Fast Commitment for Geo-Distributed Transactions via Decentralized Co-coordinators

Zihao Zhang
East China Normal University†
zihaozhang@stu.ecnu.edu.cn

Huiqi Hu*
East China Normal University†
hqhu@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University†
xzhou@dase.ecnu.edu.cn

Yaofeng Tu
ZTE Corporation
tu.yaofeng@zte.com.cn

Weining Qian
East China Normal University†
wnqian@dase.ecnu.edu.cn

Aoying Zhou
East China Normal University†
ayzhou@dase.ecnu.edu.cn

ABSTRACT

In a geo-distributed database, data shards and their respective replicas are deployed in distinct datacenters across multiple regions, enabling regional-level disaster recovery and the ability to serve global users locally. However, transaction processing in geo-distributed databases requires multiple cross-region communications, especially during the commit phase, which can significantly impact system performance.

To optimize the performance of geo-distributed transactions, we propose Decentralized Two-phase Commit (D2PC), a new transaction commit protocol aiming to minimize the negative impact of cross-region communication. In D2PC, we employ multiple co-coordinators that perform commit coordination in parallel. Each co-coordinator is responsible for collecting 2PC votes and making a `PreCommit` decision in its local region. This approach allows for the concurrent invocation of multiple cross-region network round trips, and each region can end its concurrency control locally before replication is complete, thus significantly reducing the chances of blocking and enhancing system concurrency. Moreover, we propose the bypass leader replication reply method, leveraging decentralized co-coordinators to bypass the leader for message transmission, thereby reducing the commit latency. Experimental results have demonstrated that D2PC can reduce commit latency by 43% and improve throughput by up to 2.43 × compared to the geo-distributed transaction processing methods based on 2PC.

PVLDB Reference Format:
Zihao Zhang, Huiqi Hu, Xuan Zhou, Yaofeng Tu, Weining Qian, and Aoying Zhou. Fast Commitment for Geo-Distributed Transactions via Decentralized Co-coordinators. PVLDB, 17(10): 2555 - 2567, 2024.
doi:10.14778/3675034.3675046

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZhangZihao270/D2PC.

*represents the corresponding author.
† Shanghai Engineering Research Center of Big Data Management.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675046

1 INTRODUCTION

Geo-Distributed databases have become a vital infrastructure for hosting cross-region applications, such as international banking, popular e-commerce platforms, social media, etc. Prominent examples of geo-distributed databases include Spanner [4] and CockroachDB [36]. These databases employ the strategy of partitioning data into shards and replicating them across datacenters globally.

The competitiveness of a geo-distributed database relies on its ability to handle geo-distributed transactions that span multiple regions. To ensure the atomicity of distributed transactions, it is necessary to employ an atomic commitment protocol like Two-phase Commit (2PC) to coordinate the commit phases. Additionally, to ensure high availability, a consensus protocol is used to replicate the transaction result across regional replicas.

A typical example is Spanner [4, 5], which is the layered architecture that employs the 2PC protocol over the Multi-Paxos [3] protocol. However, both commitment and consensus protocols involve multiple rounds of cross-region communication. In the geo-distributed deployment where servers span different regions, the latency is significant, lasting hundreds of microseconds. This extended latency can prevent critical applications from meeting their required service level agreements (SLA). It can also increase lock holding time and thus the chances of contention, posing a serious threat to overall performance. This often forces upper-layer applications to give up using geo-distributed transactions for more sustainable performance. Therefore, to make geo-distributed transactions practical, it is essential to minimize the negative impact of cross-region communication on transaction processing.

Several recent research works [10, 26, 30, 38–40] have proposed tightly integrating consensus protocols and commit protocols to minimize network round trips needed for transaction commitment. This integration aims to reduce latency by allowing all replicas to process transactions in a decentralized manner, in contrast to the layered architecture that relies on the leader for transaction processing. In non-conflict scenarios, these approaches typically require only a single round trip to commit a transaction, resulting in significantly reduced latency. However, in the presence of conflicts, the impact of cross-region communication on performance remains outstanding. On one hand, these approaches still require a minimum of two round trips to commit a conflicting transaction. On the other hand, multiple cross-region communications may still occur long concurrency control period, leading to high contention or abort rates. Meanwhile, involving all replicas in transaction processing is

fundamentally deviating from the architecture in existing databases that require a leader node for transaction processing.

In this work, we introduce Decentralized Two-phase Commit (D2PC), a novel commit protocol tailored for the common layered architecture in geo-distributed databases. The primary objective of D2PC is to mitigate the impact of cross-region communication on concurrency, thereby enhancing the overall throughput of geo-distributed transactions. Additionally, D2PC also works to minimize the effects of cross-region communication on transaction latency.

In the layered architecture, we have identified that the single coordinator frequently incurs cross-region communication during the commit phase. To address this, D2PC distributes the commit coordination duty among regions, thereby each server can learn the commit decision early from the local coordinator. Furthermore, D2PC decouples and parallelizes the processes of commit decision-making and replication, reducing the overall message delays included in the concurrency control period length. Through the decentralized coordinators, the replication reply can be relayed in a bypass leader manner, which reduces the commit latency.

In a nutshell, D2PC deploys a set of co-coordinators across all datacenters. Each co-coordinator works independently, collecting votes from participant shards and making a `PreCommit` decision in parallel with ongoing replication. This parallel processing allows the co-coordinator to quickly move to the `PreCommit` stage and promptly notifies the local participant leaders to end concurrency control, without waiting for replication completion. This action effectively reduces the concurrency control duration in the commit phase to 0.5 cross-region round trip. Besides, during the commit phase, each co-coordinator directly relays the replication reply from its co-located followers to the correspondent coordinator, bypassing communication with the shard leader. This reduces the commit latency to 1 - 1.5 cross-region round trips. Consequently, D2PC outperforms existing alternative methods in both concurrency control duration and latency.

The contributions of this paper can be summarized as follows:

- We observed that the single coordinator in the 2PC protocol can lead to additional cross-region communication during transaction commits. To eliminate it, we decentralized the coordinator into a group of co-coordinators distributed across every datacenter.
- Building upon co-coordinators, we proposed the decentralized transaction commitment which can minimize the impact of cross-region communication on concurrency and commit latency for geo-distributed transactions.
- We conducted extensive experiments in a multi-cloud scenario, using benchmarks of Retwis, TPC-C, and a micro-benchmark. We combined D2PC with OCC and 2PL. D2PC improves the throughput by up to 2.43 × than 2PC+OCC. Similarly, it demonstrates an improved throughput of 1.73 × compared to 2PC+2PL, and this improvement increases to 2.33 × when read optimization is enabled.

The rest of the paper is organized as follows. § 2 introduces the background and motivations of D2PC. § 3 sketches ideas and the architecture of D2PC. § 4 presents D2PC in greater detail, elaborating on its commit protocol and the techniques to shorten concurrency control period. § 5 reports the experimental results. Finally, we conclude the paper in § 6.

Table 1: Notations and notions in the paper. "✓" denotes exist in protocol, while "−" denotes not exist.

Role Symbol	Definition	D2PC	2PC
`CCoor`	the correspondent coordinator	✓	−
`Co-Coor`	the co-coordinator	✓	−
`Coor`	the coordinator	−	✓
S_n	the n-th shard	✓	✓
`L`	the leader replica of shard	✓	✓
`F`	the follower replica of shard	✓	✓
Interfaces for Client	**Description**		
Begin()	create a transaction	✓	✓
Read(t, k)	txn *t* read on *k*, returns *val* and *version*	✓	✓
Write(t, k, v)	update *k*, buffered in local	✓	✓
Commit(t, rset, wset)	commit *t*'s results to database	✓	✓
Data Store Functions	**Description**		
Prepare(t, rset, wset)	returns *vote(commit/abort)*	✓	✓
PreCommit(t)	end concurrency control early	✓	−
Commit(t) & Abort(t)	end transaction	✓	✓
Transaction State	**Description**		
`Executed`	finished execution	✓	✓
`Prepared`	`L` agree to commit	✓	✓
`PreCommit`	`Co-Coor` makes a pre-commit decision	✓	−
`Commit`	`CCoor` makes final commit decision	✓	✓
`Abort`	`CCoor` makes abort decision	✓	✓

2 BACKGROUND AND RELATED WORK

2.1 Two-phase Commit and the Blocking Issue

In distributed database management systems (DDBMS), data is partitioned into shards to achieve scalability. However, transactions that span multiple shards necessitate an atomic commitment protocol to ensure atomicity. This protocol ensures that all involved servers achieve a consensus on the commit decision.

Two-phase commit (2PC) is widely used in many DDBMSs to ensure atomicity. 2PC consists of two phases: the Prepare phase and the Commit phase. In the Prepare phase, participants send votes to the coordinator, who can proceed to the Commit phase only if all participants vote to commit. However, the 2PC protocol faces a blocking issue. As only one coordinator can make the final commit decision, if the coordinator fails before notifying the decision, participants may be blocked until the coordinator recovers. This block prevents the release of transaction resources, such as locks, impeding system progress.

Some solutions have been proposed to address the blocking issue. 3PC [33] introduces a new phase called the "prepared to commit" phase. In this phase, all participants must acknowledge the commit decision before actually committing. This ensures that all participants, not just the coordinator, are aware of the decision. E3PC [23] further enhances the availability of 3PC by introducing a quorum. Paxos commit [12] combines Paxos [27] with 2PC, using Paxos to replicate the commit decision to a set of replicas. Easy Commit [18] mandates participants to forward the coordinator's decision to all others before committing. Cornus [15] is a one-phase commit protocol designed for disaggregated-storage architecture with the assumption that the storage is high-available.

In summary, these protocols ensure fault-tolerant commit decisions to overcome the blocking issue. Following this principle, we deploy a set of co-coordinators to tolerate failures, simplifying the solution of the blocking issue without compromising transaction processing performance.

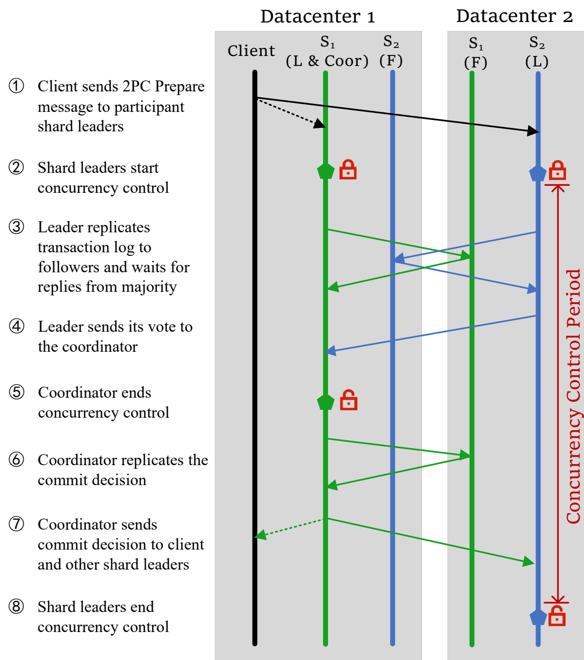


Figure 1: Example of committing a transaction using 2PC and Paxos. Solid and dashed arrows stand for inter-datacenter and intra-datacenter messages, respectively.

2.2 Transaction Commit in Geo-Distributed Databases

High availability is a crucial requirement for many applications, leading databases to adopt consensus protocols like Paxos [27] or Raft [31] for fault-tolerant replication of shards across multiple replicas. Consequently, in DDBMSs, the transaction protocols and consensus protocols need to work together. The collaboration between these two protocols can be summarized in two patterns.

Layered mode. In layered mode, transaction processing and replication are separated, and connected by the leader. Many commercial geo-distributed databases, including Spanner [4, 5], CockroachDB [36, 37], and TiDB [19], construct their transaction layer, encompassing the 2PC protocol and concurrency control protocol, atop leader-based consensus protocols. Specifically, the leader replica is responsible for processing transactions and replicating the results to other replicas by using the underlying consensus protocol. Then, 2PC is used to coordinate commit decisions among multiple shards. Both the 2PC and replication need coordination, the over-coordination introduces significant overhead.

An example in Fig. 1 demonstrates the performance issues caused by over-coordination. All the notations are listed and explained in Table 1. To simplify the explanation, we only consider two data shards, S_1 and S_2, each with two replicas located in different datacenters. In practice, there are typically many data shards, each with at least three replicas. Among participant shard leaders, a leader is selected as the coordinator (the leader of S_1). The process begins with the client sending a Prepare message to leaders (Process ①). The leader validates whether the transaction can be committed and then replicates the transaction log to the follower replicas (Process ③). After receiving replies from a majority of followers, the leader sends its vote to the coordinator (Process ④). The coordinator collects votes from all participant shards and decides whether to commit or abort. The commit decision is also replicated for fault tolerance (Process ⑥). Finally, the coordinator sends the commit decision to the client and other leaders (Process ⑦). Therefore, committing transactions incurs a total of 3 inter-datacenter round-trip times (RTTs).

Over-coordination also causes a long concurrency control period. The concurrency control period refers to the duration in which a transaction can affect others. Recall the example in Fig. 1, when the participant leader receives the Prepare message, it performs concurrency control like acquire locks (Process ②). Then the concurrency control period ends when the `Commit` message is made or received (Process ⑤ and ⑧). Consequently, the concurrency control period extends over a duration of 3 inter-datacenter RTTs on non-coordinator shards (e.g., S_2 in the example). This extended duration of exclusive resource access severely restricts the concurrency of the database system.

In practice, each shard's vote is replicated to a majority of replicas, ensuring that in the event of coordinator failure, the commit decision can be recovered by obtaining votes from the replicas of each participant. Therefore, replicating the commit decision in Process ⑥ is unnecessary, which reduces both the commit latency and concurrency control period length to 2 inter-datacenter RTTs.

The over-coordination prompts many efforts to optimize it. For instance, CockroachDB [37] introduces parallel commit to achieve single-round commit latency, but this assumes that the coordinator and shard leaders are co-located, which doesn't align with geo-distributed shard scenarios. Replicated Commit [29] conversely builds Paxos over 2PC, allows each datacenter to make a commit decision independently then uses Paxos to reach a consensus. This converts the coordination of 2PC within the datacenter, causing the cross-datacenter round trips reduced.

Integrated mode. Many works [10, 26, 30, 38–41] tightly combined the processes of transaction processing and consensus to reduce over-coordination. In these protocols, the 2PC Prepare message and replication message are combined and sent to all replicas, not the leader. Each replica will do concurrency control to process the transaction, and then try to reach a consensus about the commit decision on the coordinator. If consensus is reached, the transaction can be committed in one round of communication. Essentially, integrated mode converts the centralized transaction processing on the leader to decentralized transaction processing on all replicas, which eliminates a round of communication, but at the cost of inefficient conflict resolution.

When conflict occurs, the decisions made by each replica are difficult to make a consensus, causing extra costs to be incurred to resolve the inconsistency. For example, MDCC [26] and TAPIR [39] will abort conflicts, causing high abort rates. Carousel [38] and STARRY [40] need more communication to let a central node resolve conflicts, resulting in the latencies and concurrency control period lengths being at least 2 RTTs. Janus [30], which handles deterministic transactions, also relies on extra coordination to order transactions according to prior knowledge about read and write

sets. Compared to them, D2PC reduces the concurrency control period length, leading to fewer conflicts. Even if conflicts occur, D2PC relies on the leader to centrally resolve them, which avoids inconsistency. Therefore, D2PC is less impacted by conflicts than integrated mode approaches.

In summary, the optimization goal of integrated mode is to reduce the commit latency, but sacrifices the efficiency of conflict resolution. The fundamental design behind integrated mode diverges from the traditional architecture that relies on a single leader for transaction processing, necessitating significant customization of the transaction and storage layers. This customization poses a challenge for seamless integration into existing databases.

In contrast, our method D2PC can achieve the minimum concurrency control period length, and a comparable reduction in commit latency as integrated mode. Moreover, D2PC is based on the layered architecture, it relies on leaders to process transactions, which maintains compatibility with existing database infrastructures.

2.3 Timing of Write to be Visible

Databases usually delay the writes to be visible until the transaction is committed to ensure *recoverability*, which also prolongs the concurrency control period length. Consider a transaction T_1 write on data x, then transaction T_2 reads T_1's write on x, the system must ensures that T_2 is committed after T_1, thus preventing T_1 from aborting after T_2 is committed. A schedule is *recoverable* [2] if the commit order follows the read-after-write dependencies. Therefore, the concurrency control usually ends until the transaction is committed, to ensure a correct commit order.

Many works have investigated *early write visibility* in concurrency control protocol [7, 11, 16, 24, 25, 34], which enables transactions to read uncommitted writes, and then forces the commit order to follow the read-after-write dependencies to maintain recoverability [2, 17, 21, 32]. Most of them are designed for locking-based protocols. For example, ELR [8, 20, 24, 35] allows the transactions that have finished execution to release locks before logging. CLV [11] is proposed for the same goal and it re-designs the lock table to track and enforce the commit order of dependent transactions. Bamboo [14] explores violating 2PL by allowing lock release during execution to further enhance concurrency. There are also some works that employ early write visibility in non-locking protocols. PWV [9] is designed for deterministic databases and leverages the determinism that orders transactions before execution. This allows updates made by transactions to be visible before their execution is completed. Hekaton [6, 28] proposes a protocol for multi-version concurrency control that allows uncommitted dirty data to be read if it is in the "preparing" state. However, research on applying early write visibility for distributed transactions is sparse. DLV [13] investigated four timings to perform the lock violation in distributed databases and analyzed their performance and impact on transaction correctness.

Compared to these works, D2PC is designed for geo-distributed databases and offers a comprehensive solution to reduce both concurrency control period length and commit latency. D2PC is compatible with both 2PL and OCC. Most importantly, D2PC effectively minimizes the concurrency control period length without violating serializability.

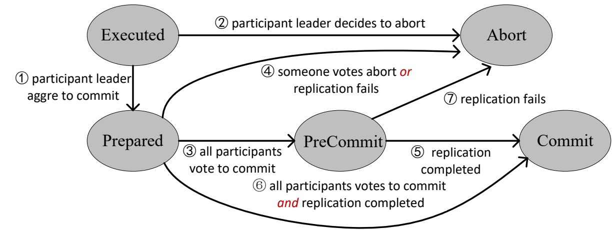


Figure 2: Transition Graph of Transaction State.

3 DESIGN OVERVIEW

In this section, we provide a concise overview of the design of D2PC, highlighting the rationale behind decentralized commit, the parallelization of processes, and the strategies employed to minimize concurrency control period length and reduce commit latency.

3.1 Notions in D2PC

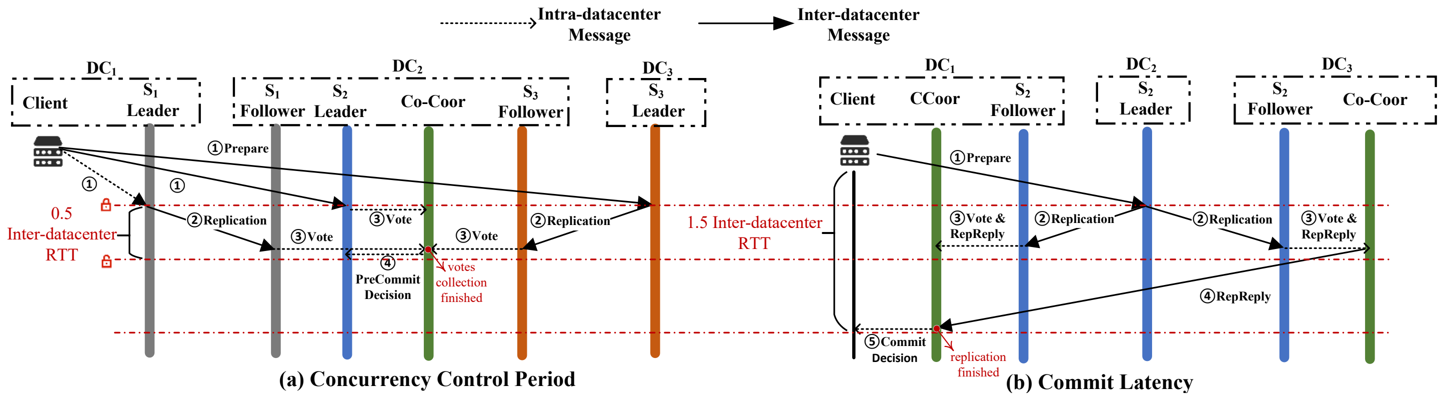
Before introducing our idea, we first detail notions listed in Table 1.

Server Setup. As the backend for global applications, the geo-distributed database is deployed in multiple datacenters across regions. Specifically, the database is partitioned into *data shards*, with $2\mathcal{F}+1$ replicas per shard, each replica located in a distinct datacenter. Among replicas of a shard, there is a leader to process transactions, and then replicate results to other follower replicas.

In D2PC, we separate the coordinator in 2PC into a group of *co-coordinators*, with each datacenter having one. Co-coordinators collect votes and make `PreCommit` decisions in a decentralized manner, which facilitates the early termination of concurrency control period for participant leaders. For each individual transaction, there is a *correspondent coordinator*, which is the co-coordinator located in the same datacenter as the client. The correspondent coordinator is responsible for making the final commit decision since it can ensure the replication is completed.

There are also multiple clients in each datacenters, which are application servers that execute transactions. They interact with databases through interfaces in Table 1. When starting a transaction, the client invokes *Begin()* to create a transaction object with a unique ID (*tid*). Then the client calls the *Read(t, k)* function to read data, either from the shard leader or the local replica (details in § 4.6), and buffers the results locally. For write operations, the client does not interact with the database, write results will be buffered locally and be committed to the database through the *Commit(t)* function after execution is finished.

Data Store. D2PC is agnostic to the underlying data store. In this work, we use key-value store as the backend. Each shard replica contains a key-value store for data storage and concurrency control. The data store provides several functions for transactions as shown in Table 1. On receiving the commit request from the client, the data store invokes *Prepare(t, rset, wset)* function to perform concurrency control and generate a vote, the vote and transaction log will be replicated to all datacenters. Once a shard leader receives the `PreCommit` decision, it invokes the *PreCommit(t)* function to end the concurrency control period. Upon receiving the final commit result, the shard leader will invoke either the *Commit(t)* or *Abort(t)* function to end the transaction accordingly.



(a) Concurrency Control Period (b) Commit Latency

Figure 3: Example of how to shorten the concurrency control period and reduce the commit latency in D2PC.

Transaction State. Fig. 2 illustrates the transitions of transaction states. The main difference of D2PC from 2PC is the `PreCommit` state. After the co-coordinator collects votes and makes a `PreCommit` decision, the transaction is set to `PreCommit` state. Then when the correspondent coordinator makes the final commit decision after it receives enough replication replies, it transitions the transaction to the `Commit` state. If some shard vote to abort or replication fails, the transaction will be set to `Abort` state.

3.2 Decentralized Commit and Processes Decoupling

The location of the coordinator significantly impacts the length of concurrency control period. When the coordinator is located in a different region than the participant leader, the communications between them are inter-datacenter, resulting in extended time for participant leaders to end the concurrency control period.

This issue primarily arises from the reliance on a single coordinator in the 2PC protocol, which makes the location of the coordinator critical. To address this, we propose a shift from a centralized to a decentralized commit coordination pattern. This involves deploying a set of *co-coordinators* across all datacenters. By doing so, each server can be co-located with a co-coordinator, enabling intra-datacenter communication between the participant leader and the co-coordinator. As a result, the waiting time for the commit decision can be reduced.

Another reason that causes long concurrency control period is the interleaving of the 2PC and replication processes. Specifically, two essential steps must be completed before committing a transaction: 2PC vote collection (receiving votes from all participants), and replication (where all participant leaders replicate the transaction log). As shown in Fig. 1, two steps are interleaved, with processes ① and ④ for 2PC, and process ③ for replication. The interleaving results in all message delays being included in the concurrency control period length.

However, we observe that the processes of 2PC and replication are independent of each other. Hence, we decouple and parallel their processes and illustrate them in Fig. 3. In this figure, (a) depicts the reduction in concurrency control period length, and (b) shows the optimization of commit latency (details in § 3.3).

The core idea of D2PC is that *if the 2PC vote collection occurs in parallel with the replication, the vote collection can be completed before replication is done via decentralized co-coordinators.* As shown in Fig. 3, the second dashed red line can be regarded as the completion of vote collection, and the third dashed red line represents the completion of replication. Therefore, we intend to *decouple and parallelize the processes of 2PC and replication, allowing each datacenter to collect votes and make a* `PreCommit` *decision early via the co-coordinator,* thereby shortening the length of concurrency control period.

To be specific, in Fig. 3(a), when a participant leader receives the `Prepare` message(message ①), it replicates the transaction log and its vote to followers (message ②). Each follower then forwards the vote to the co-located co-coordinator (message ③), e.g., the followers of S_1 and S_3 send the vote to the co-located co-coordinator in DC_2 upon receiving the replication message from their respective leaders. Upon the co-coordinator collects votes from the co-located replicas, it can make `PreCommit` decision before the replication is completed, and promptly notify the co-located participant leaders about the decision (message ④).

At this stage, the global serial order of transactions has already been determined, as all participant shards have agreed to commit. Therefore, ending concurrency control at this point not only excludes aborts caused by violating serializable, but also achieves a much shorter concurrency control period length. Considering that, participant leaders end concurrency control early.

As shown in Fig. 3(a), concurrency control period starts when leader receives the Prepare message (message ①), and ends when the co-coordinator receives all votes (message ③). Therefore, the concurrency control period length is only 0.5 inter-datacenter RTT.

3.3 Bypass Leader Replication Reply

As in leaderless replication methods [10, 26, 30, 38–40], they make the replication bypass leader to reduce commit latency. We also propose the bypass leader replication reply method by utilizing co-coordinators. In this approach, replication replies are directed to bypass the leader to the correspondent coordinator, which can reduce the commit latency.

As depicted in Fig. 3(b), after the leader replicates to followers, each follower notifies the co-located co-coordinator about the vote and includes the replication reply in the message (message ③). Subsequently, the co-located co-coordinator transfers the replication reply directly to the correspondent coordinator (message ④). In this way, the replication reply can bypass the leader, which reduces one inter-datacenter network delay for the correspondent coordinator to be aware of the replication result. Therefore, after 1.5

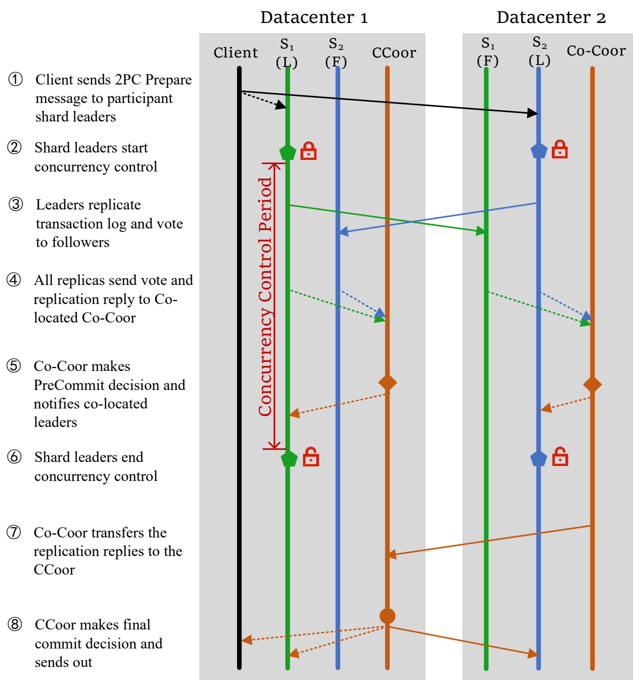


Figure 4: Example of committing transaction in D2PC. Solid and dashed arrows stand for inter-datacenter and intra-datacenter messages, respectively.

inter-datacenter RTTs, the correspondent coordinator has received the replies from each participant's majority of followers, which indicates that replication has been completed.

4 D2PC PROTOCOL

4.1 Commit Processes in D2PC

After a new transaction is initiated, the client executes the transaction logic and generates the read and write sets of the transaction. The read set contains tuples of ⟨ *key, version* ⟩ for each read data, while the write set includes tuples of the format ⟨ *key, new_value* ⟩. Once the transaction is `Executed`, it enters the PreCommit phase of the D2PC protocol.

PreCommit Phase. When the client invokes the *Commit* function, a `Prepare` request is sent to all participant leaders (Process ① in Fig. 4). Each participant leader receiving the `Prepare` request retrieves the transaction ID (tid) along with the transaction's read and write set specific to that shard. Subsequently, the participant leader invokes the *Prepare* function of the data store to validate whether the transaction can be successfully committed. If the participant leader agrees to commit the transaction, it is marked as `Prepared`. The leader then generates a log entry for the transaction, which includes tid, read and write sets. The log entry, along with the vote and involved shard list, is subsequently replicated to all replicas, including the leader itself (Process ③). In cases where a datacenter lacks a replica, the leader directly sends its vote to the co-coordinator of that datacenter.

Upon receiving the replication message, each replica notifies the co-located co-coordinator of the vote, its replication reply, and the involved shard list (Process ④). At this point, each co-coordinator can collect votes of each shard and make the `PreCommit` decision (Process ⑤). After that, each co-coordinator forwards the replication replies it collects to the correspondent coordinator (Process ⑦). Once the correspondent coordinator receives replication replies from the majority of replicas for each participant, it confirms the fault-tolerance of the transaction result. At this point, the correspondent coordinator can safely commit the transaction and the transaction enters the *Commit* phase.

The process described above, where the correspondent coordinator directly obtains the replication reply bypassing the leader, can be considered the fast path. Additionally, there is a slow path that acts the same as commit processes in layered mode as described in Fig. 1, where the follows also send the replication replies to the leader. After the leader receives majority replies, it informs the correspondent coordinator of its vote and replication result. Since both fast and slow paths make commit decision according to the vote of each shard, they always produce the same output. This slow path is retained to ensure that the correspondent coordinator can still learn the replication results even if bypass leader replies are interrupted due to failures. In the slow path, the correspondent coordinator will directly transform the state from `Prepared` to `Commit`.

Commit Phase. Once the correspondent coordinator has made the decision to either commit or abort, it promptly sends a response to the client (Process ⑥). Subsequently, it asynchronously notifies all participant leaders of the decision. This notification step is not part of the commit path and does not impact the commit latency.

Commit Latency Analysis. The transaction commit process begins when the client sends a `Prepare` message and ends upon receiving a notification from the correspondent coordinator. As the notification from the correspondent coordinator to client is an intra-datacenter, which is trivial compared to the delay required for inter-datacenter communication, it can be assumed that the transaction commit ends when the correspondent coordinator makes the final commit decision.

The correspondent coordinator must meet two conditions to make the commit decision. (*i*) It must receive votes from all participant shards. As shown in Fig. 4, to collect votes, it must go through Process ①, ③, and ④. Since messages in Process ① and ③ are inter-datacenter, and messages in ④ are intra-datacenter, which is negligible, the total message delay is 1 round of inter-datacenter communication. (*ii*) The correspondent coordinator must receive the replication replies from a majority of followers of each shard. This must go through Process ①, ③, ④, and ⑦. As only messages in Process ①, ③, and ⑥ are inter-datacenter, it requires 1.5 rounds of inter-datacenter communication to ensure replications are finished. Therefore, the overall commit latency is 1.5 inter-datacenter RTTs.

An important observation to highlight is that in the commonly adopted three-replica deployment, if each shard has a replica co-locates with the correspondent coordinator, the commit latency is only 1 inter-datacenter RTT. In this deployment, two replicas, including the leader and a follower, can form a majority. Since one follower receives a replication message indicating that the leader has persisted the log before, a majority already formed. Therefore, in Process ⑤ in Fig. 4, when the correspondent coordinator receives the replication reply from the local follower, it ensures the replication has been finished. Meanwhile, at this point, it also has collected

votes from all participants, the transaction can be safely committed, which only takes 1 inter-datacenter RTT.

4.2 Decentralized Commit via Co-coordinators

As shown in Fig. 4, upon collecting votes of all participant shards from local followers, the co-coordinator independently makes the PreCommit decision. Precisely, if abort exists in votes, the transaction will be `Abort`. Otherwise, the transaction is `PreCommit`. Since each co-coordinator can only make the decision according to the votes of each participant, consistent `PreCommit` decisions are always reached. After that, they notify the co-located participant leaders immediately. When a participant leader receives the `PreCommit` decision, it promptly ends the concurrency control period, e.g., releases locks in case of 2PL and removes the transaction from the validation list in OCC. Then transaction updates are visible.

By leveraging decentralized commit, the concurrency control period starts when the participant leader receives the `Prepare` message, and ends at the time of receiving the `PreCommit` decision. The message path includes Process ③ and Process ④. Only the messages in Process ③ are inter-datacenter, therefore, the concurrency control period length is 0.5 inter-datacenter RTT.

Note that when 2PL is adopted, the concurrency control period starts when acquiring read locks during execution, which will extend the concurrency control period length. But with the read optimization that will be introduced in § 4.6, the concurrency control period length of D2PC+2PL can also be reduced to only 0.5 inter-datacenter RTT.

After a transaction is committed, co-coordinators establish consensus to ensure the fault tolerance of the commit decision, which simplifies the resolution of 2PC blocking. As outlined in § 4.4, in the event of a correspondent coordinator failure, the commit decision can be recovered from other co-coordinators.

To achieve this, once the correspondent coordinator makes the final commit decision, it notifies all co-coordinators. When the correspondent coordinator receives replies from a majority of co-coordinators, the commit decision reaches fault-tolerant. It is important to note that the replication of commit decisions occurs asynchronously after the transaction is committed, which means it is not on the transaction commit path and has no impact on the commit latency.

Because each co-coordinator needs to handle all transactions, to prevent co-coordinators from becoming bottlenecks, we design a co-coordinator sharding strategy that shards co-coordinators into multiple co-coordinator groups. For example, when co-coordinators become the bottleneck, they can be partitioned into N groups, with N co-coordinators in each datacenter. In this way, the load on one co-coordinator can be balanced on N co-coordinators by taking the transaction ID modulo N.

4.3 Dependency Tracking

If the concurrency control period is ended before the transaction is committed, it is crucial to have strategies to ensure both *serializability* and *recoverability*. To ensure serializability, all dependencies, including *war*, *raw*, and *waw* dependencies, should be tracked to prevent them from forming cycles. To ensure recoverability, we should prevent a transaction that reads uncommitted data from being committed. For example, T_2 reads the uncommitted updates of T_1 and commits before T_1. If T_1 is later aborted and the server failure occurs, T_2 cannot be recovered as it has read non-existent data. Hence, to ensure recoverability, *raw* dependencies should be tracked. In D2PC, we choose to end the concurrency control period when the `PreCommit` decision is made. At this stage, the global serial order has been established. Therefore, concluding the concurrency control period will not violate serializability. Thus, D2PC only needs to track *raw* dependencies.

Algorithm 1: Function calls in D2PC

```
/* tuple.precommit # List of PreCommit
   transactions on the tuple                    */
1  Function Read(txn, tuple)
2      t ← the last entry in tuple.precommit
3      t.out.add(txn)
4      txn.in++
   // calls when receiving the PreCommit decision
      from the co-located co-coordinator.
5  Function PreCommit(txn)
6      for ∀ tuple ∈ txn.wset do
7          tuple.precommit.add(txn)
   // calls when receiving the commit decision from
      the correspondent coordinator.
8  Function Commit(txn, decision)
9      for ∀ t ∈ txn.out do
10         if decision == Commit then
11             t.in−−
12         else if decision == Abort then
13             t.in ← −1
14     for ∀ tuple ∈ txn.wset do
15         tuple.precommit.remove(txn)
```

D2PC introduces a `PreCommit` list for each tuple, which can efficiently identify and manage *raw* dependencies. When a participant leader receives the `PreCommit` decision of a transaction, it ends the concurrency control period and adds the transaction id to the `PreCommit` list of each write key. For a subsequent transaction that reads a key, it checks the key's `PreCommit` list to determine if any transaction has updated this key but not committed yet. If so, a *raw* dependency is identified. After identifying *raw* dependencies, D2PC effectively manages them using the register-and-report method [28]. On each participant shard, each transaction maintains a counter called *in*, which records the number of transactions it depends on (i.e., the uncommitted transactions whose updates are read by it). Additionally, each transaction maintains a list named *out*, which captures all the transactions that depend on it (i.e., the transactions that have read its updates).

The maintenance of *raw* dependencies is illustrated in Algorithm 1. For each read operation, the system checks if the read tuple has any `PreCommit` transactions. If so, a *raw* dependency is detected, and the corresponding *in* and *out* counters are updated accordingly (lines 2-4). Upon receiving the `PreCommit` decision, the participant leader invokes the *PreCommit* function to add the

transaction to the `PreCommit` list of all write keys (lines 6-7). When the final commit message is received, the *Commit* function is called, which removes the transaction from `PreCommit` lists (line 15). If the transaction is committed, all transactions that depend on it decrement their *in* counters by one. Conversely, if the transaction is aborted due to replication failure (arrow ⑦ in Fig. 2), the *in* counter of the dependent transaction is set to -1 (lines 9-13).

To ensure that transactions are committed in accordance with the *raw* dependency order, we enforce a rule that when a transaction's *in* counter is greater than 0, the participant leader should indicate that it has uncommitted dependencies in its vote. This prevents correspondent coordinator from actually committing it, but co-coordinators can still make the `PreCommit` decision. Only when all its dependent transactions are committed, it will notify the correspondent coordinator to be committed.

One issue raised by making writes visible early is the cascading abort [1], where the failure of a transaction causes aborting all subsequent transactions that have read its writes. However, in D2PC, updates made by `PreCommit` transactions are stored in memory within a write set, rather than taking effect in place. Therefore, when a new transaction reads a key, it locates the last `PreCommit` transaction on this key and uses the transaction ID to retrieve the write set to obtain the up-to-date value.

When a transaction is aborted, all following transactions that have read it can be identified by dependencies. Since these transactions have not been committed and their updates have not been applied to the database, they can be just aborted by setting the *in* counters to -1. These transactions will be directly removed from the memory. This design simplifies the process of aborting since there is no need to maintain the undo log for rolling back the database to a old state.

4.4 Failure and Recovery

Correspondent Coordinator Failure. The correspondent coordinator is the only node that can make the final commit decision, and as such, its failure may cause participant leaders and other co-coordinators to time out waiting for the decision. We will discuss how to handle correspondent coordinator failure in different cases.

Case 1. If the correspondent coordinator fails after the final decision has been sent to some participants and co-coordinators but not all of them, some co-coordinators will timeout while awaiting the decision. In such a scenario, a co-coordinator is elected as the correspondent coordinator and initiates the recovery phase by asking other co-coordinators to determine if anyone has received the decision. Upon receiving a reply containing the decision, the decision will be accepted and notified to participant leaders and other co-coordinators.

Case 2. In the event that the correspondent coordinator fails before the final decision is sent out, a co-coordinator is elected as the correspondent coordinator and initiates the recovery phase as distributed in **Case 1**. Since the decision was not made or sent out before the previous correspondent coordinator failed, none of co-coordinators has received the decision. Consequently, this co-coordinator proceeds with the termination protocol, which needs to ask all participant shard leaders for their votes and replication results. Once the replies are received from all participant shards, this co-coordinator makes the commit decision. It then notifies all participant leaders and co-coordinators, and ends the process of the undetermined transaction.

Co-coordinator Failure. If a co-coordinator fails, the participant leader co-located with it may not receive the `PreCommit` decision. When the leader eventually receives the decision from the correspondent coordinator, it can safely end the transaction. If more than $\mathcal{F}$ co-coordinators fail, there is a possibility that the correspondent coordinator may not receive the replication replies through the fast path. Nevertheless, the correspondent coordinator can still obtain the replication results from participant leaders via the slow path, as described in § 4.1. Thus, the failure of a co-coordinator does not bother the successful commit of transactions.

Participant Shard Replica Failure. When a participant shard leader fails, the correspondent coordinator can still receive the vote and replication replies from shard followers. Once the correspondent coordinator has collected votes and replication replies from a majority of followers, it can ensure that the replication has been completed and makes the final commit decision. If less than a majority of replies are received, the correspondent coordinator is not certain about the replication result. In this scenario, the correspondent coordinator will wait for a new leader to be elected and process this transaction. Once the new leader notifies the replication result, the correspondent coordinator can make the final decision and end the recovery of this transaction.

If a shard follower fails, the co-located co-coordinator may not receive the vote of this shard. This prevents participant leaders from concluding concurrency control period early because the co-located co-coordinator cannot make the `PreCommit` decision. Therefore, only after receiving the final commit decision from the correspondent coordinator, the participant leaders in this datacenter can safely end the transaction.

4.5 Correctness

Follows the properties proposed in [2] that ensures the correctness of atomic commitment protocol, D2PC is also proved to satisfy these five properties. Properties AC3 and AC4 are inherently maintained in D2PC since we do not change them at all. Therefore, we only prove D2PC satisfies other properties due to limited space.

AC1: All processes that reach a decision reach the same one.

Proof: Each participant can only learn the final decision in two ways: (i) from the correspondent coordinator or (ii) by executing the recovery protocol. In the first case, as proved in proof of AC2, the decision made by the correspondent coordinator is unique, the decision learned from the correspondent coordinator will be identical for all participants. The second case will be proven in the proof of AC5.

AC2: A process cannot reverse its decision after reaching one.

Proof: The correspondent coordinator can only make the Commit decision based on votes and replication results. Once a leader votes for a transaction, it has determined the order of the transaction on this shard, and the vote will not change later. After replication is successful, each shard's vote has been stored on at least a majority of replicas, and will never lose. Therefore, the commit decision will never be reversed.

Table 2: Network latency between datacenters (ms). Datacenters are located in Hangzhou (East China), Beijing (North China), Shenzhen (South China), San Francisco (US West), Virginia (US East), and Frankfurt (Europe).

	Hangzhou	San Francisco	Frankfurt	Beijing	Virginia	Shenzhen
Hangzhou	0.2	140	231	30	203	29
San Francisco		0.2	151	150	67	160
Frankfurt			0.25	240	98	270
Beijing				0.2	215	40
Virginia					0.3	229
Shenzhen						0.3

AC5: Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Proof: AC5 describes that the decision can survive from failures. In D2PC, as described in § 4.2, the correspondent coordinator will replicate the final decision to co-coordinators. According to the recovery rule in Case 1 in § 4.4, this decision can be recovered from alive co-coordinators, which is the decision made by the correspondent coordinator before.

If none of the co-coordinators have received the decision, each shard's vote, which is the essential information to make the decision, is persisted on a majority of replicas through replication. As shown in Case 2 in § 4.4, the decision can always be recovered by collecting votes of each participant shard from their replicas, and the recovery protocol will make a decision that is identical to the correspondent coordinator's decision since shards' votes are unchanged.

Therefore, during recovery, the decision can always be restored and is identical to the previously made by the correspondent coordinator, which proves both AC5 and AC1 are satisfied.

4.6 Read Optimization

During transaction execution, read operations are typically served by shard leaders to ensure data recency, but accessing remote leaders incurs substantial costs. Considering that each shard may have a replica that co-locates with the client, we allow the client to directly read from local replicas to reduce inter-datacenter communication, which is a common optimization in many protocols [10, 22, 38].

However, reading from follower replicas may read stale data. To ensure serializability, we will perform a read version verification during committing. The Prepare message will be accompanied by a read set containing the read versions of the transaction. When a leader receives the Prepare message, it first verifies if read versions are up-to-date, and then aborts those who have read stale data. Only when all read operations pass the recency validation, the leader continues to process the transaction.

The local read strategy is compatible with both 2PL and OCC. Since OCC is a verification-based approach, integrating the read optimization aligns naturally with OCC. However, adopting it with 2PL can potentially compromise fairness and lead to transaction starvation. Since no read locks are required during execution, transactions with large reads could frequently abort due to stale reads, which could be avoided in standard 2PL protocol. To address this, the second execution of aborted transactions is enforced to acquire read locks from the shard leader. This allows D2PC to mitigate performance penalties associated with remote reads while preserving fairness in 2PL. In our evaluation, detailed in § 5, we separately evaluate the performance of 2PL with and without read optimization, whereas OCC consistently incorporates read optimization.

Table 3: Transaction profile for Retwis workload.

Transaction Type	# gets	# puts	workload%
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1,10)	0	50%

5 EVALUATION

5.1 Experimental Setup

The Testbeds. All experiments were conducted on Alibaba Cloud ECS instances distributed across six datacenters. Table 2 outlines the network latencies between datacenters. By default, the experiments were conducted with a 3-replica deployment, where the replicas were located in Hangzhou, San Francisco, and Frankfurt. In § 5.4, the replicas in all datacenters were also involved in the evaluation.

Each cloud server was equipped with 4 CPU cores and 8GB of memory. The system configuration involved 3 - 7 data shards, with each shard consisting of 3 - 5 replicas. Each cloud server hosted a replica for each shard. The leaders of shards were distributed among different datacenters. Furthermore, within each datacenter, a server acted as the co-coordinator to serve co-located servers.

Candidates for Comparative Study. First, we compared D2PC with standard 2PC-based transaction protocol named 2PC+2PL, which is the combination of 2PC, 2PL, and Multi-Paxos. There is also an approach named 2PC+OCC, which replaces the concurrency control protocol with OCC. Both 2PC+2PL and 2PC+OCC represent layered mode architecture and exhibit a commit latency and concurrency control period length of 2 inter-datacenter RTTs.

Additionally, we conducted a comparison between D2PC and integrated mode protocols including Carousel [38], TAPIR [39], and STARRY [40]. In conflict-free scenarios, all three protocols can reach a consensus in 1 inter-datacenter RTT. However, in conflict scenarios, they must put more effort into resolving conflicts. Carousel and STARRY fall back to slow paths which take 2 and 2.5 inter-datacenter RTTs to commit. TAPIR will abort many conflicting transactions, leading to a high abort rate. As the concurrency control protocol used in all three protocols is OCC, we will compare D2PC+OCC with them.

To ensure a fair comparison, all protocols are implemented in the same code prototype and integrated with the read optimization detailed in § 4.6. For 2PL, we also conducted evaluations without read optimization, denoted as 2PC+2PL-NRO and D2PC+2PL-NRO.

Workload. We used three workloads for evaluation. The first was a synthetic workload for the Retwis application, which simulates Twitter's functionality. The Retwis workload contains four types of transactions, as outlined in Table 3. On average, each transaction accesses 4-10 data items across 2-3 shards. The second workload was TPC-C, which is the most commonly used workload for transaction processing. TPC-C contains five transaction types. In our evaluation, we only choose the three read-write transaction types. The third was a micro-benchmark which we can adjust the transaction length.

5.2 Retwis

Performance with increasing load. Firstly, we evaluated the performance of different protocols under the Retwis workload with medium contention (Zipf coefficient = 0.7).

Fig. 5 illustrates the performance with 2PL. We first set both D2PC+2PL and 2PC+2PL disable the read optimization to clearly

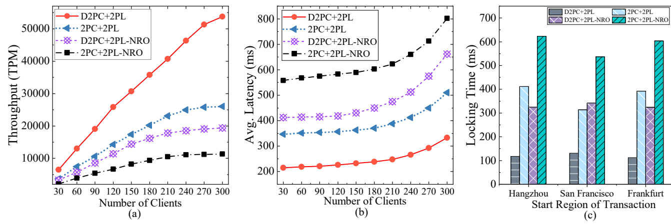


Figure 5: Retwis: increasing the number of clients (2PL).

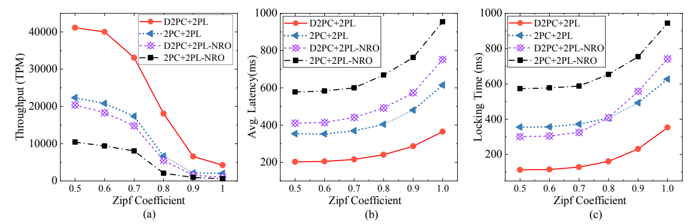


Figure 6: Retwis: varied contention levels (2PL).

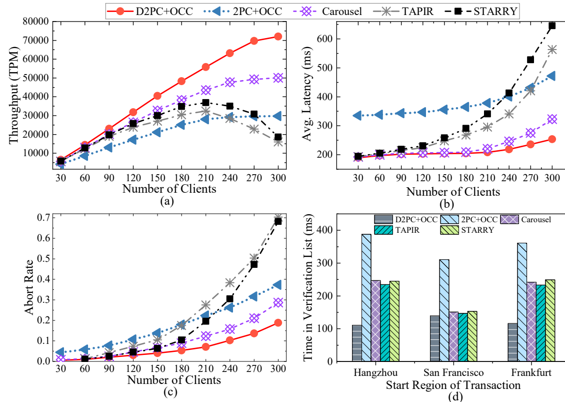


Figure 7: Retwis: increasing the number of clients (OCC).

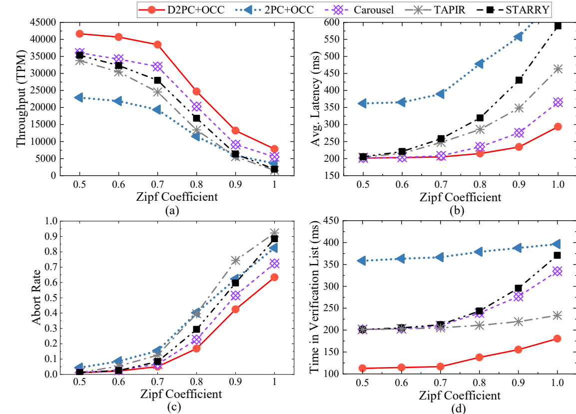


Figure 8: Retwis: varied contention levels (OCC).

show the improvement brought by D2PC. We can see that D2PC+2PL-NRO achieves 1.73 × throughput than 2PC+2PL-NRO. This is because D2PC+2PL-NRO can reduce the latency by 25% and shorten the concurrency control period length by 44%. After enabling read optimization, both D2PC+2PL and 2PC+2PL achieve significant improvement in throughput and latency, and the performance advantages of D2PC+2PL expand further. As shown in Fig. 5(a), (b), and (c), compared to 2PC+2PL, D2PC+2PL can realize a 2.33 × improvement in throughput, 42% reduction in latency, and 66% reduction in locking time.

Fig. 7 illustrates the performance under OCC. D2PC+OCC demonstrates a substantial performance improvement compared to the other protocols. As the number of clients increases, the throughput of D2PC+OCC exhibits a more rapid growth rate compared 2PC+OCC. The enhanced performance of D2PC+OCC can be attributed to low latency and short concurrency control period length. For example, with 300 clients, the commit latency of D2PC+OCC is approximately 250 milliseconds, which is reduced by about 43% compared to 2PC+OCC, and D2PC+OCC also achieves a reduction in the concurrency control period length of 64%, enabling higher concurrency and resulting in the throughput increases to 2.43 × that of 2PC+OCC.

We also compared D2PC with Carousel, TAPIR, and STARRY, all of which can achieve fast commitment. In comparison, D2PC+OCC achieves similar or lower commit latency. As depicted in Fig. 7 (b), when the load is low, the latency is comparable, because all of them can commit a transaction in one inter-datacenter RTT. As the client number increases, the likelihood of conflicts rises, the comparison protocols have to pay more cost to resolve conflicts.

For example, Carousel and STARRY switch to the slow path to let a central node resolve conflicts, which introduces extra communication, and TAPIR will abort conflicts and retry. Therefore, D2PC+OCC outperforms these protocols in transaction latency as the concurrency increases. In terms of concurrency control period length, D2PC+OCC demonstrates a significant advantage. The concurrency control period lengths of the comparison protocols are at least one inter-datacenter RTT, D2PC+OCC is only 0.5. This reduced concurrency control period length contributes to D2PC+OCC's higher throughput.

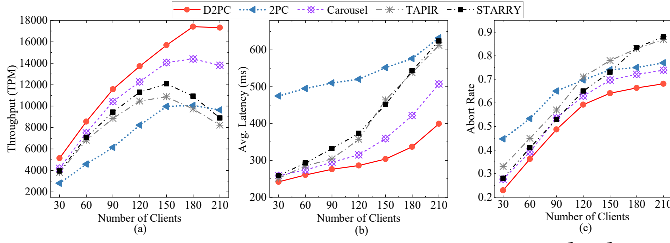
Performance under contention. By varying the Zipf coefficient, we simulated different contention levels and evaluated the performance. The workload adopted was Retwis, and the client number was fixed at 150.

Fig. 6 and 8 depict the performance under contention with 2PL and OCC respectively. As the contention increases, the throughput of all protocols gradually decreases and shows a sharp decline trend when the Zipf coefficient exceeds 0.7. Comparatively, the throughput of D2PC remains significantly higher than other approaches in both two 2PL and OCC under high contention. In the case of 2PL, Fig. 6 shows that D2PC outperforms 2PC+2PL with a throughput that is at most 3.06 × higher. When read optimization is disabled, D2PC+2PL-NRO also achieves 1.8 × throughput than 2PC+2PL-NRO. Similarly, in OCC, as shown in Fig. 8, when setting Zipf coefficient to 0.9, the throughput of D2PC surpasses 2PC+OCC by 2.35 ×. Compared with integrated mode protocols, D2PC consistently achieves higher throughput, surpassing Caoursel, TAPIR, and STARRY by 1.41 ×, 3.1 ×, and 2.79 × respectively.

The performance advantages of D2PC can be attributed to its shorter concurrency control period, as evident from Fig. 6 (c) and 8 (d). Regardless of the contention load, D2PC consistently exhibits the shortest concurrency control period. This reduction in concurrency control period mitigates contention, resulting in the lowest transaction abort rate, as demonstrated in Fig. 8 (c).

5.3 TPC-C

In the TPC-C evaluation, we only used three read-write transaction types with a proposition of neworder (48%), payment (47%), and delivery (5%).



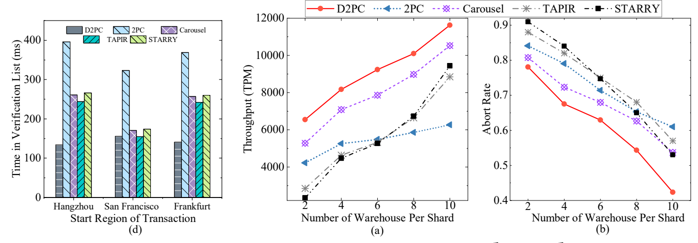


Figure 9: TPC-C: increasing loads.

Figure 10: TPC-C: varied warehouse num.

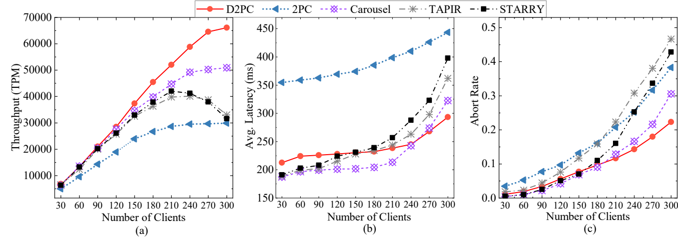


Figure 11: Performance under a 5-replica setup across the world.

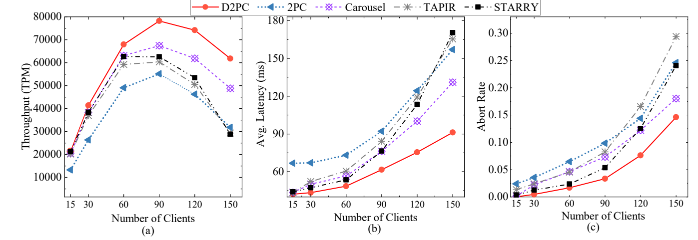


Figure 12: Performance under single continent setting.

Performance with varied client numbers. Fig. 9 shows the performance under increasing loads, and each shard contains 10 warehouses. As the number of clients increases, the throughput of D2PC climbs until reaches 17410 TPM, surpassing all competing protocols. Carousel, despite its fast commit path, still performs 18% lower than D2PC due to its longer concurrency control period, as depicted in Fig. 9(d). For 2PC, since it requires two rounds of communication to commit, its peak is only 61% of D2PC. For TAPIR and STARRY, under low conflict, they can achieve fast commitment, and their throughputs are similar to D2PC. However, as the conflict increases, their abort rates increase rapidly, causing the throughput to drop more rapidly than all other protocols.

Peformance with varied warehouse numbers. In the TPC-C workload, the number of warehouses is critical for tuning the contention level. With fewer warehouses, more transactions are competing to operate in the same warehouse, causing high contention. Fig 10 shows the performance with varied warehouse numbers per shard. D2PC, Carousel, and 2PC, which centrally resolve conflicts, cause their throughputs to be less impacted by contention than TAPIR and STARRY. Regardless of the level of contention, D2PC always shows the strongest ability in handling conflicts due to its shortest concurrency control period.

5.4 Varied Replica Locations

We further measured the performance with varied replica locations to better demonstrate the applicability of our approach under various setups. The concurrency control protocol was OCC, and the workload was Retwis with Zipf coefficient set to 0.7.

Five replicas under wide geo-distribution. Firstly, we measured the performance with a 5-replica setup that includes servers in the US East, US West, Europe, East China, and North China, to simulate a more extensive geo-distribution.

As depicted in Fig 11, D2PC achieves the highest peak throughput (Fig. 11(a)), outperforms 2PC, Carousel, TAPIR, and STARRY by 2.2 ×, 1.29 ×, 1.64 ×, and 1.57 × resepctively. As for latency, D2PC's commit latency slightly increases in the 5-replica setup due to the requirement of 1.5 inter-datacenter RTTs to complete the commit. Nevertheless, D2PC still maintains commit latency comparable to protocols with fast commit paths and significantly lower latency than 2PC+OCC. As the number of clients increases, D2PC+OCC exhibits a lower abort rate and average latency compared to Carousel, TAPIR, and STARRY because all of these protocols need extra costs to resolve conflicts.

Single continent setting. We further measured the performance under a single continent setup including replicas in East China, North China, and South China. Fig 12 shows that the transaction latencies are significantly reduced due to lower communication latency. Therefore, all protocols reach peak throughput faster than in previous experiments, with D2PC still achieving the highest peak throughput. As the client number further increases, the performance gap between D2PC and other protocols becomes wide, demonstrating the stable advantage of D2PC in handling contention under various settings.

5.5 Sensitivity Analysis of Performance

In this experiment, we used various situations to evaluate the robustness of D2PC.

Firstly, we used a micro-benchmark and controlled the operation number in a transaction. The number of clients is fixed to 150. In Fig. 13(a) and (b), as the transaction length increases, more conflicts occur, causing a rapid decline in throughput across all protocols. However, D2PC still shows an advantage in handling conflicts.

Secondly, we varied the number of shards from 3 to 7. In Fig 13(c) and (d), the increase in shard number doesn't have a huge impact on the throughput of all protocols, but the abort rates show a slow decline, because the loads are balanced on more shards, causing a reduction on conflicts.

Then, we evaluated the throughput of D2PC under co-coordinator failures. In Fig. 14(a), we artificially shut down two co-coordinator servers. Co-coordinator failure causes some transactions to fail to PreCommit, the throughput of D2PC will decrease. When only one co-coordinator is alive, D2PC essentially behaves like 2PC, requiring 2 RTTs to commit a transaction, leading to lower throughput than Carousel. However, participant leaders co-located with the alive co-coordinator can still benefit from early `PreCommit` decisions, D2PC maintains an advantage over 2PC.

Finally, we evaluated the efficacy of the co-coordinator sharding strategy in improving co-coordinator scalability. To simulate the

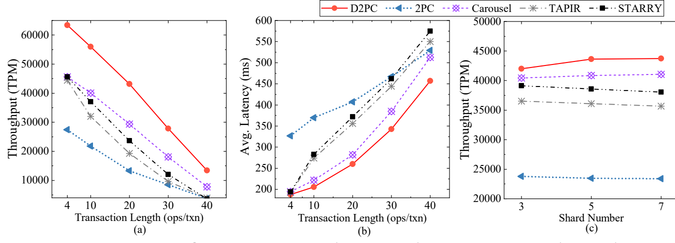


Figure 13: Performance with varied transaction lengths and shard numbers.

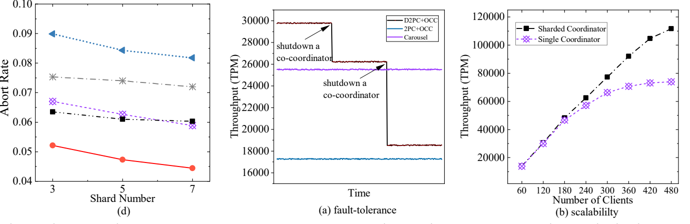


Figure 14: Fault-tolerance and scalability of co-coordinators.

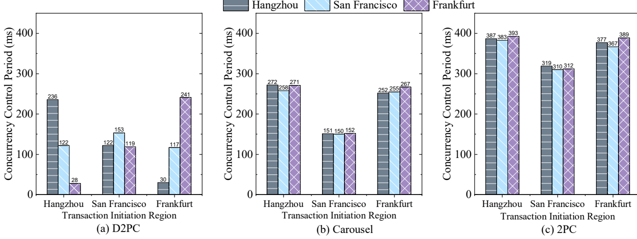


Figure 15: Comparsion of concurrency control period length.

Table 4: Analysis of concurrency control period length of shard leaders at b (transaction initiated at a).

D2PC	Carousel	2PC
$\max_{i \in p} \frac{L_{a-i}+L_{i-b}-L_{a-b}}{2}$	L_{a-max}	$\max_{i \in p}(L_{a-i} + L_{i-majority})$

scenario of co-coordinator bottleneck, we allocated only 1 CPU and 512 MB memory to co-coordinator servers. Fig. 14(b) shows that when a single co-coordinator group is deployed, throughput stabilizes as the client number increases. After sharding co-coordinators into two groups (with two co-coordinators in each datacenter), the load will be balanced, resulting in much higher throughput.

5.6 Detailed Study of Concurrency Control Period Length

In this experiment, we conducted a detailed analysis of concurrency control period lengths. The experiment used the default 3-replica setup, and each transaction access all three shards. The number of clients is fixed at 150, distributed evenly among all three datacenters.

We first define three kinds of time. L_{i-j} denotes the RTT between datacenter i and j. L_{i-max} refers to the maximum time required for datacenter i to receive replies from all datacenters, determined by the network latency with the most distant datacenter. $L_{i-majority}$ indicates the time required for datacenter i to receive replies from a majority of datacenters.

Table 4 presents an overview of the concurrency control period length for different protocols. We assume that the transaction is initiated at datacenter a, and analyze the concurrency control period length of shard leaders at datacenter b.

Theoretically, the number of round trips in the concurrency control period can be identified by calculating the coefficient of the formulas in Table 4, the coefficient is n indicates contains n round trips. The coefficient of Carousel's concurrency control period length is 1, while 2PC's coefficient is 2, therefore, their concurrency control period lengths are 1 inter-datacenter RTT and 2 inter-datacenter RTTs, respectively. Consequently, their actual concurrency control period lengths follow the same increasing order, as depicted in Fig.15(b) and (c).

In contrast, D2PC's coefficient is 1/2 after simplification, which means the concurrency control period length is only 0.5 inter-datacenter RTT. However, the concurrency control period length of D2PC exhibits significant variation across different datacenters. This is because the concurrency control period of D2PC is related to the locations of both the client and servers.

We set the timing of commit initiation as 0. For the shard leader in datacenter b, the concurrency control period starts when it receives the Prepare message at time $\frac{L_{a-b}}{2}$, and ends when receives all vote messages from other participant shards at the time $\max_{i \in p} \frac{L_{a-i}+L_{i-b}}{2}$, where p is the set of participant datacenters. Therefore, the concurrency control period length is $\max_{i \in p} \frac{L_{a-i}+L_{i-b}-L_{a-b}}{2}$.

This formula shows that the concurrency control period of D2PC is negatively correlated to the network latency between a and b. For example, taking Hangzhou as the initiated datacenter a, and Frankfurt as b. Referring to the network latencies in Table 2, the network latency between them is the highest. Therefore, the concurrency control period of Frankfurt is the shortest.

As a result, the concurrency control period length in D2PC exhibits variation depending on the locations. However, the overall concurrency control period length of D2PC is still significantly lower than other protocols.

6 CONCLUSION

This paper proposed the decentralized transaction commit protocol that offers several key insights. The primary objective is to minimize the impact of cross-region communication on system currency and commit latency. To achieve this, D2PC leverages decentralized commit via co-coordinators, and parallels processes of 2PC and replication. Compared to commit approaches based on 2PC, D2PC demonstrates significant performance improvements in terms of throughput and latency. Furthermore, when compared to integrated mode approaches that achieve fast commitment, D2PC realizes a similar reduction in commit latency while also delivering higher improvements in throughput. Our experimental study demonstrated these promising characteristics.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (2023YFB4502905), NSFC Project No. 92270202, and the Natural Science Foundation of Shanghai (23ZR1418300). It was also supported by ZTE Industry-University-Institute Cooperation Funds under Grant No. HC-CN-20220721010.

REFERENCES

[1] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. 1995. Ordered Shared Locks for Real-Time Databases. *VLDB J.* 4, 1 (1995), 87–126.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[3] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *PODC*. 398–407.

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

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8:1–8:22.

[6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 1243–1254.

[7] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report MSR-TR-2016-1001. https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/

[8] Tamer Eldeeb and Philip A Bernstein. 2016. Transactions for Distributed Actors in the Cloud. (2016).

[9] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (2017), 613–624.

[10] Hua Fan and Wojciech M. Golab. 2019. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[11] Goetz Graefe, Mark Lillibridge, Harumi A. Kuno, Joseph Tucek, and Alistair C. Veitch. 2013. Controlled lock violation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 85–96.

[12] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Trans. Database Syst.* 31, 1 (2006), 133–160.

[13] Hua Guo, Xuan Zhou, and Le Cai. 2021. Lock Violation for Fault-tolerant Distributed Database System. In *ICDE*. IEEE, 1416–1427.

[14] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* 658–670.

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

[16] Ramesh Gupta, Jayant R. Haritsa, and Krithi Ramamritham. 1997. Revisiting Commit Processing in Distributed Database Systems. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.* ACM Press, 486–497.

[17] Ramesh Gupta, Jayant R. Haritsa, and Krithi Ramamritham. 1997. Revisiting Commit Processing in Distributed Database Systems. In *SIGMOD*. 486–497.

[18] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.* OpenProceedings.org, 157–168.

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

[20] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.* 3, 1 (2010), 681–692.

[21] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. 603–614.

[22] 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 *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021.* ACM, 145–161.

[23] Idit Keidar and Danny Dolev. 1995. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Mihalis Yannakakis and Serge Abiteboul (Eds.). ACM Press, 245–254.

[24] Hideaki Kimura, Goetz Graefe, and Harumi A. Kuno. 2012. Efficient Locking Techniques for Databases on Modern Hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012.* 1–12.

[25] Hideaki Kimura, Goetz Graefe, and Harumi A. Kuno. 2012. Efficient Locking Techniques for Databases on Modern Hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012.* 1–12.

[26] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan D. Fekete. 2013. MDCC: multi-data center consistency. In *EuroSys*. 113–126.

[27] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[28] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. [n.d.]. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 ([n. d.]), 298–309.

[29] Hatem A. Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-Latency Multi-Datacenter Databases using Replicated Commit. *Proc. VLDB Endow.* 6, 9 (2013), 661–672.

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

[31] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *ATC*. 305–319.

[32] P. Krishna Reddy and Masaru Kitsuregawa. 2004. Speculative Locking Protocols to Improve Performance for Distributed Database System. *IEEE Trans. Knowl. Data Eng.* 16, 2 (2004), 154–169.

[33] Dale Skeen. 1981. Nonblocking Commit Protocols. In *SIGMOD*. 133–142.

[34] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial Strictness in Two-Phase Locking. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 893. Springer, 139–147.

[35] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial Strictness in Two-Phase Locking. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 893. 139–147.

[36] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*. 1493–1509.

[37] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2312–2325.

[38] 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 *SIGMOD*. 231–243.

[39] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *SOSP*. 263–278.

[40] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. 2022. STARRY: Multi-master Transaction Processing on Semi-leader Architecture. *Proc. VLDB Endow.* 16, 1 (2022), 77–89.

[41] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, Shangjun Han, Shengyi Wang, Guoliang Li, and Ge Yu. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proc. ACM Manag. Data* 1, 1 (2023), 62:1–62:27.