Construct a causal dependency graph C from L'.
(2) Since $\pi_{\texttt{proxy}}$ ensures π'_b would not be triggered after any transition on V_b (the transducer of π_b) that depends on π'_c, there is no path from (M_a, π'_c) to (M_a, π'_b) in C; hence, a topological order L'' of C exists where (M_a, π'_b) is ordered before (M_a, π'_c).
(3) Construct L from L'' by replacing (M_a, π'_b) with (M_a, π_b) in L'', replacing (M_a, π'_c) with (M_b, π_c) in L'', and removing events that trigger $\pi_{\texttt{proxy}}$.

$R = (S_0, M_0, L)$ is then a legitimate run in G by construction. R and R' are equivalent because, except for the transducer of π_c (denoted as V_c), each transducer V in R has the same local log as its counterpart in R', and their state-histories are also the same. For V_c, since it takes on the job of τ_b after the reduction, the resulting state-histories of V_c in R and R' are the same. Therefore, $R = (S_0, M_0, L)$ is equivalent to R'.

In conclusion, the above discussion demonstrates the correctness of Lemma 1, which implies Theorem 1.

4 RAZOR IN ACTION

Optimizing distributed protocols has traditionally been a labor-intensive process, involving the expertise of knowledgeable individuals to identify potential areas for improvement, rigorous correctness proofs, and meticulous implementation [25, 93, 105]. With Reduction Theory, this process can be simplified as two systematic steps: (1) identifying message paths that satisfy conditions C1 and C2 (i.e., has a weak dependency); and (2) applying the corresponding reductions. Step (1) can be fully automated if a TLA+ [56] specification of the protocol is available (Section 4.1). Step (2) can be done by implementing a message adapter without touching the core protocol modules (Section 4.2). Occasionally, one can add an optional step (1.5) to manually uncover more reduction opportunities (Section 4.3). We demonstrate how we apply all these to the 2PC&Paxos protocol used by Spanner [29] and obtain our highly efficient protocol, Occam (Section 4.4). Case studies of two other RAC protocols are also presented (Section 4.5).

4.1 Condition Checking (Step 1)

Given a transducer network G, the conditions of our reduction rule can be systemically validated as follows. For every three-hop subpath $\Pi_a \xrightarrow{M_a} \pi_b \xrightarrow{M_b} \Pi_c$ in G:

(1) Check the output transition τ_b of π_b. If τ_b only requires M_a to evaluate, condition C1 is satisfied. For example, if τ_b is an identity output transition that passes input M_a to output, then τ_b satisfies C1. Consider a counter-example using the highlighted path in Figure 3:

$$\Pi^{P_1,P_2}_{\texttt{prepare}} \xrightarrow{\texttt{2PC-vote}} \pi_{\texttt{decide}} \xrightarrow{\texttt{2PC-decide}} \Pi^{U,P_1,P_2}_{\texttt{learn}}$$

C1 is violated because the output transition function of $\pi_{\texttt{decide}}$ is $\texttt{decide}(\{\texttt{timeout}, \texttt{vote}_{P1}, \texttt{vote}_{P2}\})$, which requires `timeout` as an input, more than just the messages output by $\Pi^{P_1,P_2}_{\texttt{prepare}}$.

(2) If C1 holds, verify C2 next. While C2 can be verified by a model checker, if one wants to inspect G directly (N.B. not G'), checking the following condition is sufficient:

C2': G provides liveness and π_b exists in every message path in G starting from Π_a to a liveness-desirable state.

In words, if G eventually reaches a liveness-desirable state and π_b is present in every message path from Π_a to that state, triggering Π_a would eventually trigger π_b. The same holds in G', i.e., triggering Π_a would eventually trigger π'_b. With the fact that triggering Π'_c implies the presence of Π_a in the reduced path $\Pi_a \xrightarrow{M_a} \Pi'_c$, π_b would be eventually triggered in G', i.e., C2 is satisfied. Using the same highlighted path in Figure 3 as an example. C2' is satisfied because $\pi_{\texttt{decide}}$ is in every path from $\Pi_{\texttt{prepare}}$ to $\pi_{\texttt{learn}}$ and 2PC would eventually trigger $\pi_{\texttt{learn}}$ under the crash-recovery fault model adopted in our theory.

Alternatively, if the state transition function is an identity function, i.e., δ_b does not update any state, δ_b also trivially satisfies condition C2.[1]

If both conditions are satisfied, apply the reduction (Section 4.2) to the subpath.

Condition checking can be automated when given the protocol's TLA+ [56] specification, where a message buffer in the transducer network would be mapped as a TLA+ *set*, a transition π as a TLA+ *action*, and τ in π as a TLA+ *expression* in the action that updates the message buffer.

The checking can be implemented as a script that begins by enumerating all 3-hop subpaths in the protocol. For each of them, it verifies condition C1 through a simple static analysis that checks whether the expression corresponding to τ_b contains only elements from the message buffer. If this condition is met, it proceeds to check condition C2.

To check condition C2, the script first changes the subpath of interest into its reduced version in the protocol's TLA+ specification. It then introduces two variables, `i_b` and `i_c`, to the specification, which serve as indicators for the triggering of π'_b and Π'_c, respectively (e.g., setting `i_b=true` in the action of π'_b). Next, it adds an eventual invariant `<>(i_c => i_b)` to represent condition C2 in the specification. Finally, the script

[1] Even if π'_b is not triggered in a particular run, one can imagine adding a dummy trigger (which has no actual effect as δ_b is an identity function) for π'_b.

invokes the TLC model checker [3], using the modified specification as input. Condition C2 is considered satisfied if the model checker reports no error.

4.2 Implementing the reduction (Step 2)

Optimizations derived from our Reduction Theory can be implemented by adding a message adapter. Specifically, for a reduction that transforms the path $\pi_a \xrightarrow{M_a} \pi_b \xrightarrow{M_b} \pi_c$ among three transducers (V_a, V_b, V_c) to $\pi_a \xrightarrow{M_a} \pi'_b \cup \pi'_c$, the message adapter for this reduction can be implemented as follows:

- On V_a, the message adapter identifies the outbound message M_a by matching its message header. It then creates a copy of M_a and sets its destination to V_c.
- On V_b, the message adapter buffers the inbound message M_x induced by π'_c, and only after receiving M_a, M_x is delivered to V_b to trigger its corresponding RPC handlers.
- On V_c: The message adapter identifies M_a in V_c's inbound messages and rewrites it as $M_b = \tau_b(M_a, \cdot)$.

Adding a message adapter is non-invasive to the core protocol modules (e.g., Raft/Paxos library). Recall that Google experienced a loss of 15 hours' worth of data due to an inadvertent upgrade to their replication module [23]. The non-invasive nature of our theory is a key factor in bringing true impact to the industry. For example, since Spanner, CockroachDB, PolarDB-X, and OceanBase are all based on 2PC&Paxos, these industrial-grade systems can be upgraded to have Occam's performance by just adding a message adapter.

4.3 Edify more reductions

Sometimes, a violation of a condition can be easily remedied by a simple tweak, leading to more reductions. This bonus step targets individuals who have a good understanding of the protocol of interest. The following is a simple guideline to approach this process in a more systematic manner.

- On violating C1:
  - If that is due to τ_b requiring a heartbeat (i.e., timestamp) as input for timeout detection, remove the heartbeat if the transducers of Π_a are highly available. That is because a highly available transducer (e.g., a Paxos-backed participant) does not cause timeout in practice (Section 2.3).
  - If that is due to τ_b requiring state s_b as input,
    - identify any state-independent part from the original message path and rewrite it into two parallel subpaths: one state-dependent subpath and one state-independent subpath. Apply the reduction rule to the latter (See Section 4.4.2); or
    - if τ_b requires only a portion of s_b that is also available on Π_a, make τ_b stateless by having Π_a piggyback the necessary part of the state into message M_a (use it with discretion because it may cost additional bandwidth consumption).
- On violating C2: add a retry mechanism to re-trigger π_b after a failure (do it with discretion; consider whether the gain is worth the cost of the additional retry).

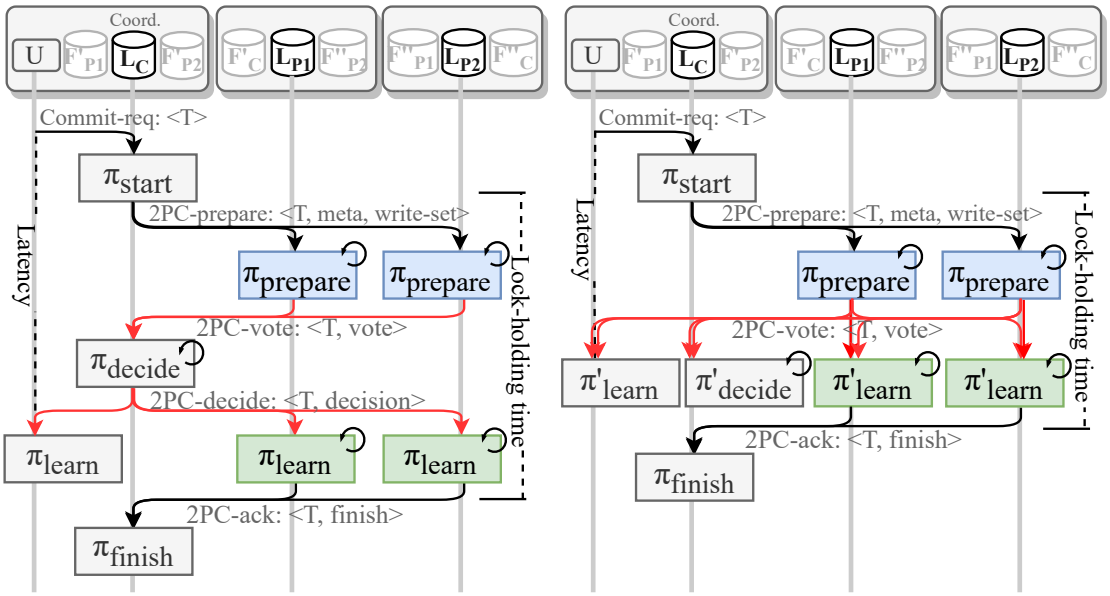
4.4 Applying to 2PC & Paxos

For 2PC&Paxos, we apply the reduction theory at two levels, first at a macroscopic level where each logical partition is modeled as a transducer (an RSM having multiple replicas). At that level, Figure 4a is the transducer network of 2PC&Paxos under its normal operation (i.e., no failure). Next, we apply the reduction theory at a microscopic level where each physical replica of each partition is modeled as a standalone transducer.

4.4.1 Reduction 1: Taking coordinator's $\pi_{\texttt{decide}}$ off the critical path.

Following Section 4.1, we enumerate all 3-hop message paths in Figure 4a. Initially, no path can directly fulfill both conditions at the macroscopic level. However, following the guideline provided in Section 4.3, the highlighted path Figure 4a is eligible for a reduction:

$$\Pi^{L_{P1}, L_{P2}}_{\texttt{prepare}} \xrightarrow{\texttt{2PC-vote}} \pi_{\texttt{decide}} \xrightarrow{\texttt{2PC-decide}} \Pi^{U, L_{P1}, L_{P2}}_{\texttt{learn}}$$



(a) Before Reduction 1

(b) After Reduction 1

Fig. 4. Transducer network of 2PC&Paxos before-and-after Reduction 1 at macroscopic level. At this level, the transducer network of 2PC&Paxos before reduction is similar to the one in Figure 3, except the log functions in Figure 3 are replaced by paxos-log functions. We use ↻ to denote transitions containing 1 RTT of paxos-log.

- Initially, C1 is violated because the output transition function of $\pi_{\texttt{decide}}$ is $\texttt{decide}(\{\texttt{timeout}, \texttt{vote}_{P1}, \texttt{vote}_{P2}\})$, requiring `timeout` as an input. Nonetheless, our guideline in Section 4.3 suggests that if the participants are highly available, we can remove the `timeout` in `decide` and then C1 is satisfied.
- As we are now inspecting the transducer network G directly, we examine the sufficient condition C2'. Since 2PC&Paxos provides liveness with a liveness-desirable state[2] at $\Pi_{\texttt{learn}}$, and $\pi_{\texttt{decide}}$ is present in every path from $\Pi_{\texttt{prepare}}$ to $\Pi_{\texttt{learn}}$, we confirm that C2' is satisfied, which implies C2.

Since both conditions are satisfied, the reduction can be applied by implementing a message adapter (Step 2; Section 4.2). With that, the message path is reduced to:

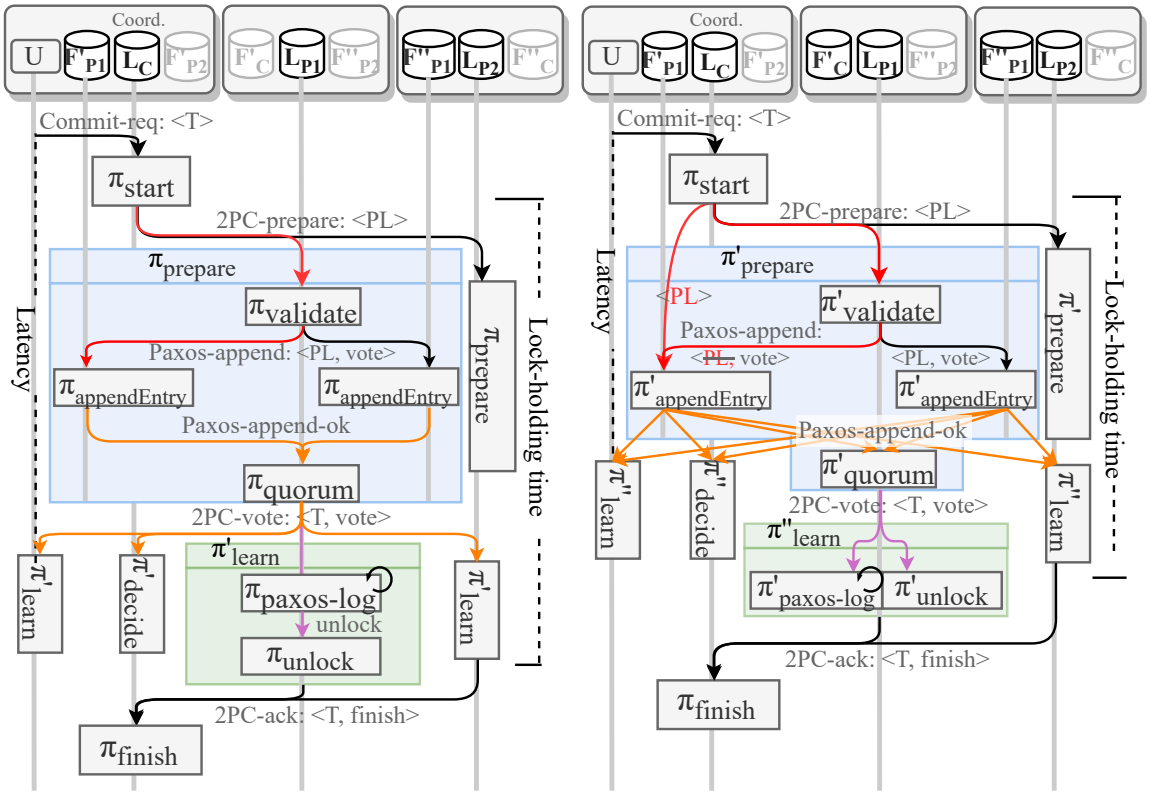
$$\Pi^{L_{P1}, L_{P2}}_{\texttt{prepare}} \xrightarrow{\texttt{2PC-vote}} \pi'_{\texttt{decide}} \cup \Pi'^{U, L_{P1}, L_{P2}}_{\texttt{learn}}$$

Figure 4b shows the reduced message path after applying Reduction 1 (without $\pi_{\texttt{proxy}}$ for brevity). Now the commit/abort decision learning step is *decentralized* — the client learns the `2PC-votes` directly from the participants, allowing the coordinator L_C leaves the critical path.

Reduction 1 essentially optimizes both the latency and system throughput (via reducing the lock-holding time). First, it reduces user-perceived latency from 3 to 2 RTTs because the client can learn the 2PC decisions straight from the participants, without waiting for the coordinator (which used to spend 1 RTT to log the decision using Paxos). Second, it reduces the lock-holding time from 4.5 to 3 RTTs. It is because now the participants can learn each other's 2PC votes directly, without waiting for the coordinator to 1 OWTT to notify them. Therefore, in addition to the 1 RTT saved in the aforementioned discussion, it can also save an additional 1 OWTT in the asynchronous phase, i.e., a saving of 1.5 RTT on the lock-holding time in total.

Beyond performance optimization, Reduction 1 can be considered anecdotal evidence of the soundness and practicality of our theory because the latency optimization of Reduction 1 is exactly the optimization reported in PaxosCommit, CockroachDB, PolarDB-X, and OceanBase! Interestingly enough, even though we base our application of the reduction rule on the same observation as those systems (i.e., the timeout is unnecessary), CockroachDB, PolarDB-X, and OceanBase failed to recognize the lock-holding time reduction opportunity and continued to have their participants wait for and learn the decision from the coordinator. This underscores the significance of developing a theory like ours, which can steer the industry toward complete solutions instead of relying on piecemeal optimizations.

[2]As mentioned in Section 3.1, $\pi_{\texttt{finish}}$ is only a state for clean-up, an asynchronous operation unrelated to liveness.



(a) After Reduction 1 ($\pi_{\texttt{prepare}}$ on L_{P1} at microscopic level)

(b) Applying all reductions: Occam

Fig. 5. Reductions 2 to 4 (for brevity, definitions of transition functions and details in P_{L2} is omitted)

4.4.2 *Reduction 2: Sending 2PC-prepare payload via LAN.*

Next, we enumerate all 3-hop message paths at the microscopic level. Consider the red message path $\mathcal{P}_1$ in Figure 5a:

$$\pi_{\texttt{start}} \xrightarrow{\texttt{2PC-prepare:}\langle PL\rangle} \pi_{\texttt{validate}} \xrightarrow{\texttt{Paxos-append:}\langle PL,\texttt{vote}\rangle} \pi^{F'_{P1}}_{\texttt{appendEntry}}$$

The figure shows the transducer network after Reduction 1 with the transition $\pi_{\texttt{prepare}}$ of $P1$ presented at a microscopic level, where the transitions inside the leader (L_{P1}) and follower replicas (F'_{P1} and F''_{P1}) of participant $P1$ become visible. When L_{P1} receives the 2PC-prepare message with a payload $PL = \langle T, \texttt{meta}, \texttt{write-set}\rangle$, it internally executes three transitions in sequence: (1) it validates the incoming write-set and acquires locks for them (i.e., $\pi_{\texttt{validate}}$). (2) If the validation is successful, it logs the validation result (i.e., $\texttt{vote} = \texttt{yes}$) using Paxos via appending the log entry to the followers' logs (i.e., $\pi_{\texttt{appendEntry}}$) and (3) waiting for a majority of them to reply (i.e., $\pi_{\texttt{quorum}}$).

Initially, we cannot apply the reduction rule to $\mathcal{P}_1$ because $\pi_{\texttt{validate}}$ is obviously state-dependent and violates C1 — it requires validating the incoming write-set against L_{P1}'s state. However, following the guideline in Section 4.3, we can rewrite $\mathcal{P}_1$ equivalently as $\mathcal{P}_2$:

$$\pi_{\texttt{start}} \xrightarrow{\texttt{2PC-prepare:}\langle PL\rangle} \begin{matrix} \pi'_{\texttt{validate}} \xrightarrow{\texttt{Paxos-append:}\langle \cancel{PL},\texttt{vote}\rangle} \pi^{F'_{P1}}_{\texttt{appendEntry}} \\ \pi_{\mathbb{I}} \xrightarrow[\langle PL\rangle]{} \end{matrix}$$

where $\pi_{\mathbb{I}}$ is an identity transition that simply relays its input to output. While the upper path is state-dependent and cannot be reduced, the lower path is state-independent and can be optimized because $\pi_{\mathbb{I}}$ trivially satisfies both conditions C1 and C2. With that, $\mathcal{P}_2$ is reduced into $\mathcal{P}'_2$:

$$\pi_{\texttt{start}} \xrightarrow[\langle PL\rangle]{\texttt{2PC-prepare:}\langle PL\rangle \quad \pi'_{\texttt{validate}} \quad \texttt{Paxos-append:}\langle \texttt{vote}\rangle} \pi'^{F'_{P1}}_{\texttt{appendEntry}}$$

The resulting message paths are as highlighted in red in Figure 5b. For $\pi'_{\texttt{validate}}$ on L_{P1}, notice that it is rewritten so that it no longer outputs the large payload PL to F'_{P1}. Instead, the reduction enables a *shortcut* that $\pi'_{\texttt{appendEntry}}$ on F'_{P1} can directly receive the large payload PL from L_C. Since L_C and F'_{P1} actually come from the same data center, the use of the reduced path $\mathcal{P}'_2$ would reduce certain WAN bandwidth consumption originally in $\mathcal{P}_2$ into the local ones in $\mathcal{P}'_2$.

Overall, Reduction 2 reduces the WAN bandwidth consumption, which could increase the system throughput when the contention is low. As we can apply Reduction 2 to participant $P2$ as well, the total bandwidth consumption would reduce from MNP to $M(N-1)P$.

4.4.3 *Reduction 3: Taking leader's $\pi_{\texttt{quorum}}$ off the critical path.*

Continuing to examine more 3-hop message paths could next identify the orange path in Figure 5a, which directly satisfies the reduction conditions given that Presumed-abort [78] is adopted in most practical systems:

$$\Pi^{F'_{P1},F''_{P1}}_{\texttt{appendEntry}} \xrightarrow{\texttt{Paxos-append-ok}} \pi^{L_{P1}}_{\texttt{quorum}} \xrightarrow{\texttt{2PC-vote}} \Pi'^{U,L_{P2}}_{\texttt{learn}} \cup \pi'^{L_C}_{\texttt{decide}}$$

- C1 is satisfied because $\pi_{\texttt{quorum}}$'s only waits for paxos-append-ok messages and ouputs $\langle T, \texttt{vote=yes}\rangle$. That is because, if the validation failed in $\pi_{\texttt{validate}}$, with presumed-abort, 2PC would *not* log that event. In other words, when $\pi_{\texttt{quorum}}$ happens, it means logging is happening and therefore the validation $\pi_{\texttt{validate}}$ must have outputted `vote=yes`.
- C2 is also satisfied via C2'. After the previous reductions, the liveness-desirable state is now at $\Pi'_{\texttt{learn}}$. Since $\pi_{\texttt{quorum}}$ exists in every path from $\Pi_{\texttt{appendEntry}}$ to $\Pi'_{\texttt{learn}}$, C2' is satisfied.

The orange paths in Figure 5b illustrate the resulting message path after Reduction 3. With Reduction 3, the decision learning step is further decentralized, allowing the client to directly learn `Paxos-append-ok` from followers. Moreover, $\pi_{\texttt{quorum}}$ is now out of the critical path, further reducing the user-perceived latency.

4.4.4 *Reduction 4: Parallelizing $\pi_{\texttt{paxos-log}}$ and $\pi_{\texttt{unlock}}$.*

The last eligible 3-hop message path could be reduced is the purple one in Figure 5a:

$$\pi'_{\texttt{quorum}} \xrightarrow{\texttt{2PC-vote}} \pi_{\texttt{paxos-log}} \xrightarrow{\texttt{unlock}} \pi_{\texttt{unlock}}$$

Similar to Reduction 3, this reduction can also be directly applied:

- C1: the output transition of $\pi_{\texttt{paxos-log}}$ is a constant (i.e., only waits for input message) because it always generates an unlock message as a result.
- C2: the liveness-desirable state is $\pi_{\texttt{unlock}} \in \pi'_{\texttt{learn}}$. As $\pi_{\texttt{paxos-log}}$ is present in every path from $\pi'_{\texttt{quorum}}$ to $\pi_{\texttt{unlock}}$, C2' (and thus C2) is satisfied.

The reduced path is shown in purple in Figure 5b, *parallelizing* $\pi_{\texttt{paxos-log}}$ and $\pi_{\texttt{unlock}}$, which helps to reduce the lock-holding time. Intuitively it is made possible largely thanks to the fact that the 2PC decision has been made durable on a quorum of replicas during the prepare phase of 2PC&Paxos protocol.

4.4.5 *Occam and its properties.*

We name the reduced protocol in Figure 5b, Occam. Since Occam is reduced from 2PC&Paxos, it is non-blocking and inherits consistency guarantees from Paxos and atomicity from 2PC. Occam can tolerate f failure with $2f+1$ replicas. Its recovery path is the same as 2PC&Paxos as no reduction is applied there. When combined with a serializable concurrency control (CC) algorithm, Occam can provide 1-copy serializability, or external consistency if using the same CC as Spanner.

- **User-perceived latency**: Referring to Figure 5b and focus on partition $P1$ (because the other partitions run in parallel), the critical path of committing a transaction T in Occam now only involves the following steps: (1) L_C sends 2PC-prepare to the leader L_{P1} (1 OWTT). (2) The leader L_{P1} validates T and log via `Paxos-append` to its followers F'_{P1} and F''_{P1} (1 OWTT). (3) The follower F'_{P1}, who co-located with the client U, sends `Paxos-append-ok` to the client (no WAN RTT because messages are sent via LAN). For $\pi''_{\texttt{learn}}$ at the client U, it is sufficient to wait for only one closest follower (i.e., F'_{P1}) for the common 3-way replication setup ($f=1$) because the single vote received actually has stopped by once at the leader (the message sender) and once at the follower, making the same vote already endorsed by a quorum of $f+1=2$. Altogether that

explains the 1 RTT user latency when $f = 1$. For $f > 1$, the client needs an additional 1 OWTT for receiving `Paxos-append-ok` from remote replicas.

- **Lock-holding time**: In addition to steps (1) and (2) in the critical path (2 OWTT), L_{P1} needs to spend another 1 OWTT for receiving `Paxos-append-ok` from a remote replica to trigger $\pi'_{\texttt{quorum}}$ and $\pi'_{\texttt{unlock}}$. Hence, the lock-holding time is 1.5 RTTs.
- **Bandwidth consumption**: With Reduction 2, we reduce the bandwidth consumption from MNP to $M(N-1)P$ in the setting with M-way replication of N partitions. Note that Reduction 1 has an overhead that increases the number of `2PC-vote` messages from M to $M(M+1)$, as each partition leader needs to send its vote to the other $(M-1)$ partitions, the coordinator, and the client after the reduction. However, the size of a `2PC-vote` message (e.g., 32 B) is negligible in comparison to the size of a 2PC-prepare message P (up to 7KB in TPC-C). Reduction 3 would also increase the number of `Paxos-append-ok` messages when a transaction can commit, but that is also negligible in practice.
- **Tail Latency.** Occam has the same $f+1$ quorum as 2PC&Paxos. Furthermore, Reductions 1 and 3 further reduce its latency variance because in the decision learning step, the client now only experiences variance from replicas. In contrast, clients in 2PC&Paxos perceive variance from replicas, leaders, and the coordinator.

4.5 Applying to TAPIR and GPAC

In TAPIR's fast-path, the coordinator (1) initiates 2PC by sending 2PC-prepare messages to all replicas, (2) collects 2PC-votes from them, and (3) makes a commit/abort decision, (4) sends the 2PC-decide messages to the replicas, and finally the replicas (5) release the locks. The path from steps (2) to (4) is: replicas $\xrightarrow{\texttt{2PC-vote}}$ coordinator $\xrightarrow{\texttt{2PC-decide}}$ replicas, which is qualified for applying the reduction rule because the coordinator generates the 2PC-decision solely based on the input `2PC-vote`. Applying our reduction theory can decentralize and offload the decision process to the replicas, reducing the lock-holding time by 0.5 RTT. GPAC has a similar path. Applying our theory to it can result in a 0.5 RTT reduction in lock-holding time as well.

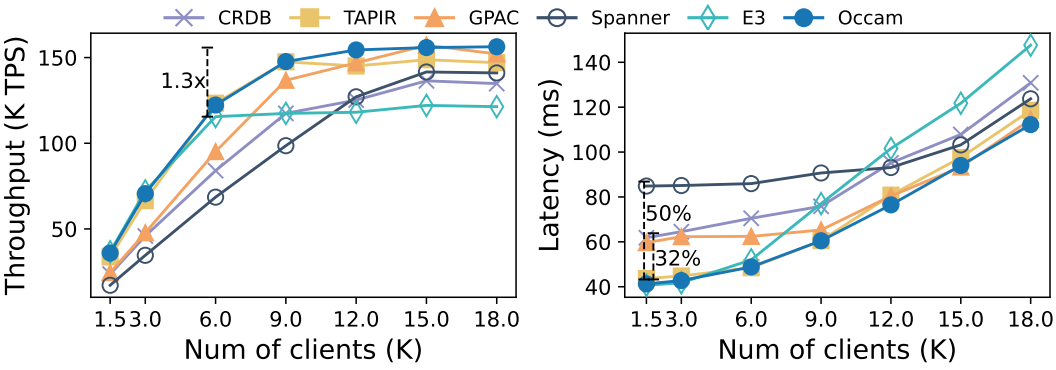
5 EVALUATION

We evaluate Occam against non-blocking protocols that support general workloads. Based on Section 2, CockroachDB (CRDB), PolarDB-X, and Oceanbase have the same properties and we choose CRDB as the representative. MDCC and TAPIR are similar and we choose the latter because it has more optimizations [118]. Since E3 is based on PaxosCommit and offers improvements over it, we choose E3 as a representative. Consequently, we compare Occam with: 2PC&Paxos (Spanner), CRDB, TAPIR, GPAC, and E3. For fairness, all protocols are implemented using DBx1000 framework [116] with the same OCC implementation based on [9]. We implemented failover for Occam and Spanner but not for others. This actually favors the others by excluding the overhead for potential failure handling. We used the RPC library in DBx1000 for communication, optimized with batching, lock-free buffers [61], and epoll.

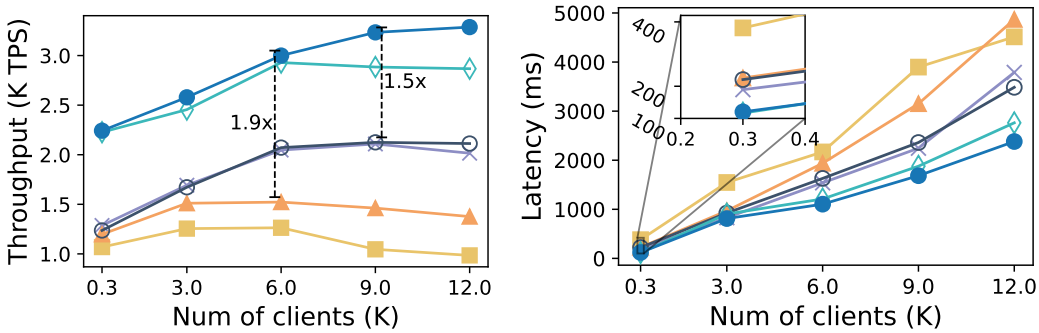
We use both YCSB [28] and TPC-C [1] workloads in our experiments. For both workloads, each client issues one transaction and waits for its commit before issuing the next. Aborted transactions are automatically retried using the exponential backoff strategy [49]. For YCSB, we use 10 billion tuples per partition. Each transaction accesses 10 keys, with a 5:5 mix of read and read-modify-write operations. Keys are selected using a Zipf distribution, with a parameter θ controlling skewness. We set $\theta = 0$ as *low contention*, and for stress testing set $\theta = 0.8$ as *high contention* [38, 54, 115]. Following [117], 10% of the accesses are uniformly selected to be remote by default.

For TPC-C, we follow [72, 73, 101] to focus on `NEW-ORDER` and `PAYMENT` transactions, as they comprise 88% of TPC-C workload and are most crucial to demonstrate performance.

All experiments are conducted on Alibaba Cloud ecs.r5.4xlarge instances. Each instance has 16 vCPUs (Intel Xeon Platinum 8269CY), 128GB DRAM, 10Gbps LAN for intra-region connections, and 1Gbps WAN for cross-region connections. By default, we use a 3-way-replicated-partitioning setup on three regions (each region hosts a partition leader and two followers of other partitions). In this setup, each pair of regions has a round trip latency of around 30ms. We report latency and throughput as the end-to-end latency from issuing



(a) Low contention



(b) High contention

Fig. 6. Throughput and Latency

the transaction until it commits and as the commit rate (i.e., goodput), respectively. All results are averages from five runs.

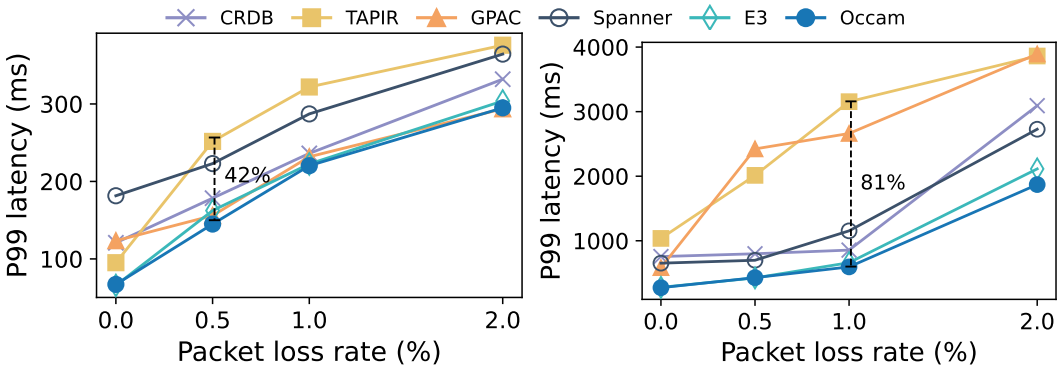
5.1 Throughput and Latency

Figure 6 shows the YCSB throughput and average latency of all evaluated protocols under the default setup. Occam is the only system that consistently works best under different contentions and system loads (numbers of clients).

Under low contention (Figure 6a), the peak throughput is primarily determined by the WAN bandwidth. That explains why Occam, GPAC, and TAPIR, whose bandwidth consumption is $M(N-1)P$, have better peak throughput than CRDB and Spanner, whose bandwidth consumption is MNP. E3 has the lowest peak throughput as it consumes more than MNP bandwidth (Section 2.3). When the number of clients has increased to around 9K, more protocols start to be (WAN)-bandwidth-bound because our profiling shows that their WAN bandwidth utilization has reached 100% while their CPU utilization is around 30%.

In terms of latency under low contention, Occam has the best latency because it can commit in 1 RTT with low bandwidth consumption. TAPIR can come close because its fast-path also uses 1 RTT although sometimes it requires 2 (slow-path). Before bandwidth-bound, Occam's latency is around 50% better than 2PC&Paxos (3 RTTs) and 32% lower than CRDB and GPAC (2 RTTs). The latency speedup is smaller than the RTT ratio because around 35% of transactions are local in this setting. After bandwidth-bound, the queuing time for the WAN network contributes to the latency increase.

Under high contention (Figure 6b), the throughput and latency of the protocols would degrade because more transactions would abort. When the number of clients has increased to around 3K, we found that the abort rate of some protocols has reached over 83% with 70% WAN bandwidth and 60% CPU utilizations, i.e., not fully utilized. That confirms that the lock-holding time is one major throughput bottleneck under high contention [54]. Yet, with the lowest 1.5 RTT lock-holding time, Occam shows the best throughput. E3's throughput comes close to Occam when the number of clients is below 6K because it has the same commit latency as Occam (hence, the same 1.5 RTT lock-holding time). However, since E3 requires a client to receive messages from replicas for all transactions regardless of their status (committed or aborted), the client may receive an overwhelming number of messages, especially during periods of high contention when there are



(a) Low contention

(b) High contention

Fig. 7. P99 Tail Latency

numerous aborted transactions. Our profiling shows that when the number of clients is more than 6K, the message queue of an E3 client becomes saturated and congested with messages, despite we have already used a highly efficient lock-free buffer [61] as the message queue. Occam does not have this issue since a client receives `Paxos-append-ok` messages from replicas only for the *committed* transactions. Although TAPIR and GPAC have better lock-holding time (1.5–2.5 RTTs) than 2PC&Paxos and CRDB (3.5–4.5 RRTs), their throughputs are worst and get bound earlier than the others because some replicas hold locks for a transaction that will eventually abort (Sec 2.2). In terms of latency, Occam also shows significant advantages over Spanner, CoackroachDB, GPAC, and TAPIR because of its 1 RTT commit latency and 1.5 RTT lock-holding time.

5.2 Tail Latency

Figure 7 shows the p99 tail latency of all protocols under different network conditions. Specifically, we ensure the system is not saturated by using only 100 clients but manually dropping 0–2% packets, which can occur in congested data-centers [47]. From the figure, we can see that Occam has the best p99 tail latency under all conditions. TAPIR has a poor tail latency as expected because it requires a larger quorum size that is especially sensitive to the network condition.

5.3 Ablation Study

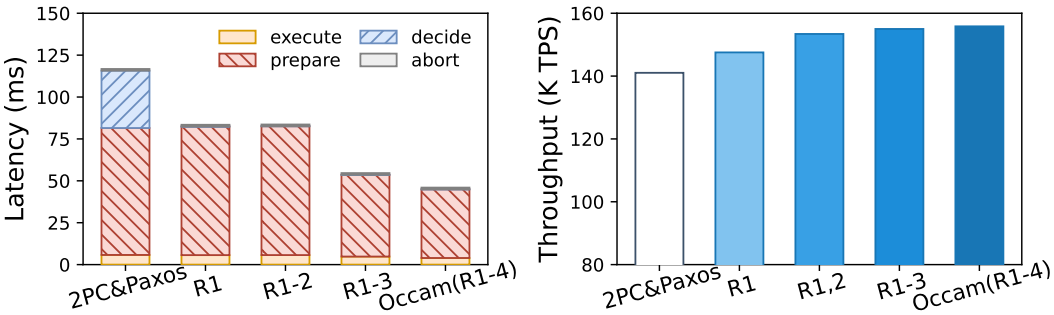
To understand the effectiveness of each reduction, we conduct an ablation study that incrementally applies reductions to 2PC&Paxos until we obtain Occam. Figure 8 (left) shows the latency breakdown of cross-partition transactions when the system is not saturated with clients. Figure 8 (right) shows the peak throughput when the system is saturated.

When breaking down the latency, execute refers to the time of the execution phase of a successful transaction, which is small because reads in OCC are done locally [118] (reading a local replica that is not a leader is still fine because stale reads will be validated during the commit phase) and writes are buffered locally as well and only applied during commit. prepare refers to the time of a successful transaction starting from the coordinator sending the 2PC-prepare message till it can make the 2PC-decision. All the remaining time in the critical path is counted as decide. Any time spent on aborted transactions including the execution time and the back-off wait-time is counted towards abort.

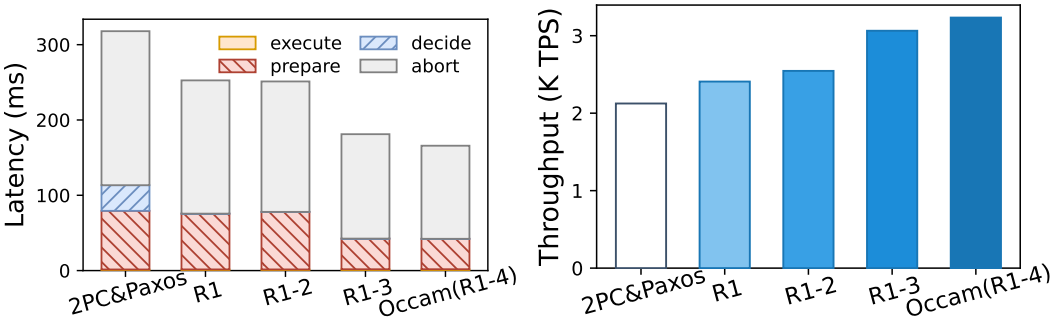
Figure 8 shows that each reduction can individually help in reducing the latency and throughput, except for Reduction 2 which was uncovered for improving the throughput only. Although more transactions are aborted under high contention (which is normal), it does not reduce the effectiveness of our reductions.

5.4 Impact of Deployment Setup

In this experiment, we evaluate the performance using different deployment setups. `3R-3P(equal dist.)` has been our default 3-way replication with 3 partitions across 3 equally distant regions (~30ms round trip latency). `3R-3P(unequal dist.)` uses the same setup but across regions with unequal distances (15–30ms

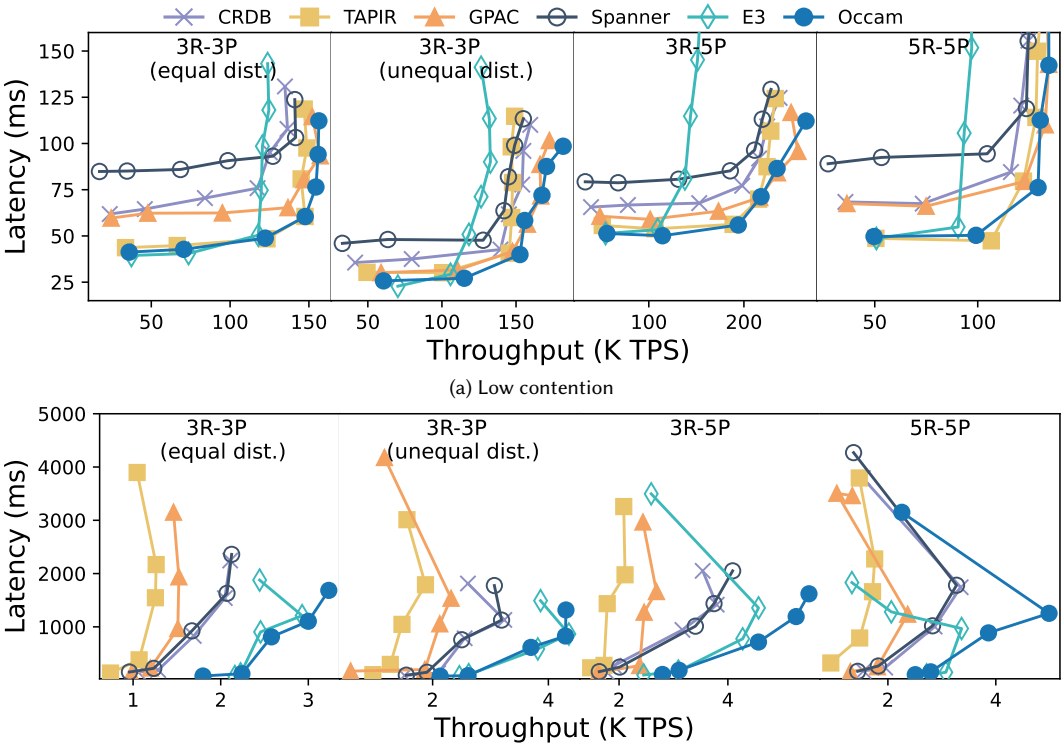


(a) Low contention



(b) High contention

Fig. 8. Ablation Study



(a) Low contention

(b) High contention

Fig. 9. Impact of different deployment setups

round trip latency). 3R−5P has 3-way replication of 5 partitions across 5 regions, with partition leaders in separate regions and two followers in the closest regions. 5R−5P has 5-way replication of 5 partitions across 5 regions, with each region hosting a partition leader and one follower for each of the other partitions. Figure 9 presents the latency and throughput by controlling the number of clients. Areas closer to the bottom-right

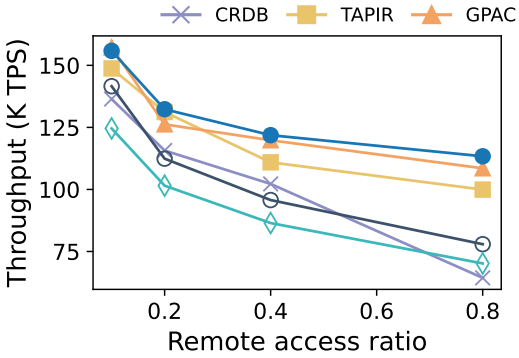


Fig. 11. Impact of remote access ratio

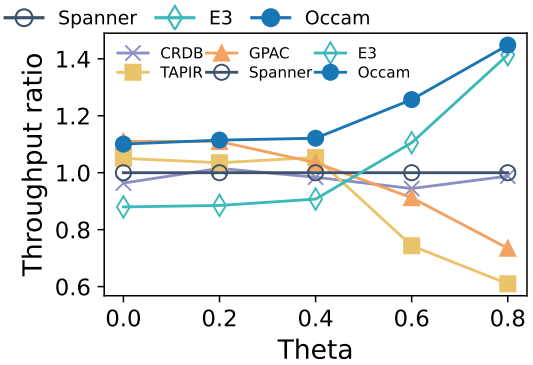


Fig. 12. Vary θ (throughput ratio to Spanner)

corner indicate better performance in both latency and throughput dimensions. As shown in the figure, Occam achieves the best performance in nearly all deployment setups, with a particularly significant margin under high contention. Under high contention, adding more clients would eventually hurt both the throughput and latency due to higher contention between clients which leads to more aborts. Under low contention, adding more clients beyond the system limit would hurt the latency only because the clients would just line up to wait for the WAN bandwidth without contending with each other like in the high contention case. In `5R-5P`, Occam's user-perceived latency is 1.5 RTT due to $f = 2$ under 5-way replication. While this places Occam slightly behind TAPIR in one case of our experiment, Occam remains dominant in all other cases.

5.5 Impact of Workload Parameters and TPC-C

We study the impact of different workloads by (1) varying the remote access ratio in YCSB, (2) varying the contention (θ) in YCSB, and (3) changing the workload to TPC-C. Figure 11 shows the peak throughput under various remote access ratios with θ fixed to 0. As the number of remote accesses increases, Occam's advantage over CRDB and Spanner grows larger. This is because more remote accesses lead to larger 2PC-prepare payloads P (due to larger write-sets), allowing Occam to save more bandwidth. Figure 12 shows the peak throughput ratios over Spanner under different θ values, with the remote access ratio fixed at 0.1. Occam's improvement becomes larger under higher contention, as the benefit of a shorter lock-holding time becomes increasingly prominent under those situations. Figure 13 shows the peak throughput of TPC-C using different numbers of warehouses. Occam has better throughput than the others in all cases because it has a short lock-holding time. More warehouses reduce contention, favoring bandwidth-efficient protocols like Occam and widening the performance gap with Spanner.

5.6 Reduction on TAPIR and GPAC

In this experiment, we study the effectiveness of applying the theory to TAPIR and GPAC. As our reduction primarily optimizes the lock-holding time of GPAC and TAPIR, we only assess the effectiveness under high contention workloads. Figure 14 shows the results, with `TAPIR-R` and `GPAC-R` representing the optimized versions. The results confirm that our reduction can further optimize TAPIR and GPAC by 1.06–1.22x, even though they are already well-optimized.

6 RELATED WORK

State-machine replication (SMR) protocols provide high availability and fault tolerance for a system. Starting from (classic) Paxos [57], numerous variants of Paxos have been developed (e.g., [58, 59, 84]), which improve Paxos in terms of latency and clarity. Recent SMR protocols are optimized for the WAN settings (e.g., [10, 15, 79]) or rely on specialized software/hardware (e.g., [64, 102, 109, 110]).

Atomic commit (AC) protocols guarantee the A (atomicity) and D (durability) of an ACID-compliance transaction when the data it updated are partitioned onto different failure domains (i.e., the partitions could fail independently). Starting from 2PC [77], which is blocking under coordinator failures, various non-blocking atomic commit protocols (e.g., [17, 43, 46, 95]) have been proposed, but they lead to higher latency or lower throughput. Recent studies on atomic commit protocols focus on optimizing for specific system architecture

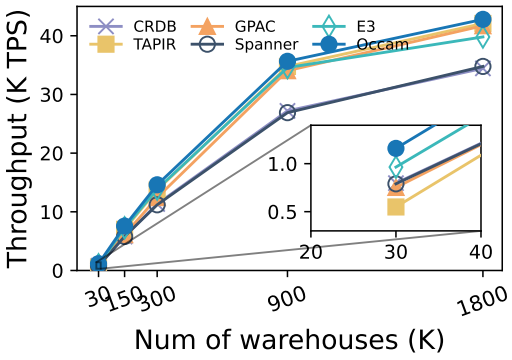


Fig. 13. TPC-C result

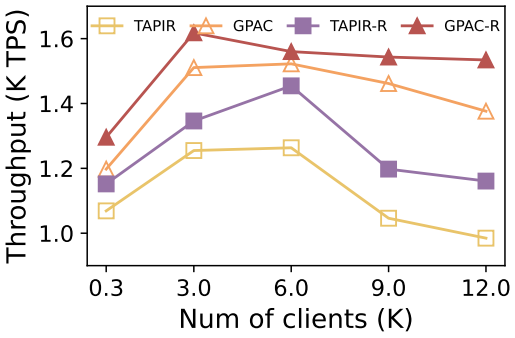


Fig. 14. Effectivness of reduction on GPAC and TAPIR

(e.g., disaggregated storage in the cloud [44, 104, 121]). In any case, our reduction theory could be applied to any of the protocols above (and their replicated-partitioned mix).

Modern geo-distributed databases require both SMR and AC protocols. When weaker replication consistency is acceptable, one could trade consistency for lower latency (e.g., [14, 20, 24, 27, 32, 55, 69, 70, 96, 100, 106, 107]). Occam targets for strong consistency. When data is not distributed across multiple geographic locations, it is possible to trade latency for throughput (e.g., through batching [35, 54, 72, 73, 123]) because the difference in microseconds [37] is imperceptible to users. However, in a geo-distributed setting, latency cannot be compromised as the latency of WAN is typically measured in milliseconds. When focusing on a restrictive set of workloads, distributed protocols can be simplified to improve performance. Beyond deterministic databases [38, 39, 41, 66, 67, 81, 83, 89, 91, 101, 108, 108, 114, 114], some other works pose various restrictions on the workload [30, 31, 40, 42, 65, 80]. They share the same generality problem that Occam does not have. Some recent work aims to remove generality problems in deterministic databases including Aria [72] and Primo [54]. Nonetheless, they belong to the category that trades latency for throughput, unsuitable for geo-distributed settings. Protocols can also be optimized with special hardware [36, 62–64, 92, 119], but we do not rely on this assumption. Another line of work focuses on the execution phase of distributed transactions [34, 50, 68, 75, 115, 117, 124] in order to reduce aborts and achieve a higher commit rate (i.e., throughput). We do not expect to use our theory there because operations in the execution phase are usually state-dependent. When a workload is perfectly partitionable, there are no cross-partition transactions and hence works like [97, 98] can achieve linear scalability without using 2PC. Some works [7, 8, 31, 52, 65] dynamically re-partition the database to reduce the number of 2PCs, although re-partitioning itself would also carry overheads. However, when a workload is not perfectly partitionable, 2PC is still the bottleneck in WAN. Beyond partitioning, full replication [7, 12, 13, 18, 71, 82, 86, 122] and shared-storage [2, 11, 22, 103] are alternative database architectures that favor read scalability. In this paper, we apply our theory to the geo-replicated-partitioning setup. Applying the theory to the other setups is our next step.

7 BEYOND GEO-DISTRIBUTED COMMIT

So far we have been focusing on replicated atomic commit protocols in our discussion. Our reduction theory in fact is general and can apply to any distributed protocol. To demonstrate, we use WormLog [93] as an example.

WormLog is an append-only log with replications. It supports two operations: `append` and `read`. Appending a log entry in WormLog involves (1) obtaining an ordered `seq_num` from a sequencer S, (2) reserving the log position `seq_num` on all replicas, and (3) replicating the log entry to the replicas.

Figure 15 shows the transducer network of WormLog.

- $\pi_{\texttt{seq}}$ is triggered when the sequencer S receives a request from the client. The output transition $\tau_{\texttt{seq}}$ determines the `seq_num` with its local counter in s_S. The state transition $\delta_{\texttt{seq}}$ increments the local counter.
- $\pi_{\texttt{reserve}}$ is triggered when the replica receives a reservation request from the sequencer. $\tau_{\texttt{reserve}}$ verifies if `seq_num` is occupied, generating an output message that consists of both the verification result and `seq_num`. $\delta_{\texttt{reserve}}$ updates `seq_num` as occupied in its state.

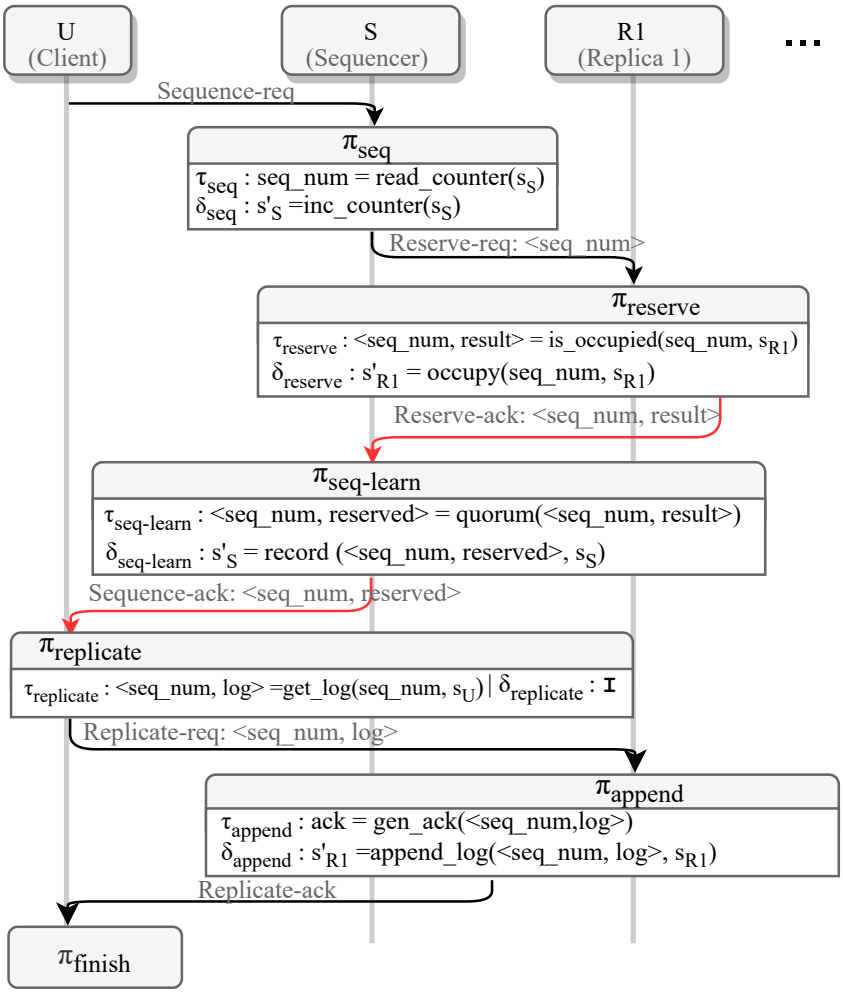


Fig. 15. The transducer network of WormLog (other replicas are omitted as they are the same as R1)

- $\pi_{\texttt{seq-learn}}$ is triggered when the sequencer receives reservation responses from the replicas. $\tau_{\texttt{seq-learn}}$ outputs `seq_num` with `reserved=true` if a quorum of replicas have reserved it. $\delta_{\texttt{seq-learn}}$ records the `seq_num` and `reserved` result in the sequencer's state.
- $\pi_{\texttt{replicate}}$ is triggered when the client receives `<seq_num,reserved=true>` from the sequencer. (The reservation failure path is omitted in the figure for brevity). Its $\tau_{\texttt{replicate}}$ outputs a message consisting of the `seq_num` and the log entry. Its state transition function $\delta_{\texttt{replicate}}$ is an identity function.
- $\pi_{\texttt{append}}$ is triggered on receiving the replication request from the client. $\tau_{\texttt{append}}$ generates an ack and $\delta_{\texttt{append}}$ appends the log entry at the slot `seq_num`.
- $\pi_{\texttt{finish}}$ at the client receives the ack's and records that the log entry has been appended to the shared log.

WormLog has proven its liveness that it eventually terminates at $\pi_{\texttt{finish}}$. Following the condition checking step as presented in Section 4.1, one can enumerate all 3-hop message paths in Figure 7 and identify that the highlighted path satisfies both conditions C1 and C2:

$$\pi_{\texttt{reserve}} \xrightarrow{\texttt{<seq\_num, result>}} \pi_{\texttt{seq-learn}} \xrightarrow{\texttt{<seq\_num, reserved>}} \pi_{\texttt{replicate}}$$

Specifically, C1 is satisfied because $\tau_{\texttt{seq-learn}}$ only requires messages from replicas. Since we are now inspecting G directly, we inspect the sufficient condition C2' (c.f. Section 4.1). Specifically, as WormLog provides liveness and $\pi_{\texttt{seq-learn}}$ is present in every message path from $\pi_{\texttt{reserve}}$ to $\pi_{\texttt{learn}}$, C2' is satisfied. C2' implies C2. Hence, the highlighted path is a weak dependency and a reduction can be applied:

$$\pi_{\texttt{reserve}} \xrightarrow{\texttt{<seq\_num, result>}} \pi'_{\texttt{seq-learn}} \cup \pi'_{\texttt{replicate}}$$

Experimental results show that the reduction above can reduce WormLog's latency by 25.7% under the 3-replica setting.

8 CONCLUSION

This paper presents a new theory that can optimize any distributed protocol by simply adding message adapters. The theory is short and sweet — with just a single reduction rule, the theory can upgrade the latency and throughput of Spanner, GPAC, and TAPIR without touching their original implementations.

REFERENCES

[1] 2010. TPC-C BENCHMARK Revision 5.11.

[2] 2023. *AlloyDB for PostgreSQL.* https://cloud.google.com/alloydb

[3] 2023. *TLC.* https://github.com/tlaplus/tlaplus/releases/tag/v1.7.3

[4] [Online]. *Technical Report and TLA+ Specification for Occam.* https://github.com/Occam-Project/Occam

[5] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. 1998. One-phase commit: does it make sense?. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250).* IEEE, 182–192.

[6] Maha Abdallah and Philippe Pucheral. 1998. A non-blocking single-phase commit protocol for rigorous participants. *Networking and information systems journal* 1, 2-3 (1998), 195–212.

[7] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 1381–1392.

[8] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[9] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record* 24, 2 (1995), 23–34.

[10] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

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

[12] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. (2011).

[13] 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.* 325–340.

[14] Valter Balegas, Nuno Preguiça, Sérgio Duarte, Carla Ferreira, and Rodrigo Rodrigues. 2018. IPA: Invariant-preserving applications for weakly-consistent replicated databases. *arXiv preprint arXiv:1802.08474* (2018).

[15] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08).*

[16] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. {RobinHood}: Tail Latency Aware Caching–Dynamic Reallocation from {Cache-Rich} to {Cache-Poor}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* 195–212.

[17] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems.* Vol. 370. Addison-wesley Reading.

[18] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *CIDR*, Vol. 11. 9–20.

[19] Romain Boichat and Rachid Guerraoui. 2005. Reliable and total order broadcast in the crash-recovery model. *J. Parallel and Distrib. Comput.* 65, 4 (2005), 397–413.

[20] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13).* 49–60.

[21] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 2859–2872.

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

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

[24] 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 Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[25] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data.* 235–247.

[26] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data.* 19–33.

[27] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008),

1277–1288.

[28] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[29] 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.

[30] James A Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX Annual Technical Conference*, Vol. 12.

[31] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*. 163–174.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[33] Diego Didona and Willy Zwaenepoel. 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 79–94.

[34] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. 2015. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 262–275.

[35] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943.

[36] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*. 54–70.

[37] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for {On-Disk} Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 343–360.

[38] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015).

[39] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[40] Hua Fan and Wojciech Golab. 2018. Scalable transaction processing using functors. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1004–1016.

[41] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[42] Hua Fan, Wojciech Golab, and Charles B Morrey III. 2017. ALOHA-KV: high performance read-only and write-only distributed transactions. In *Proceedings of the 2017 Symposium on Cloud Computing*. 561–572.

[43] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160.

[44] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A Bernstein. 2022. Cornus: atomic commit for a cloud DBMS with storage disaggregation. *Proceedings of the VLDB Endowment* 16, 2 (2022), 379–392.

[45] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160* (2020).

[46] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol.. In *EDBT*. 157–168.

[47] Pat Helland. 2022. Decoupled Transactions: Low Tail Latency Online Transactions Atop Jittery Servers. In *12th Conference on Innovative Data Systems Research*.

[48] Joseph M Hellerstein. 2010. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record* 39, 1 (2010), 5–19.

[49] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. (2020).

[50] Evan PC Jones, Daniel J Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 603–614.

[51] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[52] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 145–161.

[53] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 113–126.

[54] Ziliang Lai, Hua Fan, Wenchao Zhou, Zhanfeng Ma, Xiang Peng, Feifei Li, and Eric Lo. 2023. Knock Out 2PC with Practicality Intact: a High-performance and General Distributed Transaction Protocol. *arXiv preprint arXiv:2302.12517* (2023).

[55] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[56] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

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

[58] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[59] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19 (2006), 79–103.

[60] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1091–1104.

[61] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A cache/storage subsystem for modern hardware. In *Proceedings of the International Conference on Very Large Databases, VLDB 2013*.

[62] Junru Li, Youyou Lu, Yiming Zhang, Qing Wang, Zhuo Cheng, Keji Huang, and Jiwu Shu. 2022. SwitchTx: scalable in-network coordination for distributed transaction processing. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2881–2894.

[63] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 104–120.

[64] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*. 467–483.

[65] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.

[66] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. 2019. MgCrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment* 12, 5 (2019), 597–610.

[67] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1156–1168.

[68] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. 1999. Providing persistent objects in distributed systems. In *ECOOP'99—Object-Oriented Programming: 13th European Conference Lisbon, Portugal, June 14–18, 1999 Proceedings*. Springer, 230–257.

[69] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.

[70] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation*.

[71] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The {FuzzyLog}: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 357–372.

[72] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.

[73] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. (2021).

[74] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672.

[75] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment* 7, 5 (2014), 329–340.

[76] Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2019. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment* 12, 5 (2019), 611–623.

[77] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[78] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[79] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 358–372.

[80] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 479–494.

[81] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts.. In *OSDI*. 517–532.

[82] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2013. Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments. In *CIDR*.

[83] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[84] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[85] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[86] Stacy Patterson, Aaron J Elmore, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2012. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *arXiv preprint arXiv:1208.0270* (2012).

[87] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 325–341.

[88] George Pîrlea. 2021. *Errors found in distributed protocols*. https://github.com/dranov/protocol-bugs-list

[89] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[90] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 145–160.

[91] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[92] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 740–755.

[93] Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. Wormspace: a modular foundation for simple, verifiable distributed systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 299–311.

[94] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. 2013. F1: A distributed SQL database that scales. (2013).

[95] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 133–142.

[96] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 385–400.

[97] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2018. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 463–489.

[98] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[99] 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.

[100] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182.

[101] 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.

[102] Sarah Tollman, Seo Jin Park, and John K Ousterhout. 2021. EPaxos Revisited. In *NSDI*. 613–632.

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

[104] Ruihong Wang, Jianguo Wang, Stratos Idreos, M Tamer Özsu, and Walid G Aref. 2022. The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. *Proceedings of the VLDB Endowment* 16, 1 (2022), 15–22.

[105] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2203–2215.

[106] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. 2019. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2019), 344–358.

[107] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.

[108] Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, and Yu-Shan Lin. 2016. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1553–1565.

[109] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. 2022. Nezha: Exploiting Concurrency for Transaction Processing in DAG-based Blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 269–279.

[110] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: using network measurements to reduce state machine replication latency in wans. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 351–363.

[111] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[112] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing distributed transaction prioritization for high-contention workloads. In *Proceedings of the 2022 International Conference on Management of Data*. 715–729.

[113] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[114] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.

[115] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[116] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. (2014).

[117] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[118] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 263–278.

[119] 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.

[120] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 276–291.

[121] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.

[122] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, et al. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[123] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.

[124] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data*. 685–699.