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

Global enterprises deploy databases across multiple regions to achieve high availability, strong scalability, and efficient service localization. However, supporting serializable cross-region transactions in such databases is significantly challenging. Existing efforts aim to optimize coordination costs by reducing the number of WAN (Wide Area Network) round-trips but still incur cross-region latency for conflicting transactions.

This paper argues that monolithic serializable consistency models are not well-suited for multi-region systems. Cross-region transactions inherently experience much higher latency than intra-region transactions. When a cross-region transaction conflicts with an intra-region transaction, it inevitably leads to head-of-line blocking in pessimistic protocols or significantly high abort rates in optimistic protocols.

In response, we propose a new approach called *consistency tiering*, which enables different consistency guarantees for transactions based on their characteristics. We introduce Relay, a tiered consistency model for multi-region systems that provides linearizability for intra-region transactions and regularity for cross-region transactions, both ensuring serializability for isolation. We design and implement two prototypes based on Spanner and CockroachDB. Experimental results show that Relay significantly outperforms monolithic serializable models across various workloads.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Information systems → Distributed database transactions**.

Additional Key Words and Phrases: Geo distributed database system, Consistency

ACM Reference Format:
. 2025. Relay: High-performance Transactions in Heterogeneous Networks via Consistency Tiering. 1, 1 (March 2025), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Today, cloud providers (e.g., AWS [6], Azure [7], and Google Cloud [13]) host computing infrastructures across multiple geographic regions. Consequently, multi-region deployment has emerged as a prominent choice for cloud-native applications pursuing low latency, high availability, and strong scalability [11, 21, 51, 55, 63]. To achieve scalability, these applications typically use a multi-region database as their backend and partition it into multiple shards. For availability, each shard is replicated across multiple regions.

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

Supporting serializable transactions in this deployment model presents significant challenges. Due to geographic distances, coordinating cross-region transactions (CRTs) is inherently slower than intra-region transactions (IRTs).

Author's Contact Information:

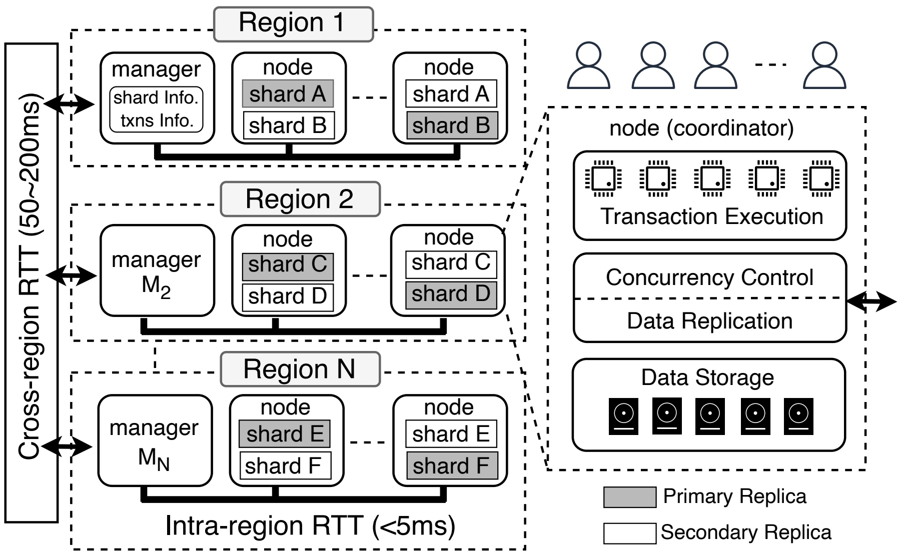


Fig. 1. A typical deployment model for multi-region systems. The system store is partitioned into multiple shards spanning multiple regions. Each shard consists of a primary replica and several secondary replicas. Intra-region network latency (e.g., $< 5ms$) is significantly shorter than intra-region one (e.g., $50 \sim 200ms$).

To optimize CRT performance, several notable works have been proposed (e.g., [21, 44, 45, 48, 53]). We classify these works into two categories.

The first category optimizes the coordination cost of cross-region transactions by reducing the number of WAN round trips. A representative approach is deterministic concurrency control [41, 45, 48, 53], which avoids 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 during the commit phase (e.g., Aria [41]). As a result, they cannot or have very limited support for interactive transactions. These limitations largely prevent their adoption in industry products.

The second category eliminates CRTs by making certain assumptions about the workloads [38, 65]. For example, Leap [38] always migrates the primary replicas from the remote region to the local one before executing a CRT. Therefore, its performance heavily depends on the assumption that CRTs are the exception rather than the norm.

We emphasize that CRTs remain essential for general workloads, especially when there is limited prior knowledge of application semantics. Moreover, the performance disparity between CRTs and IRTs remains significant despite the optimizations proposed in the first category. For instance, Detock [45], a state-of-the-art geo-distributed transaction protocol, can execute and commit CRTs with just a single cross-region network communication. However, even with this optimization, CRTs still have an average latency of approximately 100ms, significantly higher than the latency of IRTs under default experimental setups. When both IRTs and CRTs use Detock's consistency model (e.g., strict serializability), even a few slow CRTs can entangle numerous IRTs, leading to deadlocks or aborts. Consequently, this substantially degrades overall database performance (i.e., the contention window can be further amplified when stragglers occur). We present an experimental study in Figure 2, which is also evidenced by multiple real-world studies [11, 21, 45].

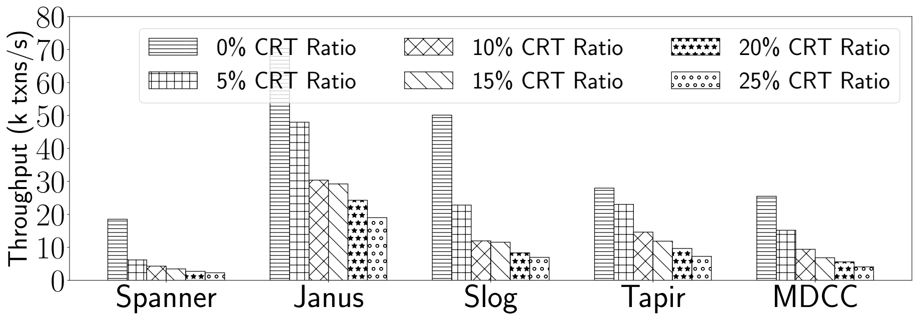


Fig. 2. Impact of CRT ratio on the throughput.

Departing from existing work, we address multi-region transactions through ***consistency tiering***. We argue that monolithic serializable consistency models (that ensure the same consistency guarantees for both IRTs and CRTs) are ill-suited for multi-region deployments. The primary issue stems from the inherent heterogeneity in transaction types introduced by such deployments. Since data is closely tied to its home region via primary replicas, data access costs vary for transactions originating from different regions. To accommodate this heterogeneity, a diverse range of consistency guarantees should be available for different transaction types, while upholding serializability and the strongest possible consistency where necessary.

Our observation is that monolithic consistency models tend to be either overly strong for cross-region transactions (CRTs) or could be enhanced without degrading the performance of intra-region transactions (IRTs). For instance, the strict serializability model [46] abstracts the entire system as a single node, requiring heavy synchronization among all computing servers. As a result, this model fails to preserve the benefits of near-client computing: even an IRT must be ordered alongside CRTs. When a CRT conflicts with an IRT, it unavoidably leads to prolonged head-of-line blocking in pessimistic protocols (e.g., two-phase locking) or introduces significantly higher abort rates in optimistic protocols (e.g., OCC [33]).

On the other hand, weak consistency models (e.g., serializable snapshot isolation [17]) support serializability but not linearizability reduce the consistency guarantees for all types of transactions, harming the model's applicability and usability. Through consistency tiering, we have found that the weak consistency guarantees for local transactions can be enhanced with a small performance overhead as the communication cost for coordinating IRTs is low. Additionally, multi-region applications demonstrate a desire to enforce stronger consistency for local transactions, which aligns with their service localization (see Section 2.3).

By adopting consistency tiering for multi-region applications, we introduce ***Regional Linearizable Serializability*** (Relay), the first tiered consistency model that provides the strongest possible consistency for both intra-region transactions (IRTs) and cross-region transactions (CRTs). Relay ensures strict serializability (i.e., the strongest consistency guarantee) for IRTs within the same region while providing regular serializability for CRTs. We define Relay in Section 3.

To demonstrate the efficiency and applicability of Relay, we designed, implemented, and evaluated variations of two notable industrial systems: Spanner and CockroachDB (CRDB). We refer to these derived variants as Spanner-Relay and CRDB-Relay, respectively. We chose these two database systems because they complement each other in both their consistency models and concurrency control protocols. For example, Spanner adopts strict serializability, stronger than Spanner-Relay, whereas CRDB adopts single-key linearizability, weaker than CRDB-Relay. Spanner

System	Transaction Protocol	Consistency Model	Isoaltion Level	Blocking
Spanner [14]	read-write transaction: 2PL + 2PC read-only transaction: timestamp ordering	strict serializability	serializable	Yes
Calvin [54]	centralized coordinator	strict serializability	serializable	Yes
Slog [48]	IRT: intra-region sequencer CRT: centralized coordinator	strict serializability	serializable	Yes
Detock [45]	IRT: intra-region Sequencer CRT: dependency-graph	strict serializability	serializable	Yes
Janus [44]	dependency-graph	strict serializability	serializable	Yes
Ocean Vista [21]	timestamp ordering (watermark)	strict serializability	serializable	Yes
CRDB [51]	timestamp ordering (HLC)	single-Key linearizability	serializable	Yes
RedT [64]	2PL + 2PC	serializable snapshot isolation	serializable	Yes
Tapir [62]	variant of OCC	serializable snapshot isolation	serializable	No (abort)
MDCC [32]	Paxos	snapshot isolation	snapshot isolation	Yes
GentleRain [20]	timestamp ordering	causal consistency	serializable	Yes
PaRiS [50]	timestamp ordering	causal consistency	serializable	Yes
Cure [4]	timestamp ordering	causal consistency	serializable	Yes

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

uses conventional two-phase locking (a pessimistic concurrency control protocol), while CRDB employs hybrid logical clocks for concurrency control (a timestamp ordering protocol). We believe these 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:

- By systematically analyzing multi-region deployments, we have gained the 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 that by slightly reducing the consistency guarantees, Relay significantly improves Spanner's throughput and provides more robust consistency guarantees for CRDB with minor performance degradation.

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

2 Background and Motivation

2.1 System Model and Heterogeneous Network

As illustrated in Figure 1, the database is partitioned into multiple data shards that span across several regions. Each shard consists of a primary replica and several secondary replicas. Depending on replication policies, replicas can be configured to reside on cross-region or intra-region nodes (servers). For instance, ***partial replication*** is a common strategy in a geo-distributed database for replicating only a subset of the data across different geographical locations [11, 51]. This approach optimizes performance and complies with data regulations (e.g., the EU's General Data Protection Regulation).

Fundamentally, our deployment model is compatible with a wide range of possible data placement policies that allow users to comply with data domiciling requirements and also make trade-offs between performance and fault tolerance. However, for simplicity, our paper first follows a partial replication design that deploys replicas intra-region or inside a nearby region so that IRTs can have significantly shorter latency than CRTs. We study the performance sensitivity of Relay to RTTs in Section 4.2 later.

Each region includes a centralized transaction manager responsible for maintaining globally consistent metadata, such as table schemas, data placement policies, and globally unique transaction IDs. Nodes communicate with each other over the network. We assume a partially synchronized network where every message is eventually delivered and processed.

The deployment model exhibits significant heterogeneity in network performance between different nodes, both within the same region and across different regions. For example, a cross-region network (also known as a wide-area network, WAN) round trip can 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). In contrast, the network delay within a region is normally less than $5ms$ [6]. Meanwhile, cross-region networks can also be less stable than intra-region ones [64]. 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 enhance the performance of geo-distributed transaction processing. Table 1 show representative ones, with all supporting serializability for isolation or capable of being modified for serializability (e.g., MDCC). These systems focus on reducing the cost of CRTs. Some efforts aim to minimize the number of WAN round-trips in transaction coordination. For example, RedT [64] implements a pre-write-log mechanism to bypass the synchronization of `prepare` messages (i.e., the first phase in two-phase commits) from the coordinator to primary replicas. Calvin [54], Slog [48], Janus [44], Detock [45], MDCC [32], and Ocean Vista [21] employ deterministic concurrency control protocols to cut down the execution cost of CRTs in WAN environments. By eliminating non-deterministic logic, they create a global log containing all transactions introduced into the system, ensuring a concurrent execution schedule that mirrors processing all transactions serially in the order of their appearance in the log. Once the transaction order is established, both CRTs and IRTs can be executed locally, as the executors can simply follow the order indicated in their received logs.

Despite various proposals, none effectively prevent an IRT from being blocked by CRTs. Spanner and RedT utilize two-phase locking for transaction ordering and commit transactions via a two-phase commit. When an IRT conflicts with an ongoing CRT, it must wait for the necessary locks. Tapir employs a variation of OCC, causing the IRT to abort during the validation phase, leading to a high abort rate, as noted in previous studies [11, 21]. Deterministic databases either sequence IRTs and CRTs together (e.g., Calvin, Janus, and Ocean Vista) or require an IRT to wait for CRTs scheduled earlier to execute (e.g., Slog and Detock).

As a result, these systems can experience significant performance issues when CRTs occur in the database, especially when contention between IRTs and CRTs is high. Our experimental study, depicted in Figure 2, examined the impact of CRT ratios on five recent representative systems using YCSB-T workloads with a Zipf parameter of 0.8. The key observation from our results is that even a small percentage of CRTs can drastically degrade overall system performance, with up to 86% degradation observed with just 5% CRTs.

Relay can 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 [14, 18, 40]. Operations such as transferring funds between accounts, updating balances, and processing payments must be processed consistently within a region. This ensures that once a transaction is completed, the user can see the same updated state immediately. However, weaker consistency could be used for non-critical, read-heavy operations like displaying transaction history or account summaries over WAN [10, 30]. For instance, activities such as displaying transaction history, querying account summaries, and other read-focused queries where immediate consistency is not as critical, especially over wide area networks (WAN).

Content Platforms. In platforms that involve user interactions and content dissemination, the choice between strong and weaker consistency models can significantly impact both user experience and system performance. When users post updates or comments on social media platforms, achieving immediate consistency is crucial to maintaining a seamless user experience. Users expect their content to appear instantly after posting, reflecting changes across all instances of the platform. Yet, weaker consistency models are desirable to enhance performance and scalability for distributing content like feeds or recommendations across a global audience (i.e., cross-region queries) [8].

3 Regional Linearizable Serializability

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

3.1 Definition of Relay

We adopted the formalism from existing works [26] for clarity. Table 2 summarizes the notations. Without loss of generality, we consider an OLTP (On-Line Transaction Processing) service 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 intra-group operations.

Grouping and sharding create a two-tier 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 [45]. Conversely, the shard group is a consistency strategy where cross-group external ordering requirements are typically less important. This two-level framework is crucial for high-performance transactions in heterogeneous networks, as it effectively connects divisions based on application semantics (e.g., data items grouped by warehouse ID in TPC-C [15]) with those based on deployment topology (e.g., 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
	$\xrightarrow{so}$	Order for operations in $\mathcal{S}$
	$<_S$	Order 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.

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 [60]).

Real-time order. Following Lamport's formalism, an order of transactions is usually considered as a set of *return before* relations [34]. 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 implies 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 consistency guarantees can incur high-performance costs, while weak consistency may compromise application quality and demand more engineering effort and domain-specific knowledge for new applications. We explore the desirability of Relay by examining the practical implications of existing consistency models.

Strict serializability (SS) is the strongest consistency model, ensuring serializability for isolation and linearizability for ordering in all transactions, including CRTs and IRTs. The guarantee of strict serializability (SS) is often excessive and costly to implement for many applications. It requires real-time ordering for all transactions, assuming all pairs without overlapping lifetimes are potentially causally related, even though most are independent. This approach is overly strong and expensive [26]. Consequently, many transactional systems use weaker consistency models, which have successfully met industrial needs for decades [2, 47].

Unlike SS, Relay ensures the “no stale read” property or regularity, effectively preventing most application-level anomalies [56] while maintaining high performance. Relay enforces real-time ordering (linearizability) for transactions within the same region, as intra-region network costs are typically manageable with modern hardware. Consequently, compared to SS, the only anomalies in Relay may result from disrupted real-time ordering among transactions in non-overlapping regions. These anomalies do not compromise the correctness of multi-region applications for two main reasons. First, multi-region databases manage shard groups with data locality, where each region holds or leads

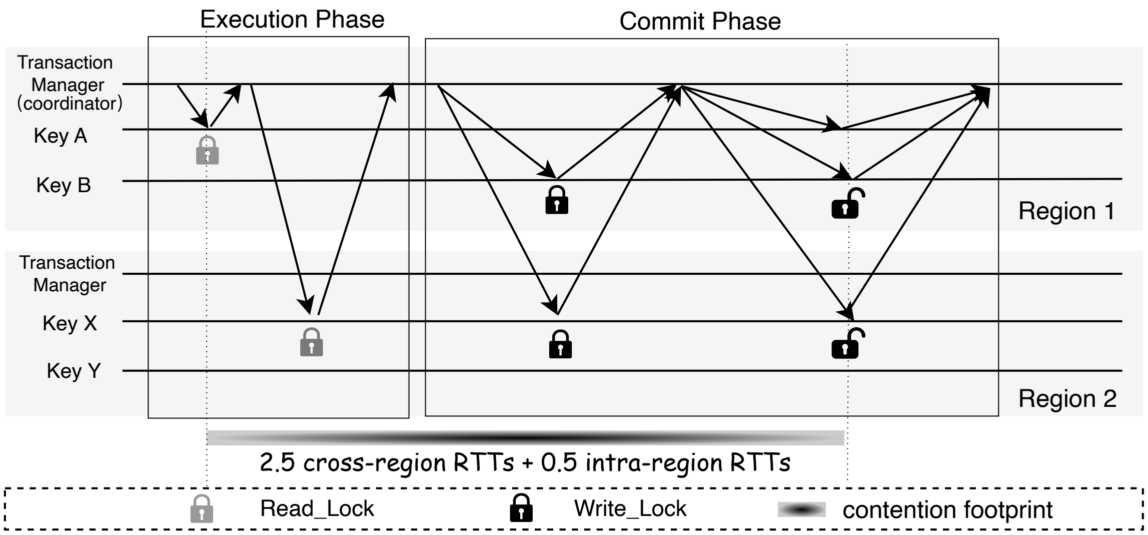


Fig. 3. This diagram shows how Spanner orders and commits a CRT. Replicas are removed for readability.

shards containing data for nearby clients. Consequently, transactions accessing non-overlapping regions are likely causally unrelated, so violating real-time orders for such transactions will not introduce application-level anomalies.

Second, the time window for breaking causal relations is narrow. Relay requires "no stale reads" for all IRTs and CRTs. To sever the causal relationship between two transactions, external communication must conclude faster than a transaction's lifetime. For instance, 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 boost the performance of multi-region databases while ensuring correctness and programmability by considering real-world deployments and leveraging the inherent locality feature of data.

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

4.1.1 Spanner Background. Spanner adheres to strict serializability, coordinating read-write transactions using two-phase locking (2PL) and committing them with two-phase commit (2PC). In the example depicted in Figure 3, a transaction T reads the keys A and X, then updates the keys B and X. Keys A and B reside at different data nodes in *region* 1, while key X is in *region* 2. To commence, T is forwarded to the transaction manager of the first accessed node, which acts as the coordinator and assigns a globally unique TID to T. The transaction manager for *region* 1 then becomes the coordinator of T. The coordinator sequentially executes all transaction operations, acquiring read locks for each read and buffering write operations in temporary memory. Once all writes are buffered, the coordinator

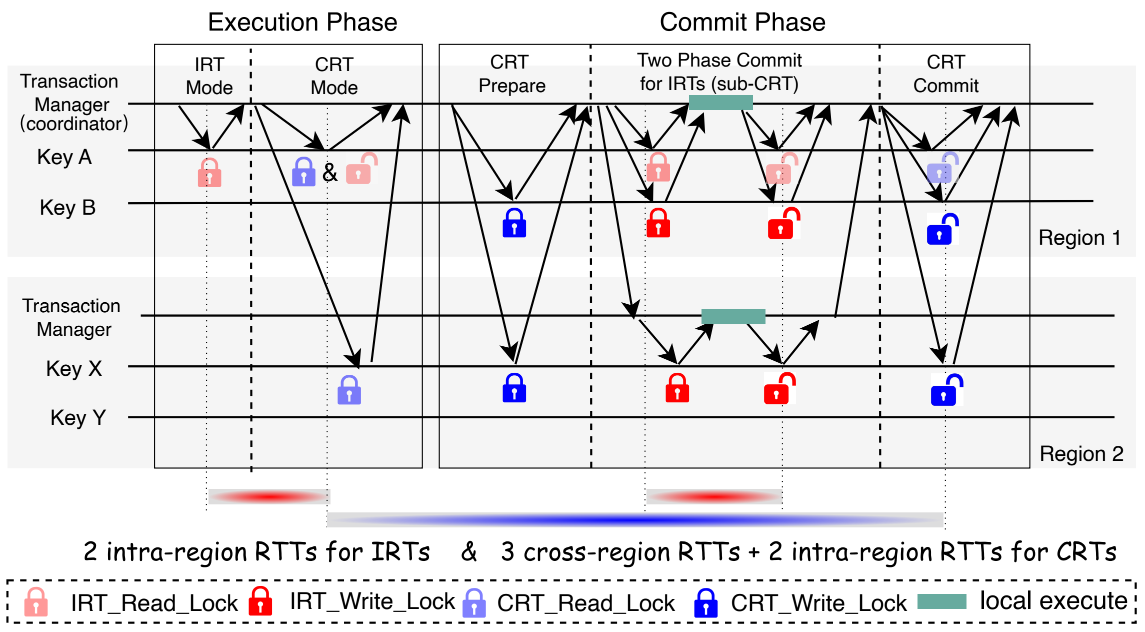


Fig. 4. This diagram shows how Spanner-Relay orders a CRT using a variant of 2PL and commits it using 2PC.

obtains exclusive locks for all write keys(i.e., keys B and X) and installs the writes if all required locks are acquired. Subsequently, all locks (both read and exclusive) are released. Finally, T is successfully committed.

Spanner faces performance bottlenecks due to its coordination of intra-region transactions (IRTs) and cross-region transactions (CRTs) in a similar manner. A read lock held by a CRT blocks all writes from both CRTs and IRTs, while an exclusive lock held by a CRT prevents all reads and writes. This results in a large contention window for CRTs, meaning the lock duration is extensive. More than two cross-region network round trips can block all conflicting transactions.

In the example from Figure 3, the contention window of T takes 2.5 cross-region network round trips and 0.5 intra-region round trips. This blocking can quickly 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. Consequently, all other IRTs and CRTs that access the key Y have to compete with T' for the lock, enlarging the affected key space.

4.1.2 Spanner-Relay. By applying our new consistency model, we differentiate between intra-region transactions (IRTs) and cross-region transactions (CRTs). This involves distinguishing the locks acquired by IRTs and CRTs. The two types of locks order transactions independently, meaning a CRT lock does not block IRTs and vice versa. In our design, CRT locks are used solely for reservation purposes and to maintain the partial order between CRTs.

The pseudocode of Spanner-Relay is shown in Algorithm 1, and Figure 4 illustrates how Spanner-Relay executes and commits T. To ensure general applicability, we assume that the read/write sets of the transaction are unknown to the transaction manager, and sets are inferred during execution.

Execution Phase. At the beginning, T initially executed as an IRT and acquired IRT_Read_Lock for the key A. T switches to the CRT mode when it attempts to perform remote reads (i.e., reads the key X in *region* 2). Before switching,

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             CRT_Read_Lock(k) for all k ∈ read_set
13             Release_IRT_Read_Lock(k) for all k ∈ read_set
14         end
15         if txnType == IRT then
16             IRT_Read_Lock(key)
17         else
18             CRT_Read_Lock(key)
19         end
20     end
21     Event write (key, value)                             ▷ Writes are only buffered
22         write_set.append(<key, value>)
23         Execute Line 9 ∼ 13
24     end
25 function Commit phase:
26     if txnType == IRT then
27         IRT_Write_Lock(k) for all k ∈ write_set
28         wait for all ACKs from storage                   ▷ Abort if fail.
29         Commit(txn)
30         Release_IRT_Read_Lock(k) for all k ∈ read_set
31         Release_IRT_Write_Lock(k) for all k ∈ write_set
32     else
33         CRT_Write_Lock(k) for all k ∈ write_set
34         wait for all ACKs from storage                   ▷ Abort if fail
35         Send Commit to txn managers in r, r ∈ touchedRegions
36         ▷ Each transaction manager commits the transaction as IRT
37         wait for all ACKs from the txn managers          ▷ Abort if fail
38         Commit(txn)
39         Release_CRT_Read_Lock(k) for all k ∈ read_set
40         Release_CRT_Write_Lock(k) for all k ∈ write_set
41     end
```

it releases the acquired IRT_Read_Lock for key A and updates the lock type to CRT_Read_Lock. If key A is already exclusively locked by other CRTs, T aborts and retries by directly 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 changes are managed

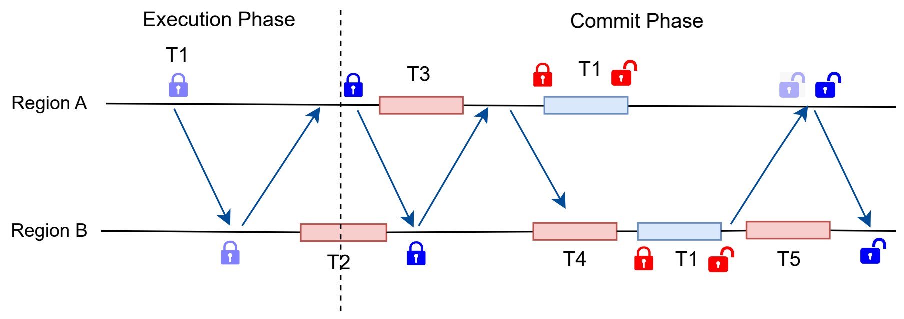


Fig. 5. A case study of Spanner-Relay's non-blocking feature.

through intra-region communication, there is no WAN overhead. Following the transaction logic, T computes its write set and proceeds to the commit phase.

Commit Phase. To commit, T acquire `CRT_Write_Lock` for the keys 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. That is, the CRT will acquire IRT locks within each region for the corresponding transaction piece of T in the region.

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.

4.1.3 The Case for Blocking. We illustrate the non-blocking feature of Spanner-Relay with an example illustrated in Figure 5. Assume T_1 is a CRT, T_2 to T_5 are IRTs executed using 2PC. When T_1 holds CRT locks, IRTs T_2 and T_3 can still acquire locks and execute as usual. When T_1 tries to acquire IRT locks, T_3 is executing, causing T_3 to be temporarily blocked until T_3 finishes. Subsequently, T_1 acquiring IRT locks and executes; it temporarily blocks the subsequent IRT T_5, ensuring sequential execution between IRTs and CRTs. Since this blocking does not exceed the execution time of a single IRT, it does not significantly affect CRT latency.

Trade-off Analysis. By differentiating between CRT and IRT locks, Spanner-Relay eliminates both "commit blocking" and "coordination blocking" for IRTs. The contention footprint for conflicting IRTs is minimized to two intra-region network round-trip communications in our example. While CRTs might incur slightly more communication costs between the coordinator and transaction managers, data locality ensures that IRTs are typically more dominant, critical, and sensitive to performance. In scenarios where CRTs make up the majority of the workload, the performance impact of the heterogeneous network is less significant. Therefore, Spanner-Relay can revert to the classic Spanner approach at runtime by using IRT locks for all transactions if needed.

4.2 Evaluation and Discussion

4.2.1 Implementation Details. We implemented Spanner-Relay in C++ on a third-party framwork [1]. 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% (i.e., a typical ratio observed in real-world applications [12, 15]) 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 5 min, with the first 30 s and the last 30 s excluded from results to avoid performance fluctuations during start-up and cool-down.

Testbed on AWS. All experiments, except for the sensitivity study, were conducted on Amazon EC2 using the m5.2xlarge instance type. Each node had 8 virtual CPU cores and 32GB of RAM. We used five 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 presented in a referenced table.

The database was partitioned into 60 data shards, with each region containing 12 shards and a replication level of 5 , alongside transaction managers. By default, 35 clients per region were used to achieve the peak throughput for both Spanner and Spanner-Relay.

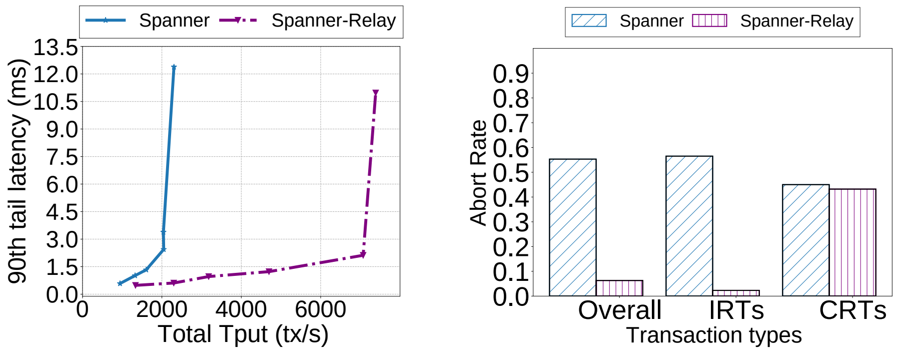
Sensitivity Study. We examined the sensitivity of intra-region latency because Relay relies on network heterogeneity to realize its potential. Since intra-region network latency cannot be adjusted in real-world deployments like AWS, we conducted a sensitivity study through simulation on a local cluster. The experiments were performed on a cluster of 10 machines, each equipped with a 2.6 GHz Intel E5-2690 CPU with 24 cores, a 40 Gbps NIC, and 64 GB of memory. Each data node was executed in a Docker container, and we used Linux tc [28] to control the round-trip time (RTT) among nodes. Each server was abstracted as an individual region, and the partition and replication policy mirrored the experiments conducted on AWS.

4.2.2 Performance Overview. We first use the default settings. To ensure a fair comparison, we avoided using prior knowledge of read and write sets in our experiments, even though the read and compose set of YCSB-T could be determined before execution. As shown in Figure 6a and Figure 7a, 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 6b and Figure 7b illustrate the results. For `NO_WAIT`, Spanner-Relay can efficiently reduce the abort rate for both IRTs and

	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)



(a) Latency-throughput Curve

(b) Abort Rate.

Fig. 6. Overall performance and abort rate of Spanner and Spanner-Relay on YCSB-T (default setting) using NO_WAIT.

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.

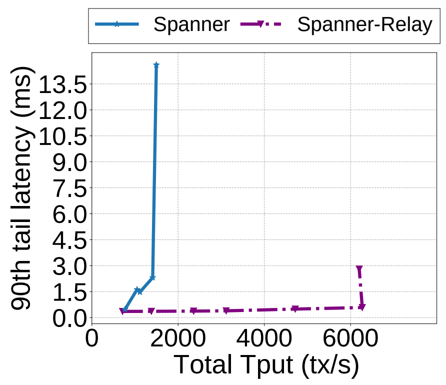
For WAIT_DIE, Spanner-Relay achieved a significantly lower average latency for IRTs, while the average latency of CRTs was roughly the same as in Spanner. This is because, in Spanner-Relay, CRTs will never block an IRT. The results on 90*th* latency support this claim. Both Spanner and Spanner-Relay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

4.2.3 *Impact of Experimental Parameters.*

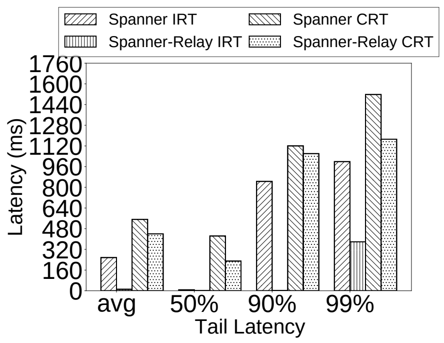
Next, we delve into understanding how Spanner-Relay and Spanner are affected by various workload parameters.

Concurrency. We first compare the performance of Spanner-Relay and Spanner under various concurrencies. As illustrated in Figure 8a and Figure 9a, 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, 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 8b and Figure 9b, when CRTs were enabled, Spanner experienced severe performance degradation (e.g., throughput dropping from 23,396 transactions per second to 5032 transactions per second when the CRT ratio increased from 0% to 5%),

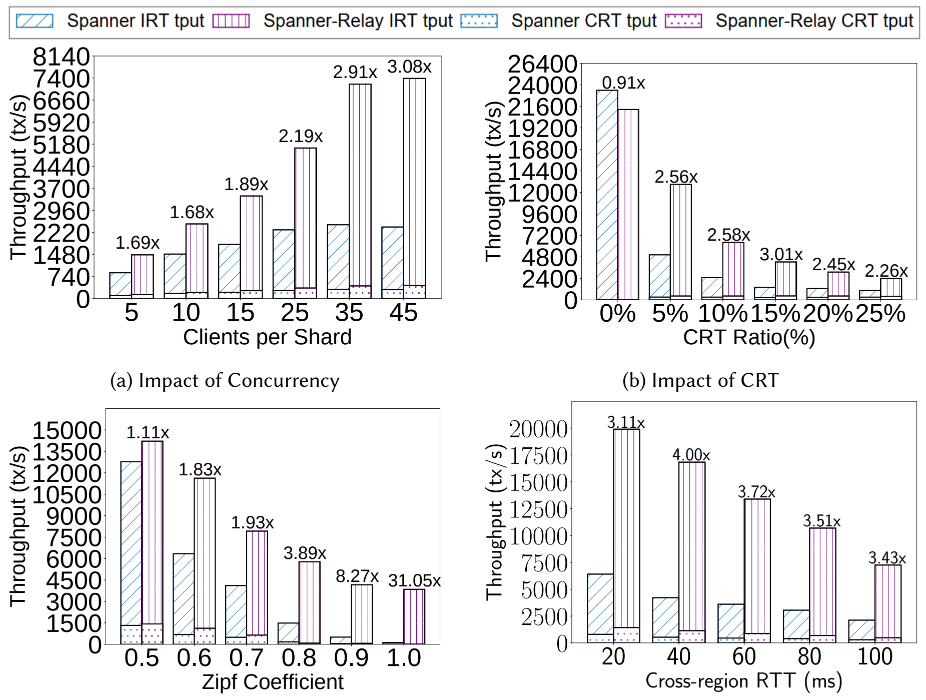


(a) Latency-throughput Curve



(b) Tail latency.

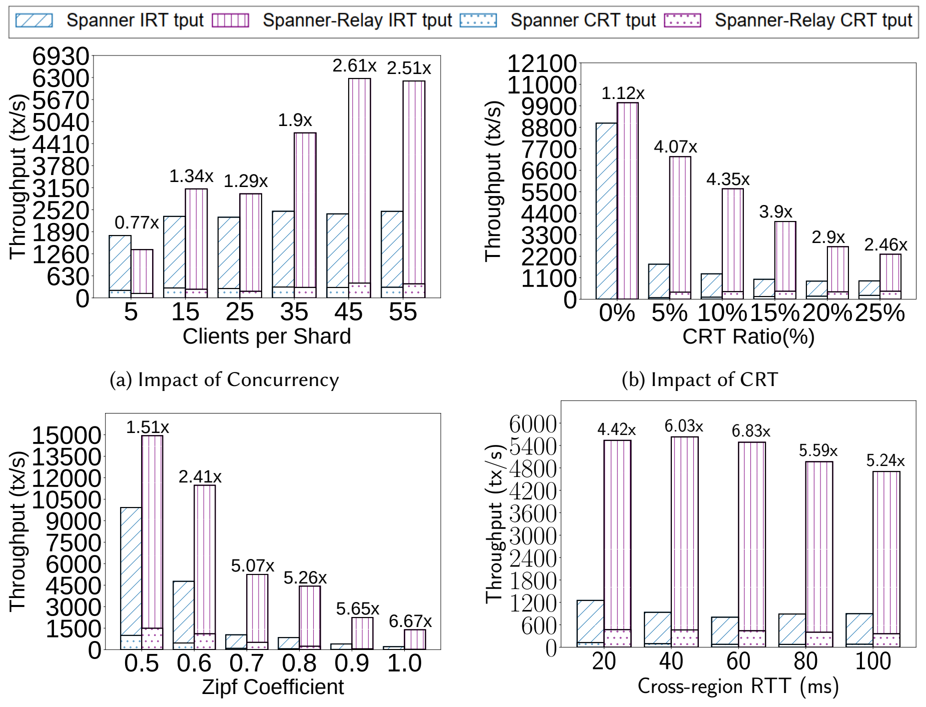
Fig. 7. Overall performance and latency of Spanner and Spanner-Relay on YCSB-T (default setting) using `WAIT_DIE`.



(c) Impact of Conflicts (d) Impact of RTT

Fig. 8. Performance of Spanner and Spanner-Relay on YCSB-T with different experimental parameters using `NO_WAIT`.

aligning with our discussion in Section 1 and Section 2.2. In contrast, Spanner-Relay's performance degraded slightly, attributed to eliminating cross-region costs for IRTs. In scenarios where all transactions were IRTs (i.e., a special case



(a) Impact of Concurrency

(b) Impact of CRT

(c) Impact of Conflicts

(d) Impact of RTT

Fig. 9. Performance of Spanner and Spanner-Relay on YCSB-T with different experimental parameters using `WAIT_DIE`.

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 (Section 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 8c and Figure 9c 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 Section 2.2, the contention window 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.

4.2.4 *Sensitivity Study.* We studied the sensitivity of cross-region network delays by simulation (see Section 4.2). Larger cross-region network delays generally result in longer transaction coordination and commit times for CRTs. The results, illustrated in Figure 8d and Figure 9d, indicate that Spanner-Relay outperforms Spanner under the different cross-region network delays. Spanner-Relay shows more improvements when the network delays are moderate (e.g., 40 s and 60 s for a cross-region RTT). 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 tps (transactions per second) to 476 tps as the network delay increased from 20 s to 100 s using `NO_WAIT`.

4.2.5 Evaluation Conclusion and Discussion. Our evaluation mainly considered YCSB benchmarks, which are widely used in the database community [9, 27, 37, 41]. Although other benchmarks, such as TPC-C, could potentially provide more insights, our codebase framework does not support them. Moreover, a recent study indicates that TPC-C, inherently as an I/O benchmark, may not be suitable for benchmarking concurrency control algorithms [58]. The improvement of Spanner-Relay comes from the high-skew that exists in real-world transactions, but TPC-C does not support directly setting high-skew data. While tuning TPC-C to remove wait time and using a small number of warehouses can introduce high contention, it introduces other issues such as a data set being too small or lacking locality.

Our evaluation shows that using Relay significantly enhances Spanner's performance in throughput, tail latency, and abort rate. The benefits of Spanner-Relay vary with different deadlock mechanisms. For example, with `WAIT_DIE`, Spanner-Relay reduces head-of-line blocking by prioritizing IRTs over CRTs. With `NO_WAIT`, it lowers the abort rate by using two lock types and independently ordering IRTs and CRTs.

Spanner-Relay can be a practical example, illustrating how Relay can assist multi-region databases in achieving a balance between consistency and performance. We compare Spanner-Relay to Spanner, not other geo-distributed systems, as our goal is to showcase Relay's potential using Spanner. Existing optimization techniques, like pre-write-log mechanism [64], can further enhance Spanner-Relay's performance and are complementary to our approach. 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 largely sacrificing performance.

5.1 Protocols and Implementations

5.1.1 CRDB Background. CockroachDB [51] (CRDB) is an open source production-grade database system that began as an external Spanner clone. Like Spanner, CRDB aims to build a resilient geo-distributed SQL Database with serializable ACID transactions. CRDB provides serializability for isolation and single-key linearizability (i.e., "no stale reads" for each key) by multi-version timestamp ordering. The transaction manager nodes in CRDB interact with clients, assign timestamps to transactions, and drive transaction coordination.

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 Section 6.1 for detailed comparisons between Relay and SKL. CRDB chose this design because, without Relay's consistency tiering, developers must pick between two extremes: enforcing strict serializability for all non-conflicting transactions or none. Strict serializability across regions can negate data locality benefits and degrade performance, so CRDB opted for the latter. As a result, CRDB cannot ensure causal relations between two transactions from the same client accessing different keys in the same region.

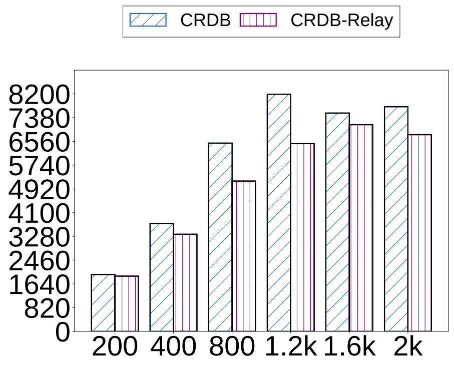
Algorithm 2: Algorithm of CRDB-Relay Coordinator

```
function CRDB-Relay Coordinator:
    inflightOps ← ∅                                    ▷ Ongoing operations.
    touchedRegions ← ∅                                 ▷ Regions involved in the transaction.
    txnTS ← now()                                      ▷ Timestamp of the transaction.
    for op ← KV operation received from SQL layer do
        if op.commit then
            op.deps ← inflightOps
            send ⟨commit, txnTS⟩ to transaction managers
            wait for all ACKs
        else
            r ← op.key.region
            if r ∉ touchedRegions then
                txnTS ← max(txnTS, GetFinishedTs(r))
                VerifyReads(txnTS)
                touchedRegions ← touchedRegions ∪ {r}
            end
            op.deps ← {x ∈ inflightOps |x.key = op.key}
            inflightOps ← (inflightOps- op.deps) ∪ {op} resp ← send(op, keyLeader(op.key))
            txnTS ← max(txnTS, GetFinishedTs(r))
            VerifyReads(txnTS)
        end
    end
```

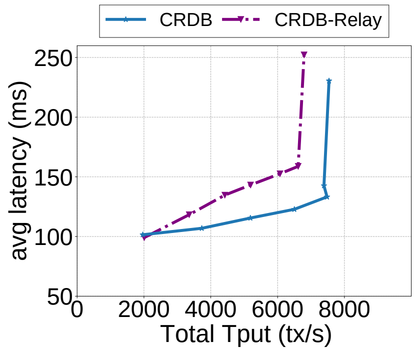
Relay offers a better design by ensuring linearizable consistency between transactions in the same region. To evaluate Relay, we redesigned CRDB's protocol to include multi-region semantics in conflict detection. We first discuss how CRDB coordinates transactions.

CRDB uses multi-version concurrency control (MVCC) to handle concurrent requests by performing reads and writes at commit timestamps. When a transaction is in conflict with other transactions (e.g., a write to a key at a 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). As conflict detection is conducted by key, CRDB does not guarantee the order between non-conflict transactions.

5.1.2 CRDB-Relay. Algorithm 2 shows the pseudocode of CRDB-Relay. We design Relay by reusing the timestamp adjustment mechanism in CRDB. In particular, 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, 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, required by hybrid logical lock). 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.

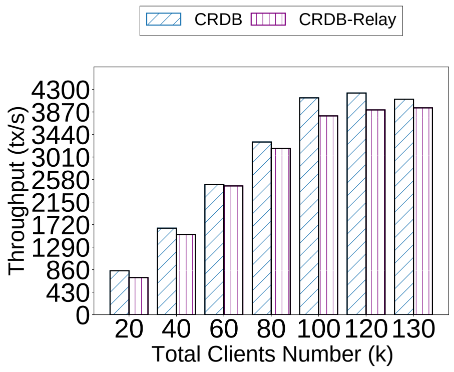


(a) Throughput.

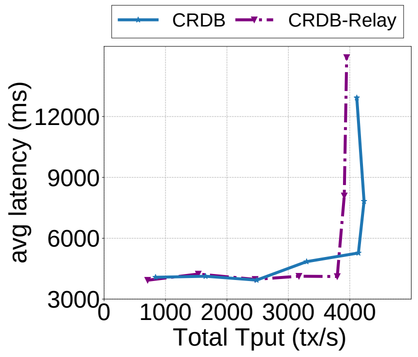


(b) Latency-tput Curve.

Fig. 10. Performance Comparison on YCSB-T (skewed).



(a) Throughput.



(b) Latency-tput Curve.

Fig. 11. Performance Comparison on TPC-C

Instead of using CRDB's passive mechanism, CRDB-Relay employs active inquiry to determine if a transaction T_1 may have finished before T_2 starts. Formally, T_2 should ask all data nodes inside a region to ensure that there does not exist 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.

CRDB uses transaction managers to maintain the status of each transaction, making this information accessible. CRDB doesn't use active inquiry for transaction ordering since it can be inefficient for all transactions to communicate with a single node in a geo-distributed setup. 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 ordering (Algorithm 2, Line 14). Using the active inquiry, CRDB-Relay can efficiently track ongoing transactions, thus tightening the consistency guarantees.

5.2 Evaluation and Discussion

Implementations Details. We implemented CRDB-Relay using the open source CRDB codebase 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 [51] (e.g., write pipelining, parallel commits, and follower read).
Workloads. We used YCSB-T and TPC-C [15] for our evaluation. To partition the Tables across multi-regions while compromising with data locality, we used the fine-grained partition framework supported by CRDB's code. 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 [11].
Testbed. We used the cloud environment recommended by CRDB for running the TPC-C benchmark(15 nodes on c5d.4xlarge machines). Each instance runs as a CRDB or a CRDB-Relay node. Besides, we run a transaction manager for each region coated with CRDB nodes.

5.2.1 Performance Overview. **Performance on YCSB-T.** We compared the performance of CRDB and CRDB-Relay on default YCSB-T (Section 4.2). The results are shown in Figure 10. 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 the 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 less than $\sim$ 10%.
Performance on TPC-C. Figure 11 shows the results. We first calibrated the performance of CRDB with the results presented in its original paper. As shown in Figure 11a, the peak throughput of CRDB was $\sim$ 4200 tps (113,520 tpmC), roughly the same as [55]. Then, we compare the performance between CRDB and CRDB-Relay. The results show that CRDB-Relay 's peak throughput (4080 tps) is slightly lower ($\sim$ 3%) than CRDB, while the latency results are similar.
Conclusion. Our evaluation indicates that CRDB-Relay matches the performance of CRDB while offering strong consistency guarantees for real-time orders. CRDB-Relay serves as a practical example of how Relay can help multi-region databases using weaker consistency models to enhance 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) [27] 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 13b). 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 13b, 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

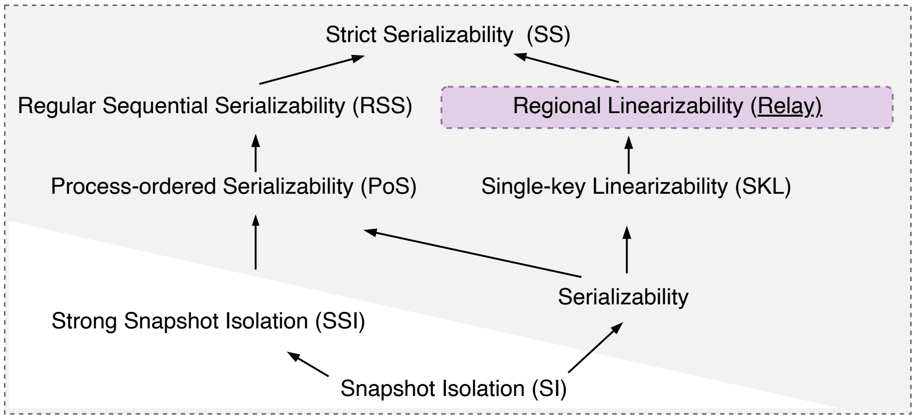
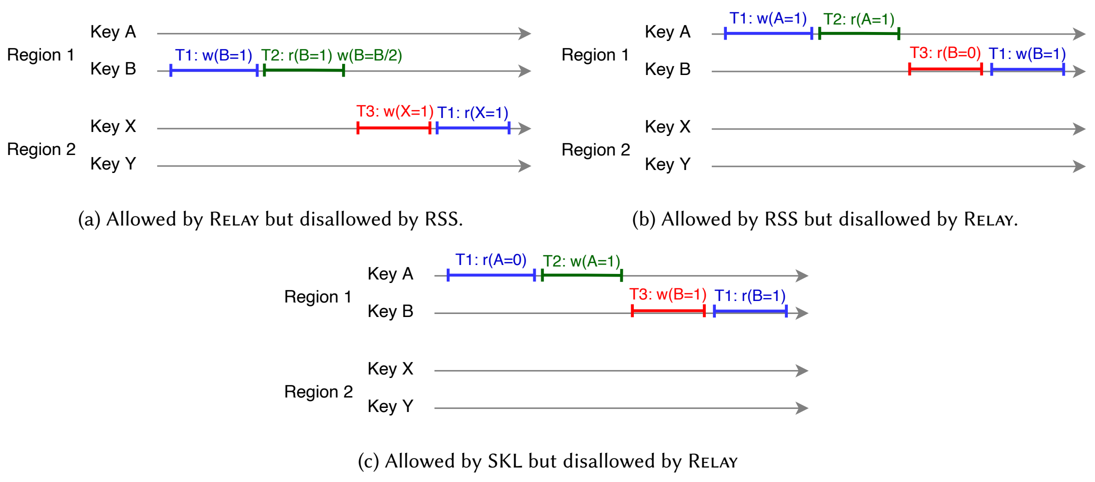

Strict Serializability (SS)
Regular Sequential Serializability (RSS)
Regional Linearizability (Relay)
Process-ordered Serializability (PoS)
Single-key Linearizability (SKL)
Serializability
Strong Snapshot Isolation (SSI)
Snapshot Isolation (SI)

Fig. 12. This diagram shows how Relay compared to its proximal consistency models: SS [46], RSS [26], SKL [51], PoS [16, 39], SSI [22], serializability [56], and SI [5]. The model in a grey background ensures serializable isolation.



(a) Allowed by Relay but disallowed by RSS.

(b) Allowed by RSS but disallowed by Relay.

(c) Allowed by SKL but disallowed by Relay

Fig. 13. Comparison of Relay with proximal levels of consistency models.

between T_2 and T_3 is disrupted: the real-time order between T_2 and T_3 is $T_2 \longrightarrow T_3$; the serializable order enforced by RSS is $T_3 \longrightarrow T_1 \longrightarrow T_2$, which implies $T_3 \longrightarrow T_2$. 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 13a).

Single-key Linearizability (SKL) was proposed by [51]. 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 13c 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 $T_3 \longrightarrow T_1 \longrightarrow T_2$. 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 implemented along with PoS by associating each client to a home region.

6.2 Transaction Priority

Compared to strict serializability (SS), one of the pivotal innovations of Relay is its ability to prioritize transactions based on their type and current state dynamically. Specifically, Relay schedules IRTs ahead of CRTs until the order of CRTs is firmly established. This prioritization ensures that IRTs, which typically require faster processing, are not delayed by the more complex ordering requirements of CRTs. Therefore, Relay effectively reduces latency for time-sensitive operations. Once the order of CRTs is established, Relay adjusts its scheduling to treat both IRTs and CRTs with equal priority, ensuring fairness and efficiency in execution. This dynamic adjustment not only enhances performance but also maintains the integrity and consistency of the database.

In this context, multi-region system developers can leverage existing transaction priority protocols [3, 19, 24, 25, 29] to transition from a monolithic consistency model (e.g., SS) to Relay. For instance, many lock-based DBMSs expose interfaces for users to specify transaction priority levels. Each transaction is assigned a unique timestamp before execution, with smaller timestamps indicating higher priority. When a transaction requests a lock and is denied, it will enter the lock wait queue if it has a higher priority than the current lock holder [42, 51]. Polaris [61] 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, these priority designs can inspire or be used by monolithic concurrency control protocols to achieve Relay.

6.3 Mixed Consistency Models

Several prior works [23, 31, 32, 35, 36, 43, 57, 59] have been proposed to manage both weakly and strongly consistent transactions within a database.

Mixed Replication Consistency. Pileus [52] allows users to specify consistency guarantees in the level of service agreements. Red-Blue consistency [35] allows strongly and causally consistent operations to co-exist in a single system and rely on application developers to make choices. AutoGR [57] automatically identifies the minimal set of the required consistency guarantees based on applications using the Z3 theorem prover. Walter [49] introduces parallel snapshot isolation to replicate data asynchronously. Different from these works, Relay studies the blocking issues in transaction coordination, which is orthogonal to replication policies. In particular, Relay provides serializability (i.e., the strongest isolation levels) for all transactions and tailors real-time properties to achieve high performance.

Mixed Consistency in Transaction Order. MixT [43] advocates that consistency is a property of information. It proposes a new embedded language enabling users to configure the consistency guarantees for each operation manually. Differently, Relay does not depend on prior application knowledge to manually set distinct consistency guarantees for different transactions or operations inside transactions while approaching best-of-effort performance and consistency.

7 Conclusion

This work analyzes the limitations of monolithic consistency models, which are ill-suited for heterogeneous networks due to their rigidity. We propose Relay, a novel consistency model by consistency tiering. We design, implement, and evaluate two system variants: Spanner-Relay and CRDB-Relay, to showcase the usage of Relay. Our evaluation shows that Relay significantly improves Spanner's performance and strengthens CRDB's consistency, highlighting the potential of consistency tiering in distributed databases.

References

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

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

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

[4] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414. IEEE, 2016.

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

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

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

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

[9] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

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

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

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

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

[14] 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. Spanner: Google's globally-distributed database. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012.

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

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

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

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

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

[20] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2014.

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

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

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

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

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

[26] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 163–179, 2021.

[27] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 163–179, 2021.

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

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

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

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

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

[33] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

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

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

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

[37] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

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

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

[40] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. Ncc: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall. *arXiv preprint arXiv:2305.14270*, 2023.

[41] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. 2020.

[42] Andrei Matei. Cockroachdb's consistency model. https://www.cockroachlabs.com/blog/consistency-model/, 2019.

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

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

[45] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

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

[47] Postgresql. https://www.postgresql.org, 2012.

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

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

[50] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 304–316. IEEE, 2019.

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

[52] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 309–324, 2013.

[53] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[54] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 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*, May 2014.

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

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

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

[58] Yang Wang, Miao Yu, Yujie Hui, Fang Zhou, Yuyang Huang, Rui Zhu, Xueyuan Ren, Tianxi Li, and Xiaoyi Lu. A study of database performance sensitivity to experiment settings. *Proceedings of the VLDB Endowment*, 15(7), 2022.

[59] Yingyi Yang, Yi You, and Bochuan Gu. 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)*, volume 1, pages 183–190. IEEE, 2017.

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

[61] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. Polaris: Enabling transaction priority in optimistic concurrency control. *Proceedings of the ACM on Management of Data*, 1(1):1–24, 2023.

[62] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.*, 35(4):1–37, December 2018.

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

[64] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment*, 16(6):1372–1385, 2023.

[65] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. Lion: Minimizing distributed transactions through adaptive replica provision (extended version). *arXiv preprint arXiv:2403.11221*, 2024.