**STARRY: Multi-master Transaction Processing on Semi-leader Architecture**

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

Huiqi Hu†  
Xuan Zhou  
East China Normal University†  
{hqu,xzhou}@dase.ecnu.edu.cn

Jiang Wang  
Huawei Co., Ltd.  
wangjiang16@huawei.com

**ABSTRACT**

Multi-master architecture is desirable for cloud databases in supporting large-scale transaction processing. To enable concurrent transaction execution on multiple computing nodes, we need an efficient transaction commit protocol on the storage layer that ensures ACID as well as consensus among replicas. A leader-based protocol is easy to implement. However, it faces the single-node bottleneck and suffers from high transaction latency in cross-region deployment. While a leaderless protocol can achieve a higher degree of parallelism, it is inefficient in resolving conflicts.

This paper proposes the semi-leader protocol, which is a new type of transaction commit protocol for multi-master transaction processing. In a nutshell, the semi-leader protocol is a hybrid protocol that offers separate commit paths for conflicting transactions and non-conflicting transactions. A centralized node, known as the sequencer, is employed to perform precise conflict resolution for conflicting transactions, while non-conflicting transactions can be committed timely in a decentralized manner. Based on the semi-leader protocol, we designed **STARRY**, a multi-master transaction processing mechanism. Experimental results demonstrate that **STARRY** is 1.4 × and 4.21 × as performant as the leaderless and leader-based protocols respectively in throughput. When dealing with high-contention workloads, **STARRY** can significantly reduce the abort rates.

**PVLDB Reference Format:**

doi:10.14778/3561261.3561268

**1 INTRODUCTION**

Resource separation and elasticity are the preeminent design principles to cloud database systems. Most recent cloud databases[1–3, 9, 10, 14, 36, 37] have chosen to disaggregate the computation and storage into separate layers, so that both layers can expand and shrink independently.

Under the disaggregated architecture, most cloud database systems claim to support high availability, strong consistency, and scalability of transaction processing.  

- To provide high availability, the storage layer must maintain multiple replicas. To tolerate regional failures, the replicas even need to be geographically distributed.  
- To ensure consistency of data, the storage layer needs to perform global concurrency control to achieve serializability of transactions. At the same time, a consensus protocol is required to reach a consistent transaction order among replicas, thus achieving linearizability.  
- To enable scalability of transaction processing, the computing layer needs to support adding additional computing nodes to execute transactions concurrently. Following the notion proposed in Aurora[4, 36], this is called multi-master transaction processing in cloud databases. Other than improving the throughput of transaction processing, a multi-master architecture can also enhance the availability, as each computing node can provide individual transaction services[4, 5], especially when the computing nodes are deployed in different regions.

In this paper, we study how to design a transaction processing mechanism to meet the aforementioned properties in an efficient way. Fig. 2 illustrates three ways to support cross-region multi-master transaction processing on the disaggregated-storage architecture. The storage layer consists of multiple replicas, which can provide unified data access services for all computing nodes. In a cross-region deployment, the computing nodes and storage replicas are distributed in multiple regions to enable high availability. When the workload increases, more computing nodes can be deployed to execute transactions concurrently. This enables scalability of transaction processing.

When multiple computing nodes process transactions concurrently, inter-node conflicts can become a major setback for performance. To verify this, we conducted a simple set of experiments on Aurora (in May 2022). We measured the performance variation of Aurora’s single-master and multi-master clusters in dealing with varying degrees of contention. As shown in Fig. 1, when contention intensifies, the performance of both clusters drops. However, the performance of the multi-master cluster drops substantially faster than that of the single-master cluster. This clearly indicates that
inter-node conflicts have a significant negative impact on performance. It motivates us to devise a mechanism to efficiently resolve inter-node conflicts.

In a disaggregated-storage architecture, it is impractical to rely on computing nodes to perform conflict resolution, as they are designed to be stateless and independent. A common approach is to resolve conflicts at the storage layer [4, 36], since transactions should be committed to the storage layer to persist results. This requires a transaction commit protocol that combines both concurrency control and consensus protocols, the former is required to handle inter-node conflicts and the latter ensures the consistency of storage replicas. Regarding concurrency control, as all transactions must be committed on the storage layer, it is natural to choose OCC [20] which will detect conflicts during committing. As for consensus protocols, different consensus mechanisms can result in completely different effects. Existing consensus protocols can be classified into two categories. One is known as leader-based protocols, which requires a centralized node, called leader, to process all commands. Examples include Multi-Paxos [11] and Raft [31]. The other is known as leaderless protocols, which allow all replicas to process commands and collectively reach an agreement about the order of commands. An example is EPaxos [28].

Fig. 2(a) and (b) illustrate two multi-master transaction processing architectures based on leader-based and leaderless protocols respectively. As we can see, if a leader-based protocol is adopted, all transaction commit requests are submitted to the leader to be processed. Therefore, the leader will be more loaded than the other replicas and become a potential bottleneck. Moreover, remote clients have to bear the latency of cross-region communication with the leader. In a leaderless protocol, all replicas can share the workload evenly, enabling better scalability of transaction processing. In addition, each client can be served by the nearest replica, which helps reduce the latency. However, a leaderless protocol has to resort to a decentralized approach for conflict resolution, which can be either imprecise or costly.

Fig. 3 illustrates the problem of decentralized conflict resolution. After execution, three concurrent conflicting transactions are sent to three different replicas for committing. Due to the different arrival times, each replica may see different orders of transactions (consider R1 and R2). One may even miss some transactions (consider R3). One approach to resolve conflicts is that each replica makes decision independently according to some rules, and tries to reach a consensus on the decision. If OCC is applied, the commit of one transaction will force all its conflicting transactions to abort (e.g., the commit of T1 will force R1 to abort T2 and T3). As a result, after collecting decisions from all replicas, none of the three transactions can commit, since none of them reaches a commit decision on a majority. The other approach is to let all replicas communicate and reach an agreement about the order of conflicting transactions. However, this may incur prohibitive communication costs. Thus, the limitation of leaderless protocols in resolving conflicts motivates us to design a protocol that can achieve precise conflict resolution while ensuring efficient transaction processing.

In this paper, we introduce STARRY, an efficient mechanism for multi-master transaction processing that is built upon a new transaction commit protocol called semi-leader protocol. As illustrated in Fig. 2(c), the core idea of semi-leader protocol is to offer two separate commit paths, a centralized one for conflicting transactions and a decentralized one for non-conflicting transactions. For non-conflicting transactions, the protocol works in a decentralized manner, so that transactions can commit on any replica to achieve good scalability and fast committing. When conflicts occur, it turns into the conflict path, which employs a special replica known as sequencer to perform precise centralized conflict resolution. Once a conflict is detected on a replica, the conflict information will be messaged to the sequencer, who uses its global view to identify an optimal serial order for conflicting transactions by reordering technology. This enables STARRY to minimize the negative impact of inter-node conflicts on performance.

Such a hybrid protocol changes the coordination pattern of existing protocols, since it combines decentralized and centralized coordination. Moreover, the separation of commit paths poses a number of challenges to the design of the semi-leader protocol. First, as a transaction may reach inconsistent commit decisions on two paths, it is important to ensure the uniqueness of the final decision. Second, the sequencer should collect as complete conflict information as possible in time, to enable more precise conflict resolution. Third, we need a new recovery protocol to ensure that transactions on both paths survive failure. Last, these methods must be integrated into a correct and efficient protocol. In this paper, we show how semi-leader protocol can cope with these challenges.
Table 1 summarizes the advantages of semi-leader protocol in supporting multi-master transaction processing. In comparison with leader-based protocols, semi-leader protocol allows for better scalability and lower latency. In comparison with leaderless protocols, semi-leader protocol offers more precise conflict resolution.

The contributions of this paper are summarized as follows:

- We proposed a semi-leader transaction commit protocol, which enables the combination of fast decentralized committing and precise centralized conflict resolution. Based on it, we designed 
  
  \textit{Starry}, a new multi-master transaction processing mechanism for disaggregated-storage architecture that can minimize the impact of inter-node conflicts.
- We further extended \textit{Starry} to support distributed transactions, and optimized the read-only transaction algorithm for better performance.
- We conducted extensive experiments to evaluate \textit{Starry}’s performance in multi-master transaction processing.

The rest of the paper is organized as follows: § 2 introduces the background and related work. § 3 describes \textit{Starry} in detail, including the semi-leader protocol, the conflict reordering technique and the recovery mechanism. Their correctness is also analyzed. § 4 presents the designs to support distributed transactions and read-only transactions. Experimental results are presented in § 5.

2 BACKGROUND AND RELATED WORK

\textit{Starry} aims to support multi-master transaction processing on the architecture of typical cloud database, in which storage and computation are disaggregated. The key lies in how to integrate the concurrency control and consensus protocols. This section introduces the current development of transactional cloud databases and the related work on concurrency control and consensus protocols.

2.1 Transaction Processing on Cloud Databases

In cloud-native databases with separated computation and storage layers, each computing node runs an instance to process requests from clients, and the storage layer provides unified data access interfaces to the computing nodes. Systems such as Aurora [36, 37] and PolarDB [1, 9, 10] lay out computing nodes as one primary read/write (RW) node and multiple read-only (RO) nodes. Only the RW node can process read/write transactions, and the RO nodes only serve read-only transactions. Data is synchronized through redo logs. In Aurora, after the storage layer applies the redo logs from the RW node, the updates are visible to all RO nodes.

As a single RW node has limited capacity, some systems explored ways to support multiple RW nodes. Some early on-premise databases, such as Oracle RAC, use synchronization techniques, such as cache fusion [21], to enable concurrent transaction processing on multiple DB instances. Some cloud databases, such as Aurora and PolarDB have recently renewed their architectures to support multiple masters. However, conflict resolution between multiple RW nodes is a challenge. In Aurora’s multi-master cluster, the work of conflict detection is pushed down to the storage. Aurora only checks write conflicts at a coarser granularity of page (fixed as 16KB). On receiving redo logs from RW nodes, the storage node checks if multiple transactions modify the same page. If a conflict is detected, the transaction has to be rolled back. So far, Aurora’s multi-master cluster can support single region deployment with up to 4 RW nodes [4].

2.2 Concurrency Control and Consensus

In distributed database systems, concurrency control mechanisms and consensus protocols need to work together to ensure the correctness of transaction processing [12, 16, 18, 19, 30, 33, 38, 39]. In the literature, two types of consensus protocols were studied for transaction processing.

**Leader-based consensus protocol.** Many database systems build concurrency control mechanisms over leader-based consensus protocols such as Paxos [23] and Raft [31]. For example, Spanner [12, 13] adopts two-phase-locking (2PL) over Paxos, while CockroachDB [33] and TiDB [18] adopt multi-version concurrency control (MVCC) over Raft. In both approaches, concurrency control and consensus work independently, that is, the leader first applies its concurrency control protocol to schedule transactions and then replicates updates to other servers using the consensus protocol.

However, in supporting multi-master transaction processing (Fig. 2(a)), leader-based protocols may cause severe performance penalties. As summarized in Table 1, the overall performance is limited by the capacity of the leader, which hurts the scalability of the computing layer. Besides, as Fig. 2(a) shows, a transaction usually needs N rounds of cross-region communication to read data from the leader (where N is the number of read operations in the transaction) and additional two rounds for transaction commit and replication (and in Fig. 2(a)). The overall latency of N+2 wide-area RTTs may be high in some application scenarios.

**Leaderless consensus protocol.** Leaderless consensus protocols such as EPaxos [28] allow all replicas to process requests. A number of systems, such as TAPIR [39], MDCC [19], Carousel [38] and Janus [30], have applied the idea to distributed database systems. In these systems, concurrency control mechanisms and consensus protocols are integrated. When a transaction is committed, conflict detection and replication are performed at the same time. If no conflicts are detected in a super quorum, the transaction can be committed directly. This integration reduces the communication cost in the commit phase to one round trip, and thus significantly shortens the latency.

However, these methods struggle in handling conflicts. MDCC [19] suffers from both transaction conflicts and Paxos collisions to fail on fast commits [19, 24, 25] (collision occurs when replicas receive transactions in different orders), and it simply aborts conflicting transactions for conflict resolution. TAPIR [39] adopts the same

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Performance limitation</th>
<th>Wide-area transaction latency (RTT)</th>
<th>Conflict resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader-based</td>
<td>leader</td>
<td>N+2</td>
<td>centralized abort &amp; retry</td>
</tr>
<tr>
<td>Leaderless</td>
<td>/</td>
<td>non-conflict: 1 conflict: 2</td>
<td>decentralized abort &amp; retry</td>
</tr>
<tr>
<td>Semi-leader</td>
<td>/</td>
<td>non-conflict: 1 conflict: 2.5</td>
<td>centralized reorder &amp; re-commit</td>
</tr>
</tbody>
</table>
strategy except that it is free from Paxos collisions. More specifically, TAPIR employs an inconsistent replication (IR) protocol, which allows operations to be executed in any order, while the final consensus decision is made in the application layer. In TAPIR, transaction logs are sent to each replica, where the transaction will be validated using OCC-liked rules. After receiving replies from replicas, the application layer invokes the DECIDE function to decide the results. Transactions that are Prepare-OK on majority replicas can be committed. As for conflicting transactions, unlike OCC that directly aborts them, TAPIR gives some conflicts a chance to re-commit. For example, in Fig. 3, T1’s write keys have been read by others. On receiving T1, R2 replies to re-commit it with a larger timestamp, which orders T1 after T2 and T3. But for T2 and T3, they are not eligible for re-committing and are eventually aborted.

Another way to resolve conflict is to order the transactions prior to their execution. For instance, Janus [30] and Carousel [38] adopt this approach. However, this requires that each transaction knows its read and write sets in advance, which is not generally applicable.

If we adopt leaderless protocol for multi-master transaction processing (Fig. 2(b)), it can eliminate the single-leader bottleneck, and enable low latency of one wide-area RTT (8 in Fig. 2(b)) when conflict-free. However, as illustrated by the example shown in Fig. 3, the different transaction orders seen by replicas will lead to inconsistent decisions, which may lead to a large number of aborts. Even though TAPIR tries to re-commit a transaction when conflicts occur, it still faces high abort rates. This is a fundamental limitation of a leaderless protocol in handling conflicts. Without a global view of conflicts, it is unable to perform precise conflict resolution.

2.3 Transaction Reordering in OCC

OCC has been widely adopted in recent database systems because it offers excellent performance when there is little conflict. However, studies [6, 17] showed that OCC performs poorly under the workload of high contention. To address this issue, some systems adopted the strategy of transaction reordering [8, 15, 29, 30].

For instance, Rococo [29] and Janus [30] reorder transactions based on their dependencies before execution and let each server executes transactions in the same order. Reordering before execution targets one-shot transactions that the write and read sets are known in advance. Post-execution reordering can get rid of such prerequisites by reordering transactions in batches in the validation phase [15]. This inspires us to design the conflict resolution strategy of Starrry.

Since Starrry aims to serve general-purpose transactions, the sequencer in Starrry performs post-execution reordering during the transaction commit phase. Recalling the example in Fig. 3, after the sequencer has collected three transactions, Starrry only aborts T3 to break the dependency cycle, and reorders T1 and T2 to be able to commit T1 and re-commit T2. This results in an optimal decision, that is hardly achievable with leaderless methods.

3 DESIGN OF STARRY

In this section, we describe Starrry in detail, including the process of the commit protocol, the technique for reordering conflicting transactions to reduce the abort rate, the recovery approach, as well as correctness guarantees.

<table>
<thead>
<tr>
<th>Table 2: Replica state and log structure in Starrry.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State on all Replicas</strong></td>
</tr>
<tr>
<td>counter - incremented counter for assigning timestamps</td>
</tr>
<tr>
<td>active list - log entries of all received active transactions</td>
</tr>
<tr>
<td>Log[[]] - a two-dimensional array to store transaction log entries</td>
</tr>
<tr>
<td><strong>State on Sequencer</strong></td>
</tr>
<tr>
<td>txm_graph - the dependency graph of conflicting transactions</td>
</tr>
<tr>
<td>results - decisions of conflicting transactions made by sequencer</td>
</tr>
<tr>
<td><strong>Log Entry Format</strong></td>
</tr>
<tr>
<td>ts - the commit timestamp of the transaction</td>
</tr>
<tr>
<td>wset - the write set of the transaction</td>
</tr>
<tr>
<td>rset - the read set of the transaction</td>
</tr>
</tbody>
</table>

3.1 Preliminaries

This subsection presents the basics for understanding the protocol of Starrry.

Roles of Replicas. The storage layer consists of 2F + 1 replicas (or storage servers) that can tolerate up to F non-Byzantine failures. All replicas act as normal replicas that can receive and process transaction requests. When a replica receives the commit request, it acts as the proposer to replicate and commit the transaction. Although each replica is able to process requests, only one replica, known as sequencer, is responsible for centrally resolving conflicts when they occur.

Transaction Order. Transactions are ordered according to the commit timestamp assigned by replicas. Since there is no synchronized clock across regions, we use the Lamport logical clock [22] to generate timestamps. Each replica maintains a local monotonically increasing counter C_i for generating logical timestamps, C_i is updated as follows:

- Each time replica R_i assigns a timestamp for a new committing transaction, C_i is incremented by 1.
- Each time R_i sends a message to other replicas, C_i is attached to the message.
- Each time R_i receives a message from R_j, C_i is updated to max(C_i, C_j).

First, such a logical timestamp can capture the happened before relations between transactions. In other words, the logical timestamps assigned by our system satisfy the constraint that the transaction order is consistent with the real-time order, which means, if t_i commits before t_j starts, then t_i’s timestamp is less than t_j’s. This is required by linearizability.

Second, both linearizability and serializability require a total order for global transactions, which means that the timestamp of each transaction must be unique. To this end, we set each logical timestamp as a tuple (C_i, R_i), in which R_i represents the id of the i-th replica, and C_i represents the counter on the replica. Timestamps are first ordered by counter values. If they have identical counter values, they are ordered by the replica ids. For instance, (2, 3) is less than (3, 1), and (3, 1) is less than (3, 2).

Transaction Lifecycle. When a computing node receives a transaction request from the client, the transaction enters the execute phase. The computing node reads data from the closest replica and caches the value into its private space for future reading. As
for write operations, new values is also stored locally. The commit phase starts after finishing execution. First, a new log entry, which contains the read and write sets of the transaction, is generated. Then, the computing node sends the new entry to the closest replica \( R \) to commit. \( R \) assigns the entry a logical timestamp as the commit timestamp, and adds it into the log array, then replicates it to other replicas to check if it can be committed. After replicas reach a consensus on the final result of the transaction, the result will be notified to the computing node, which will in turn respond to the client and end the transaction.

**Coping With Conflicts.** Conflicts can occur among concurrent transactions. Given two concurrent transactions \( \alpha \) and \( \beta \), their write and read sets are represented as \( \omega set \) and \( \rho set \) respectively, they conflict if one of the following conditions holds:

- Write-write conflict: if \( \omega set_\alpha \cap \omega set_\beta \neq \emptyset \), they are considered to have a write-write conflict (ww-conflict).
- Read-write conflict: if \( \omega set_\alpha \cap \rho set_\beta \neq \emptyset \), we say there is a read-write dependency from \( \beta \) to \( \alpha \). Because two concurrent transactions cannot see each other’s writes, \( \beta \) cannot be serialized after \( \alpha \). Therefore, if the commit timestamp is \( ts_\alpha < ts_\beta \), which means that the commit order is \( \alpha < \beta \), the two transactions are considered to have a read-write conflict (rw-conflict).

Each record in \textit{Starry} is attached with timestamps of the last transactions that update and read on it, represented by \textit{write}\_\textit{ts} and \textit{read}\_\textit{ts} respectively. According to the Thomas write rule [35], if the data has been modified by a transaction with a larger timestamp, we can safely ignore the write of an earlier transaction. Therefore, \textit{Starry} ignores \textit{ww}-conflicts, since it can always use \textit{write}\_\textit{ts} attached on each record to decide if a transaction’s update should take effect.

**States of Replicas.** As shown in Table 2, each replica maintains a series of metadata to record its state. First, it needs a \textit{counter} for generating logical timestamps. Second, it maintains an \textit{active list} which records all the transactions that are in the commit phase. For the sequencer, it also needs to store the conflicting transactions in a directed graph denoted by \textit{txn}\_\textit{graph}, in which vertices represent transactions and edges represent read-write dependencies. After resolving all conflicts in the graph (see details in § 3.4), the sequencer determines whether each transaction should commit, re-commit, or abort, and records decisions in the structure called \textit{results}.

Besides, all replicas maintain an array named \textit{Log} \([\square]\), where each instance in the log array recording the updates of a transaction. In \textit{Starry}, all replicas can act as a proposer to propose log entries. To avoid different replicas compete for the same position in a single log sequence, the log structure is designed as a two-dimensional array, each row in the array is dedicated to one replica. When receiving a new log entry, the replica simply appends it to its own log sequence, thus avoiding the competition. The format of a log entry is also shown in Table 2. It contains three variables \textit{ts}, \textit{wset} and \textit{rset}, which represent the commit timestamp of the transaction and its write set and read set respectively. The write set contains the keys of the updated records and their new values, and the read set contains the keys and the versions that have been read.

### 3.2 An Intuitive Example

In theory, if there is no conflict among transactions, they can be committed in one round trip of communication among replicas. If there is a conflict, extra rounds of communication are required to resolve it on the sequencer. Fig. 4 provides an example to illustrate how our commit protocol works. The detailed messages in Fig. 4 are described in Table 3.

**Example 3.1.** Example in Fig. 4(a) shows the commit process of non-conflicting transactions. \( L_1 \) and \( L_2 \) represent the log entries of transactions \( T_1 \) and \( T_2 \) respectively. \( L_1 \) is sent to \( R_0 \) and be assigned a timestamp of \( \langle 1, 0 \rangle \). Similarly, \( L_2 \) is sent to \( R_4 \) which assigns it a timestamp of \( \langle 1, 4 \rangle \). As \( T_1 \) and \( T_2 \) do not conflict, during replicating, \( R_1, R_2 \) and \( R_3 \) all reply \textit{pre-commit} to \( R_0 \) and \( R_4 \). Thus, both of them commit in a single round trip of communication. Then \( R_0 \) and \( R_4 \) notify other replicas of the commit decision.

The commit process of conflicting transactions is shown in Fig. 4(b). Suppose \( R_3 \) acts as the sequencer. \( T_3 \) and \( T_4 \) are \textit{rw}-conflict and their log entries are assigned the timestamps of \( \langle 2, 0 \rangle \) and \( \langle 2, 4 \rangle \).
respectively. As the replication messages arrive at each replica in different orders, the decisions made by the replicas are also different. As $R_1$ and $R_2$ (the sequencer) receive $L_3$ before $L_4$, they attempt to commit $L_3$ and reply pre-commit to $R_0$ and conf to $R_4$. On the contrary, $R_3$ attempts to commit $L_4$ and identifies $L_3$ as a conflict. Since both transactions are identified as conflicts by some replicas, they fail to commit in the first round of communication. Instead, they launch the second round of communication to ask the sequencer for conflict resolution. Therefore, $R_0$ and $R_4$ request $R_2$ to make the final decision. Based on the received conflict information (red dotted arrow in Fig. 4(b)), $R_2$ reorders two transactions (details in §3.4), then decides to commit $L_4$ and re-commit $L_3$ with timestamp $(3, 0)$. Please note that re-commit only needs to restart the commit phase, instead of re-executing the entire transaction. Through the conflict resolution on the sequencer, both transactions avoid aborting.

3.3 The Semi-leader Transaction Commit Protocol

Fig. 5 shows the complete protocol in the absence of failures. Once a replica $R$ receives the commit request of transaction $\alpha$ (denote as $T_\alpha$), it acts as the proposer (denote as $P$) and starts phase 1 to commit $T_\alpha$’s log entry (denote as $L_\alpha$).

**Phase 1: Replication and conflict detection.** In phase 1, $L_\alpha$ will be replicated to all replicas and be checked if conflict occurs. Phase 1 consists of the following 3 processes.

Process Θ: TryCommit. $P$ first verifies that if $L_\alpha$ conflict with other local transactions it has received. If not, $P$ assigns $L_\alpha$ a commit timestamp and adds $L_\alpha$ into its log array. Then, $P$ sends a RepTxn message to all replicas and waits for replies from a super quorum (set as $\lceil \frac{2}{3}F \rceil +1$, explained in §3.5.1). Note that, after sending out the RepTxn message, the proposer adds $L_\alpha$ to a pending list and continues to process other transactions. The whole process contains no blocking point.

Process Θ: HandleRepTxn. When a replica $R$ receives a RepTxn message, it applies Algorithm 1 to process the message. $R$ first adds $L_\alpha$ into its log array and calls the function OCC_Check to validate the transaction (lines 1-2). The validation works as follows:

1. A running transaction is deemed to read stale data if a record in its read set has been updated by a committed transaction (lines 1-2). If this occurs, the transaction has to be aborted.
2. A transaction will be re-committed with a new timestamp, if its write set has been read or overwritten by a committed transaction with a larger timestamp (lines 19-20). The new timestamp should be greater than that of the conflicting transaction, to ensure a correct order.
3. A transaction is identified as a conflict if it has an rw-conflict with an active transaction (line 14 and line 21).

If $L_\alpha$ is not identified as conflict (conf), replica $R$ directly replies to $P$ its intentions, which can be pre-commit, abort or re-commit (line 8); otherwise, $R$ will enumerate all transactions that rw-conflict with it, and add them as dependencies into $dep_\alpha$ (line 4). After that, $R$ sends the conflict information to the sequencer $S$ through a
Algorithm 2: HandleRepReply(status)

24 replies ← Union(status in all Reply)
25 if contains at least \( \lceil \frac{3}{2} F \rceil + 1 \) pre-commit then
26 \( L_\alpha.status \leftarrow \) pre-commit
27 reply commit to the computing node
28 else if abort ∈ replies then
29 \( L_\alpha.status \leftarrow \) abort
30 reply abort to the computing node
31 else if \( \langle \text{re-commit}.new_ts \rangle \in \) replies then
32 \( L_\alpha.ts \leftarrow \max(\text{new}_ts \text{ in replies})\)
33 resends RepTxn(\( L_\alpha \)) to all replicas
34 else
35 // identified as conflict to be resolved
36 dep_\alpha ← Union(dep_\alpha in all Reply)
37 send ReqDec(\( L_\alpha, dep_\alpha \)) to S

NotConf message (line 5), this is a faster way for the sequencer to be aware of conflict (the other way is ReqDec message sent by the proposer), which can help the sequencer collects conflicts in time. Then, \( R \) replies conf and \( dep_\alpha \) to \( P \) (line 6).

Process \( \emptyset \): HandleReply. After the proposer \( P \) receives replies from the majority, it checks if \( L_\alpha \) can pass the non-conflict path or goes into the conflict path (shown in Algorithm 2):

1. If \( P \) receives pre-commit from a super quorum, it is guaranteed that none of its conflicting transactions can pass the validation on a super quorum. Therefore, \( L_\alpha \) is directly committed through the non-conflict path (lines 25-27).
2. If abort exists in replies, it means that transaction \( a \) reads stale data. Therefore, \( L_\alpha \) must be aborted (lines 28-30).
3. If re-commit exists in replies, \( L_\alpha \) will restart Phase 1 with the new timestamp (line 31-33).

If \( L_\alpha \) is decided to commit or abort, \( P \) first replies the result to the computing node and then notifies other replicas of the decision.

In other situation, \( L_\alpha \) will turn to conflict path and enter the conflict resolution phase. \( P \) takes the union of all \( dep_\alpha \) and sends ReqDec message to the sequencer (lines 34-36), then waits asynchronously for the decision of \( L_\alpha \).

Phase 2: Conflict resolution. The sequencer maintains a conflict dependency graph (\( txn_graph \)). When receiving NotConf messages, it updates the graph accordingly. When reordering is triggered, the sequencer reorders the conflicting transactions and notifies all other replicas about the final decisions.

Process \( \emptyset \): HandleConflict. When the sequencer receives at least \( \lceil \frac{3}{2} F \rceil + 1 \) NotConf messages of \( L_\alpha \), it knows for sure that \( L_\alpha \) has not been committed on the non-conflict path. Then, \( S \) will add \( L_\alpha \) and \( dep_\alpha \) to \( txn_graph \).

Process \( \emptyset \): ResolveConflict. After the \( S \) receives a ReqDec message from a proposer \( P \), it unions the \( dep_\alpha \) collected on the proposer to gain a more complete view of conflicts. Then, it invokes the Reordering function (details in §3.4) to reorder conflicting transactions and decide the fate of each transaction.

1. If it decides to re-commit \( L_\alpha \), it does not need to notify other replicas, but directly sends a NotDec message to the proposer \( P \). Then, \( P \) updates \( L_\alpha \)'s timestamp as \( \text{new}_ts \) and restarts TryCommit.
2. If it decides to commit or abort \( L_\alpha \), it notifies other replicas about its decision. Each replica that receives the decision enters Process \( \emptyset \), which persists the decision of \( L_\alpha \) and route the decision to the proposer. If the proposer \( P \) receives at least \( F \) notifications, it enters Process \( \emptyset \) to finish the commit of \( L_\alpha \) and replies the result to the computing node.

After entries are committed, they will be applied in timestamp order, so that all updates in their write sets will take effect.

Latency analysis. The green arrows in Fig. 5 show the commit path of the non-conflicting transaction. After phase 1, if at least \( \lceil \frac{3}{2} F \rceil + 1 \) replicas reply pre-commit, the proposer can safely notify the computing node about the commit of the transaction, which will in turn respond to the client. In this case, it takes only one wide-area RTT to finish the transaction.

If conflict occurs, the proposer requests the sequencer for final decision and waits for notifications from \( F \) replicas. The whole process of phase 2 will take 1.5 wide-area RTTs (as shown by the red arrows in Fig. 5). In this case, the overall latency to finish a transaction will be 2.5 wide-area RTTs.

3.4 Conflict Resolution on Sequencer

Conflict resolution is the process of serializing conflicting transactions. For a given set of conflicting transactions (denote as \( S \)), we aim at finding a serially ordered subset \( C \), such that the complement set \( A = S \backslash C \) (the set of aborted transactions) is minimized. To select aborted transactions, we need to know the conflicting relationships among transactions, which are actually read-write dependencies (denoted as \( \text{rw-dependencies} \)). As shown in Fig. 6, the sequencer builds a graph (\(txn_graph\)), in which each node denotes a conflicting transaction and each edge denotes an \( \text{rw-dependency} \). If the graph is cyclic, some transactions must be aborted to break the dependency cycle. Then, a serial order can be found by topologically sorting the remaining transactions in the graph. If the conflicting transaction’s timestamp violates the serial order, the sequencer will assign a new timestamp to it. We call this operation reordering.

When receiving a ReqDec about a transaction, the sequencer will extract a subgraph to be reordered from the \(txn_graph\) (those can connect to the transaction and it can connect to). A transaction that has been committed on non-conflict path may also have been added to the \(txn_graph\) as the dependency of a transaction that requires conflict resolution. To ensure the uniqueness of the final decision on the two paths, the status of committed transaction cannot be changed by reordering. Therefore, before reordering, the sequencer must check the status of transactions to be reordered.

Specifically, if a transaction \( T_a \) has already committed on the non-conflict path (i.e., received the commit decision from its proposer), the sequencer will mark it as committed that cannot be changed during reordering. Then, its in-dependency \( T_b \) (\( T_a.write \cap T_b.read = \emptyset \)) will be aborted, since \( T_b \) does not see the new value written by \( T_a \). The aborted transactions will be removed from \(txn_graph\). \( T_a \)'s out-dependency \( T_c \) (\( T_a.read \cap T_c.write = \emptyset \)) will be assigned a larger timestamp and re-committed, so that \( T_c \) can be ordered after \( T_a \). For other pending conflicting transactions (i.e., received a ReqDec message or more than \( \lceil \frac{3}{2} F \rceil + 1 \) NotConf messages), the sequencer can safely reorder them. If a network failure
Algorithm 3: Reordering

\begin{verbatim}
37  SCC ← Tarjan(subgraph)
38  for component ∈ SCC do
39    if component.size > 1 then
40      e ← entry with the largest prod_degree
41      results[e] ← abort
42    remove e from the graph
43    sort ← TopologySort(remain entries in SCC)
44    new_ts ← 0
45    for e ← sort.front() do
46      if e.in_degree == 0 then
47        results[e] ← commit
48        new_ts ← max(new_ts, e.ts)
49      else
50        new_ts ← new_ts + 1
51        results[e] ← (re-commit, new_ts)
52    remove e from txn_graph
\end{verbatim}

occurs, which is rare, the sequencer fails to receive the commit decision or NotConf messages in time, and the status of a transaction can be uncertain. In this case, the sequencer will execute a status confirmation process to learn its status from other replicas.

The sequencer executes reordering as follows (the pseudocode is shown in Algorithm 3):

1. Because the dependency cycle must be contained in a strongly connected component (SCC), the sequencer divides the subgraph into SCCs by using the Tarjan SCC algorithm [34] (line 37).
2. For the SCC of more than one entry, it contains a dependency cycle. To break the cycle, sequencer chooses the entry with the largest prod_degree (product of in-degree and out-degree) to abort and removes it from the graph (lines 39-42).
3. After breaking the dependency cycle, the sequencer topologically sorts remaining entries and stores the results in an array named sort (line 43). After that, the sequencer traverses the entries in sort. For an entry that does not have in-dependency, the sequencer decides to commit it with the initial timestamp (lines 46-48); otherwise, the sequencer assigns it a new_ts, which is larger than all in-dependencies’ timestamps, and re-commits it (lines 50-51).

As conflicting transactions constantly arrive, the size of txn_graph continues to grow. The sequencer chooses to reorder a bunch of connected conflicting transactions when it receives the first ReqDec message. Following that, it removes them from the graph. Periodic reordering divides conflicting transactions into batches, which avoids the graph growing to a size that we cannot manage. As reordering can only resolve conflicts within a batch, transactions that conflict with previous batches have to be aborted.

Example 3.2. Fig. 6 shows a case of reordering. There are eight transactions in txn_graph, represented by T_1, T_2, ..., T_8 respectively, with timestamps of 1-8 (note that the replica id in timestamp is not shown here). Their dependencies are shown in (a). During reordering, the sequencer first divides the graph into four SCCs, as shown by the shading in (b). According to the algorithm, the transactions with the largest prod_degree, i.e., T_4 and T_6, are chosen to abort. After trimming the aborted transactions and their dependencies, the graph becomes the one in (c). Then, the sequencer topologically sorts the graph, and gets two new orders as {T_1 < T_5 < T_2 < T_1} and {T_8 < T_7}. As T_3, T_5 and T_8 do not have in-dependencies, their timestamps remain unchanged and can be committed. In contrast, T_2, T_1 and T_7 are assigned new timestamps of 6, 7 and 9 respectively and to be re-committed later. The new commit order after reordering is shown in (d).

3.5 Failure Recovery

Node failure is inevitable in distributed systems. We designed a recovery approach to ensure fault tolerance. In the following, we describe how to handle the failures of normal replicas and the sequencer respectively.

3.5.1 Normal Replica Failure. When a normal replica fails, its proposed transaction commits requires another replica to take over. Since more than one replica may detect the replica failure and take over its entries, this will cause confusion, thus we only allow the sequencer to handle normal replica failures. Therefore, when an active replica times out when waiting for the result of L_{α}, they will notify the sequencer. The sequencer then starts the recovery phase by sending Recovery(L_{α}) to other replicas (including itself) and waits for at least \( F + 1 \) replicas reply the status of L_{α}. After that, the sequencer determines the status of L_{α} according to the following rules:

1. If any replica replies with the final result of L_{α}, the sequencer will choose the result and sync it to other replicas. Then, it waits for acks from F replicas to end the process of L_{α}.
2. If no one has received the finalized result, and less than \( \lceil \frac{F}{2} \rceil + 1 \) replicas respond pre-commit, L_{α} cannot pass non-conflict path, the sequencer can safely abort it. If at least \( \lceil \frac{F}{2} \rceil + 1 \) replicas respond pre-commit, in case that L_{α} has passed the non-conflict path and has been replied to the client, the sequencer will commit it and abort all others that conflict with L_{α}.

Assuming that T_1 commits on the non-conflict path, and then F replicas fail. If the sequencer wants to recover T_1, we must ensure that in the remaining F + 1 replicas, there are still a majority of
replicas that decide to reply pre-commit. To guarantee this, \(T_1\) needs to receive \(\left\lceil \frac{2}{3}F \right\rceil + 1\) pre-commit replies to pass the non-conflict path, which is why the super quorum is introduced.

3.5.2 Sequencer Failure. In the event of a sequencer failure, a new sequencer should be elected to continue to resolve conflicts. The states of the failed sequencer to be recovered include its undetermined log entries and \(txn\_graph\). The former can be handled in the same way as a normal replica failure after the new sequencer can provide services. Therefore, the focus of recovering the sequencer is on reconstructing \(txn\_graph\).

As what Raft[31] does, we call the periods hosted by different sequencers as terms. Only one replica can act as the sequencer in one term. Each replica maintains a current term, which contains a monotonically increasing term number and the sequencer id. The term is attached to every message between replicas. Once a replica finds a higher term number, it requests the majority of replicas for the newest term. The process of sequencer recovery can be divided into two phases:

(i) Sequencer election. When a replica times out waiting for the response from the sequencer, it will increment its term number and becomes a candidate. Then it sends the RequestVote message with the term number to others. On receiving RequestVote message with a higher term number, a replica will reply a vote to the candidate. Each replica can vote for at most one candidate in a given term. The vote also piggybacks all undetermined conflicting entries’ information, which can help the sequencer to reconstruct \(txn\_graph\). After receiving votes from more than \(F\) replicas, the candidate becomes the new sequencer. It then notifies other replicas of the new term number and sequencer id.

(ii) Dealing with undetermined conflicts. The key requirement of sequencer recovery is that the new sequencer cannot change the commit and abort decisions made by the last sequencer, because those may have been replied to the client. To ensure this, new sequencer collects all conflicting entries that are attached to votes as a set \(C\), then asks other replicas for the status of entries in \(C\) and waits for replies from a majority. If an entry has been decided by the last sequencer, the new sequencer accepts the decision. For those entries which have not seen the decision in replies, they will be added into \(txn\_graph\) to be decided later by reordering.

3.6 Correctness

Similar to other Paxos variants, STARRY guarantees the properties of non-triviality, linearizability and fault tolerance for safety.

Non-triviality. Non-triviality requires that the committed transaction must be proposed by a client rather than predetermined transactions. As replicas only accept transaction logs from computing nodes, which is the result of transaction requests from clients, this property is naturally satisfied.

Linearizability. Linearizability can be further decomposed into two properties: (1) all replicas apply the same entries in the same order; (2) the order is consistent with the real-time order.

Property (1) means that each log entry should be placed in the same slot of log arrays in all replicas, and the timestamp of a committed log entry is consistent among replicas. As each proposer in STARRY owns an exclusive row in the log array, the slot of each log entry can be uniquely determined. Besides, as the timestamp of a log entry is determined either by only the proposer or the sequencer, it must be unique and consistent in the entire system. Thus we can guarantee that all replicas apply the same committed entries in the same order.

Property (2) means that if two transactions \(\alpha\) and \(\beta\) operate on the same data, and \(\beta\) is proposed after \(\alpha\) is committed, \(\alpha\) must be executed before \(\beta\). According to our protocol, after \(\alpha\) commits, the set of replicas that have seen \(\alpha\) (denote as \(S\)) contains at least \(F + 1\) replicas. If \(\beta\) is proposed by \(R\) and \(R \in S\), the proposed timestamp of \(\beta\) must be larger than \(\alpha\)’s commit timestamp. Thus, \(\alpha\) will be ordered before \(\beta\). If \(R \not\in S\), which means that \(R\) falls behind, \(\beta\) can be assigned a timestamp that is smaller than \(\alpha\). In this situation, \(\beta\) cannot commit before \(\alpha\) in the following three possible cases:

(i) \(\alpha\).write ∩ \(\beta\).read ≠ \(\emptyset\). In this case, as majority replicas have already committed \(\alpha\), \(\beta\) will be aborted when other replicas run OCC_Check.

(ii) \(\alpha\).read ∩ \(\beta\).write ≠ \(\emptyset\). In this case, \(\beta\) will be assigned a new timestamp that is larger than \(\alpha\) during OCC_Check (line 19-20 in Algorithm 1). Thus, \(\beta\) can only commit after \(\alpha\).

(iii) \(\alpha\).write ∩ \(\beta\).write ≠ \(\emptyset\). Similar to Case (ii), \(\beta\) will be re-assigned a larger timestamp and ordered after \(\alpha\).

In summary, regardless of the relationship between \(\alpha\) and \(\beta\), if \(\beta\) can be committed, the commit order is always \(\alpha < \beta\). Because the log entries are applied according to the transaction commit order, thus the operation of \(\beta\) is always executed after \(\alpha\), which ensures the real-time order.

Fault tolerance. The recovery protocol must ensure that no committed transaction is lost after a failure. For a transaction \(T_1\) that has committed on non-conflict path, at least \(\left\lceil \frac{2}{3}F \right\rceil + 1\) replicas have replied pre-commit. Even if \(F\) replicas fail, the sequencer can still receive at least \(\left\lceil \frac{2}{3} \right\rceil + 1\) pre-commit replies when recovering \(T_1\). According the rule ③ in §3.5.1, sequencer will eventually commit it. For a transaction \(T_2\) that has committed on the conflict path, the commit decision must have been stored on \(F + 1\) replicas. Therefore, during recovery, the sequencer will see the commit decision on at least one replica, which ensures that \(T_2\) can be recovered. In summary, the recovery protocol can make sure no committed transaction will be lost.

Serializability. STARRY also guarantees serializability among transactions. In STARRY, once a transaction \(\alpha\) is committed, no other concurrent transactions that conflict with it, say \(\beta\), can be committed. No matter whether \(\alpha\) is committed on non-conflict path or conflict path, its status is already stored on the majority of nodes. Therefore, \(\beta\) cannot get enough pre-commit from a super quorum and be committed on the non-conflict path. If \(\beta\) enters the conflict path, it can only be aborted or reordered after \(\alpha\) by the sequencer. In both two cases, \(\beta\) no longer conflicts with \(\alpha\).

4 EXTENSIONS

STARRY also adopts measures to further optimize the performance of distributed transactions and read-only transactions.

4.1 Distributed Transactions

To handle continuously growing data volumes, a distributed database system usually partition the storage into shards to gain scalability. So does a cloud database with disaggregated storage. Distributed transaction processing requires an atomic commitment protocol, such as two-phase commit (2PC), to ensure atomicity.
In 2PC, a coordinator is in charge of collecting votes from all participants to decide whether a transaction should commit or abort. In STARRY, the computing node works as a coordinator. Before starting to commit, the computing node acquires the current timestamp from each participant (a shard), and chooses the largest one as the timestamp of the entire transaction. Since the coordination of timestamp only interacts with local replicas, it does not hurt the latency. In the first phase of 2PC, each participant shard makes the decision according to the protocol in § 3.3. If all participant shards reply pre-commit or anyone replies abort, the computing node decides to commit or abort the transaction respectively, then the notifications to all participants are asynchronous. Besides, a participant can also reply re-commit. After receiving all re-commit messages, the computing node will choose the largest re-commit timestamp as the new timestamp and restart the commit phase.

With the high availability design in the storage layer, STARRY can avoid the blocking problem caused by coordinator failure [7, 32]. It is unnecessary to resort to complicated solutions such as three-phase commit [32] and Paxos commit [16]. When a participant times out waiting for the coordinator’s decision, it will execute the termination protocol to check the transaction status on other participants to learn the final decision, and then terminate the transaction. Since each participant’s decision is already stored on at least $F+1$ servers, the final decisions can always be found as long as less than $F$ replicas fail. Similar to the approaches in [19, 39], STARRY does not allow the coordinator to abort transactions unilaterally (if all participants decided to commit a transaction, the coordinator must commit it). This prevents the termination protocol from making a decision that is different from the one made by the failed coordinator.

### 4.2 Read-only Transactions

Read-only (RO) transactions are common in practice [26]. STARRY can treat RO transactions and read/write (RW) transactions equally to achieve strict-serializability. When a computing node executes a RO transaction, it replicates it to other replicas to detect conflict. Once a RO transaction commits, it can be ensured that no other concurrent transactions which update the read set commit before the transaction. Therefore, a RO transaction only reads the most up-to-date values, which guarantees strict-serializability.

As RO transactions do not perform updates, it is unnecessary for them to conduct replication. In most applications, read operations are latency-sensitive. This motivates us to skip as much cross-region communication as possible in RO transactions. Therefore, we extended STARRY to support the consistency level of process-ordered serializability (POS) [27] for RO transactions. POS only requires that each replica executes RW transactions in the same order, without demanding that RO transactions follow the real-time order. This allows RO transactions to read out-of-date data.

When relaxing the consistency level to POS, RO transactions can be served in the local replica without cross-region communication, which significantly reduces the transaction latency. For single-shard RO transactions, there is no need for coordinating a read timestamp among multiple shards. The computing node can directly read the local replica. As to cross-shard RO transactions, we must ensure consistent reads on multiple shards. This requires all shards to agree on a read timestamp before execution. Therefore, STARRY only allows transactions whose read set can be predetermined to run at the POS level. In POS, the computing node first asks each involved shard for the latest write ts of each read key in a RO transaction, and chooses the largest one as the read timestamp for all reads. Then, the computing node directly reads data on each involved shard’s local replica.

### Table 4: Network latency between data centers (ms).

<table>
<thead>
<tr>
<th></th>
<th>Shanghai</th>
<th>San Francisco</th>
<th>Frankfurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shanghai</td>
<td>0.3</td>
<td>140</td>
<td>231</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.3</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>Frankfurt</td>
<td></td>
<td>0.25</td>
<td></td>
</tr>
</tbody>
</table>

5 **EVALUATION**

5.1 **Experimental Setup**

5.1.1 **The Testbeds.** We conducted experiments on both local cluster and cross-region cloud servers. The local cluster was deployed on servers with two Intel Xeon Silver 4110 processors with 32 cores and 196 GB RAM. We configured the system as 5 shards. Each shard had 3-9 replicas. One server ran a replica of each shard or multiple computing nodes.

Experiments on the cloud servers were conducted on Alibaba Cloud ECSs instances across three data centers: Asia (Shanghai), US West (San Francisco), and Europe (Frankfurt). The network latencies between data centers are shown in Table 4. Each instance in the experiments had 2 virtual CPU cores and 8 GB of memory. We set the configuration to 3 shards, each with 3 replicas. This requires a total of 9 servers used as storage servers. In each data center, only one replica of a shard was deployed. Therefore, the setup can tolerate data center failures.

#### Table 5: Transaction profile for Retwis workload.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th># gets</th>
<th># puts</th>
<th>workload%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add User</td>
<td>1</td>
<td>3</td>
<td>5%</td>
</tr>
<tr>
<td>Follow/Unfollow</td>
<td>2</td>
<td>2</td>
<td>15%</td>
</tr>
<tr>
<td>Post Tweet</td>
<td>3</td>
<td>5</td>
<td>30%</td>
</tr>
<tr>
<td>Load Timeline</td>
<td>rand(1,10)</td>
<td>0</td>
<td>50%</td>
</tr>
</tbody>
</table>
5.1.3 Workload. We used two workloads for evaluation. The first one was a synthetic workload of Retwis application, which simulates Twitter’s functionality. The workload of Retwis contains 4 types of transactions as shown in Table 5, each accessing 4-10 data items across 2-3 shards on average. The second workload was YCSB+T, which is the extension of YCSB to support transactions and has been widely used for evaluating NoSQL databases. The evaluation of TAPIR also used them as the main workload.

5.2 Performance on the Local Cluster

We first ran experiments on the local cluster to demonstrate the performance improvement made by STARRY. The workload adopted was Retwis. Its Zipf coefficient was set to 0.7.

We evaluated the performance of the protocols on single-shard (Fig. 7(a) and (b)) and multi-shard (Fig. 7(c) and (d)) deployments respectively. It can be seen that STARRY always performed the best. In the single shard deployment, STARRY could scale to 10 computing nodes and achieve the peak throughput of 8258 tps, which is 1.4x and 3.42x as high as the peak throughputs of TAPIR and Raft-based respectively. When we partitioned data into 5 shards, STARRY could scale to 20 computing nodes and achieve 1.33× the throughput of TAPIR and 4.21× that of Raft-based.

STARRY and TAPIR were more scalable than Raft-based, because they could make use of the computation resources of all replicas. The performance gap between STARRY and TAPIR can be attributed to their differences in conflict resolution. Benefiting from the centralized reordering strategy, STARRY could avoid a large fraction of unnecessary aborts. As shown in Fig. 7(b) and (d), STARRY managed to reduce the abort rate by more than 60%.

We also varied the number of replicas per shard to see how the size of consensus group impacts performance. The results are presented in Fig. 7(e). We can see that when the replica number was less than 7, more replicas could contribute more computation resources to STARRY and TAPIR, thus gaining higher throughputs. However, when there were more than 7 replicas, the throughputs started to decline. This is expected, as each replica needs to send at least \( O(n) \) \((n\) is the replica number\) network messages in each round of consensus. When the number of replicas reaches a certain threshold, the network could be saturated, so that further increase of this number would only hurt the overall throughput. As for Raft-based, more replicas would burden the leader, which means that increasing the replica number always hurts performance.

5.3 Performance under Contention

We varied the degree of contention by adjusting the coefficient of the Zipf distribution, to see how the performance is affected by contention. The experiments were conducted on the local cluster using the Retwis workload. The number of computing nodes was fixed to 10.

Fig. 8(a) and (b) show the throughputs and abort rates with different Zipf coefficients. We excluded the cases where the Zipf coefficient is less than 0.5, as contention is rare in these cases. When the Zipf coefficients were greater than 0.5, the abort rates of Raft-based and TAPIR increased sharply, causing the throughputs to drop quickly. As STARRY could reorder conflicting transactions to minimize the abort rate, its abort rate was stable before the Zipf coefficient reached 0.7. Even after it passed 0.7, STARRY was subject to fewer aborts than TAPIR (by up to 70%). The throughput of STARRY was 1.43× and 1.5× as good as that of TAPIR when the Zipf coefficient was set to 0.8 and 0.9 respectively. When the Zipf factor was greater than 0.9, STARRY’s abort rate also rose rapidly, because transaction reordering became almost ineffective under such a high degree of contention.

Fig. 8(c) shows the impact of reordering on the sequencer’s throughput. As the Zipf coefficient increased, the performance gap between the sequencer and a normal replica widened. This can be attributed to the growing workload of reordering on the sequencer, as depicted in Table 6. We also noticed that the batch size for reordering rose quickly at Zipf=0.9 in Table 6, which explains the rise of the abort rates in Fig. 8(b). When resolving a large batch of conflicts, a large number of transactions can be forced to re-commit, they may be identified as conflict again. This may cause
5.4 Performance on the Cross-Region Cloud

Our third set of experiments were conducted on the cross-region cloud of Alibaba ECSs. For Raft-based and Starry, we set the leader and sequencer to be in US West by default. The workload we adopted was YCSB Workload A (50% write and 50% read), in which each transaction contained 4 operations.

Fig. 10(a) shows the average latency of transactions at different throughputs. We varied the number of computing nodes to adjust the workload. When the load was low, Starry and TAPIR were 50% faster than Raft-based in latency, because they could commit transactions in a single wide-area round-trip. In contrast, Raft-based needs more than two if the computing node is not co-located with the leader. As the load increased, Starry showed its advantages. Because its abort rate was smaller, its throughput was 1.25× as high as that of TAPIR.

Fig. 10(b) and (c) show the performance under contention. We can see that Starry performed the best. Its abort rate was 60% of TAPIR, which helped it achieve better throughput. The abort rate of Raft-based was the highest because its long-duration transactions caused more conflicts. We also compared the latencies between Starry and TAPIR. As shown in Fig. 10(d), the median latencies of them were close, while Starry’s tail latency was higher than TAPIR’s under high-contention load. This is because conflict resolution on Starry requires 2.5 RTTs, while TAPIR takes only 2 RTTs.

5.5 Performance of Different Consistency Levels

We evaluated the performance of Starry at two consistency levels - strict serializable (SS) and process-ordered serializable (POS). The experiments were conducted on the cross-region cloud. The workload adopted was YCSB Workload B, which is a read-heavy workload (5% write and 95% read).

Fig. 11 shows the performance difference. In POS, RO transactions are not required to read the up-to-date values. Thus, RO transactions can be served at the closest replica without communicating with remote servers. The elimination of cross-region communication significantly reduced the latency. As shown in Fig. 11(b), under a read-heavy workload, POS achieved the average latency of 12-19 ms. In contrast, transactions at SS needed hundreds of milliseconds to commit, because SS requires RO transactions to conduct replication for detecting conflicts. The low latency of POS also helped in improving its throughput, which was 10× higher than that of SS, as shown in Fig. 11(a).

6 CONCLUSION AND FUTURE WORK

This paper proposed the semi-leader transaction commit protocol. The key insight is that the combination of decentralized transaction processing and centralized conflict resolution can boost the performance of multi-master transaction processing. Compared to the pure centralized approaches, it can significantly improve the performance in terms of scalability and latency in a cross-region setup. Compared to the pure decentralized approaches, it is significantly more robust against contention. Based on the semi-leader protocol, we designed Starry, a mechanism of multi-master transaction processing for typical cloud database architecture, with disaggregated storage and computation layers. Our experimental study demonstrated its promising characteristics. In the future, we plan to further evaluate its practicality in real-world cloud databases, and explore ways (e.g., caching and sharding on the computing layer) to improve its applicability to a variety of real-world workloads.

ACKNOWLEDGMENTS

This work was sponsored by the National Science Foundation of China under grant number 61772202. It was also sponsored by CCF-Huawei Database System Innovation Research Plan.
REFERENCES


