Relay: High-performance Transactions in Heterogeneous Networks via Consistency Tiering

Abstract

Global enterprises deploy databases across multiple regions for high availability, strong scalability, and efficient service localization. However, supporting serializable cross-region transactions is significantly challenging in such databases, and they often degrade the performance of the whole system. Existing works optimize the coordination cost by cutting down the number of WAN round-trips but still incur cross-region latency for conflict transactions.

This paper contends that monolithic serializable consistency models are not well-tailored for multi-region systems. As cross-region transactions fundamentally experience significantly higher latency than intra-region transactions, when a cross-region transaction conflicts with an intra-region transaction, it unavoidably leads to a long-time head-of-line blocking for pessimistic protocols or incurs a significantly higher abort rate for optimistic protocols.

In response, we propose a new approach called *consistency tiering*, which enables different consistency guarantees for different transactions based on their characteristics. We derive Relay, a tired consistency model for multi-regions, which provides linearizability for intra-region transactions and regularity for cross-region transactions. Both of them provide serializability for isolation. We then design and implement two prototypes based on Spanner and CockroachDB. The experimental results show that Relay greatly outperforms monolithic serializable models on many workloads.

1 Introduction

Nowadays, cloud providers (e.g., AWS [5], Azure [6], and Google Cloud [11]) host computing infrastructures across multiple geographic regions. As a result, multi-region deployment has emerged as a prominent choice for cloud-native applications in pursuit of short latency, high availability, and strong scalability [10, 18, 46, 50, 57]. For scalability, these applications typically use a multi-region database as their backend and partition the database into multiple shards. For availability, each shard is replicated into multiple regions.

Figure 1 shows a typical multi-region deployment model. Each shard has a primary replica (dark-colored) and several secondary replicas. Generally, the primary replica is deployed in the region where the majority of client requests originate to ensure good data locality and short latency.

Supporting serializable ACID transactions in such a deployment model presents significant challenges. Coordinating a cross-region transaction (for short, CRT) is always slow due to geographic distances. To optimize the performance of

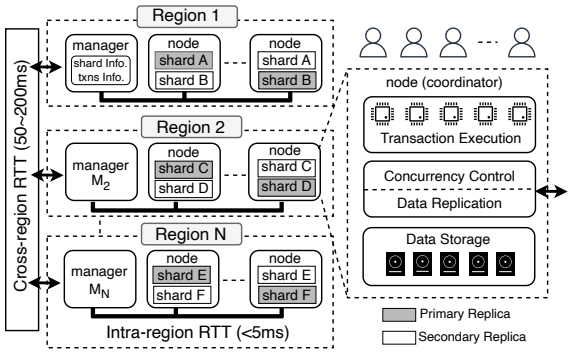


Figure 1. A typical deployment model for multi-region systems. The system store is partitioned into multiple shards spanning multiple regions. Each shard consists of a primary replica and several secondary replicas. Intra-region network latency is significantly shorter than inter-region one.

CRTs, several impressive works (e.g., [18, 40, 41, 44, 48]) have been proposed. We classify these works into two categories.

The first category optimizes the coordination cost of CRTs by cutting down the number of WAN round trips. A representative research line is deterministic concurrency controls [36, 41, 44, 48], which avoid expensive commit and replication protocols by eliminating nondeterministic race conditions. However, existing works either require a predetermined read/write set or rely on serializability checks in the commit phase (e.g., Aria [36]). The second category eliminates CRTs by making certain assumptions about the workloads [33, 60]. For example, Leap [33] always migrate the primary replicas from the remote region to the local before executing a CRT. Its performance highly counts on the assumption that CRT is the exception rather than a norm.

We emphasize that CRTs remain essential for general workloads, such as those with limited prior knowledge of application semantics or those employing interactive transactions. Moreover, the performance disparity between CRTs and intra-region transactions (for short, IRTs) is still significant even with the optimizations proposed in the first category. For instance, Detock [41], a state-of-the-art geo-distributed transaction protocol, can execute and commit CRTs in a single cross-region network communication. However, the average latency of CRTs is still $\sim 100ms$, which is significantly larger than IRTs' ($\sim 5ms$) under default experimental setups. Using the default consistency model (e.g., strict serializability) for IRTs and CRTs, even a few slow CRTs can entangle numerous IRTs, leading to deadlocks or aborts. This substantially degrades the overall database performance.

We show an experimental study in Figure 2, which is also validated by multiple real-world studies [10, 18, 41].

Diverging from existing works, we address multi-region transactions by ***consistency tiering***. We contend that monolithic serializable consistency models are inadequately designed for multi-region deployment. The primary issue arises from the inherent heterogeneity introduced by multi-region deployment in various transaction types. As data is closely associated with their respective home region through primary replicas, data access costs vary for transactions originating from distinct regions. To match the heterogeneity, a diverse range of consistency guarantees should be available for different transaction types while upholding serializability and as strong as possible consistency in requisite scenarios.

Monolithic consistency models tend to be either overly strong for CRTs or can be further enhanced without performance degradation for IRTs. For instance, the strict serializability model [42] abstracts the entire system as a single node, necessitating heavy synchronizations between all computing servers. As a result, this model fails to preserve the advantages of near-client computing: even an IRT has to be ordered with CRTs. When a CRT conflicts with an IRT, it unavoidably leads to a long-time head-of-line blocking for pessimistic protocols or incurs a significantly higher abort rate for optimistic protocols.

On the other hand, weak consistency models (e.g., serializable snapshot isolation [15]) tune down the consistency guarantees for all types of transactions, hurting the applicability and usability of the model. By consistency tiering, we found that the weak consistency guarantees for local transactions can be further enhanced with a neglectable performance overhead because the communication cost for coordinating IRTs can be cheap when using modern hardware. Besides, we also found that multi-region applications indeed show desirability for enforcing stronger consistency for local transactions, which is in cord with service localization of global enterprises (to be illustrated in §2.3).

By adopting consistency tiering for multi-region applications, we derive regional linearizable serializability (for short, Relay), the first tiered consistency model to provide as strong as possible consistency for both IRTs and CRTs. Relay ensures strict serializability (i.e., the strongest consistency guarantees) for IRTs from the same region and provides regular serializability for CRTs. We define Relay in §3.

To demonstrate the efficiency and applicability of Relay, instead of creating a new transaction protocol from scratch, we design, implement, and evaluate two variations of two notable industrial systems: Spanner and CockroachDB (for short, CRDB). We name the derived variants as Spanner-Relay and CRDB-Relay, respectively. We chose these two database systems because they complement each other in both the consistency model and the design of concurrency control protocols. For instance, Spanner adopts strict serializability, which is stronger than Spanner-Relay. CRDB adopts

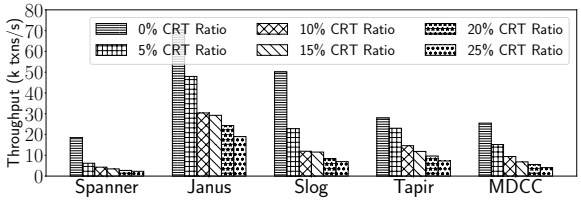


Figure 2. Impact of CRT ratio on the throughput of the state-of-the-art geo-distributed transaction systems.

single-key linearizability, which is weaker than CRDB-Relay. Spanner follows conventional two-phase locking (i.e., a pessimistic concurrency control protocol), while CRDB implements hybrid-logical clocks for concurrency control (i.e., a timestamp ordering protocol). We believe these two proof-of-concept prototypes can pave the way for adopting Relay in practical distributed transaction protocols based on a correct-by-construction approach.

Contributions. Our contributions are three-fold:

- We systematically analyze the multi-region deployment and stem an insight that monolithic consistency models may not be desirable for heterogeneous networks.
- We propose Relay, the first tailored consistency model for multi-region transactional processing.
- We designed, implemented, and evaluated Spanner-Relay and CRDB-Relay. The evaluation shows Relay improves Spanner's throughput by up to 31× under high conflict scenarios and provides robust consistency guarantees for CRDB with minor performance degradation.

The rest of the paper is organized as follows. §2 discusses our system model, background, and motivating applications. §3 details Relay. §4 and §5 delves two prototypes that implements Relay: Spanner-Relay and CRDB-Relay. §6 discusses related works, and §7 concludes the paper.

2 Background and Motivation

2.1 System Model and Heterogeneous Network

We show a typical multi-region system model in Figure 1. The database is partitioned into multiple data shards spanning over multiple regions. Each shard comprises a primary replica and several secondary replicas. Replicas can be configured to reside in cross-region or intra-region nodes (servers) based on replication policies. Each region has a centralized transaction manager responsible for globally consistent metadata (e.g., table schema, data placement policy, and globally unique transaction IDs). Nodes can communicate with each other over the network. Our assumptions include a partially synchronized network, and every message in the database is eventually delivered and processed.

The deployment model shows strong heterogeneity in the network between different nodes (both in the same region or from different regions). For example, a cross-region network (also known as a wide-area network, WAN) round trip can

Systems.	Transaction Protocol.	Consistency Models.	Coordination Blocking.
Spanner [12]	read-write transaction: 2PL + 2PC read-only transaction: timestamp ordering	strict serializability	Yes
Calvin [49]	centralized coordinator	strict serializability	Yes
Slog [44]	IRT: intra-region sequencer CRT: centralized coordinator	strict serializability	Yes
Detock [41]	IRT: intra-region Sequencer CRT: dependency-graph	strict serializability	Yes
Janus [40]	dependency-graph	strict serializability	Yes
Epaxos [39]	dependency-graph	strict serializability	Yes
Ocean Vista [18]	timestamp ordering (watermark)	strict serializability	Yes
CRDB [46]	timestamp ordering (HLC)	Single-Key linearizability	Yes
RedT [59]	2PL + 2PC	serializable snapshot isolation	Yes
Tapir [58]	variant of OCC	serializable snapshot isolation	No (by aborting IRTs)
MDCC [28]	Paxos	snapshot isolation	Yes

Table 1. This table summarizes the state-of-the-art geo-distributed transaction systems in the literature. These existing systems either block IRTs or enforce the IRTs to abort when the IRTs conflict with an ongoing CRT.

incur a $\sim 53.25ms$ network delay from US East (Ohio) to US West (N. California) and a $\sim 152.52ms$ network delay from US West (N. California) to EU (Frankfurt). On the contrary, the network delay within a region is normally less than $2ms$ [5]. Moreover, according to real-world statistics [59], cross-region networks can also be less stable than intra-region ones. The heterogeneous nature poses significant challenges when designing a distributed system and, in turn, motivates us to rethink the existing consistency models.

2.2 Geo-distributed Transaction Protocols

Many influential works have been proposed to optimize the performance of geo-distributed transaction processing (see Table 1). All of these systems support serializability for isolation or can be enhanced to be serializable with minor modifications (e.g., MDCC).

Generally, these systems focus on optimizing the cost of CRTs by redesigning several aspects of concurrency control protocols. Among them, there are some works that attempt to reduce the number of WAN round-trips in transaction coordination. For instance, RedT [59] employs a pre-write-log mechanism to eliminate the synchronization of `prepare` messages (the first phase in two-phase commits) from the coordinator to primary replicas. Calvin[49], Slog [44], Janus[40], Detock [41], Epaxos [39], MDCC [28], and Ocean Vista [18] adopts deterministic concurrency control protocols to reduce the execution cost of CRTs in WAN, which logically create a global log containing all transactions that have been input into the system. The system then ensures a concurrent execution schedule equivalent to processing all transactions serially in the order they appear in this log (i.e., a partial order). Consequently, after the transaction order is determined, the execution of both CRTs and IRTs can be local. The executors adhere to orders in the logs they receive.

However, all these proposals can not essentially prevent an IRT from being blocked by CRTs. Briefly, Spanner and RedT employ two-phase locking for transaction ordering and commit transactions using two-phase commit. When an IRT conflicts with an ongoing CRT, the IRT has to be blocked for the required locks. Tapir uses a variation of OCC and enforces the IRT to abort in the validation phase, resulting in a high abort rate (as confirmed by other previous papers [10, 18]). Determinisct databases either order IRTs and CRTs together (e.g., Calvin, Janus, Epaxos, and Ocean Vista) or require an IRT to be blocked, waiting for the execution of CRTs that are scheduled ahead (e.g., Slog and Detock).

Consequently, these systems can still cause severe performance issues when CRTs occur in the database, and the contention between the IRTs and CRTs is relatively high. We experimentally studied the impact of CRT ratios on the five latest representative systems in Figure 2. We used YCSB-T workloads with a Zipf parameter of 0.8. From the experimental results, our key observation is that even a few CRTs can significantly degrade the whole system's performance (e.g., up to 86% degradation with only 5% CRTs).

Relay can potentially address such issues by consistency tiering and thus is fundamentally different from these proposals. Relay trades off consistency for performance with minimal intrusion (i.e., the consistency tradeoff in Relay is tightly necessary for addressing blocking issues).

2.3 Motivating Applications

Several popular multi-region applications favor stronger consistency for local data accesses while tolerating weaker consistency guarantees for cross-region operations.

Financial Services. Financial transactions within the same banking institution or localized service require strong consistency to ensure the integrity of accounts and transactions [12, 16, 35]. However, weaker consistency could be used for non-critical, read-heavy operations like displaying transaction history or account summaries over WAN [8, 26].

Content Platforms. For operations like posting updates or comments where immediate consistency is essential to the user experience, strong consistency is preferred [9]. Yet, weaker consistency models are desirable to enhance performance and scalability for distributing content like feeds or recommendations across a global audience [7].

Gaming. In multiplayer online games, actions within the same game server or instance (e.g., trading items, player interactions) require strong consistency. However, weaker consistency should be sufficient for leaderboards or achievements that are updated less frequently and viewed by players across different regions [37, 45, 53].

Collaboration Tools. Strong consistency ensures all participants have the same view for real-time collaborative editing within a small group. However, a weaker consistency model can be adopted for syncing documents between groups across globally distributed data centers [30].

3 Regional Linearizable Serializability

In this section, we formally define Relay. For comparisons with other models, we refer readers to §6.1.

3.1 Definition of Relay

We adopted the formalism from existing works [23] for clarity. Table 2 summarizes the notations. Without loss of generality, we consider an OLTP service (either a relational database or a transactional key-value store) handling data objects identified by unique keys. We use $\mathcal{K}$ to represent the global key spaces. $\mathcal{K}$ is divided into multiple disjoint *shards* to facilitate scalable transaction processing.

Shard Groups (e.g., grouped by regions). Grouping semantics is one of the foremost concepts of Relay, which fundamentally distinguishes it from other monolithic consistency models. Specifically, Relay divides data shards into disjoint groups (known as regions in practice). Then, Relay ensures linearizability for intra-group operations and provides regularity for inter-group operations.

Note that grouping and sharding construct a two-level division of the global key space ($\mathcal{K}$): the OLTP service has many disjoint groups, and each group contains a number of (not necessarily equivalent) disjoint shards. These two divisions serve different purposes; sharding is for horizontally scaling the service to run on many servers, and thus, the division policy is usually designed for reducing the ratio of cross-shard transactions to achieve high efficiency [41]. On the other hand, the shard group is a consistency strategy where cross-group external ordering requirements are typically less important (see our motivating example in §2.3).

This two-level framework is essential for achieving high-performance transactions in heterogeneous networks because it efficiently bridges the division based on application semantics (e.g., the data items grouped by warehouse ID in TPC-C [13]) and division based on deployment topology (e.g., the data shards grouped by regions).

Type	Symbol	Description
Ops	o	A database operation, e.g., read, write, insert, scan, etc.
	$r(k, v)$	Read the value v using key k
	$w(k, v)$	Write value v for key k
	Σ_T	Operations of transaction T
Txns	T	A transaction consists of Σ_T with operation order ($\xrightarrow{to}$)
	$\mathcal{R}_T$	Read Set of Transaction T
	$\mathcal{W}_T$	Write Set of Transaction T
	$\mathcal{G}_T$	The set of all shard groups relevant to T
Data	$\mathcal{K}$	Global Key Space
	g	A shard group contains multiple shards
Order	$\mathcal{H}_i$	Transaction history on $node_i$, $\mathcal{H}_i = (\mathcal{E}_i, po_i, \tau_i)$
	$\mathcal{H}$	Transaction history of the whole system, $\mathcal{H} = \bigcup \mathcal{H}_i$
	$\mathcal{S}$	A serializable schedule for transactions
	$\xrightarrow{rb}$	Real-time order imposed by runtime execution
	$\xrightarrow{so}$	Oreder for operations in $\mathcal{S}$
	$<_S$	Oreder for transactions in $\mathcal{S}$

Table 2. Preliminaries and notations for Relay.

Transactions and Operations. Clients interact with the OLTP service through transactions. Each transaction comprises several single-key read or single-key write *operations*. Formally, each transaction T is a tuple $(\Sigma_T, \xrightarrow{to})$, where Σ_T is the set of operations in T, and $\xrightarrow{to}$ is a total order of all operations in Σ_T. Each operation is either a read (denoted as $o_1 = r(k_1, v_1)$) or a write (denoted as $o_2 = w(k_2, v_2)$). We use $\mathcal{R}_T = \{k | r(k, v) \in \Sigma_T\}$ to denote T's read set and $\mathcal{W}_T = \{k | w(k, v)\} \in \Sigma_T\}$ as T's write set. It should be noted that Relay, as a consistency model, does not essentially require the read and write set of each transaction to be determined upfront, which is a common but restrictive assumption in existing deterministic databases [28, 39, 41, 44, 49]. Essentially, Relay supports general transaction semantics (to be illustrated in Spanner-Relay, §4).

Conflicts and Relevance. We say two transactions conflict with each other if they access the same key, and at least one of the two accesses is "write" (which is known as read-write conflicts and write-write conflicts). We say a transaction T is relevant to shard group g if T accesses at least one key owned by g, and we then use $\mathcal{G}_T$ to represent the set of all groups relevant to T. Formally,

$$\mathcal{G}_T = \{g \mid \exists k : k \in g \wedge k \in (\mathcal{W}_T \cup \mathcal{R}_T)\}$$

History and Equivalence. A history of a data $node_i$ ($server_i$) is an associative triple $\mathcal{H}_i = (\mathcal{E}_i, po_i, \tau_i)$, where $\mathcal{E}$ is a set of operations; po is a partial ordering on $\mathcal{E}$ into processes; and τ divides $\mathcal{E}$ into transactions. We say two histories ($\mathcal{H}_1$ and $\mathcal{H}_2$) are equivalent if they have the same $\mathcal{E}$, po, and τ. Intuitively,

two equivalent histories have the same sequence of operations for each client process and thus are indistinguishable inside the database (also known as view-equivalence [55]).

Real-time order. Following Lamport's formalism, an order of transactions is usually considered as a set of *return before* relations [29]. In our paper, we say a transaction T_1 precedes another transaction T_2 if T_1 finishes (commits) before T_2 starts (i.e., arrives at the database system), denoted as $T_1 \xrightarrow{rb} T_2$.

Definition of RELAY. We define RELAY using the notations above. We say that a transaction processing service ensures RELAY, if for all execution histories, $\mathcal{H} = \bigcup \mathcal{H}_i$, are view-equivalent to a serial schedule $\mathcal{S}$ and the following three properties hold for $\mathcal{S}$.

- *Serializability.* There exists serial schedule $\mathcal{S}$ with total ordering *so* on $\mathcal{E}$ such that ❶ S is equivalent to H; and ❷ no two transactions overlap in *so*, i.e., either

  $$o_1 \xrightarrow{so} o_2, \forall o_1 \in T_1, \forall o_2 \in T_2$$

  or

  $$o_2 \xrightarrow{so} o_1, \forall o_1 \in T_1, \forall o_2 \in T_2$$

  Note that the property ❷ infers for any two transactions T_1 and T_2, either $T_1 <_S T_2$ or $T_2 <_S T_1$ holds. Thus, it essentially defines a total order $<_S$ among all transactions.
- *No Stale Reads.* Formally, for any two transactions T_1 and T_2, the following statements holds.

  $$\mathcal{W}_{T_1} \cap (\mathcal{W}_{T_2} \cup \mathcal{R}_{T_2}) \neq \emptyset \wedge T_1 \xrightarrow{rb} T_2 \implies T_1 <_S T_2$$

  Intuitively, the statement implies regularity. If a transaction T_1 is in read-write conflicts or write-write conflicts with another transaction T_2, and T_1 commits before T_2 in wall-clock time (i.e., elapsed real-time), then the transaction processing service must order T_1 before T_2.
- *Real-time Ordering inside all Shard Groups.* Formally,

  $$\mathcal{G}_{T_1} \cap \mathcal{G}_{T_2} \neq \emptyset \wedge T_1 \xrightarrow{rb} T_2 \implies T_1 <_S T_2$$

  Intuitively, the statement implies for any two transactions from the same shard groups, if a transaction T_1 commits before another transaction T_2 in wall-clock time, then the transaction processing service must order T_1 before T_2, even if they are not conflicts at all. Note that $T_1 \xrightarrow{rb} T_2 \implies T_1 <_S T_2$ essentially means linerlizability.

3.2 Practical Implications

Strong guarantees of consistency model can come up with high-performance costs. On the contrary, a weak consistency model may sacrifice the quality of applications and require more engineering effort and domain-specific knowledge when developing new applications. In the rest, we discuss the desirability of RELAY by studying the practical implications of existing consistency models.

Strict serializability (SS) is the strongest consistency model, which provides all transactions (both CRTs and IRTs) with serializability for isolation and linearizability for ordering. In essence, the guarantee of SS is considered excessive for many applications and expensive to implement. For example, ensuring real-time ordering between all transactions should always pertain to external (out-of-band) causal relations among transactions. However, since a system is unaware of external relations (e.g., real-world communication between two clients), SS must regard *all* pairs of transactions without overlapping lifetime as potentially causally related and then pertain to their ordering, albeit most transactions are independent. In practice, such a guarantee is overly strong and expensive [23]. As evidence, massive existing transactional systems are built atop weaker consistency models and have worked for industrial needs in decades [2, 43].

Different from SS, RELAY ensures the "no stale reads" property or regularity, which effectively prevents most application-level anomalies [51] while leaving the potential for achieving high performance. Note that RELAY indeed enforce real-time ordering (linearizability) among transactions accessing the same region since intra-region network cost is generally tolerable when using modern hardware. As a result, compared to SS, the only anomalies in RELAY may arise from the potential disruption of real-time ordering among transactions happening independently within non-overlapping regions.

Such anomalies do not compromise the correctness of multi-region applications for two key reasons. First, multi-region databases manage shard groups (regions) with data locality. Each region holds (e.g., being the leader of) the shards containing data of nearby clients. Two transactions that access non-overlapped regions are likely to be causally unrelated. Therefore, violating the real-time order for transactions executed in non-overlapping regions will not introduce application-level anomalies.

Second, the time window for breaking causal relations is narrow. RELAY necessitates "no stale reads" for all intra-region or cross-region transactions. To sever the causal relationship between two transactions, external communication must conclude faster than a transaction's lifetime. Specifically, consider two transactions T_2, T_3 accessing non-overlapped regions, where $T_2 \xrightarrow{rb} T_3$. If anomalies were present, it would imply the existence of another transaction T_1 accessing both T_2 and T_3's regions, leading to a final serial order of $T_3 <_S T_1 <_S T_2$. However, as RELAY also mandates "no stale reads," the external causal relation must conducted within T_1's lifetime period.

Overall, RELAY has the potential to enhance the performance of multi-region databases while maintaining correctness and programmability. RELAY stands out as the pioneering consistency model that takes into account real-world deployments and the inherent locality feature of data.

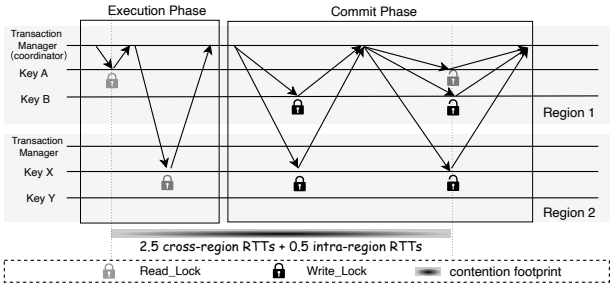


Figure 3. This diagram shows how Spanner orders and commits a CRT. Replicas are removed for readability.

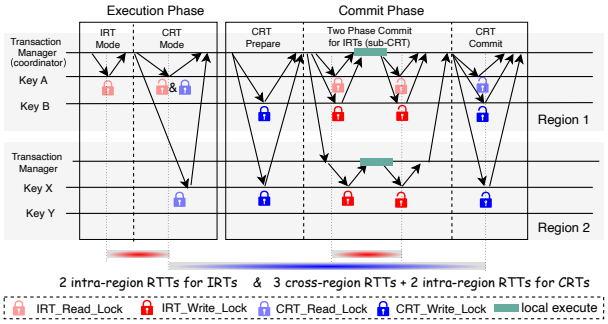


Figure 4. This diagram shows how Spanner-Relay orders a CRT using a variant of 2PL and commits it using 2PC.

Algorithm 1: Algorithm of Spanner-Relay

```
1  function Execution phase:
2      read_set & write_set ← ∅
3      txnType ← IRT                          ▷ Start a new transaction as IRT.
4      touchedRegions ← ∅   ▷ Regions involved in the transaction.
5      ▷ Execute the transaction commands by events:
6      Event read(key)
7          value = find_record(key)
8          read_set.append(key)
9          touchedRegions ← touchedRegions ∪ key.region
10         if |touchedRegions| ≥ 2 then
11             txnType ← CRT
12             Release_IRT_Read_Lock(k) for all k ∈
                 read_set
13             CRT_Read_Lock(k) for all k ∈ read_set
14         if txnType == IRT then
15             IRT_Read_Lock(key)
16         else
17             CRT_Read_Lock(key)
18     Event write (key, value)               ▷ Writes are only buffered
19         write_set.append(<key, value>)
20         Execute Line 9 ~ 13
21 function Commit phase:
22     if txnType == IRT then
23         IRT_Write_Lock(k) for all k ∈ write_set
24         wait for all ACKs from storage      ▷ Abort if fail.
25         Commit(txn)
26         Release_IRT_Read_Lock(k) for all k ∈ read_set
27         Release_IRT_Write_Lock(k) for all k ∈ write_set
28     else
29         CRT_Write_Lock(k) for all k ∈ write_set
30         wait for all ACKs from storage      ▷ Abort if fail
31         Send Commit to txn managers in r, r ∈ touchedRegions
32         ▷ Each transaction manager commits the transaction as
             IRT
33         wait for all ACKs from the txn managers   ▷ Abort if fail
34         Commit(txn)
35         Release_CRT_Read_Lock(k) for all k ∈ read_set
36         Release_CRT_Write_Lock(k) for all k ∈ write_set
```

4 Spanner and Spanner-Relay

Spanner-Relay is a specific example of Relay to optimize the performance of existing strictly serializable databases.

4.1 Protocols and Implementations

Spanner Background. Spanner sticks to strict serializability. It coordinates read-write transactions using two-phase locking (2PL) and committing the transactions using two-phase commit (2PC). Figure 3 presents an example of transaction processing in Spanner. In this example, a transaction T reads the keys A and X, and then updates the keys B and X. The keys A and B are located at different data nodes in *region* 1, and the key X is located at a data node in *region* 2. To commence, T is forwarded to the transaction manager that T firstly accesses, acting as the coordinator and assigning a globally unique `TID` to T. The transaction manager of *region* 1 serves as the coordinator since T reads the key A in the first access. Subsequently, the coordinator sequentially executes all the transaction operations. During execution, it acquires read locks for each read operation and buffers write operations in temporary memory. After buffering all writes, the coordinator obtains exclusive locks for all write keys (i.e., the keys B and X) and installs the writes if all the required locks are acquired. Following this, all the locks (both read and exclusive locks) are released immediately. Finally, T is successfully executed and committed.

We now discuss the performance bottlenecks in Spanner. As Spanner coordinates IRTs and CRTs similarly, where a read lock held by a CRT will block all writes from both CRTs and IRTs, an exclusive lock held by a CRT will prevent all reads and writes. Consequently, a CRT's contention window (i.e., the lock duration) is extremely large. More than two cross-region network round trips will block all conflict transactions. In our example (Figure 3), the contention window of T takes 2.5 cross-region network round trips and 0.5 intra-region round trips. Worse still, such blockings can rapidly accumulate through transitive relations. For instance, considering another CRT T' that accesses the key B and Y, T' can successfully acquire the exclusive lock on the key Y while having to wait for the lock on the key B. As such, all other IRTs and CRTs that access the key Y have to compete with T' for ownership of the lock on the key Y, significantly enlarging the affected key space.

Spanner-Relay. By applying our new consistency model, we treat the IRTs and CRTs differently. To implement this, we distinguish the locks acquired by IRTs and CRTs. The two types of locks order transactions independently. A CRT lock does not block IRTs, and vice versa. In our design, CRT locks

only provide the functionality for reservation and maintain the partial order between CRTs.

Algorithm 1 shows the pseudocode of Spanner-Relay. Figure 4 illustrates an example of how Spanner-Relay executes and commits the same transaction T. Without loss of generality, we assume that the read and write sets of the transaction are unknown to the transaction manager. T is firstly executed as an IRT and acquire `IRT_Read_Lock` for the key A during the execution. T switches to the CRT mode when it attempts to perform remote reads (i.e., reads the key X in *region* 2). Before that, it releases the acquired `IRT_Read_Lock` for the key A and updates the lock type to `CRT_Read_Lock`. If the key A is already exclusively locked by other CRTs, T aborts and directly retries using CRT mode. Otherwise, T successfully enters the CRT mode and employs `CRT_Read_Lock` for the remaining reads (e.g., reads the key X). Since all the changes are handled by intra-region communication, it will not incur WAN overhead. Then, following the transaction logic, T computes its write set and proceeds to the commit phase.

In the commit phase, T acquire `CRT_Write_Lock` for the key B and X. When all `CRT_Write_Lock` is successfully acquired (i.e., the order between T and other CRTs has been determined), the coordinator notifies all transaction managers of the region that T accessed. Each transaction manager commits T independently using IRT mode. As we already allow T to hold the read locks for all read keys (i.e., the keys A and X), the read keys of T can not be modified by any other CRTs. In case any IRTs have modified the read keys of T, we re-execute it locally. The tricky thing is that even if the re-execution depends on remote reads, we can defer the IRT lock acquisition of the execution until the remote reads have been finished since we have now obtained the read and write set of the transaction. One exception is that the transaction T's read and write set may differ during re-execution, or T has cycle dependency in the transaction logic (e.g., the execution in *region* 1 depends on the reads in *region* 2, the execution in *region* 2 depends on the reads in *region* 1, and both the read keys of *region* 1 and *region* 2 has been changed). In such cases, we revert from Spanner-Relay to use IRT locks for the CRT.

Trade-off Analysis. By differentiating CRTs and IRTs locks, Spanner-Relay eliminates both the "commit blocking" and the "coordination blocking" for IRTs. In our example, the contention footprint for conflict IRTs is reduced to two intra-region network round-trip communication. On the other hand, CRTs may incur slightly more communication costs between the coordinator and the transaction managers. However, with data locality, IRTs are always the dominators, more critical, and sensitive to performance. In fact, if CRTs constitute the majority of the workloads, the performance degradation induced by the heterogeneous network is less of a problem. Hence, Spanner-Relay can fall back to the classic Spanner at runtime (i.e., simply using IRT locks for all transactions).

	Ohio	California	Mumbai	Singapore	Paris
Ohio	1.48	52.39	218.64	217.31	91.84
California	52.67	1.12	226.92	170.14	141.52
Mumbai	209.65	227.22	1.92	58.26	125.63
Singapore	216.75	170.13	57.66	2.42	169.29
Paris	93.22	142.18	124.24	168.71	2.06

Table 3. Ping RTT between used EC2 regions (ms)

4.2 Evaluation and Discussion

4.2.1 Implementation Details We implemented Spanner-Relay in C++ utilizing the third-party implementation [1] since the original version of Spanner is not open-sourced. We employed `libevent` for message passing between processes on distinct nodes and between threads in the same process. Transactions were written as stored procedures containing read and write operations over a set of keys.

Deadlock Mechanisms. We considered two different deadlock mechanisms in our evaluation since these mechanisms impart different scopes to how Relay benefits Spanner.

- **`NO_WAIT`.** If a lock request is denied, the database will immediately abort the requesting transaction.
- **`WAIT_DIE`.** It allows a transaction to wait for the requested lock if the transaction is older than the one that holds the lock. Otherwise, the transaction is forced to abort.

Workloads. We employed the standard YCSB-T benchmark for our evaluation. We generated a total of 3,000,000 keys, distributing 500,000 keys per shard. Each transaction had 10 operations, encompassing 5 read operations and 5 read-modify-write operations. By default, we tuned the percentage of CRTs to be 10% and varied the amount of contention in the system by choosing keys according to a Zipf distribution with a Zipf coefficient = 0.75 (medium and high contention). Each of our experiments lasted 30 minutes, with the first 30s and the last 30s excluded from results to avoid performance fluctuations during start-up and cool-down.

Testbed on AWS. All experiments except the sensitivity study were conducted on Amazon EC2 using $m5.2xlarge$ instance type. Each node has 8 virtual CPU cores and 32 RAM. We use 5 EC2 regions: us-east-2 (Ohio), us-west-1 (California), ap-south-1 (Mumbai), ap-southeast-1 (Singapore), and eu-west-3 (Paris). The ping latencies among these regions are shown in Table 3. We partitioned the database into 60 data shards, with each region containing 12 data shards and a replication level of 5, alongside transaction managers. By default, we utilized 35 clients per region to attain the peak throughput for both Spanner and Spanner-Relay.

Sensitivity Study. We study the sensitivity of inter-region latency since Relay relies on the heterogeneity of networks to achieve its potential. However, the inter-region network latency is not tunable in a real-world deployment (e.g., on AWS). Then, we conducted the sensitivity study on a local cluster by simulation. The experiments were conducted on a

cluster comprising 10 machines, each with a 2.60GHz Intel E5-2690 CPU with 24 cores, 40Gbps NIC, and 64GB memory. We executed each data node in a docker container and used Linux tc [24] to regulate the RTT among nodes. We abstracted each server as an individual region. The partition and replication policy is the same as the experiments on AWS.

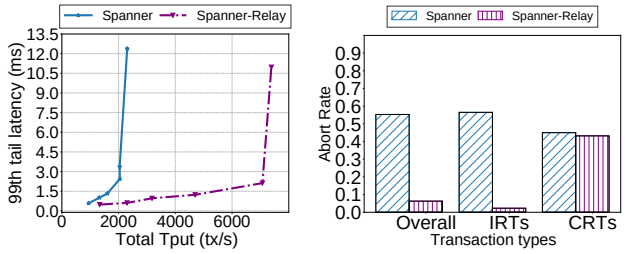
4.2.2 Performance Overview We first evaluated the performance under the default setting. For an apple-to-apple comparison, we refrained from using prior knowledge of read and write sets in our experiments, even though the read and compose set of YCSB-T can be revealed before execution. As shown in Figure 5a and Figure 6a, Spanner-Relay significantly outperformed Spanner on YCSB-T. In particular, Spanner-Relay achieved 3.21× and 4.15× higher peak throughput when utilizing NO_WAIT and WAIT_DIE. We observed that NO_WAIT mechanism resulted in substantially higher throughput than using WAIT_DIE due to the workload's write-intensive nature with medium contention.

To comprehend how our new design contributes to performance improvement, we gathered data on the abort rate for NO_WAIT and tail latency for WAIT_DIE when both Spanner and Spanner-Relay achieved the peak throughput. Figure 5b and Figure 6b illustrate the results. For NO_WAIT, Spanner-Relay can efficiently reduce the abort rate for both IRTs and CRTs. The overall abort rate decreased from 55.3% to 6.2%. In particular, Spanner-Relay achieved a more significant reduction for IRTs (from 56.5% to 2.3%) due to the "non-blocking" property in IRT coordination and commitment. It's worth mentioning that the 2.3% abort rate was only caused by the contention among IRTs. Meanwhile, the abort rate of CRTs also saw a reduction. However, compared to IRTs, CRTs still exhibited a much higher abort rate (i.e., 43.2%) due to the larger contention footprint.

For WAIT_DIE, Spanner-Relay achieved a significantly lower average latency for IRTs, while the average latency of CRTs was roughly the same as in Spanner. This is because, in Spanner-Relay, CRTs will never block an IRT. The results on $90th$ latency support this claim. Both Spanner and Spanner-Relay exhibited low $90th$ latency, about 1.4 and $0.4ms$, respectively. The low tail latency of Spanner-Relay validates that Relay can efficiently eliminate the performance bottlenecks in Spanner by removing the head-line blocking of IRTs. The $99th$ latency of Spanner-Relay increased due to the queueing effect in the software stack.

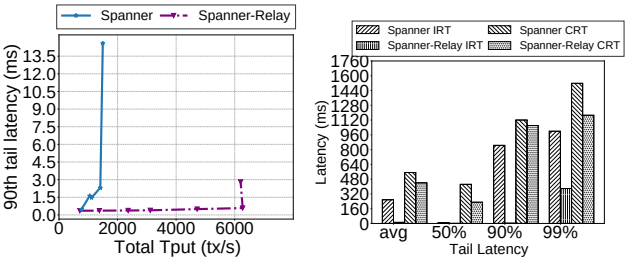
4.2.3 Impact of Experimental Parameters. Next, we delve into understanding how Spanner-Relay and Spanner are affected by various workload parameters.

Concurrency. We first compare the performance of Spanner-Relay and Spanner under various concurrencies. As illustrated in Figure 7a and Figure 8a, Spanner's throughput reached saturation rapidly as the number of clients increased. Consequently, the peak throughput of Spanner was 2303 and 1495 transactions per second using No_WAIT and WAIT_DIE,



(a) Latency-throughput Curve **(b)** Abort Rate.

Figure 5. Overall performance and abort rate of Spanner and Spanner-Relay on YCSB-T (default setting) using NO_WAIT.



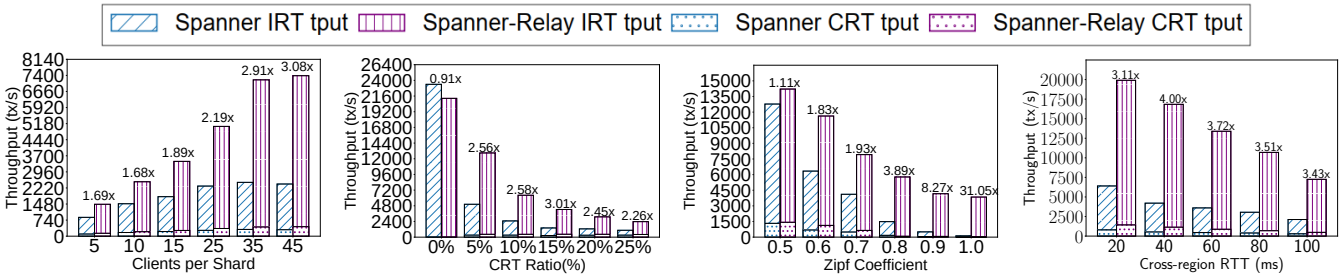
(a) Latency-throughput Curve **(b)** Tail latency.

Figure 6. Overall performance and latency of Spanner and Spanner-Relay on YCSB-T (default setting) using WAIT_DIE.

respectively. In contrast, Spanner-Relay could serve more clients and achieve a substantially higher peak throughput.

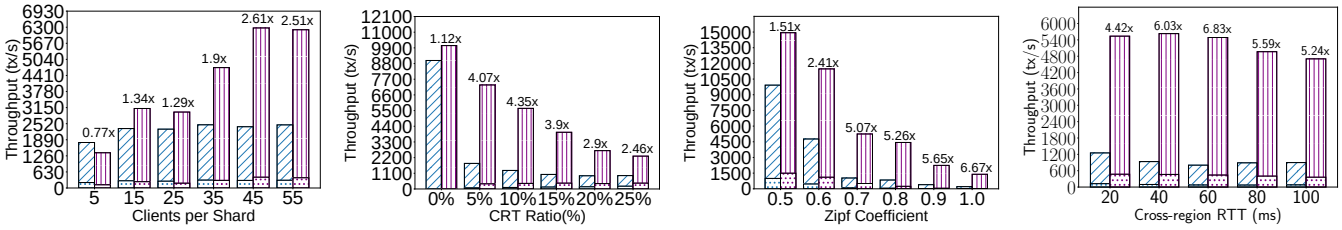
Percentages of CRTs. We studied the impact of the CRT ratio by fine-tuning the workload. As shown in Figure 7b and Figure 8b, when CRTs were enabled, Spanner experienced severe performance degradation (e.g., throughput dropping from 23396 transactions per second to 5032 transactions per second when the CRT ratio increased from 0% to 5%), aligning with our discussion in §1 and §2.2. In contrast, Spanner-Relay's performance degraded slightly, attributed to the elimination of cross-region costs for IRTs. In scenarios where all transactions were IRTs (i.e., a special case in our experiments), Spanner-Relay demonstrated slightly lower throughput than Spanner due to the extra cost (i.e., checking transaction types even if all transactions are IRTs). With a continuous increase in CRT ratios, the throughput of Spanner and Spanner-Relay decreased due to cross-region communication costs. In practice, the CRT ratio of workloads should not be too high. Real-world workloads show good data locality under multi-region deployment (§2.1), facilitating low-latency data access.

Contention (Zipf). We also compared the performance under various contention by adjusting the skewness of Zipf distribution while keeping other parameters consistent with the default settings. The results depicted in Figure 7c and Figure 8c consistently show that the Spanner-Relay's throughput outperforms Spanner's across all levels of contention. This improvement stems from Spanner-Relay efficiently reducing the contention window by ordering IRTs and CRTs independently. As discussed in §2.2, the contention window



(a) Impact of Concurrency **(b)** Impact of CRT **(c)** Impact of Conflicts **(d)** Impact of RTT

Figure 7. Performance of Spanner and Spanner-Relay on YCSB-T with different experimental parameters using NO_WAIT.



(a) Impact of Concurrency **(b)** Impact of CRT **(c)** Impact of Conflicts **(d)** Impact of RTT

Figure 8. Performance of Spanner and Spanner-Relay on YCSB-T with different experimental parameters using WAIT_DIE.

of IRTs eliminates both "coordination blocking" and "commit blocking", which is extremely expensive in multi-region deployments. As expected, Spanner-Relay gained larger improvement under high contention. The reason is that, under high contention, IRTs in Spanner have more chance to be blocked by CRTs, leading to poor performance. It should be noted that the performance of Spanner-Relay also degraded due to the cost of acquiring locks. NO_WAIT consistently outperformed WAIT_DIE since WAIT_DIE suffers more from lock thrashing and timestamp allocation.

4.2.4 Sensitivity Study We studied the sensitivity of cross-region network delays by simulation (see §4.2). Larger cross-region network delays generally result in longer transaction coordination and commit times for CRTs. The results, illustrated in Figure 7d and Figure 8d, indicate that Spanner-Relay outperforms Spanner regardless of the different cross-region network delays. In fact, Spanner-Relay shows more improvements when the network delays are moderate (e.g., 40*s* and 60*s* for a cross-region network round trip). This is because the network delay amplifies the advantages of Spanner-Relay while also affecting Spanner-Relay's CRTs. Our results show that it caused a throughput drop from 1429 transactions per second (tps) to 476 tps as the network delay increased from 20 seconds to 100 seconds using NO_WAIT.

4.2.5 Evaluation Conclusion and Discussion. Our evaluation results show that, by applying Relay, developers can significantly improve the performance of Spanner in various aspects (e.g., throughput, tail latency, and abort rate). Depending on the used deadlock mechanisms, the benefits of Spanner-Relay come from different scopes. For instance, when using WAIT_DIE, Spanner-Relay can efficiently eliminate the head-of-line blocking of IRTs by scheduling IRTs ahead of CRTs. When using No_WAIT, Spanner-Relay can significantly reduce the abort rate of IRTs by applying two types of locks and orders IRTs and CRTs independently.

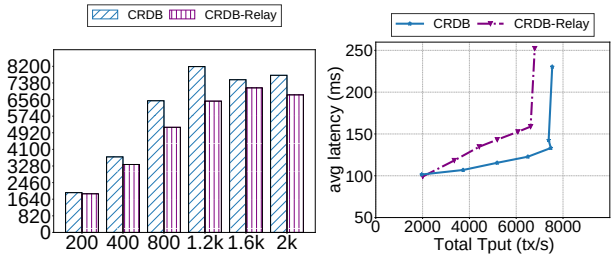
Spanner-Relay can be a practical example, illustrating how Relay can assist multi-region databases in achieving an optimal balance between consistency and performance. It should be noted that we do not directly compare the performance of Spanner-Relay with other advanced geo-distributed transactional systems (e.g., the systems discussed in §2.2) but compare the performance of Spanner-Relay to the native edition of Spanner. This is because we do not intend to build the most powerful multi-region database but take Spanner as a case study to show the potential of Relay. Other performance optimization techniques presented in existing works are orthogonal to our paper. For example, a pre-write-log mechanism in RedT [59] can be adopted by Spanner-Relay to improve the performance further. We believe our study on Spanner-Relay holds the potential to guide future research by encouraging developers to consider consistency tiering in concurrency control protocols.

5 CRDB and CRDB-Relay

CRDB-Relay is another case study of Relay. CRDB-Relay shows that Relay can tighten the monolithic consistency model (i.e., providing tighter and stronger consistency guarantees) without sacrificing performance.

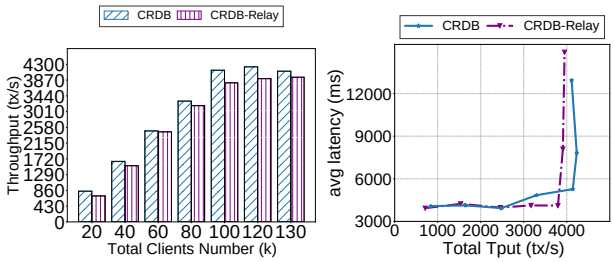
Algorithm 2: Algorithm of CRDB-Relay Coordinator

```
1  function CRDB-Relay Coordinator:
2      inflightOps ← ∅                          ▷ Ongoing operations.
3      touchedRegions ← ∅   ▷ Regions involved in the transaction.
4      txnTS ← now()                ▷ Timestamp of the transaction.
5      for op ← KV operation received from SQL layer do
6          if op.commit then
7              op.deps ← inflightOps
8              send ⟨commit, txnTS⟩ to transaction managers
9              wait for all ACKs
10         else
11             r ← op.key.region
12             if r ∉ touchedRegions then
13                 txnTS ← max(txnTS, GetFinishedTs(r))
14                 VerifyReads(txnTS)
15                 touchedRegions ← touchedRegions ∪ {r}
16             op.deps ← {x ∈ inflightOps | x.key = op.key}
17             inflightOps ← (inflightOps- op.deps) ∪ {op} resp
                 ← send(op, keyLeader(op.key))
18             txnTS ← max(txnTS, GetFinishedTs(r))
19             VerifyReads(txnTS)
```



(a) Throughput. **(b)** Latency-tput Curve.

Figure 9. Performance Comparison on YCSB-T (skewed).



(a) Throughput. **(b)** Latency-tput Curve.

Figure 10. Performance Comparison on TPC-C

5.1 Protocols and Implementations

CRDB Background. CRDB [46] is an open-source production-grade database system that began as an external Spanner clone. Same as Spanner, CRDB aims to build a resilient geo-distributed SQL Database with serializable ACID transactions. CRDB provides single-key linearizability (i.e., "no stale reads" for each key) by multi-version timestamp ordering. The transaction manager nodes in CRDB are the special nodes for interacting with clients, assigning timestamps to transactions, and driving the coordination of transactions.

CRDB's consistency model is strictly weaker than Relay as CRDB ensures only a subset of Relay's guarantees: CRDB does not preserve the real-time ordering for any pair of non-conflicting transactions, while Relay provides real-time order among IRTs in the same region. We refer readers to §6.1 for detailed comparisons between Relay and SKL.

CRDB makes such a design choice because, without the consistency tiering of Relay, the developers could only choose the consistency model from two extremes on the spectrum: developers should either enforce all real-time constraints among non-conflicting transactions (i.e., strict serializability) or enforcing none of them. Since implementing strict serializability across regions can easily overwhelm the benefits of data locality and lead to terrible performance, CRDB had to select the latter choice. Consequentially, CRDB cannot even ensure the causal relation of two transactions from the same client when they access different keys in the same region.

Relay provides a better design point in the spectrum. To study the pros and cons of Relay, we re-design the protocol of CRDB by incorporating multi-region semantics into the part of conflict detection. Before we dive into CRDB-Relay, we discuss how CRDB coordinates transactions below.

Briefly, CRDB performs its reads and writes at its commit timestamp and relies on multi-version concurrency control (MVCC) to process concurrent requests. When a transaction is detected as in conflict with other transactions (e.g., a write to a key at timestamp t_a finds there's already been a read on the same key at a higher timestamp t_b >= t_a), CRDB adjusts the commit timestamp of the transaction to ensure single-key linearizability (e.g., forces the writing transaction to advance its commit timestamp past t_b). Since the detection is conducted by key, CRDB does not guarantee the real-time order between non-conflict transactions.

CRDB-Relay. Algorithm 2 shows the pseudocode of CRDB-Relay. We design Relay by reusing the timestamp adjustment mechanism in CRDB. Using the mechanism, CRDB-Relay detects conflict transactions in the granularity of shard groups (by the transaction manager inside the region) instead of in the granularity of single keys.

That is, to achieve Relay, for any two transactions T_1 and T_2 that access overlapped regions, if T_1 have finished, CRDB-Relay must ensure that T_2's timestamp is larger than T_1's. CRDB-Relay achieves such guarantees by comparing with T_1's write timestamp when T_2's read arrives. Specifically, if $T_2.ts > T_1.ts$, T_2 must see T_1's write; otherwise, if $T_2.ts < T_1.ts$, T_1 may still finish before T_2 starts due to clock skewness, but the skewness should have an assumed bound (i.e., $500ms$ for the uncertainty window, same as CRDB). Consequentially, if $T_1.ts - T_2.ts < bound$, CRDB-Relay cannot determine the order between T_1 and T_2. In such a case, CRDB-Relay should enforce T_2 to abort and then let it retry automatically. However, the uncertainty window may still be too heavy compared to the short execution time of IRTs.

Therefore, instead of sticking to such a passive mechanism implemented in CRDB, our observation of CRDB-Relay is that, to know whether a transaction T_1 may have finished before T_2 starts, a more efficient method should be using active inquiry. Formally, T_2 should ask all data nodes inside a region to ensure that there does not exist such a T_1 that commits before T_2 while taking a larger commit timestamp; otherwise, T_2 should adjust its commit timestamp to past the commit timestamp of T_1.

Fortunately, as CRDB uses transaction managers to maintain the status of each transaction, one can get the status of each transaction from the managers. It should be noted that CRDB does not adopt this active inquiry design to determine the order of transactions because it can be inefficient and non-scalable to let all transactions contact a single node in a geo-distributed deployment. However, Relay's region-based approach enabled efficient management in the granularity of shard groups, which makes it possible to record transactions' state in a per-region manner (Algorithm 2, Line 13). Each transaction can only inquire about relevant regions' managers for the ordering (Algorithm 2, Line 14). Using the active inquiry, CRDB-Relay can efficiently track the ongoing transaction region, thus tightening the consistency guarantees of CRDB.

5.2 Evaluation and Discussion

Implementations Details. We implemented CRDB-Relay using the open-source codebase of CRDB with version v20.3 from the official sites. We modified the logic of obtaining a valid timestamp by recoding all used timestamps inside a region using sets. Our implementation is orthogonal to the optimizations introduced by its origin paper [46] (e.g., write pipelining, parallel commits, and follower read).

Workloads. In addition to YCSB-T, we used TPC-C [13] for our evluation. TPC-C is a widely used standard OLTP benchmark, which organizes data by data warehouse. To partition the Tables across multi-regions while compromising with data locality, we used the advanced fine-grained partition and replication framework of CRDB to ensure each warehouse. We generated a total of 12,000 warehouses and distributed 2400 warehouses per region. We kept the mixing ratio of different transaction types as default. As a result, 11% transactions are CRTs [10].

Testbed. We used the same cloud environments as those experiments of Spanner. Each instance runs as a CRDB or a CRDB-Relay node. Besides, we run a transaction manager for each region coated with a CRDB node.

5.2.1 Performance Overview

Performance on YCSB-T. We compared the performance of CRDB and CRDB-Relay on the default YCSB-T (§4.2) using the calibrated configurations according to the official guidelines [47]. The results are shown in Figure 9. CRDB-Relay achieved 0.90× to 0.93× throughput compared to CRDB with various concurrencies. The peak throughput of CRDB-Relay was 9.8% lower than CRDB, and average latency of CRDB-Relay was 1.08× to 1.32× higher than CRDB.

CRDB-Relay incurred a more server performance drop on the skewed YCSB-T workloads since, compared to CRDB, CRDB-Relay may expand the contention footprint by involving more ongoing timestamps (i.e., line 16, Algorithm2). However, the overhead is still marginal, and the performance degradation is smaller than ∼ 10%.

Performance on TPC-C. Figure 10 shows the results. We first calibrated the performance of CRDB with the results presented in its original paper. As shown in Figure 10a, the peak throughput of CRDB was ∼ 4200 tps (113, 520 tpmC), roughly the same as [50]. Then, we compare the performance between CRDB and CRDB-Relay. The results show that CRDB-Relay 's peak throughput (4080 tps) is slightly lower (∼ 3%) than CRDB, while the latency results are similar.

Conclusion. Our evaluation results show that CRDB-Relay achieved similar performance as CRDB while providing strong consistency guarantees on real-time orders. CRDB-Relay can be a practical example, illustrating how Relay can assist multi-region databases that use weaker consistency models to tighten their consistency and performance trade-offs.

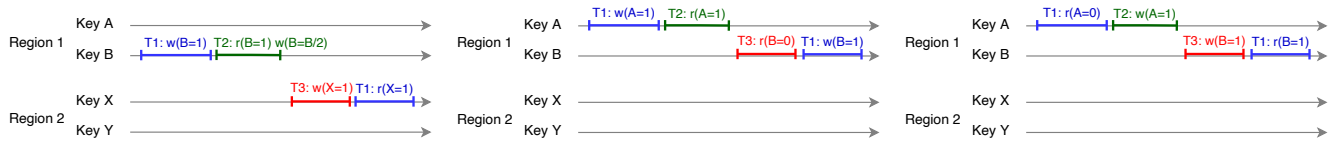
6 Related Works

Transaction processing is a well-explored research area with a plethora of influential works. We discuss the most related works in this section to position our work.

6.1 Proximal Consistency Models

Figure 12 compares Relay to its proximal consistency models. We detail three of them below. All of them are serializable.

Regular Sequential Serializability (RSS) is another tiered consistency model and complements Relay. Briefly, RSS and Relay focus on different aspects of distributed transactions and apply consistency tiering to different types of transactions. Specifically, RSS is tailored for read-only transactions, permitting two read-only transactions to observe partial results of a committed read-write transaction in arbitrary orders (see our example in Figure 11b). This behavior essentially violates the real-time ordering among read-only transactions while significantly benefiting the performance by allowing more concurrency. As illustrated in Figure 11b, RSS allows transaction T_2 to read the writes made by a concurrent transaction T_1, while T_3 following T_2 reads a version preceding T_1. Consequently, the real-time order between T_2 and T_3 is disrupted: the real-time order between T_2 and T_3 is $T_2 \rightarrow T_3$; the serializable order enforced by RSS is $T_3 \rightarrow T_1 \rightarrow T_2$, which implies $T_3 \rightarrow T_2$. Different from RSS, Relay focuses on data locality within multi-region deployments, allowing a database system to relax the real-time ordering among transactions that access non-interleaved regions (see Figure 11a).



(a) Allowed by Relay but disallowed by RSS. **(b)** Allowed by RSS but disallowed by Relay. **(c)** Allowed by SKL but disallowed by Relay

Figure 11. Comparison of Relay with proximal levels of consistency models.

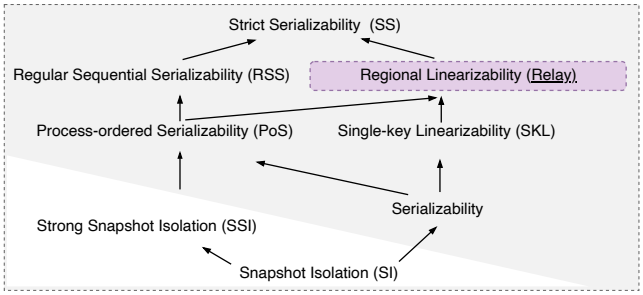


Figure 12. This diagram shows how Relay compared to its proximal consistency models: SS [42], RSS [23], SKL [46], PoS [14, 34], SSI [19], serializability [51], and SI [4]. The model in a grey background ensures serializable isolation.

Single-key Linearizability (SKL) was proposed by [46]. SKL is strictly weaker than Relay. Silimar to Relay, SKL also guarantees serializability and provides "no stale reads" for the transactions that access the same keys. However, SKL does not preserve real-time orders between non-conflicting transactions (i.e., both IRTs and CRTs) because SKL only checks the properties in key granularity. For instance, the execution shown in Figure 11c is permissible by SKL since there are "no stale reads" for each accessed key. However, it violates the real-time ordering between T_2 and T_3, as the serial order is $T_3 \rightarrow T_1 \rightarrow T_2$. Such a violation is not allowed by Relay because Relay preserves real-time orders for T_2 and T_3 in the same region.

Process-ordered Serializability (PoS) complements Relay. In particular, PoS tracks the causality of each client and ensures the system preserves the real-time ordering within the set of each client's requests. Relay does not involve client semantics by definition. However, in a real deployment, Relay can be stronger than PoS by associating each client to a home region. If each client is associated with a region, all the requests of the client are routed to its home region and executed in real-time order. Thus, in this case, Relay can be proved to be strictly stronger than PoS.

6.2 Transaction Priority

Compared to strict serializability (SS), one of the pivotal innovations of Relay is scheduling IRTs ahead of CRTs until the CRTs' order is established. Therefore, Relay operates under the assumption that IRTs have a higher priority than CRTs until CRTs have been ordered, after which both IRTs and CRTs are given equal priority for execution.

In this context, multi-region system developers can leverage existing transaction priority protocols [3, 17, 21, 22, 25] to transition from a monolithic consistency model (e.g., SS) to Relay. For instance, Polaris [56] is a transaction priority protocol rooted in a variant of OCC. Polaris embeds priority-related conflict detection within each record, permitting priority preemption during runtime. Consequently, Polaris can be a template for OCC-like concurrency control protocols to achieve Relay.

6.3 Mixed Consistency Models

Several prior works [20, 27, 28, 31, 32, 38, 52, 54] have been proposed to manage both weakly and strongly consistent transactions within a database. For instance, MixT [38] advocates that consistency is a property of information. It proposes a new embedded language enabling users to manually configure the consistency guarantees for each operation. Red-Blue consistency [31] allows strongly and causally consistent operations to co-exist in a single system and rely on application developers to make choices. AutoGR [52] automatically identifies the minimal set of the required consistency guarantees based on applications using the Z3 theorem prover.

Different from these application-semantic-based works, Relay is geared towards multi-region deployments, which integrates network semantics into consistency. Consequently, Relay does not depend on prior application knowledge to manually set distinct consistency guarantees for different transactions or operations. Relay provides serializability (i.e., the strongest isolation levels) for all transactions and tailors real-time properties to achieve high performance.

7 Conclusion

This work analyzes the inherent limitations of monolithic consistency models: they are inadequately designed for heterogeneous networks. These models are either overly strong for slow transactions or can be tightened for fast ones.

We propose Relay, the first consistency model for multi-region databases via consistency tiering. Following Relay, we design, implement, and evaluate two practical systems: Spanner-Relay and CRDB-Relay. Our evaluation shows that Relay can significantly enhance the performance of Spanner and strengthen the consistency guarantees of CRDB. The code is available at https://github.com/sosp24p269/relay.

References

[1] [n. d.]. Github: UWSysLab/tapir. https://github.com/UWSysLab/tapir.

[2] 2014. MySQL Database. http://www.mysql.com/.

[3] Robert K Abbott and Hector Garcia-Molina. 1992. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems (TODS)* 17, 3 (1992), 513–560.

[4] Todd Anderson, Yuri Breitbart, Henry F Korth, and Avishai Wool. 1998. Replication, consistency, and practicality: are these mutually exclusive?. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 484–495.

[5] AWS. [n. d.]. Regions, Availability Zones, and Local Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[6] Azure. [n. d.]. Regions and availability zones. https://docs.microsoft.com/en-us/azure/availability-zones/az-overview.

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

[8] Ngai Hang Chan. 2004. *Time series: applications to finance*. John Wiley & Sons.

[9] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[10] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 210–227.

[11] Google Cloud. [n. d.]. Google Cloud IoT Core. https://cloud.google.com/iot-core/.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*.

[13] THE TRANSACTION PROCESSING COUNCIL. 2014. TPC-C. http://www.tpc.org/tpcc/.

[14] Khuzaima Daudjee and Kenneth Salem. 2004. Lazy database replication with ordering guarantees. In *Proceedings. 20th International Conference on Data Engineering*. IEEE, 424–435.

[15] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*. 715–726.

[16] Christian Decker, Jochen Seidel, and Roger Wattenhofer. 2016. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*. 1–10.

[17] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

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

[19] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.

[20] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. 2003. Application specific data replication for edge services. In *Proceedings of the 12th international conference on World Wide Web*. 449–460.

[21] Jayant R Haritsa, Michael J Carey, and Miron Livny. 1990. Dynamic real-time optimistic concurrency control. In *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, 94–103.

[22] Jayant R Haritsa, Michael J Carey, and Miron Livny. 1990. On being optimistic about real-time constraints. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 331–343.

[23] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 163–179.

[24] Bert Hubert. [n. d.]. tc(8), Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[25] SL Hung and KY Lam. 1992. Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record* 21, 4 (1992), 22–27.

[26] Kaippallimalil J Jacob and Dennis Shasha. 1999. Fintime: a financial time series benchmark. *ACM SIGMOD Record* 28, 4 (1999), 42–48.

[27] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment* 2, 1 (2009), 253–264.

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

[29] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[30] Filippo Lanubile, Christof Ebert, Rafael Prikladnicki, and Aurora Vizcaíno. 2010. Collaboration tools for global software engineering. *IEEE software* 27, 2 (2010), 52.

[31] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[32] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. 2018. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 359–372.

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

[34] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The {SNOW} Theorem and {Latency-Optimal}{Read-Only} Transactions. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 135–150.

[35] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. *arXiv preprint arXiv:2305.14270* (2023).

[36] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).

[37] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[38] Mae Milano and Andrew C Myers. 2018. MixT: A language for mixing consistency in geodistributed transactions. *ACM SIGPLAN Notices* 53, 4 (2018), 226–241.

[39] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the*

13th ACM Symposium on Operating Systems Principles (SOSP '91).

[40] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 517–532.

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

[42] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[43] PostgreSQL 2012. PostgreSQL. https://www.postgresql.org.

[44] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12 (2019), 1747–1761.

[45] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)(July 2021), USENIX Association.*

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

[47] Cockroach Labs Documentation Team. [n. d.]. Benchmarking overview. https://www.cockroachlabs.com/docs/stable/performance

[48] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[49] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. In *SIGMOD '12: Proceedings of the 2012 ACM SIGMOD international conference on Management of data*.

[50] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, et al. 2022. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data*. 2312–2325.

[51] Paolo Viotti and Marko Vukolić. 2016. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–34.

[52] Jiawei Wang, Cheng Li, Kai Ma, Jingze Huo, Feng Yan, Xinyu Feng, and Yinlong Xu. 2021. AUTOGR: automated geo-replication with fast system performance and preserved application semantics. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1517–1530.

[53] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. PolarDB-IMCI: A cloud-native HTAP database system at alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[54] Yingyi Yang, Yi You, and Bochuan Gu. 2017. A Hierarchical Framework with Consistency Trade-off Strategies for Big Data Management. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, Vol. 1. IEEE, 183–190.

[55] Mihalis Yannakakis. 1984. Serializability by locking. *Journal of the ACM (JACM)* 31, 2 (1984), 227–244.

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

[57] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–37.

[58] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (Dec. 2018), 1–37. https://doi.org/10.1145/3269981

[59] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient Distributed Transaction Processing in Heterogeneous Networks. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1372–1385.

[60] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing Distributed Transactions through Adaptive Replica Provision (Extended Version). *arXiv preprint arXiv:2403.11221* (2024).