Region-Linearizable Serializability: A Practical Consistency Model for Multi-Region Distributed Transactions

Ruijie Gong, Haoze Song, Xusheng Chen, Sen Wang, Heming Cui

Abstract—**Deploying databases across multiple regions has become the de facto choice for cloud-native applications that desire high availability, strong scalability, and efficient service localization. However, supporting serializable transactions in such databases presents a significant challenge. Coordinating cross-region transactions (CRTs) is inherently slow due to the extensive geographic distances. Even a few slow transactions can significantly degrade the performance of the entire system. Despite various proposals introduced to optimize the performance by either eliminating CRTs or minimizing the cost of cross-region coordination, CRTs remain crucial for general workloads, with their costs still being dozens of times higher than those of intra-region transactions (IRTs).**

This paper contends that existing serializable consistency models are not well-designed for multi-region deployments. The root cause is the strong heterogeneity in deployments: certain transactions (e.g., CRTs) experience significantly higher committing latency than others. To address this, we propose a new layered consistency model specifically tailored for multi-region deployments: region-linearizable serializability (RLS). Specifically, RLS ensures strict serializability (i.e., linearizability) for IRTs from the same region and provides regular serializability for CRTs.

To demonstrate the efficiency and applicability of our new consistency model (i.e., RLS), we design, implement, and evaluate variations of two representative database systems: Spanner and CockroachDB. Our evaluation demonstrates that these variations can significantly enhance performance (e.g., $4.5\times$ throughput for Spanner on average) or provide better functionality without performance degradation (e.g., more consistency guarantees for CockroachDB).

I. Introduction

TODAY, cloud providers (e.g., AWS [1], Azure [2], and Huawei [3]) host computing infrastructures across multiple global locations. These infrastructures are typically organized into regions, each strategically designed to offer computing services in proximity to clients. As a result, multi-region deployment has emerged as a prominent choice for cloud-native applications aiming for stringent client-perceived latency, high availability, strong scalability, and effective service localization [4]–[8].

In pursuit of scalability and availability, these applications frequently partition and replicate their data storage across multiple servers (nodes). Each partition, known as a data shard, maintains a primary replica. The primary replica is consistently deployed in the region where the majority of access requests originate, ensuring optimal locality. Figure 1 provides an overview of our multi-region deployment model, which is fueled by the desire of global companies to not only build scalable applications but also control with fine granularity where data resides for good performance and meet the data governance policies (e.g., GDPR [9]).

However, supporting serializable ACID transactions in such applications presents a perpetual challenge. Coordinating a cross-region transaction (CRT) is always slow due to geographic distances. For example, a cross-region transaction incurs a $\sim 50ms$ network delay from Hong Kong to Singapore, whereas the network delay within a region is less than $2ms$ when powered by modern network technologies (e.g., dedicated inter-datacenter networks [1]).

Several impressive works (e.g., [5], [10]–[13]) have been proposed to enhance the performance of geo-distributed transactions. These strategies involve redesigning critical aspects of concurrency control protocols or eliminating CRTs by making certain assumptions about the workloads. However, 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 IRTs is still significant. For instance, Detock [10], a state-of-the-art geo-distributed transaction protocol meticulously optimized for CRTs, can execute and commit CRTs in a single cross-region network communication but still incurs an average latency of approximately $\sim 100ms$ for CRTs and $\sim 5ms$ for IRTs under default experimental setups. Furthermore, according to the real-world studies [5], [10], [14], as well as our experimental evaluations, even a few slow CRTs can entangle numerous IRTs, leading to deadlocks or aborts. This substantially degrades the overall database performance.

Diverging from these works, we address multi-region transactions by reevaluating and redesigning existing consistency models. Specifically, we contend that current 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. We contend that a diverse range of consistency guarantees should be accessible for different transaction types, all while upholding strong consistency in requisite scenarios.

In reality, existing consistency models tend to be either overly strong or can be further enhanced without performance degradation. Aggressive models (e.g., strict serializability model [15]) abstract the entire system as a single node, necessitating

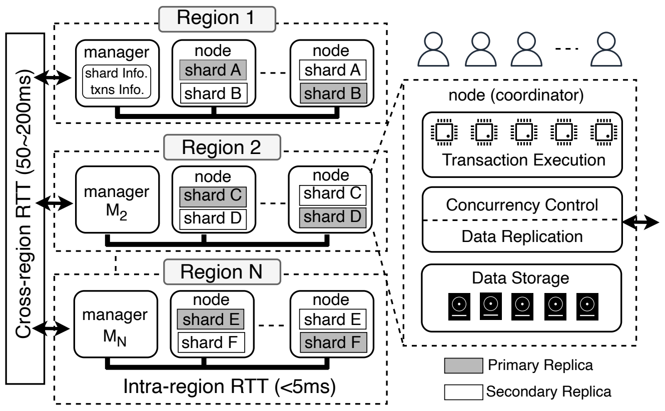


Fig. 1. This diagram presents a typical deployment for multi-region databases. The database is partitioned into multiple data shards spanning over multiple regions. Each shard includes a primary replica and several secondary replicas. Intra-region communications are much faster than inter-region communications.

heavy synchronizations between all computing servers. This approach leads to poor performance in multi-region deployments. Moreover, this model fails to preserve the advantages of near-client computing: even a local transaction has to be ordered with cross-region ones, which is at odds with the motivation of multi-region deployment. Other models provide weaker consistency (e.g., strong snapshot isolation [16]). While these models may suffice for certain application scenarios, the consistency guarantees for local transactions can be further enhanced without much performance overhead: the communication cost for coordinating local transactions can be cheap when using modern hardware and networks.

In this paper, we propose region-linearizable serializability (for short, RLS), the first consistency model to provide as strong as possible consistency for multi-region databases. RLS treats CRTs and IRTs differently. It ensures strict serializability (i.e., the strongest consistency guarantee) for IRTs from the same region and provides regular serializability for CRTs. Specifically, RLS ensures serializability and "no stale reads" property (a.k.a. regularity) for all transactions (i.e., both CRTs and IRTs) while preserving real-time order only for the transactions that at least access one same region. For a formal definition, we refer readers to §III.

To demonstrate the efficiency and applicability of RLS, instead of creating a new transaction protocol from scratch, we design, implement, and evaluate variations of two database systems: Spanner and CockroachDB (for short, CRDB). We refer to these variants as Spanner-RLS and CRDB-RLS, respectively. We chose these two database systems because they complement each other in both the consistency model (strict serializability versus single-key linearizability) and the design of concurrency control protocols (two-phase locking versus timestamp ordering). These two proof-of-concept prototypes pave the way for efficient and practical optimization of distributed protocols when deployed across multiple regions based on a correct-by-construction approach.

Specifically, Spanner and Spanner-RLS follow the design of traditional pessimistic concurrency control: it orders transactions using locks and commits transactions using the two-phase commit. Our variation (Spanner-RLS) significantly reduces CRTs' contention footprint (i.e., locking duration), thus allowing more parallel concurrency. As a result, Spanner-RLS attains $1.16\times$ to $89.01\times$ higher throughput than Spanner while having similar latency.

We present CRDB-RLS as a practical example to demonstrate that databases using weaker consistency models can also benefit from RLS by tightening their consistency guarantees. The original consistency model (i.e., single-key serializability) used by CRDB is considerably weaker than RLS. We implemented regional semantics into CRDB's conflict detection protocol, requiring all conflicting IRTs to be ordered within the regions, even if they access different keys. Given that intra-region communication is significantly cheaper than cross-region communications, the performance overhead of CRDB-RLS can be ignored (e.g., less than $\sim 15\%$ in our evaluation). Conversely, the stronger guarantee efficiently eliminates anomalies caused by violating the real-time order, thus simplifying application development [17], [18].

Contributions. In summary, this paper's contributions stem from a fundamental insight that existing consistency models are inadequately designed for multi-region deployment. To our knowledge, this paper provides the first tailored consistency model for multi-region transactional processing. Our contributions are four-fold:

- We systematically analyze the multi-region deployment model and highlight the limitations of existing consistency models.
- We implemented two distinct system variations, Spanner-RLS and CRDB-RLS, to demonstrate the efficiency and applicability of our novel consistency model (RLS). Both of the two prototypes are built on open-source codebases, and the source code is available at https://github.com/vldb24p771/spanner_rls and https://github.com/vldb24p771/crdb_rls, respectively.
- We extensively evaluate these variations and present compelling results showcasing the substantial performance improvements achieved by RLS. Specifically, RLS enhances Spanner's performance, achieving throughput improvements ranging from $1.16\times$ to $89.01\times$, and provides more robust consistency guarantees for CRDB without significant performance degradation.
- Spanner-RLS and CRDB-RLS can serve as practical templates for the future adoption of RLS to other concurrency control protocols.

The rest of the paper is organized as follows. §II discusses the system model of our multi-region databases deployment, the background of geo-distributed transaction processing, and the motivating applications. §III details our new consistency model: RLS. §IV illustrates the application of RLS to Spanner. §V delves into CRDB and CRDB-RLS. Finally, a discussion of related works is presented in §VI, and §VIII concludes the paper.

II. Background and Motivation

This section provides background on multi-region databases and the issues of existing geo-distributed transaction protocols.

System	Transaction Protocol	Consistency Models	Coordination Blocking
Spanner [17]	Read-write: 2PL + 2PC; Read-only: TSO	SS	Yes
Calvin [19]	Centralized coordinator	SS	Yes
Slog [11]	CRT: Centralized coordinator; IRT: Intra-region Sequencer	SS	Yes
Detock [10]	CRT: Dependency-graph; IRT: Intra-region Sequencer	SS	Yes
Janus [13]	Dependency-graph	SS	Yes
Epaxos [20]	Dependency-graph	SS	Yes
Ocean Vista [5]	TSO (Watermark)	SS	Yes
CRDB [6]	TSO (HLC)	SKL	Yes
RedT [21]	2PL + 2PC	Serial.	Yes
Tapir [22]	Variant of OCC	Serial.	No (by aborting IRTs)
MDCC [23]	Paxos	SI	Yes

TABLE I
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.

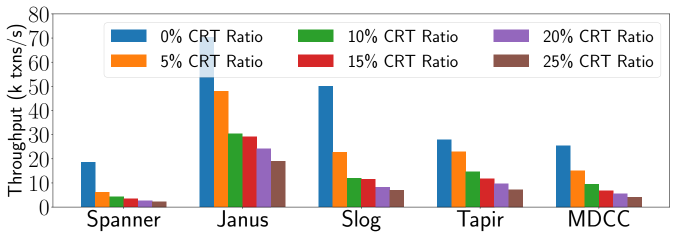


Fig. 2. Impact of CRT ratio on the throughput of the state-of-the-art geo-distributed transaction systems.

A. *Multi-region Databases*

Multi-region infrastructures motivate our design of RLS. Deploying a multi-region application has the following advantages.

- **[Low data access latency]** Multi-region deployment enables the placement of data in proximity to active client concentrations across different regions. Applications are designed to predominantly access a data shard from a single region, resorting to cross-region data access only when necessary. This approach facilitates applications in offering global data access with ultra-low latency.
- **[Meet data governance policy]** Privacy regulations impose multi-region deployments for global services with strict requirements on data residency. For instance, the General Data Protection Regulation (GDPR) forbids replicating European citizens' data outside. Thus, multi-region deployments are becoming the de facto choice for multinational companies.
- **[Flexible failure model]** Multi-region deployment allows for data replication across regions, significantly enhancing availability by tolerating complete region failures. In practice, replicating data globally can be costly. Users have the flexibility to define distinct replication policies based on data criticality, such as replicating critical data across regions and keeping other data within the same region.

A typical multi-region deployment model is illustrated 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 is equipped with 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.

B. *Geo-distributed Transaction Protocols*

Many influential works have been proposed to optimize the performance of geo-distributed transaction processing. We summarize the state-of-the-art systems in Table I. All of these systems support at least serializability as their consistency model or can be enhanced to serializability with minor modifications (e.g., MDCC).

Generally, these systems focus on optimizing the cost of coordinating transactions from the protocol scope. For instance, some works attempt to reduce either the number or the overhead of network round-trips in transaction coordination. Tapir [22] improves Spanner's performance by integrating two-phase commit and consensus protocols into a single framework, eliminating redundant coordination and reducing the WAN round-trips. RedT [21] enhances system performance by further decreasing the network round-trips. RedT targets RDMA-capable networks for local communication and employs a pre-write-log mechanism to eliminate the synchronization of `prepare` messages (i.e., the first phase in two-phase commits) from the coordinator to primary replicas.

Calvin [19], Slog [11], Detock [10], Ocean Vista [5], Epaxos [20], MDCC [23], and Janus [13] follow the design of deterministic databases, 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 (as illustrated in §III-B). 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 [5], [14]). 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) for overly strong consistency guarantees.

Consequently, these systems can 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. For detailed evaluation setups, we refer readers to §IV-B. 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).

RLS addresses such issues by rethinking the limitations within existing consistency models and thus is fundamentally different from these proposals. RLS trades off consistency for performance with minimal intrusion (i.e., the consistency tradeoff in RLS is tightly necessary for addressing blocking issues). We regard RLS as orthogonal to these advanced geo-distributed transaction protocols. Consequently, new protocols may benefit from the methodology of RLS and the key optimizations of these advanced protocols.

III. Region-linearizable Serializability

In this section, we formally define our new consistency model: Region-linearizable Serializability (RLS), which is specially tailored for the database systems that are deployed in near-client computing facilities (e.g., Regions [1] in AWS). For comparisons with other serializable consistency models, we refer readers to §VI-A.

A. Definition of RLS

For clarity, we adopted the formalism from existing works [24]. Table II summarizes the notations used in our definition, which will be further illustrated later. 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 transaction processing across many nodes (servers), which is common in multi-region deployed data-intensive applications.

Shard Groups (e.g., grouped by regions). Grouping semantics is one of the foremost distinctions of RLS compared to other serializable consistency models. Specifically, RLS divides data shards into disjoint groups and ensures linearizability (i.e., the strongest consistency) for intra-group operations. For inter-group operations, RLS provides regular serializability.

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. This division serves different purposes; sharding is for horizontally scaling the service to run on many servers, and the division policy is usually for reducing the ratio of cross-shard transactions to achieve high efficiency [10]. On the other hand, grouping is a consistency strategy where cross-group external ordering requirements are typically less important. Such semantics are usually related and directed by geo-distributed (multi-region) deployments.

We regard this two-level framework as essential because it efficiently bridges the division based on application semantics (e.g., the data items grouped by warehouse ID in TPC-C [25]) and division based on deployment topology (e.g., the data shards grouped by regions in geo-distributed deployments).

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 on 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 RLS, 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 [10], [11], [19], [20], [23]. Essentially, RLS supports general transaction semantics (to be illustrated in our example system: Spanner-RLS, §IV). In RLS, we consider a general transaction $T_1 = \{r(x, n), w(n, v)\}$ that reads the value of key x as the key for the write operation, where $\mathcal{W}_{T1}$ can not be obtained before execution.

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 in other papers). We say a transaction T is relevant to shard group g if T accesses at least one key owned by g, and we 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.

Real-time order. An order of transactions is usually considered as a set of *return before* relations [26]. In our paper, we say a transaction T_1 precedes another transaction T_2 if T_1 finishes

Ops	o	Database operation, e.g., read, write, insert, scan.
	$r(k,v)$	Read the value v using key k
	$w(k,v)$	Write value v for key k
Txns	T	Txn consists of operations (Σ_T) with 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, $\mathcal{H} = \bigcup \mathcal{H}_i$
	$\mathcal{S}$	Totally ordered serializable schedule for all txns
	$\xrightarrow{rb}$	Real-time order imposed by runtime execution
	$\xrightarrow{so}$	Oreder for operations in $\mathcal{S}$
	$<_S$	Oreder for transactions in $\mathcal{S}$

TABLE II
PRELIMINARIES AND NOTATIONS FOR RLS.

(commits) before T_2 starts (i.e., arrives at the database system), denoted as $T_1 \xrightarrow{rb} T_2$.

Definition of RLS. We then define RLS using the notations above. We say that an OLTP service ensures RLS, if for all execution histories, $\mathcal{H} = \bigcup \mathcal{H}_i$, are 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

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

Therefore, the property ❷ infers that $\xrightarrow{so}$ defines a total order $<_S$ among all transactions.

- *No Stale Reads.* Formally, for any two transactions T_1 and T_2

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

- *Real-time Ordering inside all Shard Groups.* Formally,

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

B. Performance Issues in Strict Serilizability

Strict serializability (SS), the most substantial consistency level for distributed databases, ensures that a replicated distributed database works as a single node that executes all client transactions serially. The serial order respects the real-time relations (i.e., the "return before" relation in §III-A) among all client transactions.

However, the strong guarantees of SS always come up with high-performance costs, especially when deployed in a multi-region environment. This has led both academia and industry to seek weaker consistency models. For example, numerous new consistency models were proposed in recent years (see §VI-A), and almost all industrial systems do not provide SS by default.

C. Practical Implications

In essence, the guarantee of SS is considered excessive for many multi-region application scenarios. Specifically, ensuring a real-time relation between transactions pertains to external (out-of-band) causal relations among transactions. Since a system is unaware of external relations, SS regards *all* pairs of transactions without overlapping lifetime as potentially causally related and pertains to their ordering, albeit most transactions are independent.

RLS ensures the "no stale reads" property for all transactions, effectively preventing most application-level anomalies [27]. Additionally, RLS enforces real-time ordering among transactions accessing interleaved regions (i.e., conflict IRTs and CRTs), including transaction ordering requirements inferred by transitivity. Compared to SS, the only anomalies in RLS may arise from the potential disruption of real-time ordering among transactions happening independently within non-overlapping regions.

We argue that such anomalies do not compromise the correctness of multi-region databases for two primary reasons. First, multi-region databases optimally leverage data access locality to assign shards to regions (see §II-A). Typically, each region manages (e.g., being the leader of) shards containing data of nearby clients, making two transactions accessing non-overlapped regions causally unrelated. Thus, prioritizing their real-time order will not introduce application-level anomalies.

Second, the time window for breaking causal relations is narrow. RLS necessitates "no stale reads" for all transactions, whether intra-region or cross-region. 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 $T_3 <_S T_1 <_S T_2$ (as depicted in Figure 11a). However, as RLS also mandates "no stale reads", T_1 must be concurrent with T_2 and T_3, implying that the external causal relation must conclude within T_1's lifetime.

Therefore, RLS possesses the unique potential to significantly enhance the scalability and latency of multi-region databases while maintaining correctness and programmability. RLS stands out as the pioneering consistency model that takes into account real-world deployments and the inherent locality feature of data.

IV. SPANNER AND SPANNER-RLS

In this section, we present the design, implementation, and evaluation of Spanner and Spanner-RLS. We regard Spanner-RLS as a specific instance of using our new consistency model (RLS) to advance the performance of existing strictly serializable databases.

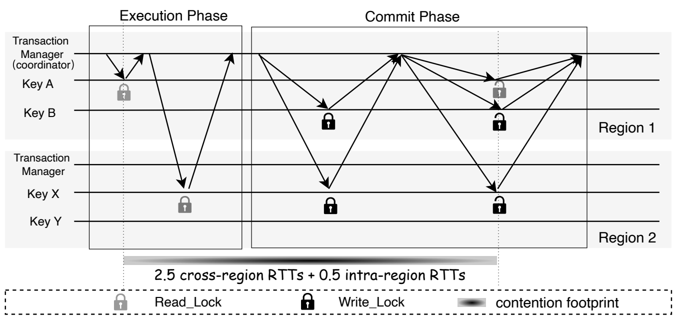


Fig. 3. This diagram shows how Spanner orders a CRT using 2PL and commits it using 2PC in a multi-region deployment. Replicas are removed for readability.

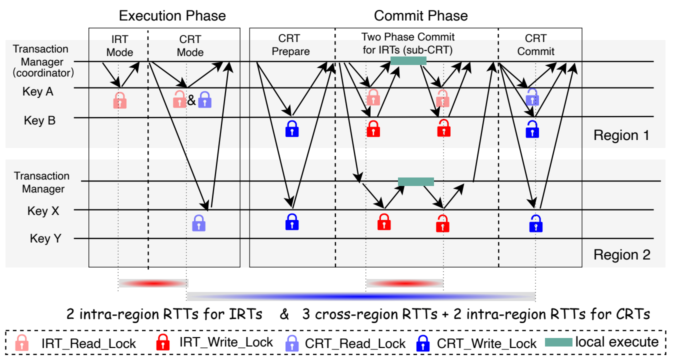


Fig. 4. This diagram shows how Spanner-RLS orders a CRT using a variant of 2PL and commits it using 2PC.

A. *Protocols and Implementations*

Spanner Background. Google's Spanner provides strict serializability (a.k.a. external consistency) for read-write transactions by coordinating them using two-phase locking (2PL) and then committing the transactions using two-phase commit (2PC).

Figure 3 presents an example, where 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. In this example, 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. As such, T is successfully executed and committed.

We are now prepared to introduce the performance issues in Spanner. Spanner coordinates intra-region transactions (IRTs) and cross-region transactions (CRTs) similarly, where a read lock held by a CRT will block all writes from both CRTs and IRTs, and an exclusive lock held by a CRT will prevent all reads and writes, correspondingly. Consequently, a CRT's contention footprint (i.e., the lock duration) is extremely large. More than two cross-region network round trips will block all conflict transactions. In our example, it takes 2.5 cross-region network round trips and 0.5 intra-region round trips. Even worse, 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. Consequently, all other IRTs and CRTs that access the key Y have to compete with T' for ownership of the lock on the key Y, enlarging the affected key space.

Spanner-RLS. Following the methodology of our new consistency model (RLS), we treat the IRTs and CRTs differently in the variation of Spanner, termed Spanner-RLS. To achieve 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-RLS, and we highlight the regional semantics in blue. Figure 4 illustrates how Spanner-RLS executes and commits the same transaction T. Without loss of generality, we assume that read and write sets of a transaction are unknown to the transaction manager. Therefore, Spanner-RLS can support general transactions without prior knowledge. 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 much overhead. Then, following the transaction logic, T computes its write set and proceeds to the commit phase.

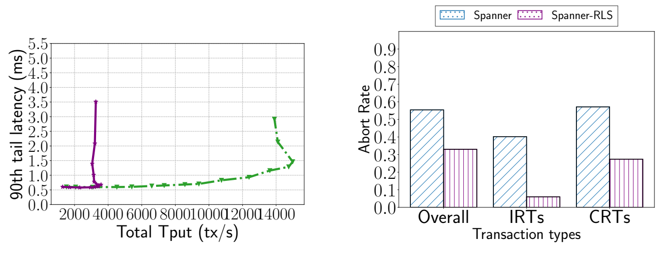
In the commit phase, T acquire `CRT_Write_Lock` for the key B and X. When all `CRT_Write_Lock` is successfully acquired (i.e., the order between T and other CRTs has been determined), the coordinator notifies all transaction managers of the region that T accessed. Each transaction manager commits T independently 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 of any IRTs that have modified the read keys of T, we re-execute it locally. The tricky 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 now have 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

Algorithm 1: Algorithm of Spanner-RLS

```
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 transaction commands, which triggers 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
37
```

read keys of $region$ 1 and $region$ 2 has been changed). In such cases, we can easily revert from Spanner-RLS to Spanner by using IRT locks directly for the CRT.

We then analyze the tradeoff in Spanner-RLS. By differentiating CRTs and IRTs locks, Spanner-RLS 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, in practical workloads, IRTs are always the dominators and are 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-RLS can fall back to the classic Spanner switchable at runtime.



(a) Overview (b) Abort Rate.

Fig. 5. Overall performance and abort rate of Spanner and Spannner-RLS on YCSB-T (default setting) using `NO_WAIT`.

Read-only Transactions. Leveraging error-bounded timing service (e.g., TrueTime API), Spanner can execute read-only transactions in a single network round trip. Using its TrueTime API, Spanner assigns a commit timestamp to each transaction, guaranteed to be between the transaction's real start and end times. Therefore, when using the TrueTime API for read-only transactions, they can safely read from the replicas without coordination. Spanner-RLS follows this design for enhanced performance. In our evaluation, we emulated TrueTime error as 10 ms, which is used in the previous paper [24] and matches the p99.9 value observed in practice.

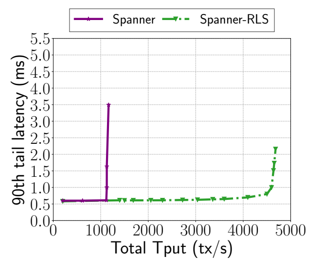
B. Evaluation and Discussion

1) Experimental Setups: We implemented Spanner-RLS in C++ utilizing the third-party implementation [28] 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 implemented as stored procedures containing read and write operations over a set of keys.

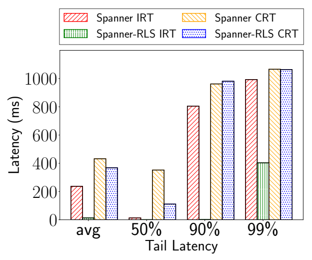
Cluster Setups. All experiments were conducted on our cluster comprising 10 machines, each with a 2.60GHz Intel E5-2690 CPU with 24 cores, 40Gbps NIC, and 64GB memory. We executed each data node (shard) in a docker container and utilized tc [29] to regulate the RTT among nodes. The server ran on the Ubuntu 18.04 operating system.

Deployments. To simulate a multi-region deployment, we abstracted each server as an individual region. Subsequently, We set the cross-region round-trip latency as $50ms$ using tc, aligning with the real-world statistics [30]. We partitioned the database into 300 data shards, with each region containing 30 data shards and a replication level of 3, alongside transaction managers. By default, we utilized 40 clients per region to attain the peak throughput for both Spanner and Spanner-RLS.

Workloads. We employed the standard YCSB-T benchmark for our evaluation. We generated a total of $3,000,000$ keys, distributing $100,000$ 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



(a) Overview



(b) Tail latency.

Fig. 6. Overall performance and latency of Spanner and Spanner-RLS on YCSB-T (default setting) using `WAIT_DIE`.

experiments lasted 3 minutes, with the first $30s$ and the last $30s$ excluded from results to avoid performance fluctuations during start-up and cool-down.

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

- **`NO_WAIT`.** When using this deadlock mechanism, if a lock request is denied, the database will immediately abort the requesting transaction, and the client will retire the transaction.
- **`WAIT_DIE`.** Unlike `NO_WAIT`, `WAIT_DIE` 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 and will be retired by the client.

We do not consider other deadlock mechanisms since they are either subsumed by the two mechanisms or will incur significant overhead in a multi-region deployment. For instance, deadlock detection necessitates a centralized deadlock detector for cycle detection, which can be expensive due to cross-region communication.

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-RLS significantly outperformed Spanner on YCSB-T. In particular, Spanner-RLS achieved 3.95× and 4.27× higher peak throughput when utilizing `NO_WAIT` and `WAIT_DIE`, respectively. Spanner-RLS's $90th$ resembled that of Spanner, essentially representing the IRTs latency. We observed that employing the `NO_WAIT` mechanism resulted in substantially higher throughput than using `WAIT_DIE`, given the default 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-RLS achieved the peak throughput. Figure 5b and Figure 6b illustrate the results.

Regarding `NO_WAIT`, Spanner-RLS can efficiently reduce the abort rate for both IRTs and CRTs. The overall abort ratedecreased from 56% to 33% (i.e., 41.1% reduction). In particular, Spanner-RLS achieved a more significant reduction for IRTs (from 40% to 6%) due to the "non-blocking" property in IRT coordination and commitment. It's worth mentioning that the 6% 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., 27.42%) due to the larger contention footprint.

For `WAIT_DIE`, Spanner-RLS achieved a significantly lower average latency for IRTs, while the average latency of CRTs was roughly the same as in Spanner. This is attributed to the fact that in Spanner-RLS, an IRT will never be blocked by CRTs. The results on $50th$ and $90th$ latency support this assertion. Both Spanner and Spanner-RLS exhibited low $50th$ latency, while the $90th$ latency of Spanner and Spanner-RLS was $805ms$ and $1.4ms$, respectively. The $99th$ latency of Spanner-RLS increased due to the queueing effect in the software stack.

Next, we delve into understanding how Spanner-RLS and Spanner are affected by various workload parameters. These experiments were conducted using YCSB's APIs as they offer flexibility in configuration.

3) Impact of Concurrency: We first compare the performance of Spanner-RLS 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 3911 and 1181 transactions per second using `No_WAIT` and `WAIT_DIE`, respectively. In contrast, Spanner-RLS could serve more clients and achieve a substantially higher peak throughput.

4) Impact of CRT Ratio: We studied the impact of the CRT ratio by fine-tuning the workload generation. As shown in Figure 7b and Figure 8b, when CRTs were enabled, Spanner experienced severe performance degradation (e.g., throughput dropping from 18526 transactions per second to 6184 transactions per second when the CRT ratio increased from 0% to 5%), aligning with our discussion in §I and §II-B. In contrast, Spanner-RLS'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-RLS demonstrated slightly lower throughput than Spanner due to the cost for extra steps in concurrency control (i.e., checking transaction types even if all transactions are IRTs). With a continuous increase in CRT ratios, the throughput of Spanner and Spanner-RLS decreased due to cross-region communication costs. In practice, the CRT ratio of workloads should not be too high since the cost of CRT itself is still relatively high compared to IRTs. Real-world workloads show good data locality under multi-region deployment (§II-A), facilitating low-latency data access.

5) Impact of Cross-Region RTT: Next, we studied the impact of the cross-region network delays, a critical factor affecting the overall cost of CRTs. Larger cross-region network delays generally result in longer transaction coordination and commit times for CRTs. The results, illustrated in Figure 7c and Figure 8c, clearly indicate that Spanner-RLS outperforms Spanner regardless of the cross-region network delays. In fact, Spanner-RLS demonstrates more when the network delays are moderate (e.g., $40s$ and $60s$ for a cross-region network round trip). In addition, the network delay amplifies the advantages of Spanner-RLS while also affecting Spanner-RLS's CRTs, causing a drop in throughput from 1429 transactions per second

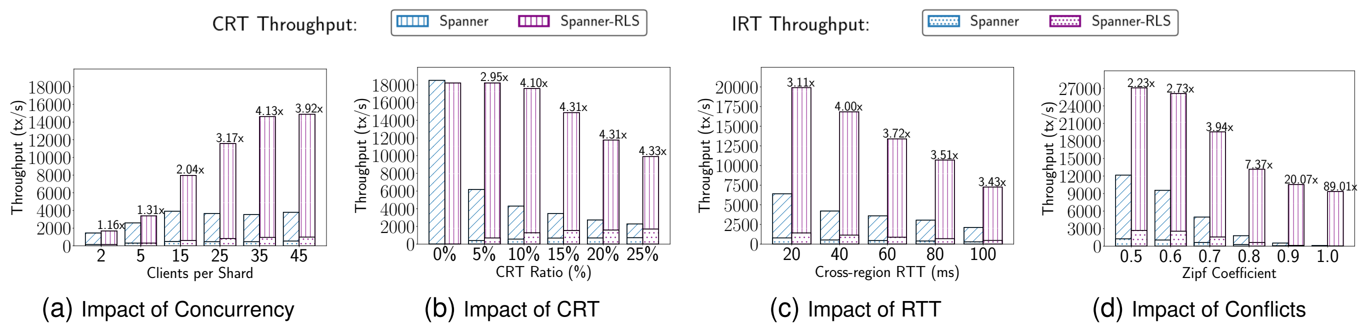


Fig. 7. Performance of Spanner and Spanner-RLS on YCSB-T with different experimental parameters using `NO_WAIT`.

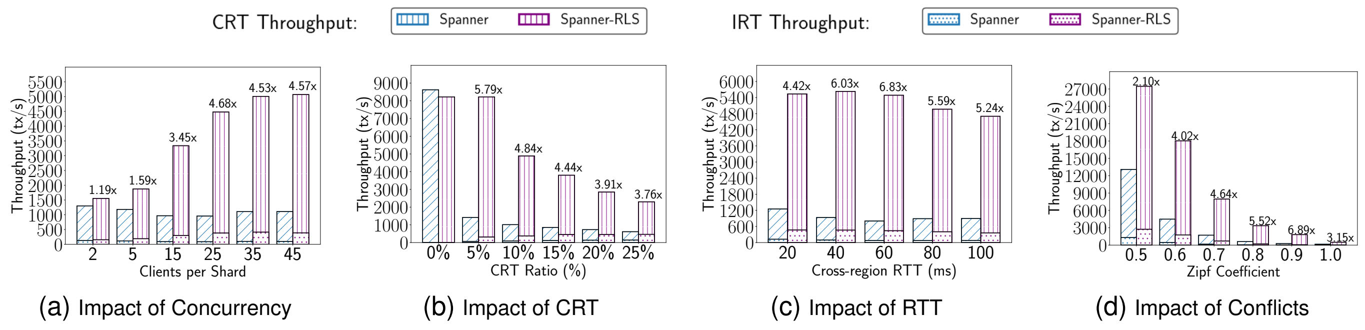


Fig. 8. Performance of Spanner and Spanner-RLS on YCSB-T with different experimental parameters using `WAIT_DIE`.

to 476 transactions per second as the network delay increased from 20 seconds to 100 seconds using `NO_WAIT`.

6) Impact of Contention: In our final experiment, we compared the performance under various contention by adjusting the skewness of Zipf distribution in YCSB-T while keeping other parameters consistent with the default settings. The results depicted in Figure 7d and Figure 8d consistently show that the Spanner-RLS's throughput outperforms Spanner's across all levels of contention. This improvement stems from Spanner-RLS efficiently reducing the contention footprint by ordering IRTs and CRTs independently. As discussed in §II-B, the contention footprint of IRTs eliminates both "coordination blocking" and "commit blocking", which is extremely expensive in multi-region deployments. As expected by our analysis, Spanner-RLS gained larger margins under high contention. This is because, 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-RLS also degraded due to the cost of acquiring locks, and `NO_WAIT` consistently outperformed `WAIT_DIE` since `WAIT_DIE` suffers more from lock thrashing and timestamp allocation when the contention is higher.

C. Takeaways

By adhering to the principles of RLS, developers can significantly enhance the performance of Spanner. Spanner-RLS serves as a practical example, illustrating how RLS can assist multi-region databases in achieving an optimal balance between consistency and performance. Further enhancements in the performance of Spanner-RLS can be achieved by implementing advanced optimizations (e.g., a pre-write-log mechanism in RedT [21]), which is orthogonal to our paper. We believe our study on Spanner-RLS holds the potential to guide future research by encouraging researchers and developers to consider multi-level two-phase locking and two-phase commit. Even though the design of Spanner-RLS may not be directly applicable to other strictly serializable concurrency protocols due to potential differences in transaction coordination mechanisms, the fundamental concept of RLS remains applicable.

V. CRDB AND CRDB-RLS

This section presents the design, implementation, and evaluation of Cockroachdb (for short, CRDB) and CRDB-RLS. We show that RLS has the potential to evolve the consistency model of existing databases (i.e., providing tighter and stronger consistency guarantees) without sacrificing performance.

A. Protocols and Implementations

CRDB Background. CRDB [6] 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.

Overall, CRDB provides single-key linearizability (i.e., no stale reads for each key) by supporting multi-version timestamp ordering (MVTO). The transaction manager nodes in CRDB are the special nodes for interacting with clients, assigning timestamps to transactions, and driving transaction coordination. CRDB assumes a maximum clock offset among transaction managers (i.e., using $500ms$ by default), which is critical to its correctness.

Algorithm 2: Algorithm of CRDB-RLS Coordinator

```
1  function CRDB-RLS 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)
```

CRDB's consistency model (i.e., single-key linearizability) is strictly weaker than RLS as CRDB ensures only a subset of RLS's guarantees: CRDB does not preserve the real-time ordering for any pair of non-conflicting transactions, while RLS provides real-time order among IRTs in the same region. We refer readers to §VI-A for detailed comparisons between RLS and Single-Key Linearizability (SKL).

CRDB makes such a design choice because, without the two-layered design of RLS, the developers could only choose the consistency model between extreme ends of the spectrum: either enforcing 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 severely impact performance, CRDB has chosen the latter approach. Consequentially, CRDB cannot even guarantee the causal relation of two transactions from the same client when they access different keys in the same region.

CRDB performs its reads and writes at its commit timestamp, relying heavily on multi-version concurrency control (MVCC) to process concurrent requests. When a transaction conflicts with other transactions, CRDB adjusts the transaction's commit timestamp to ensure single-key linearizability. Since conflict detection is only conducted in the critical granularity, CRDB does not provide any consistency guarantees when two transactions do not conflict with each other.

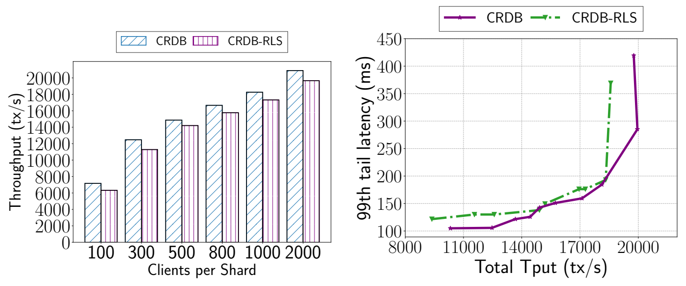
RLS provides a better design point in the spectrum. To demonstrate the pros and cons of RLS, we re-design the protocol of CRDB by incorporating multi-region semantics into the conflict detection progress, resulting in CRDB-RLS.

CRDB-RLS. Algorithm 2 illustrates the pseudocode of CRDB-RLS, with multi-region semantics highlighted in blue. To achieve RLS, for any two transactions T_1 and T_2 that access overlapped regions, if T_1 have finished, CRDB-RLS ensures that T_2's timestamp is larger than T_1's. CRDB achieves this 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., $500ms$ in the codebase of CRDB). Consequentially, if $T_1.ts - T_2.ts < bound$, CRDB cannot determine the order between T_1 and T_2. In such a case, CRDB enforces T_2 to abort and then lets it retry automatically.

Our observation of CRDB-RLS is that, to know whether a transaction T_1 may have finished before T_2 starts, instead of directly comparing the timestamps, a more intuitive method should be using active inquiry. In CRDB, transaction managers maintain the status of each transaction, and one can get the status of each transaction from the managers. Initially, CRDB does not adopt this active inquiry design because it can be inefficient and non-scalable to let all transactions contact a single node in a geo-distributed deployment. However, RLS's region-based approach enabled CRDB to record transaction states in a per-region manner (Algorithm 2, Line 13) and enabled each transaction to inquire only relevant regions' managers (Algorithm 2, Line 14), achieving both stronger consistency and high efficiency.

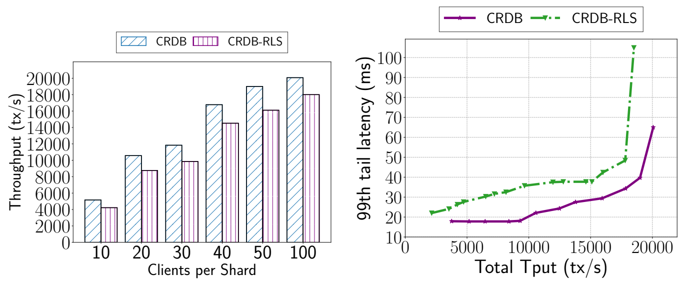
Like CRDB, CRDB-RLS allocates timestamps using hybrid-logical clocks (HLC), where physical time is based on a node's coarsely-synchronized system clock, and logical time is based on Lamport's clocks. Such a mechanism benefits the design of CRDB-RLS as valid transactions can now adjust their timestamps based on logical time without frequently re-obtaining new timestamps.

Implementations. We implemented CRDB-RLS using the open-source codebase of CRDB with version v23.1.10 from



(a) Throughput. (b) Overview.

Fig. 9. Performance Comparison on Micro Workload.



(a) Throughput. (b) Overview.

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

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 [6] (e.g., write pipelining, parallel commits, and follower read).

B. Evaluation and Discussion

We used the same hardware and cluster setups as Spanner. For the deployments, we employed 10 containers spanning uniformly over the machines, with each container running as a CRDB node. We used the multi-region SQL for Table partition and data replication.

1) Performance on Micro Workload: As the configuration file of CRDB is complex and always critical to the performance, we calibrated our results by running the most basic built-in workloads provided by the CRDB codebase (i.e., a transaction reads and writes to three keys spread uniformly across the cluster). The results are shown in Figure 9. The peak throughput of CRDB was $\sim 20k$ tps, which aligns with the results shown on the official sites. Therefore, we believe our experimental results are representative.

In addition to CRDB, we also test the performance of our implemented variation: CRDB-RLS. CRDB-RLS's peak throughput ($\sim 18.6k$ tps) is slightly lower (7%) than CRDB, while the latency is roughly the same. The slight performance degradation is caused by the higher cost of obtaining valid timestamps for IRTs.

2) Performance on YCSB-T: We further compared the performance of CRDB and CRDB-RLS on the default YCSB-T (§IV-B) using the calibrated configurations. The results are shown in Figure 10. Overall, CRDB-RLS achieved approximately $0.87\times$ to $0.91\times$ throughput compared to CRDB with various concurrencies. The peak throughput of CRDB-RLS was 12% lower than CRDB, and the 99th tail latency of CRDB-RLS was $1.2\times$ to $1.6\times$ higher than CRDB.

CRDB-RLS incurred a more server performance drop on the skewed YCSB-T workloads. Compared to CRDB, CRDB-RLS 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 $\sim 15\%$.

C. Takeaways.

CRDB-RLS achieved similar performance as CRDB while providing strong consistency guarantees on real-time orders. This is because CRDB-RLS can efficiently capture ongoing transactions inside a region due to the fast networks. Recording ongoing transactions (known as transaction tables) is a common approach for processing transactions on a single machine or in a smaller cluster due to its simplicity and easy extension. We reused such an approach but with a more lightweight tracking method for ordering IRTs.

VI. Related Works

Transaction processing represents a well-explored area of research, with a plethora of influential works. We will provide an overview of related works in this section.

A. Proximal Consistency Models

Figure 12 compares RLS to its proximal consistency models. We describe three of them in detail. All of them are serializable.

Regular Sequential Serializability (RSS) complements RLS, as they focus on different aspects of distributed databases. RSS is primarily 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. On the contrary, RLS 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 (refer to Figure 11a). 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 $T_2 \to T_3$; the serializable order enforced by RSS is $T_3 \to T_1 \to T_2$, which implies $T_3 \to T_2$. This execution is not allowed by RLS since RLS ensures strict serializability (i.e., real-time order) for IRTs within the same region.

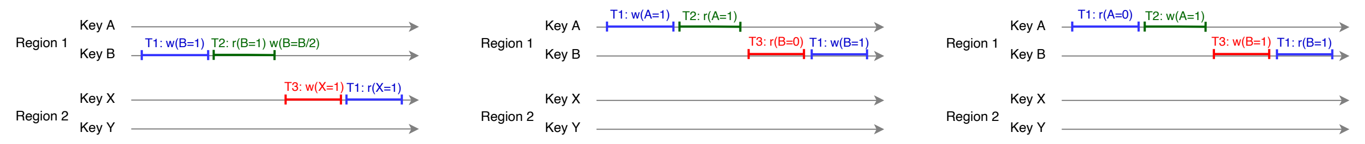
Single-key Linearizability (SKL) was initially proposed by CockroachDB [6]. SKL stands as a strictly weaker variant of RLS. Like RLS, SKL guarantees serializability and "no stale-reads". However, unlike RLS, SKL does not preserve real-time orders between non-conflicting transactions. For instance, the execution illustrated 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 in the same region, as the serial order is $T_3 \to T_1 \to T_2$, a violation not allowed by RLS. While weaker SKL may suffice for certain applications, other applications might necessitate stronger guarantees. Moreover, preventing consistency anomalies can significantly streamline application development.

Process-ordered Serializability (PoS) complements RLS. In a real deployment scenario, RLS can be stronger than PoS by associating each client with its nearby region. Specifically, PoS tracks the causal relations of each client and ensures the system preserves the ordering within each client's requests. If each client is associated with a region (e.g., sending requests to nodes within its region), RLS can prove to be strictly stronger than PoS since it guarantees real-time ordering for each client.

B. Transaction Priority

Compared to strict serializability (SS), one of the pivotal innovations of RLS is scheduling IRTs ahead of CRTs until the CRTs' order is established. Therefore, RLS 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 database developers can leverage existing transaction priority protocols to transition from SS to RLS, enhancing performance. For instance, Polaris [35] represents a transaction priority protocol rooted in a variant of OCC. Polaris embeds priority-related conflict detection within each record and permits priority preemption during runtime.



(a) Allowed by RLS but disallowed by SS, RSS. (b) Allowed by RSS but disallowed by RLS. (c) Allowed by CRDB but disallowed by RLS

Fig. 11. Comparison of RLS with proximal levels of consistency models.

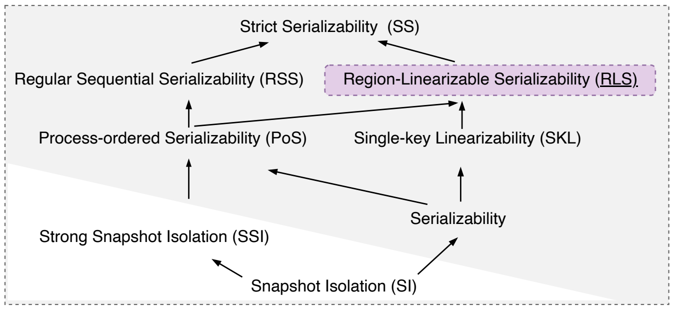

Strict Serializability (SS)
Regular Sequential Serializability (RSS)
Region-Linearizable Serializability (RLS)
Process-ordered Serializability (PoS)
Single-key Linearizability (SKL)
Serializability
Strong Snapshot Isolation (SSI)
Snapshot Isolation (SI)

Fig. 12. This diagram shows how RLS compared to its proximal consistency models: SS [15], RSS [24], SKL [6], PoS [31], [32], SSI [33], serializability [27], and SI [34]. We highlight all those serializable consistency models in grey.

Furthermore, Polaris avoids global operations, mitigating substantial overhead in a multi-region deployment. Consequently, achieving RLS should involve drawing inspiration from OCC-like concurrency control protocols.

C. *Mixed Consistency Models*

Several prior works [23], [36]–[42] have been on manipulating weakly and strongly consistent transactions within a single database. For instance, MixT [36] 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. Similarly, Red-Blue consistency [37] allows strongly and causally consistent operations to co-exist in a single system in the transaction's granularity and depends on application semantics to make "consistency choices". AutoGR [38] automatically analyzes and identifies the minimal set of the required consistency guarantees based on applications using the Z3 theorem prover. However, it relies on the application codes as inputs and only provides serializable guarantees without real-time order.

In contrast, RLS is geared towards multi-region deployments, directly integrating network semantics into the consistency model. Consequently, RLS does not depend on prior application knowledge to manually set distinct consistency guarantees for different transactions or operations. Moreover, RLS provides serializability (i.e., the strongest isolation levels) for all transactions while tailoring real-time properties to achieve heightened performance.

VII. Future Works

The multi-region transactions set three clear objectives: high throughput, low client-perceived latency (especially for IRTs), and as strong as possible consistency. Earning all three requires careful designs to strike a balance between the three individual objectives. Our exploration of RLS opens up a new design space for achieving such goals. We draw some lessons worth further research:

- **Multi-Layered Consistency Model.** Modern network exhibits a multi-layered structure [21], [43]. For instance, Cloud providers (e.g., Huawei [3]) are diligent in using CXL- and RDMA-based networks inside a data center, a dedicated network between data centers inside a region, and public networks across the regions. In this work, we have explored the consistency model for multi-region deployment by treating IRTs and CRTs differently. However, a more fine-grained design may still be desirable to tightly fit consistency guarantees into the network stack. For instance, in-network ordering technologies [20], [44]–[46], which leverage the properties of the network for order, are proposed for in-data center deployment. How to combine such technologies with geo-distributed transactions is still an open problem. A multi-layered consistency model may work as the glue to bridge the design of in-network transaction processing technologies with geo-distributed transaction processing technologies.
- **Data Locality and Partial Replication.** Even though RLS provides practical mitigation for achieving high throughput and low latency for both CRTs and IRTs, the cost of remote reads cannot be fundamentally removed. Therefore, an efficient data partition and replication policy are still critical for real-world usage: a good partition policy [47]–[51] can vastly reduce the ratio of remote reads, and a cautious replication policy [4], [6], [52] can balance the overhead of data synchronization and remote access.

VIII. Conclusion

This work uncovered fundamental bottlenecks in strictly serializable concurrency control algorithms used in multi-region deployments. As the consistency model inherently enforces these bottlenecks, many proposals turn to lift the restrictions by adopting weaker consistency. However, we found that all existing consistency models are inadequately designed for multi-region deployments: they are either overly stringent or have room for improvement without incurring significant performance penalties.

In response, we propose Region-Linearizable Serializability (RLS), the first consistency model meticulously tailored for multi-region deployment. Following the RLS methodology, we design, implement, and evaluate two practical system variations based on open-sourced codebases: Spanner-RLS and CRDB-RLS. The code of our stereotypes is available at https://github.com/vldb24p771/spanner_rls and https://github.com/vldb24p771/crdb_rls, respectively.

Our evaluation results demonstrate that RLS can significantly enhance the performance of Spanner (i.e., from $1.16\times$ to $89.01\times$ higher throughput) and further strengthen the consistency guarantees of CRDB without significant performance drop (i.e., $< 15\%$).

References

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

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

[3] H. Cloud, "HUAWEI CLOUD Regions and Service Endpoints," https://developer.huaweicloud.com/intl/en-us/endpoint.

[4] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 210–227.

[5] H. Fan and W. Golab, "Ocean vista: gossip-based visibility control for speedy geo-distributed transactions," *Proceedings of the VLDB Endowment*, vol. 12, pp. 1471–1484, 2019.

[6] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

[7] N. VanBenschoten, A. Ajmani, M. Gartner, A. Matei, A. Shah, I. Sharif, A. Shraer, A. Storm, R. Taft, O. Tan *et al.*, "Enabling the next generation of multi-region applications with cockroachdb," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2312–2325.

[8] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–37, 2018.

[9] "General data protection regulation (gdpr) – official legal text," https://gdpr-info.eu/, (Accessed on 10/01/2021).

[10] C. D. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023.

[11] K. Ren, D. Li, and D. J. Abadi, "Slog: serializable, low-latency, geo-replicated transactions," *Proceedings of the VLDB Endowment*, vol. 12, pp. 1747–1761, 2019.

[12] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–12. [Online]. Available: https://doi.org/10.1145/2213836.2213838

[13] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 517–532.

[14] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 210–227.

[15] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[16] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 715–726.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, Oct. 2012.

[18] H. Lu, S. Mu, S. Sen, and W. Lloyd, "Ncc: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall," *arXiv preprint arXiv:2305.14270*, 2023.

[19] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. 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.

[20] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[21] Q. Zhang, J. Li, H. Zhao, Q. Xu, W. Lu, J. Xiao, F. Han, C. Yang, and X. Du, "Efficient distributed transaction processing in heterogeneous networks," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1372–1385, 2023.

[22] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building Consistent Transactions with Inconsistent Replication," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 1–37, Dec. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3297862.3269981

[23] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 113–126.

[24] J. Helt, M. Burke, A. Levy, and W. Lloyd, "Regular sequential serializability and regular sequential consistency," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 163–179.

[25] T. T. P. COUNCIL, "TPC-C," http://www.tpc.org/tpcc/, 2014.

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

[27] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–34, 2016.

[28] "Github: UWSysLab/tapir," https://github.com/UWSysLab/tapir.

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

[30] M. Azure., "Azure network round-trip latency statistics," https://docs.microsoft.com/en-us/azure/networking/azure-network-latency.

[31] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," in *Proceedings. 20th International Conference on Data Engineering*. IEEE, 2004, pp. 424–435.

[32] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The {SNOW} theorem and {Latency-Optimal}{Read-Only} transactions," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 135–150.

[33] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 492–528, 2005.

[34] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, consistency, and practicality: are these mutually exclusive?" in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 484–495.

[35] C. Ye, W.-C. Hwang, K. Chen, and X. Yu, "Polaris: Enabling transaction priority in optimistic concurrency control," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–24, 2023.

[36] M. Milano and A. C. Myers, "Mixt: A language for mixing consistency in geodistributed transactions," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 226–241, 2018.

[37] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making {Geo-Replicated} systems fast as possible, consistent when necessary," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 265–278.

[38] J. Wang, C. Li, K. Ma, J. Huo, F. Yan, X. Feng, and Y. Xu, "Autogr: automated geo-replication with fast system performance and preserved application semantics," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1517–1530, 2021.

[39] Y. Yang, Y. You, and B. 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)*, vol. 1. IEEE, 2017, pp. 183–190.

[40] C. Li, N. Preguiça, and R. Rodrigues, "Fine-grained consistency for geo-replicated systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 359–372.

[41] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, "Application specific data replication for edge services," in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 449–460.

[42] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 253–264, 2009.

[43] K. C. Webb, A. C. Snoeren, and K. Yocum, "Topology switching for data center networks," in *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)*, 2011.

[44] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au, and H. Cui, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *The 28th ACM Symposium on Operating Systems Principles*, 2021.

[45] B. Li, G. Zuo, W. Bai, and L. Zhang, “1pipe: Scalable total order communication in data center networks,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 78–92.

[46] I. Choi, E. Michael, Y. Li, D. R. Ports, and J. Li, “Hydra:{Serialization-Free} network ordering for strongly consistent distributed applications,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 293–320.

[47] M. Abebe, B. Glasbergen, and K. Daudjee, “Dynamast: Adaptive dynamic mastering for replicated systems,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1381–1392.

[48] ——, “Morphosys: automatic physical design metamorphosis for distributed database systems,” *Proceedings of the VLDB Endowment*, vol. 13, no. 13, pp. 3573–3587, 2020.

[49] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden, “Schism: a workload-driven approach to database replication and partitioning,” 2010.

[50] A. Pavlo, C. Curino, and S. Zdonik, “Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 61–72.

[51] E. Zamanian, C. Binnig, and A. Salama, “Locality-aware partitioning in parallel database systems,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 17–30.

[52] N. Schiper, P. Sutra, and F. Pedone, “P-store: Genuine partial replication in wide area networks,” in *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 214–224.