Efficient Geo-Distributed Transaction Processing

by

Joshua Hildred

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

Distributed deterministic database systems support OLTP workloads over geo-replicated data. Providing these transactions with ACID guarantees requires a delay of multiple wide-area network (WAN) round trips of messaging to totally order transactions globally. This thesis presents Sloth, a geo-replicated database system that can serializably commit transactions after a delay of only a single WAN round trip of messaging. Sloth reduces the cost of determining the total global order for all transactions by leveraging deterministic merging of partial sequences of transactions per geographic region. Using popular workload benchmarks over geo-replicated Azure, this thesis shows that Sloth outperforms state-of-the-art comparison systems to deliver low-latency transaction execution.

Acknowledgements

I would first like to thank my supervisor, Professor Khuzaima Daudjee. Without Khuzaima, none of this would have been possible. Khuzaima's guidance along with his kind and caring nature allowed me to grow as a person and as a computer scientist.

I would also like to thank Michael Abebe. Michael's mentorship was instrumental in helping me become a fully functioning grad student.

Both Khuzaima and Michael challenged me and my research. Without them, my thesis would not be the same.

I would also like to thank my thesis readers, Samer Al-Kiswany and Sujaya Maiyya, whose feedback helped me improve my thesis.

Finally, I would like to thank my family: my mother, Alison, my father, Rob, and my brother, Eli. Their love and support has meant everything to me. Last but not least, I would like to gratefully acknowledge my partner Alicia. Alicia was with me through the highs and lows of being a Master's student; she celebrated my successes, comforted me in my failures, and lived through all my stress. For this, I am eternally grateful.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Geographically-replicated database systems are used in industry as the backbone to provide both good performance and fault-tolerance for a range of global client services from advertising platforms [17] and banking [5] to global travel operations [1]. Replication of data across geo-distributed data centres provides two benefits when compared with data that is replicated within only a single data centre. First, geo-replication allows copies of data to be placed geographically closer to clients. The locality of clients and data supports low-latency access to deliver improved performance [19]. Secondly, geo-replication allows database systems to be tolerant to data centre unavailability through zone/region-aware replication. In comparison, replication within a single data centre can protect against only machine-level failures. The level of fault tolerance provided by geo-replication is key for being able to handle large-scale failures such as those caused by natural disasters or core network infrastructure failures [6].

1.1 Geo-Replicated Data

Distributing work across data replicas requires coordination and communication that can pose significant challenges for geo-replicated database systems compared to on-premise (non-geo-replicated) systems. Performant ACID transactions over geo-replicated data have become desirable for users of these systems [29, 41]. To provide strong consistency and global atomicity, transactions need to be coordinated across geo-distributed regions or sites, resulting in multiple rounds of communication over a wide-area network (WAN) that incur significantly higher latency than over a local-area network (LAN). Although the latency of a LAN round-trip time (RTT) is usually in the order of milliseconds, the latency of a WAN RTT can be 200× higher. Unless a geo-distributed system can mitigate WAN latencies, these high latencies will translate into high transaction response times and low throughput, leading to poor overall distributed system performance [41].

Early work on geo-replicated database systems that support strongly consistent transactions uses a combination of Paxos [25] and Two-Phase Commit (2PC) to provide ACID transactional guarantees. For example, Spanner [17] uses Paxos and 2PC, which can take up to 4.5 WAN round trips to coordinate distributed transaction commit [29].

The number of WAN round trips for a transaction to commit can be reduced to two by running 2PC within each region while using Paxos to agree on the outcome of 2PC [29]. A geo-replicated database system can use a primary site architecture, in which a specific single region (primary) is responsible for enforcing ordered access to data items. This type of architecture can reduce transactional latency within the primary's region as communication will happen over a LAN. However, transactions outside the primary region will still incur high latency for communication over a WAN (to reach the primary region) [41, 9, 37]. Thus, the challenge of ameliorating the high cost of transaction latencies in geo-distributed database systems remains.

Systems that rely on Paxos or other consensus approaches for coordination and fault tolerance generally do not perform well when WAN latencies are large. For example, using Paxos requires communication with at least a majority of replicated sites, and even the minimum latency within a majority quorum can result in large latency overheads in the system. Consider a scenario with 5 replicas in Azure Cloud, one in each of East US, East US 2, France Central, West EU and and East Asia (Table 1.1). Based on these (Table 1.1) latencies, the best case RTT for communication between a majority quorum in Paxos would be at least 82 ms (East US, East US 2, France Central) with the worst case (East Asia, France Central, West EU) being 191 ms, resulting in high latency for geo-replicated transactions.

WAN RTT						
Originating region	East US	East US 2	South East Asia	East Asia	France Central	West EU
East US		6 ms	228 ms	202 ms	82 ms	82 ms
East US 2	6ms		228 ms	200 ms	83 ms	86 ms
South East Asia	228 ms	228 ms		34 ms	182 ms	159 ms
East Asia	202 ms	200 ms	34 ms		168 ms	191 ms
France Central	82 ms	83 ms	182 ms	168 ms		12 ms
West EU	82 ms	86 ms	159 ms	191 ms	12 ms	

Table 1.1: Azure Inter-region latencies for data centers within three continents [2].

1.2 Deterministic Transaction Processing

Subsequent work on deterministic database systems has improved upon the performance of geo-replicated database systems that use 2PC or other distributed commit protocols [41]. Deterministic database systems [41, 37] use determinism of transaction outcomes to achieve correctness. Deterministic execution relies on the creation of a global order of transactions before transaction execution can begin. The deterministic database system executes transactions in conformance to this global order. This determinism allows a geo-replicated database system to avoid communicating between regions after transactions have been globally ordered for execution.

An observation of distributed deterministic database systems [41, 37, 36] is that much of the transaction latency is incurred from WAN communication required to create a global order. For example, Paxos is used by Calvin [41] to facilitate the creation of the global order resulting in two round trips over the replicas to reach agreement to add a transaction to the global order. Calvin cannot take advantage of locality in a workload due to all transactions needing to be added to the Paxos-coordinated global order. SLOG [37] characterizes transactions into single-region and multi-region. However, similarly to Calvin, SLOG must globally order all multi-region transactions over a WAN, thereby also incurring large transaction latencies.

1.3 Contributions

This thesis presents Sloth, a geo-replicated transactional database system that significantly reduces transaction latencies by determining regional transaction orders that are consistent with global transaction execution schedules. To provide low latency ACID transactions over geo-replicated data, Sloth exhibits three properties:

- Sloth provides low latency ordering for coordination of deterministic transaction execution.
- Sloth exploits data locality in primary location for low latency transaction execution.
- Sloth provides region-level fault tolerance and durability without prohibitive latency overheads.

Sloth uses a transaction ordering protocol that requires at most a single WAN round trip before any transaction can be committed. Sloth's protocol deterministically merges partial transaction orders into a globally consistent order that preserves correctness for serializable transaction isolation and replica consistency.

Sloth exploits data locality. The deterministic merging of partial transaction orders allows any replica to begin transaction execution as soon as all ordering information for the transaction is received at the replica. The ordering information enables transactions with locality to begin executing with little or no delay as a transaction waits for only the dependent ordering information (and not all ordering information).

Sloth uses a replication protocol that individually replicates the partial transaction orders to nearby regions with low latency. The replication overlaps the deterministic merging of the partial transaction orders and transaction execution. Sloth uses a recovery protocol that guarantees no transactions will be lost and global serializability will not be violated. Importantly, the replication protocol does not change the one WAN round trip it takes to commit a transaction.

Finally, this thesis presents an evaluation of Sloth's performance using OLTP benchmark workloads TPC-C [7] and MovR [42, 4]. As shown in Chapter 5, Sloth outperforms state-of-art systems SLOG [37] by 6× and Calvin [41] by up to 38× on transaction latency.

This thesis makes 3 contributions:

- An ordering protocol that requires a single WAN round trip to globally order all transactions and allows for locality to be exploited is provided (Chapter 4). Furthermore, the correctness of the protocol is shown (Chapter 4.2.6).
- A fault tolerance and recovery scheme for region level failures that does not increase the one WAN round trip transaction commit, including their correctness, is provided (Chatper 4.2.7).
- An empirical comparison of OLTP workload performance against the state-of-the-art in geo-distributed deterministic transaction execution is provided (Chapter 5).

In addition to the above, Chapter 2 discusses related work, Chapter 3 provides background relevant to the topic of the thesis, and Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Related work

This chapter discusses the work related to this thesis. The chapter is organized based on four key characteristics of the work in this thesis.

2.1 Totally or Partially Ordered Sequence of Transactions

Many distributed data systems rely on an explicit ordering of transactions for correctness. The transaction orders can be a total order of transactions or a partial order.

Calvin [41] deterministically executes transactions at replicas according to a predetermined transaction ordering. As transactions are submitted, Calvin totally orders the transactions against each other in a distributed log. This ordering of transactions is used to guarantee a globally serializable execution schedule at each replica. SLOG [37] builds upon Calvin to reduce transaction latency. SLOG relaxes the need to order all transactions totally but still requires a consistent order at each replica for global serializable execution. SLOG has a per replica sequencer that sequences transaction that access data for which the primary copy is located at the replica. SLOG merges these orderings into a consistent global transaction order to allow globally serializable execution schedules. SLOG totally orders all transactions with primary data location at two or more replicas (multi-replica transactions); these transactions are then broken into transaction pieces. The transaction's pieces are sequenced in the local logs along with the transactions that only access data at a single primary (single-replica transactions). SLOG waits for all a transaction's pieces to appear in the merged order before the transaction can be committed.

Aria [28] and QStore [36] require that all replicas execute batches of transactions in the same order to guarantee correctness. In both systems, batches of transactions are proposed by replicas in a round-robin fashion.

Hyder [14] is a database built over a shared log with a deterministic meld [15] operation that runs independently at each site to decide if a transaction commits or aborts. The meld operation uses the total order of transactions in the shared log to decide if a transaction can commit based on the state of the database after running all the preceding transactions in the total order, and the set of data items that the transaction updates.

FuzzyLog [27] shows how to support serializable transactions over its partially ordered log. FuzzyLog maintains its partially ordered log as a DAG with vertices partitioned by colors (all vertices within a color are totally ordered). FuzzyLog uses the Tango [12] protocol to support serializable transactions; Each partition of data is assigned a color. Transaction updates and commit decisions are represented as a vertex with the colour of the corresponding partition. A transaction is committed at a replica once the replica has seen the all the commit decisions from each partition that the transaction modifies. ConfluxDB [16] merges log streams from primary sites into a unified log stream while maintaining SI. However, ConfluxDB must also wait for entries in log streams before a commit decision can be made to guarantee correctness.

2.2 WAN Round Trips

One way to decrease the latency of geo-replicated transactions is to reduce the number of WAN round trips that must occur before a transaction can commit.

A number of consensus systems can be used to sequence transactions in a single round trip of WAN communication. Fast Paxos [26] introduces a fast round, in addition to the classic round, that can sequence operations in a single round trip to $\lceil \frac{3}{2} f \rceil + 1$ replicas. However, if operations proposed in the fast rounds are received by replicas in different orders, the fast round fails.

EPaxos [31] is a leaderless protocol that has a fast path quorum which can sequence operations in a single WAN round trip to $f + \lfloor \frac{f+1}{2} \rfloor - 1$ replicas if all replicas agree on an operation's dependencies. A second round of WAN messaging is required if the dependencies returned in the fast path differ.

CURP [35] can make operations durable in a single round of WAN delay by replicating unordered operations to witnesses. If all witnesses accept the operation (witnesses reject operations if they do not commute), the operation takes one WAN RTT of delay; Otherwise, CURP needs 2 WAN RTTs to order and replicate the operation.

Mencius [30] alternates replicas, using a predetermined order, as the leader for the round. Each replica has the option to propose operations when it is the leader, with replicas being able to skip their turn. Mencius can commit operations in a single round trip of WAN delay by allowing non-conflicting operations to commit out of order. However, in the presence of conflicting operations, the system must wait on the conflicting operations from previous rounds before the most recent operation can be committed.

Several geo-replicated database systems can commit transactions with a single round of WAN communication. However, a second round is required if certain conditions do not hold. Janus [33] reduces the number of WAN round trips to commit transactions to one if all replicas return identical dependencies for the transaction. Otherwise, Janus requires a second round of WAN communication with the majority of replicas to agree on the transaction's dependencies. Starry [45] can commit transactions in a single round of WAN communication to a supermajority of $\lceil \frac{3}{2}f \rceil + 1$ replicas for transactions that do not conflict with concurrent transactions. On conflict, a second round of WAN communication to a central sequencer is required to resolve the conflict. Carousel [43], MDCC [24], and TAPIR [44] also commit transactions in a single WAN round trip to a supermajority, but if there are concurrent conflicting transactions, a second round of messaging is required.

SLOG [37] commits single-replica transactions in zero WAN RTTs, or one WAN RTT if replication to other regions is performed. SLOG achieves zero WAN RTT commit for single-region transactions as they only need to be sequenced in the local log of the corresponding replica before the transaction can be committed. To handle region-level failures, SLOG must replicate the local logs to other regions before transactions can commit. This replication results in one WAN RTT commit for single-replica transactions. For multi-replica transactions, SLOG requires 1.5 WAN round trips if the total order of multi-region transactions is not replicated and 2.5 if it is (using Paxos).

2.3 Strongly Connected Components

The strongly connected components of a conflict graph provide a convenient mechanism for database systems to handle circular dependencies.

Janus [33] tracks transaction dependencies in an identical conflict graph at each replica by ensuring all replicas agree on transaction dependency information before execution via consensus. After all replicas agree on a transaction's dependency information (by consensus), each replica independently computes the strongly connected component for the transaction. The replica executes the transactions in the strongly connected component in a deterministic order. The deterministic order of transactions within a strongly connected component allows all replicas to execute transactions with circular dependencies in a consistent order.

Starry [45] uses a central coordinator which globally tracks transaction conflicts in a conflict graph. When a transaction conflict is discovered, the conflict graph is updated to reflect the conflict through messages from replicas to the central coordinator. When a decision must be made to resolve a cyclic conflict, the central coordinator computes the strongly connected component and reorders transactions by either aborting or re-committing transactions.

EPaxos [31] uses strongly connected components to resolve conflicts between dependent operations during execution. To execute an operation, EPaxos builds the dependency graph for all of the operation's dependent operations and computes the strongly connected components of this dependency graph. EPaxos then executes all commands according to a topological ordering of the strongly connected components, and then according to the deterministic order of transactions within the strongly connected components.

Rococo [32] uses graph-based dependency tracking with reordering based on strongly connected components but does not consider geo-replicated data. Rococo requires that each transaction has a coordinator that performs two rounds of communication with participating servers before the transaction can be committed. The transaction coordinator is responsible for collecting and distributing a transaction's dependency information. Rococo uses offline transaction decomposition, based on transaction chopping [38], to ensure that transactions' pieces are reorderable. To commit a transaction, the transaction coordinator sends a commit request to servers. During transaction commit, the servers independently find the strongly connected component to which the committing transaction belongs and execute the transaction according to the deterministic order within the strongly connected component.

2.4 Reduced Replication Quorum Size

As mentioned previously, replicating data to a quorum of geographically distributed replicas can incur significant latency. Flexible Paxos [21] proposes reducing the size of the replication quorum while increasing the size of the election quorum. Flexible Paxos takes advantage of the fact that the quorum for leader election and replication only needs to intersect. Thus, a reduction in the size of the replication quorum must come with an increase in the size of the election quorum to ensure that the quorums intersect. Reducing the replication quorums decreases replication latency if leader elections are rare, for example, if a long-lived leader is used.

SLOG [37] provides fault tolerance against region-level failures by replicating its local logs to a single geographically close region with low WAN latency. SLOG does not provide details on how this is performed and how correctness is preserved in the case of failures.

WPaxos [10] looks to provide low-latency ordering over geo-replicas by reducing the size of the replication quorum to include only geographically close replicas with low WAN latency. WPaxos assigns each object to one of the replicas; a replica can only act as a leader and propose operations for the objects it is assigned. WPaxos replicates operations to a reduced replication quorum. WPaxos performs an object-stealing phase if all objects are not local to a replica proposing an operation that accesses these objects. The object-stealing phase must contact a majority of replicas in the leader election quorum.

Chapter 3

Background

A deterministic database system enforces transaction determinism and avoids randomness during execution [41, 8]. Deterministic execution means that no communication is required among replicas during transaction execution and commit to guarantee that the database system stays in a consistent state. Deterministic transaction execution generally involves the creation of a predetermined *global order* for transaction execution at each replica using transactions' read and write sets [41]. A deterministic database system uses this ordering to execute transactions in a *globally serializable order* (Definition 3.0.1) at each replica. Once a replica obtains a transaction's position in the global order, the replica can execute and commit the transaction in that order without further WAN communication. Distributed deterministic database systems therefore avoid expensive distributed commit protocols by eliminating nondeterminism within the system [41].

Definition 3.0.1. *Global serializability means that concurrent transactions executing on multiple copies of a data item have to appear as if they have executed on a single copy of the data item in some serial order [13].*

Distributed deterministic database systems typically have three core components; *sequencer, deterministic scheduler*, and *storage layer* (Figure 3.1).

The *sequencer* is responsible for generating the global transaction order, meaning the sequencer is where all coordination among replicas occurs. When a deterministic database system is geo-replicated, much of the transaction latency comes from the sequencer component needing to communicate across a WAN, to create a global transaction order, before execution can begin[1]

[1]Transactions that do not wait on WAN communication can execute in as little as 5 ms whereas a single round trip of WAN communication can be more than 40× higher.

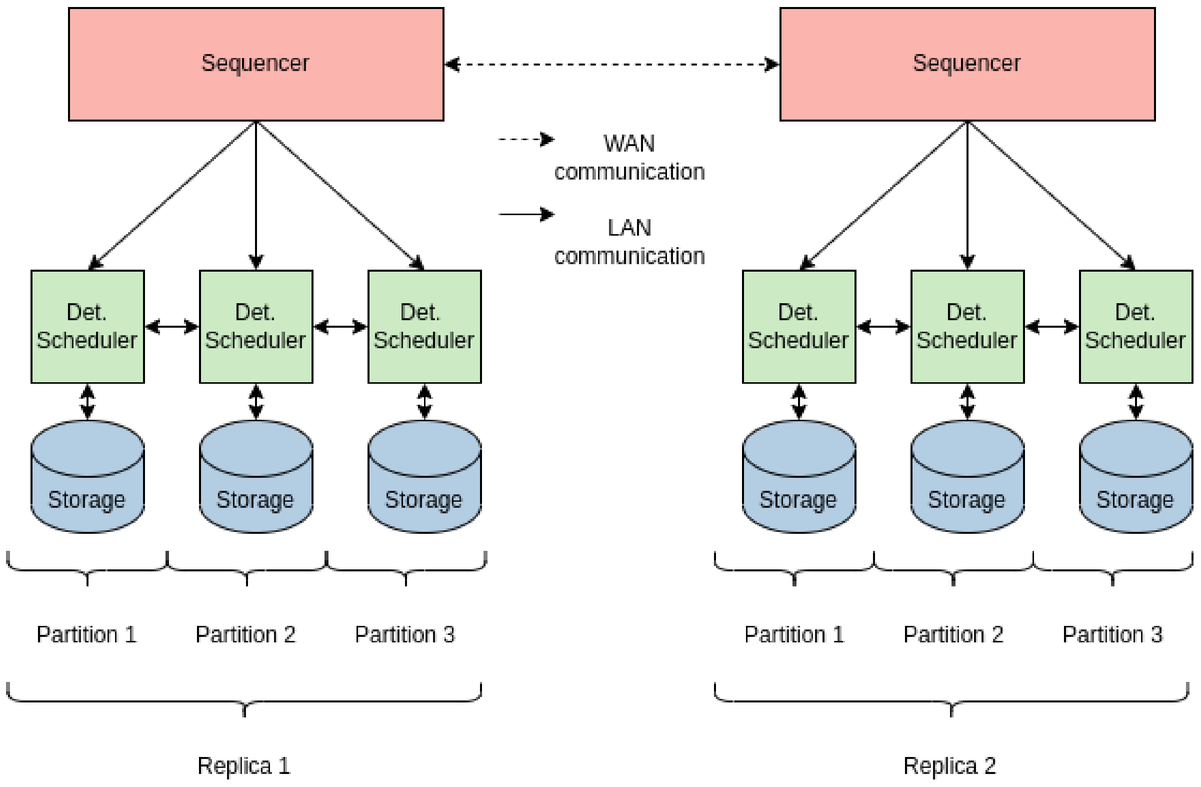


Figure 3.1: General architecture of a geo-replicated deterministic database[41]

[41, 36, 37]. Typically, the global order is created through agreement by a consensus protocol such as Paxos [41, 33]. The consensus protocol provides fault tolerance for the sequencer component and a mechanism for all replicas to agree on a global ordering of transactions. Consensus-based approaches come at the cost of multiple round trips to a majority quorum of replicas, which typically add large latency overheads.

Alternatively, the sequencer can use primary-based ordering; the simplest implementation is a system with a single machine creating a global order [41]. However, primary-based ordering schemes are more susceptible to failure than consensus-based approaches. Consensus protocols replicate both the transaction execution schedule and the transaction logic; if a replica fails, this information can be safely recovered by reading from a majority of surviving replicas. If the primary fails in primary-based ordering, the order may not persist past failures, or the time to recover may be large, leaving the system in an inconsistent state or unable to continue to operate. Thus, although primary-based ordering can save a round trip of WAN latency compared to using a consensus protocol, it is not fault tolerant.

The *deterministic scheduler* ensures that each replica deterministically schedules and executes each transaction across the replica's data partitions. Each deterministic scheduler knows that all counterpart schedulers at other replicas will execute transactions according to the chosen global transaction order. Thus, once the deterministic scheduler at a replica knows a transaction's position in the global (serializable execution) order, the transaction can be scheduled for execution and committed independently of other schedulers while ensuring a valid global serialization order [41].

The Calvin system [41] uses Paxos as the core of its sequencer to totally order all transactions. The scheduler is a per-replica distributed lock manager. The lock manager guarantees deterministic execution if locks are acquired by transactions in conformance to the global transaction order. The lock manager guarantees serializability through a deterministic locking scheme. The storage layer is an in-memory key-value store that can create, read, update, and delete data items.

Sloth uses a different design for the sequencer that allows the merging of partial transaction orders into a globally consistent order that preserves correctness for serializable transaction isolation and replica consistency. The deterministic merging of partial transaction orders allows transaction execution to begin as soon as all ordering information for the transaction is received at a replica.

Chapter 4

System Design and Architecture

This chapter presents an overview of Sloth and the system details, along with correctness.

4.1 Sloth Overview

An overview of Sloth's system model and transaction ordering is presented in this chapter. The chapter also presents some basic terminology used in the rest of the thesis, followed by an example that demonstrates how Sloth delivers transaction execution latency savings over its competitors. In Chapter 4.2 the Sloth system design is presented in detail.

4.1.1 System Model

There are N_D (unique) data items in the system. The data items are partitioned into N_P partitions. Each partition of data is fully replicated at all of the N_R regions. One replica is designated as a data item's *primary* (copy). The remaining copies of the data item are referred to as its *secondaries* or *replicas*. The set of data items for which replica R is the primary is termed R's *primary set*. When presenting Sloth's sequencer protocol, this thesis considers a single replica and a single sequencer component per region, which is not a requirement but simplifies the presentation. Therefore, if the replica at a region R contains the primary copy of a data item D, R is referred to as D's primary region or primary. Thus, the set of data items that region R is the primary for is R's primary set or $ps(R)$.

A Sloth sequencer has two key responsibilities per region:

1. A region R is responsible for ordering all transactions that access data items whose primary copy is at R. This order is referred to as R's *partial sequence*, and R *sequences* T, or R is a *sequencer* for T. A transaction T will appear in the partial sequence of all regions that are the primary for a data item in T's read or write set.

2. A region R performs *deterministic merging* of all *partial sequences* into a transaction order that is consistent with the *global order* for transaction execution. The partial sequences are merged independently at each region, with no further communication after the partial sequence is received. This merging means that each ordering of transactions at a given region may potentially be different from orders at other regions. However, importantly, if any two transactions conflict, then their relative (conflict equivalent) order is preserved in all orderings of transactions at each region (Definition 4.2.1).

4.1.2 Transaction Ordering

The per region deterministic merging of partial sequences into a transactionally consistent, and complete, order is challenging to achieve. Deterministic merging can result in transactions being added to the transaction order and executed in different orders across regions. In particular, as long as a region has all of a transaction T's dependency information, i.e., T's position in all partial sequences, T can be added to the globally consistent transaction order at the region. Deterministic merging allows a region to execute transactions while bypassing some or all of the delay incurred by coordination between regions. For example, a region R_1 can add T to its transaction order and execute T before another region R_2 knows T exists. The details of the sequencer are presented and discussed in Chapter 4.2.1 and the details of deterministic merging are described in Chapter 4.2.2.

A fundamental feature of Sloth's protocol is that it incurs *at most* a single round trip of communication to include a transaction in the global execution order, meaning the transaction can execute and commit with a single RTT of delay. As mentioned above, a region can add a transaction T to the compete sequence once it has T's position in all partial sequences. The transaction requires half a round trip to send it to each region, and half a round trip to propagate partial sequences to all regions.

Furthermore, the latency can be reduced significantly if locality exists among the regions which contain the primary copy of data items in the transaction's read and write set. In particular, for a transaction T, region R must wait for communication from only regions that contain the primary copy of data items in T's read and write set before execution can begin. Thus, no WAN communication will be performed before the transaction can be executed at R if R contains the primary copy for all data in T's read and write set. If R holds the primary copy of only part of

T's read and write set, with another region R_2 containing the primary copies for the rest, R must wait on communication with only R_2, which significantly reduces transaction latency if these regions are close to each other.

4.1.3 Example

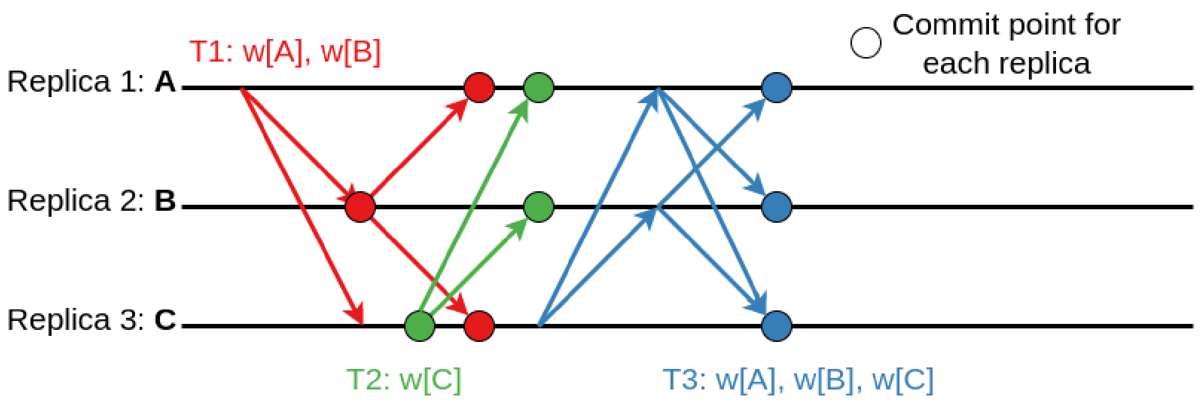
Figure 4.1a exemplifies the performance advantages of the above-mentioned properties. This example uses three regions and three data items, with each region containing the primary copy for a single data item. Region R_1 contains the primary copy for A, region R_2 contains the primary for B, and region R_3 contains the primary for C. For simplicity, the example assumes that the round trip latency between any two regions from among R_1, R_2 and R_3 is the same.

Transaction T_1, that updates data items A and B, can be committed at R_1 after only a single round trip of communication with R_2. Once T_1 has been added to R_1 and R_2's partial sequences, and the partial sequences have been propagated to R_1, T_1 can be added to R_1's order of transactions. Once the transaction is part of the transaction order at R_1, it can be executed and committed without communication with other regions. T_1 runs and commits independently at R_2 and R_3 once it has been added to each region's respective sequence (again without any communication).

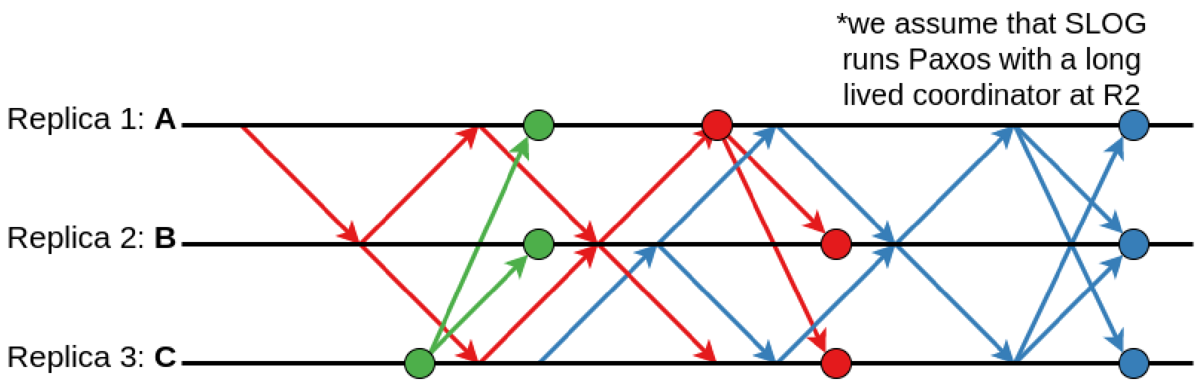
A transaction T_2 that originates at R_3 and updates only data item C can run and commit at R_3 with no WAN round trips. Because T_2 needs to be sequenced by only R_3's partial sequencer, T_2 can be added to the transaction order at R_3, executed, and committed with no WAN trips. Similar to T_1, T_2 will be executed at R_1 and R_2 after the partial sequence from R_3 has been received, and T_2 has been added to the transaction order at the respective regions.

As transaction T_3 originates at R_3 and updates all three data items, each partial sequencer must sequence it. T_3 must wait on concurrent round trips to R_1 and R_2 before being committed at R_3. While such a transaction incurs the worst-case latency as all regions must be contacted, T_3 will still benefit by executing in a single round trip of WAN communication delay, possibly with just a larger latency.

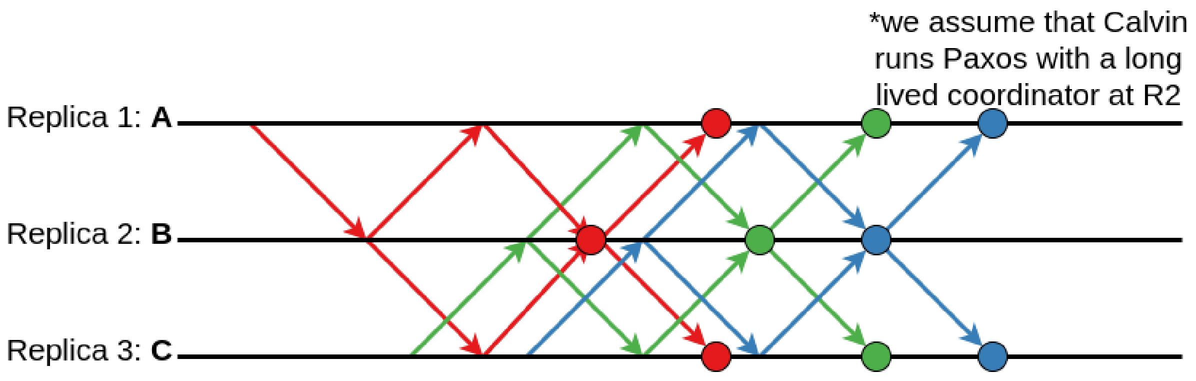
Figure 4.1b shows SLOG's sequencer operation. Transaction T_2 is single-region and can execute without a WAN round trip because T_2 will appear in a single local log and thus does not need to be totally ordered. T_2 does not wait on transaction pieces from other local logs. T_1 and T_3 are multi-region and must be totally ordered against all multi-region (**MR**) transactions before the transaction pieces can be added to the local logs. Two round trips are needed if Paxos is used for global ordering. Furthermore, waiting for transaction pieces to be propagated in local logs requires an extra one-half round trip. Thus, in SLOG, multi-region transactions require two-and-a-half WAN round trips before they can be committed, compared to only one round trip in Sloth.



(a) Sloth


*we assume that SLOG runs Paxos with a long lived coordinator at R2
Replica 1: A
Replica 2: B
Replica 3: C

(b) SLOG


*we assume that Calvin runs Paxos with a long lived coordinator at R2
Replica 1: A
Replica 2: B
Replica 3: C

(c) Calvin

Figure 4.1: Example of Calvin, SLOG and Sloth sequencer message passing

Figure 4.1c shows how Calvin must order all three transactions using Paxos. In general, Paxos incurs two round trips, which means that before a replica can execute and commit a transaction, a delay of two round trips is required. E.g., for transaction T_2, SLOG and Sloth outperform Calvin because no WAN communication is needed for T_2 to commit.

As the (Figure 4.1) example demonstrates, Sloth achieves significant latency savings over both SLOG and Calvin. SLOG and Calvin wait 2× longer than Sloth for T_1 to commit. Calvin waits 2× longer and SLOG waits 2.5× longer than Sloth for T_3 to commit. Sloth's latency decreases over SLOG and Calvin for T_1 and T_3 results from Sloth needing only a single round trip to exchange partial sequences among regions. SLOG performs worse on T_3 than Calvin due to needing to wait for all of T_3's pieces to be propagated from the local logs after T_3 is totally ordered against all MR transactions.

4.2 The Sloth System

In the rest of this chapter, the design of the Sloth system with a focus on its sequencer and related components including the provision of fault tolerance is described. To provide distributed low-latency geo-replicated transactions, this thesis presents a novel protocol that the sequencer in Sloth utilizes to generate a serializable global transaction order (Definition 3.0.1) *without* needing to know the total order of all transactions.

4.2.1 Sequencer Architecture

Sloth's sequencer has three components; the transaction batcher, the partial sequencer, and the sequencer merger (Figure 4.2). The transaction batcher receives transactions from clients, creates batches of transactions, and send the batches to all partial sequencers. In particular, transactions are sent to a region's partial sequencer only if they access at least one data item for which that region is the primary.

The partial sequencer for a region R orders all transactions with at least one data item for which R contains the primary copy. The partial sequencer sends the partial sequence to the sequence merger at each region. Formally, given the read set $rs(T_i)$ and write set $ws(T_i)$ of transaction T_i and the set $ps(R)$ of all data items for which region R contains the primary copy, the partial sequence for region R is an ordering of (all) transactions T_i s.t. $(ws(T_i) \cup rs(T_i)) \cap ps(R) \neq \emptyset$. As an optimization, the partial sequencer orders batches of transactions rather than individual transactions.

The sequence merger creates a globally consistent transaction order by performing the deterministic merging of all partial sequences from each regions' partial sequencer. The ordering of transactions are *conflict equivalent* [13] to all other orderings of the transactions at other regions.

Definition 4.2.1. *For two orderings of transactions, O_1 at region R_1 and O_2 at region R_2, O_1 and O_2 are conflict equivalent [13] if for all pairs of transactions T_i and T_j s.t. $[ws(T_i) \cap (ws(T_j) \cup rs(T_j))] \cup [ws(T_j) \cap (rs(T_i) \cup ws(T_i))] \neq \emptyset$, T_i and T_j appear in the same relative order in each of the transaction orderings O_1 and O_2.*[1]

The key idea behind the sequencer in Sloth is that a transaction T will appear in the partial sequence of a region R if and only if R is the primary for a data item that T accesses. This means that a regions's sequence merger only needs to wait for T to appear in the partial sequences for the primary of data items in T's read and write set before T can be added to the globally consistent transaction order. Thus, if T's read and write sets have primary copy locality (primary copy is held by region locally), T can be added to the transaction order with no messaging delay, and if the regions are close then the messaging delay will be small. In comparison, systems such as Calvin [41] must incur the full WAN messaging delays of Paxos before a transaction can be executed and committed at any region.

Next, how Sloth performs the merging of partial sequences to create a globally consistent transaction order is described. Furthermore, it is shown that the conflict equivalent orderings of transactions at each region guarantees correctness to provide global serializability for transaction execution.

4.2.2 Sequence Merging

Sequence merging in Sloth allows transactions at a region to be added to the globally consistent order as soon as possible without waiting for communication with other regions. Sloth takes advantage of this property to exploit the locality that exists in workloads. To create a globally consistent transaction order through deterministic sequence merging, each region's sequence merger keeps a copy of a *global directed conflict graph* that is used to keep track of all conflicts between transactions. Sequence mergers use the ordering provided by the partial sequences to dictate the direction of the edges in the global conflict graph. Furthermore, as the sequence mergers will see the same partial sequences, the global directed conflict graph is the same at each region. Thus, this graph allows the sequence mergers to reach an identical view of the ordering of all conflicting transactions.

[1]As read operations do not conflict with each other, no ordering constraints between them are required within the global order.

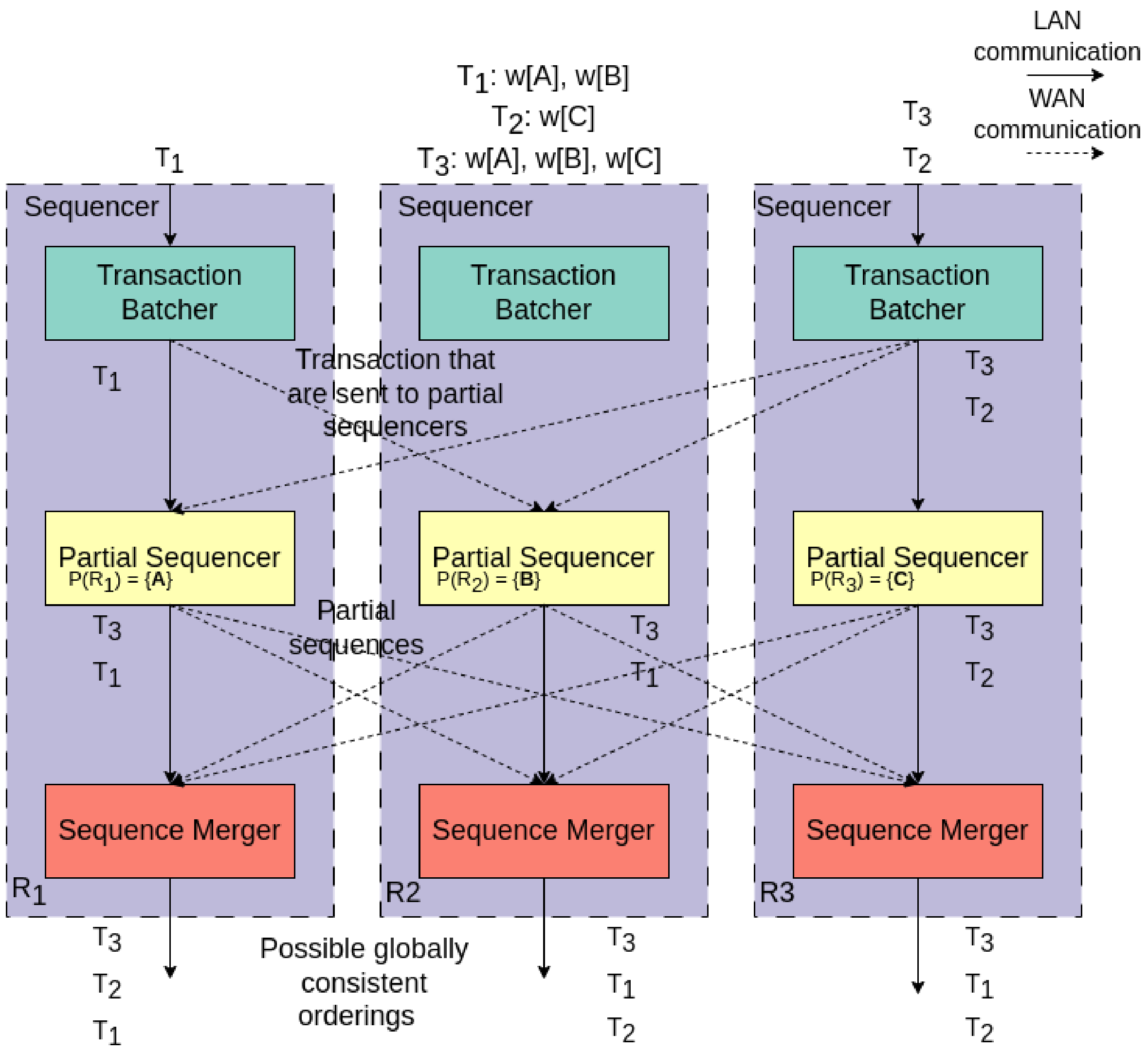


Figure 4.2: Sloth sequencer architecture with three regions: a running example.

In the directed conflict graph, each transaction is represented as a vertex, and each conflict is represented as a directed edge (V_i, V_j) where vertices V_i and V_j represent the conflicting transactions T_i and T_j. The direction of the edge between two vertices represents which transaction comes first in a partial sequence. As transactions are received in the partial sequences, the sequence mergers continuously add vertices and edges to the directed conflict graph. When the sequence merger has received all ordering information of a transaction and its conflicts, the transaction's vertex is removed from the graph, and the transaction is added to the transaction order.

When a sequence merger at region R receives a batch of transactions from a partial sequencer, the sequence merger runs Sloth's insert algorithm (Algorithm 1) that parses the batch and inserts transactions into the graph in the order of the partial sequence. When the insert algorithm encounters a transaction, it does the following:

1. The first time the sequence merger at R encounters the transaction T in any partial sequence, a vertex representing T is added to the graph.

2. Each time the sequence merger at R sees a transaction in a partial sequence, edges representing conflicts are added to the graph. The edges added depend on the current state of the graph and the partial sequences (Chapter 4.2.3 describes this in detail).

The sequence mergers remove vertices from the directed conflict graph in topologically sorted order and add the corresponding transactions to the globally consistent transaction order. In creating the transaction order, cyclic dependencies can form in the directed conflict graph, in which case a topological ordering may not exist. To derive a topological order, the sequence merger removes cycles from the directed conflict graph before sequence merging can be performed. These cycles are removed deterministically to preserve the equivalence of global directed conflict at each region. The sequence mergers monitor for *strongly connected components* (SCC) which are defined as follows:

Definition 4.2.2. *A strongly connected component is maximal subset of vertices C, of a directed graph G, which $\forall V_1, V_2 \in C$, $\exists$ a path from V_1 to V_2 and a path from V_2 to V_1 [18].*

The sequence mergers then create the *condensation* of the directed conflict graph to deterministically remove the cyclic conflicts. The condensation of a graph is define as:

Definition 4.2.3. *The condensation is the graph when each SCC is represented as a single vertex with only the edges outside of SCCs remaining [18].*

The three main steps of deterministic sequence merging – inserting transactions, resolving cyclic conflicts, and removing transactions in topologically sorted order are described next. Subsequently, the correctness of the above approach is shown.

4.2.3 Conflict Graph

When the sequence mergers receive transactions as part of a region's partial sequence, each sequence merger updates its copy of the global conflict graph to reflect the new conflict information in the partial sequence. When the sequence mergers receive a transaction T as part of a partial sequence, any conflicts among data items in T's read and write sets are discovered and the corresponding edges are added to the graph. This gives the following property:

Property 4.2.1. *All sequence mergers will identify the same set of transaction conflicts.*

Property 4.2.1 ensures that each sequence merger will identify the same set of conflicting transactions represented by their respective edges in its copy of the global conflict graph as all other sequence mergers.

There are three types of conflicts between transactions that must be considered: read follows write (RW) conflicts, write follows read (WR) conflicts, and write follows write (WW) conflicts. Edges are added for only the most recent conflicts, whereas older conflicts are captured implicitly through a directed path in the conflict graph. The sequence merger handles the different types of conflicts as follows:

- For each data item D in a transaction T's read set and write set where the primary copy of D is at region R, a directed edge is added from T's vertex to the vertex of the most recent transaction T^* that appears before T in the partial sequence at R and writes D. This edge ensures that a transaction's read/write operation must come after any conflicting write operation on D already represented in the conflict graph.
- For each data item D in a transaction T's write set, a directed edge is added from T's vertex to all transactions that read D between T and T^* in the partial sequence at R. These edges ensure that a transaction's write operations on D will come after any transactions already represented in the conflict graph that read D.

The distinction among different types of conflicts allows concurrent reads on the same data item to execute in any order at different regions provided they do not appear out of order with respect to conflicting writes. A running example is given in Figure 4.3 with the same setup as in Figure 4.2; three regions with a single data item's primary copy at each region. Consider Figure 4.3a. T_3 appears in all three partial sequences. When the sequence merger encounters T_3 in any particular partial sequence, an edge is added between T_3 and the most recent conflicting transactions in the partial sequence, except when an edge already exists (as would be the case when a second edge would be added from T_3 to T_1).

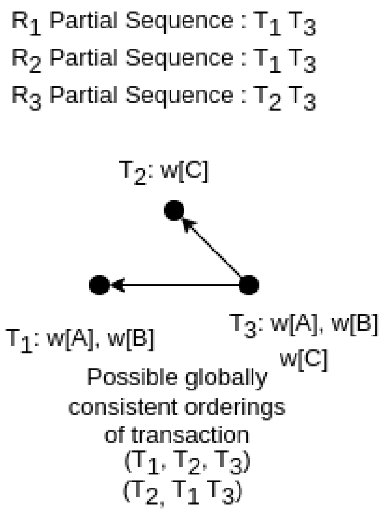
For Figure 4.3b, if transactions T_1 and T_2 are submitted (at different regions) simultaneously, they may appear in different relative orders in each of the two different partial sequences, resulting in the cycle in the global conflict graph as shown in 4.3b. The edge (T_3, T_2) is added because T_2 is the most recent conflicting transaction on data item A in the partial sequence at R_1 when T_3 is added. Finally, 4.3c shows how reads are handled. T_1 writes A, whereas T_2 and T_3 read only A. Thus, no edges are added between T_2 and T_3, but edges are added to T_1 from both T_2 and T_3. As T_4 also writes A, edges are added from T_4 to T_2 and T_3.

Algorithm 1: Insert Algorithm

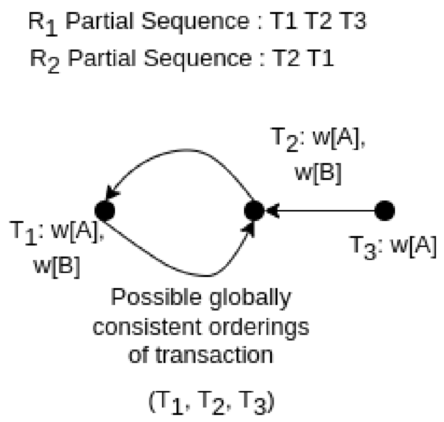
```
Transaction T;
for D in T.ReadSet do
    PrevWriter ⟵ MostRecentWriter(D);
    if !(PrevWriter.deleted) then
        addEdge(T, PrevWriter);
    end
end
for D in T.WriteSet do
    PrevWriter ⟵ MostRecentWriter(D);
    Readers ⟵ GetReaders(D) ;
    if !(PrevWriter.deleted) then
        addEdge(T, PrevWriter);
    end
    if Readers ≠ ∅ then
        for Reader in Readers do
            if !Reader.deleted then
                addEdge(T, Reader);
            end
        end
    end
end
```

4.2.4 Resolving Cyclic Conflicts

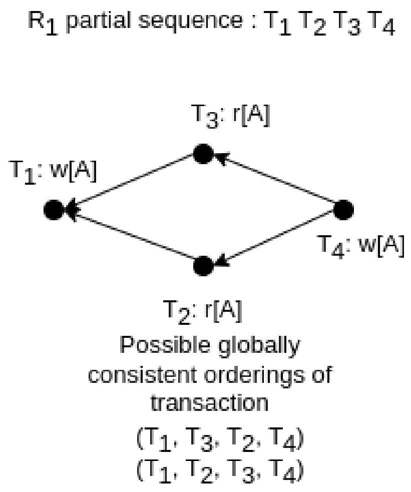
Next is description how cycles, such as in the directed conflict graph of Figure 4.3b, are eliminated by the sequence merger at each region independently and with no communication with


R1 Partial Sequence : T1 T3
R2 Partial Sequence : T1 T3
R3 Partial Sequence : T2 T3
T2: w[C]
T1: w[A], w[B]
T3: w[A], w[B] w[C]
Possible globally consistent orderings of transaction
(T1, T2, T3)
(T2, T1 T3)

(a) Running example


R1 Partial Sequence : T1 T2 T3
R2 Partial Sequence : T2 T1
T2: w[A], w[B]
T1: w[A], w[B]
T3: w[A]
Possible globally consistent orderings of transaction
(T1, T2, T3)

(b) Cyclic conflict


R1 partial sequence : T1 T2 T3 T4
T3: r[A]
T1: w[A]
T4: w[A]
T2: r[A]
Possible globally consistent orderings of transaction
(T1, T3, T2, T4)
(T1, T2, T3, T4)

(c) Reads intermixed with writes

Figure 4.3: Global conflict graph examples

other regions. The challenging task is to deterministically choose which cycle to reorder at each region. Sloth uses the global conflict graph's SCCs, rather than individual cycles, to deterministically resolve cyclic conflicts at each region. Rather than have each sequence merger choose a cycle to be reordered, the sequence mergers choose an SCC (which contains one or more cycles) to reorder. The sequence mergers eliminate SCCs by reordering conflicts. Sloth leverages two properties of graphs related to SCCs that allow the sequence merger to reorder conflicts deterministically:

Property 4.2.2. *The strongly connected components (Definition 4.2.2) in a graph is unique [18].*

Property 4.2.3. *The condensation (Definition 4.2.3) of a directed graph is a directed acyclic graph (DAG) [18].*

Property 4.2.1 together with Property 4.2.2 ensures that the set of SCCs will also be the same at each region, and therefore the condensation graph (Defintion 4.2.3) will also be the same at each region. Property 4.2.3 ensures that a topological order exists for SCCs. Thus, if the sequence mergers identify SCCs, they can create an order of transactions at each region that corresponds to a topological order of the condensation of the global conflict graph. The sequence mergers use Tarjan's SCC algorithm [40] to independently find the SCCs in the global conflict graph. Sequence mergers are continuously adding and removing vertices and edges from the conflict graph as they receive transactions in the partial sequences. Thus, the algorithm executes continuously in a loop over the (evolving) global conflict graph. After an SCC has been identified, topological sort (Algorithm 2) is used to assess whether the SCC can be safely reordered and added to the globally consistent transaction order, or if the SCC must wait for further transaction ordering information. This procedure is discussed in greater detail next.

The process of resolving cyclic conflicts can be viewed as explicitly creating the condensation of the conflict graph and then performing topological sorting on the condensation. However, in practice, SCCs are identified, and the vertices (of the conflict graph) are dynamically updated with a reference to which SCC they belong. The topological sort is then performed over the conflict graph while accounting for the vertices with a reference to an SCC. In particular, if the topological sort would consider a vertex with a reference to an SCC for removal, it instead considers the entire SCC. Not explicitly creating the condensation of conflict graph is performant as it reduces the amount of data manipulation that must occur to resolve cyclic conflicts.

4.2.5 Generating the Complete Order

The sequence merger builds on Khan's algorithm [22] to create a topologically sorted order of the condensation of the global conflict graph (Algorithm 2). The algorithm must assess whether the sequence merger can add a transaction to the globally consistent order. The number of partial sequences a transaction will appear in is extracted from a transaction's read/write sets. This allows Sloth to know when no more outgoing edges will be added to the global conflict graph for a transaction T. Since Sloth assigns each transaction a globally unique ID, this is used to identify a fixed deterministic order for transactions within an SCC. Thus, the following two properties hold:

Property 4.2.4. *If a transaction T exists in each region's partial sequence for which the region holds T's primary data item(s), then no new outgoing edges will be added by that region's sequence merger to the vertex representing T in the global conflict graph. T's vertex in the graph is referred to as being complete.*

Property 4.2.5. *If all vertices in an SCC are complete and no outgoing edges exist from a vertex within the SCC to a vertex not in the SCC, then the SCC is maximal. Such an SCC is called complete.*

If an SCC is complete, transactions in the SCC can be added to the global order deterministically by the sequence mergers. The transactions are added in sorted order based on their globally unique transaction ID. Furthermore, if the vertex for a transaction has no outgoing edges and is complete, meaning it is in a complete SCC of size 1, the transaction can be immediately added to the globally consistent transaction order.

Figure 4.3 provides an example. Both T_1 and T_2 in Figure 4.3a have no outgoing edges, meaning they do not conflict with any transactions in the graph. Furthermore, as both T_1 and T_2 are complete, they are in SCCs of size one. Thus, the sequence mergers can add both transactions in any order to the globally consistent transaction order. T_3 conflicts with both T_1 and T_2 and therefore must wait until T_1 and T_2 are added to the transaction order. Hence, T_3 will appear after T_1 and T_2 in all orders, which gives (T_1, T_2, T_3) and (T_2, T_1, T_3) as valid conflict equivalent orderings of transactions. A cycle exists in Figure 4.3b, which results in the SCC containing T_1 and T_2. T_1 and T_2 will be added to the global order in sorted order based on their globally unique transaction IDs. Again, as T_3 conflicts with the transactions in the SCC, it must appear in the transaction order after all transactions in the SCC, which results in a single valid ordering (T_1, T_2, T_3) for transactions. Finally, in Figure 4.3c, as T_3 and T_2 are read operations, they can be added to the transaction order in any order *after* T_1 is added. T_4 must be added after both T_2 and T_3 are added to the transaction order, which results in both (T_1, T_2, T_3, T_4) and (T_1, T_3, T_2, T_4) being valid, conflict equivalent, orderings of transactions.

Algorithm 2: Algorithm for topological sort

```
Q ⟵ Queue for transactions that may be ready to be added to globally consistent
  transaction order;
S ⟵ Globally consistent transaction order;
while True do
    T ⟵ Q.pop();
    if T.SCCSize = 1 then
        if T.complete and T.getOutNeighbours() = ∅ then
            S.append(T);
            Q.push(T.getInNeighbours());
        end
    else
        SCC ⟵ T.getSCC();
        if SCC.complete and SCC.outEdges = ∅ then
            DetOrder ⟵ Sort(SCC.transactions) ;
            for T in DetOrder do
                S.append(T);
            end
            Q.push(SCC.getInNeighbours()) ;
        end
    end
end
```

4.2.6 Correctness

The correctness of the aforementioned protocols in Sloth is now proven. It is shown that each ordering of transactions is conflict equivalent to all possible orderings at other replicas. Furthermore it also shown that this property, coupled with the Calvin deterministic scheduler, produces conflict equivalent serializable transaction schedules at each region.

Lemma 4.2.6. *Each region will identify the same set of strongly connected components (Defintion 4.2.2) in their copy of the global directed conflict graph.*

Proof. Each sequence merger will see the same transaction conflicts (Property 4.2.1). Thus, for each transaction, the corresponding vertex and edges will be the same in each copy of the global conflict graph. As the SCCs of a graph are unique (Property 4.2.2), these SCCs will converge to be the same at each region's sequencer. □

Theorem 4.2.7. *Each ordering of transactions at a region is conflict equivalent to all orders at the other regions.*

Proof. Consider two conflicting transactions in any regions transaction order. There are two cases for these transactions: either T_1 and T_2 appear within the same SCC (which is the same at all regions) in the conflict graph, or they appear within different SCCs.

In the first case, if T_1 and T_2 appear in the same SCC, then they will appear in the same relative order in all possible transaction orderings, at all replicas, as each transaction in an SCC is added to the transaction order in a deterministic ordering according to a globally unique transaction ID.

If T_1 and T_2 do not appear in the same SCCs then there exists some directed path between T_1 and T_2. Assume, without loss of generality, that this path is from T_2 to T_1. However, these transactions do not appear in the same SCC so there is no path from T_1 to T_2. As transactions are added to the transaction order only after all neighbours have been added, T_2 can be added only after T_1 (and all other transactions on the path from T_2 to T_1) have been added.

By Lemma 4.2.6, each region will see the same SCCs. Thus, the order of transactions will be conflict equivalent across all regions as all transactions that conflict will have the same relative order in all possible orderings. □

As mentioned in Chapter 3, the Calvin deterministic scheduler executes transactions according to the transaction order that it is given. The scheduler employs a locking scheme that guarantees serializability for local execution orders [41].

Theorem 4.2.8. *The global order results in conflict equivalent transaction execution schedules across all regions.*

Proof. By Theorem 4.2.7, the ordering of transactions at each region is conflict equivalent to all other regions. Given that the deterministic scheduler executes transactions in this order, the resulting execution schedule will be equivalent at each region. □

It should be noted that each ordering of transactions (at a region) is a serial ordering of transactions. Thus, by Theorem 4.2.8 and Theorem 4.2.7, each transaction execution schedule is conflict equivalent to a serial order.

4.2.7 Fault Tolerance

Discussed next is how Sloth tolerates failures while still committing transactions in a single WAN round trip. A two-tier approach is used in Sloth for fault tolerance: one handles failures within a region, and another handles region failures.

Sloth handles machine-level failures with no WAN RTTs by using Paxos to replicate each partial sequence within the region from which the partial sequence originates. This scheme is similar to how SLOG handles machine-level failures [37].

Handling region-level failures is more involved. Sloth utilizes a key observation from Flexible Paxos [21] for replication. Namely, the election and replication quorums do not need to be the same; rather, they need to only intersect. Thus, systems can trade-off the size of the replication quorums for lower latency replication under WAN communication [34, 21, 10].

The Sloth sequencers replicate each partial sequence to K other regions.[2] If N is the total number of regions, $(N - K)$ alive regions are required to recover from a failure (so that the replication and election quorums intersect [21]). The sequencers already propagate the partial sequence to all regions. Thus, replication simply requires a region waiting for K acknowledgements to its partial sequence from other regions, which will result in only a small delay if K is also small. The fault tolerance scheme is equivalent to running Flexible Paxos with a long-lived leader for each partial sequence among all regions. Importantly, the fault-tolerance scheme does not change the one WAN RTT transaction commit as a region simply waits for K acknowledgements to its partial sequence, which overlaps with waiting for transaction ordering information from the other regions. The scheme can lead to transactions incurring at least one WAN round

[2] K can be a small value to reduce latency.

trip to commit, but if $K = 1$, the transaction needs to wait on WAN communication with only the closest region, which can be as low as 6 ms (Table 1.1).

A region R that sequences a transaction T must wait for T's position in the partial sequence to be replicated at K regions before R can commit T. If another region R' is not a sequencer for T, R' can commit T as soon as R' receives T in all partial sequences from regions that sequence T. For correctness, it is assumed a region notifies a client of the status of a transaction T only if it is a sequencer for T. In the case of one or more region failures, the surviving regions run a recovery protocol to resume normal operation. This protocol is presented next.

Recovery protocol

To tolerate region failures, if a region stops receiving the partial sequence from another region[3], it initiates the recovery process. The recovery process needs at least $(N - K)$ regions alive and thus can recover from K or fewer region failures. If at least $(N - K)$ regions are alive, the recovery process for a failed region R_f is as follows:

- $(N - K)$ regions agree to elect a new primary region R_n for data items for which R_f is the primary.
- Each region stops accepting the partial sequence from R_f.
- Each region exchanges its copy of R_f's partial sequence to ensure that all other regions have the most up to date partial sequence.
- The new partial sequencer at R_n includes transactions to the most up to date partial sequence.
- Each region resends all missing transactions from the partial sequence to R_n.[4]

Correctness

If a region R_f fails, other regions will no longer receive transactions as part of the partial sequence from R_f. Thus, a region (other than R_f) will not be able to execute any transactions on data items for which R_f was the primary. So in the case of failure, correctness is not affected. The recovery process must recover from failures while ensuring that if a transaction T, that is in

[3]Timeouts are used to detect failures.

[4]If a transaction is not recoverable, it can be presumed failed and can be safely resubmitted to another replica.

a partial sequence from failed region R_f, is also in another region R''s partial sequence, then as long as there are fewer than $K + 1$ failures, T's position in the partial sequence will not change and thus the execution order will not change at any region.

Lemma 4.2.9. *The recovery process will not change the position of any transactions in the partial sequence of any active (non-failing) region.*

Proof. As the recovery process chooses the most up-to-date partial sequence and sends it to all active regions, as long as a transaction was in the partial sequence at one region, it will appear in the partial sequence of every active region. □

Lemma 4.2.10. *As long as at least ($N - K$) regions are alive, no transactions can be lost due to the failure of a region R_f.*

Proof. A transaction T's position in a region's partial sequence must be replicated at K other regions. Thus, as long as there are fewer than $K + 1$ region failures, then T's position in all partial sequences must be preserved since at least one surviving region will still have T in its copy of R_f's partial sequence. □

Chapter 5

Performance Evaluation

This chapter evaluates the performance of Sloth. Sloth is compared against two systems: (i) Calvin [41], a principal system that incorporates deterministic geo-replication, and (ii) SLOG [37], a state-of-the-art system for low latency transactions in geo-replicated deterministic databases. Both Sloth and SLOG build on the Calvin code base [3] that implements the scheduler and storage components.

As discussed in Chapter 3, many geo-replicated database systems, including Calvin[1] rely on consensus-based approaches such as Paxos for maintaining consistent replicas. SLOG can use a variety of methods including Paxos for ordering MR transactions. SLOG's implementation [3] orders all transactions via a single region [37]. SLOG is the closest comparison system to Sloth. Both Sloth and SLOG designate a region as the primary for a data partition. Sloth and SLOG exploit data locality based on the location of a data item's primary to reduce transaction latency. Thus, SLOG is a relevant system to compare against Sloth in terms of leveraging locality to provide low-latency transactions.

5.1 Methodology

The experiments were conducted on Microsoft's Azure Cloud using 24 Standard_D48_v5 virtual machines each with 48 vCPUs and 192 GiB RAM. Experimental measurements are shown as graphs with each graphed data point as the average of three independent runs. Each system deployment contains six replicas with four data partitions. Each replica is within a different

[1]Calvin uses Paxos for consensus.

region: US East, US East 2, France Central, EU West, Southeast Asia and East Asia. Each region contains a full replica. The regions within the same continent are referred to as *close* regions, e.g., US East and US East 2, and regions in different continents are referred to as *far* regions, e.g., US East and France Central. The system load is given by the number of clients, with each client submitting 200 transactions per second in open loop to the respective system for execution.

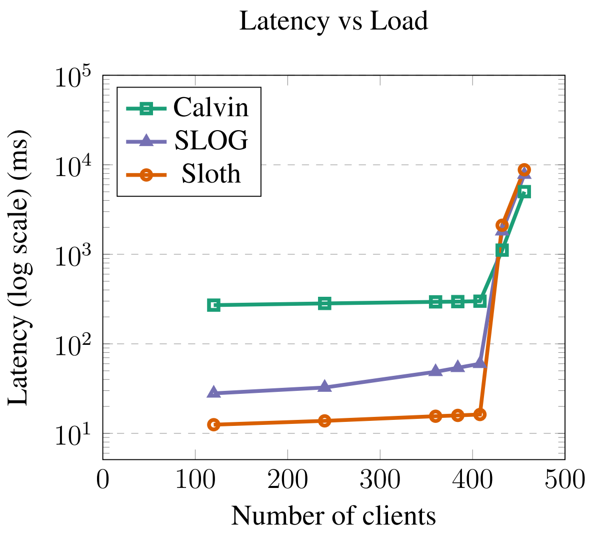
5.2 Benchmarks

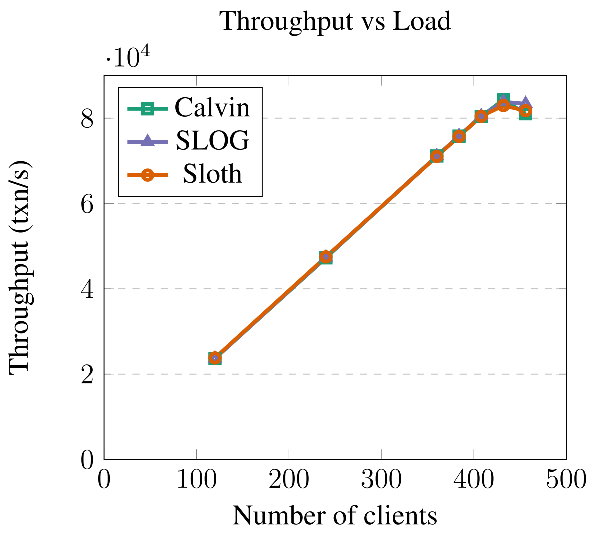
This thesis uses the TPC-C [7] and MovR [42, 4] benchmark workloads to evaluate the systems. The TPC-C benchmark is a popularly used workload representing a business application that processes orders. As in [37], every warehouse has a primary in a region with all supporting data (district records, customer records, and so on). Similarly to SLOG, this thesis focuses on the throughput and latency of NewOrder transactions as these are the transactions from the TPC-C benchmark that can be multi-region (MR) (through Multi-Warehouse (MW)) transactions. In the benchmark, each MW NewOrder transaction has a probability to involve two warehouses' records with primary copies in different regions. 10% of all NewOrder transactions are MW. Furthermore, each MW NewOrder transaction has a 5/6*th* chance of being a multi-partition transaction. 240 warehouses per region are used in the TPC-C experiments.

MovR is a carsharing application benchmark [42, 4]. For the experiments, locality is added to MovR, users live in a region but travel to close regions with probability P_c and far regions with probability P_f. Furthermore, each car belongs to a region. Thus, all user data has a primary copy in the user's region, and all vehicle data has a primary copy in the vehicle's region. Furthermore, ride data contains a primary in the region where the ride takes place. Thus, all transactions access primary data in a single region except for the BeginRide transactions that can access user and vehicle primary data in different regions. When a user travels to another region, the BeginRide transaction becomes an MR transaction between the user's primary region and the vehicle's primary region. This thesis measures the throughput and latency of the BeginRide transactions. The MovR experiments use 1 million MovR users per region and 10,000 vehicles per region.

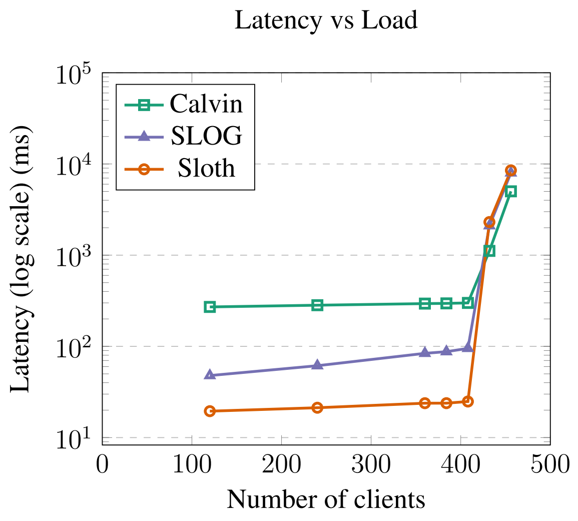
5.3 Results

The performance of Sloth, SLOG, and Calvin on the TPC-C benchmark as the percentage of MW transactions that are MR transactions increases – 10% (Figure 5.1c), 50% (Figure 5.1a),

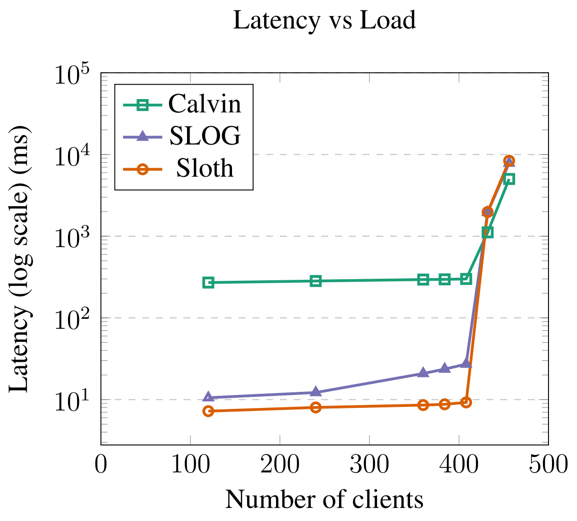




(a) 50% of MW transactions are MR



(b) 100% of MW transactions are MR



(c) 10% of MW transactions are MR

Figure 5.1: TPCC Latency & Throughput

and 100% (Figure 5.1b) is studied. The throughput graph for 50% MW transactions that are MR is shown (the throughput graphs for 10% and 100% are omitted as all 3 systems are bottlenecked by the scheduler, resulting in similar throughput graphs). Sloth has significantly lower latency than Calvin and SLOG until the saturation point (408 clients), while transaction throughput is comparable for all systems. As the number of clients increases, the gap resulting from Sloth's (lower) transaction latency and SLOG's latency also increases.

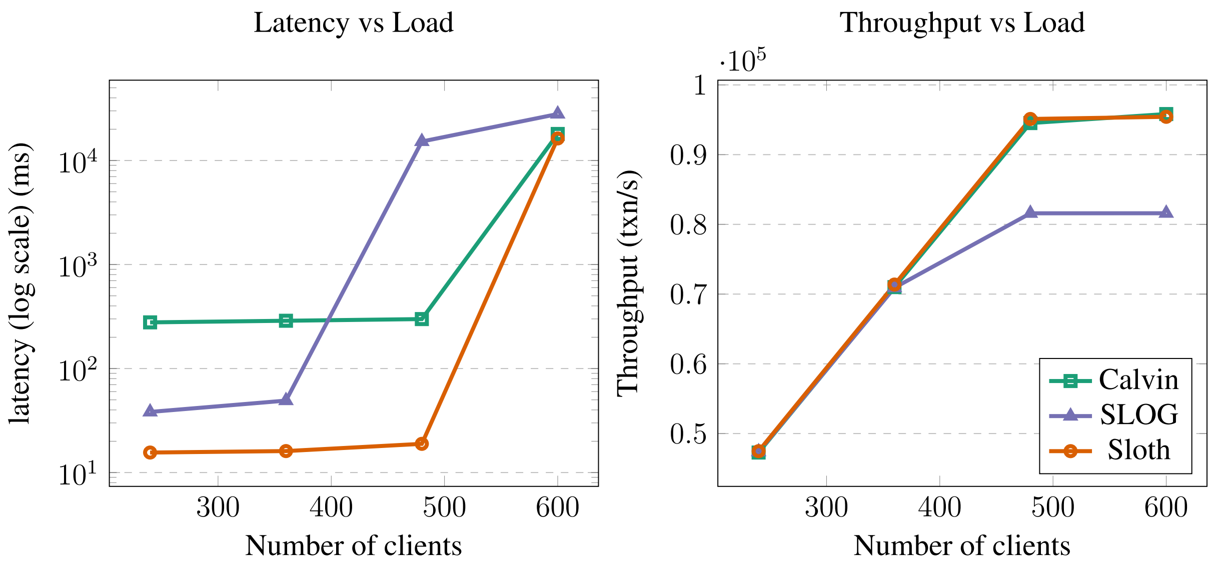
Under all load conditions, Sloth incurs significantly lower latency than the other systems (Figures 5.1b, 5.1a and 5.1c). For 100% MR transactions, these latency gains range from about 3.7× lower than SLOG and 18× lower than Calvin to almost 3× lower than SLOG and 38× lower than Calvin for 10% MR transactions.

As each client (in an open loop) continuously submits 200 transactions per second, most of the transaction latency comes from waiting on WAN communication for ordering. In particular, after all ordering information for a transaction is received, the transaction does not wait to be executed; i.e., all three systems execute and commit transactions as fast as they are received, causing throughput to be the same. As Sloth incurs the lowest latency for ordering, it performs the best on transaction latency. However, the deterministic scheduler becomes the bottleneck for all three systems past the saturation point, which means that the systems can no longer execute transactions as fast as they are submitted (and throughput is capped by the maximum throughput of the deterministic schedulers). After the saturation point, transaction latency is affected by waiting for execution by the deterministic scheduler. The effect of this waiting can be seen in Figure 5.5a.

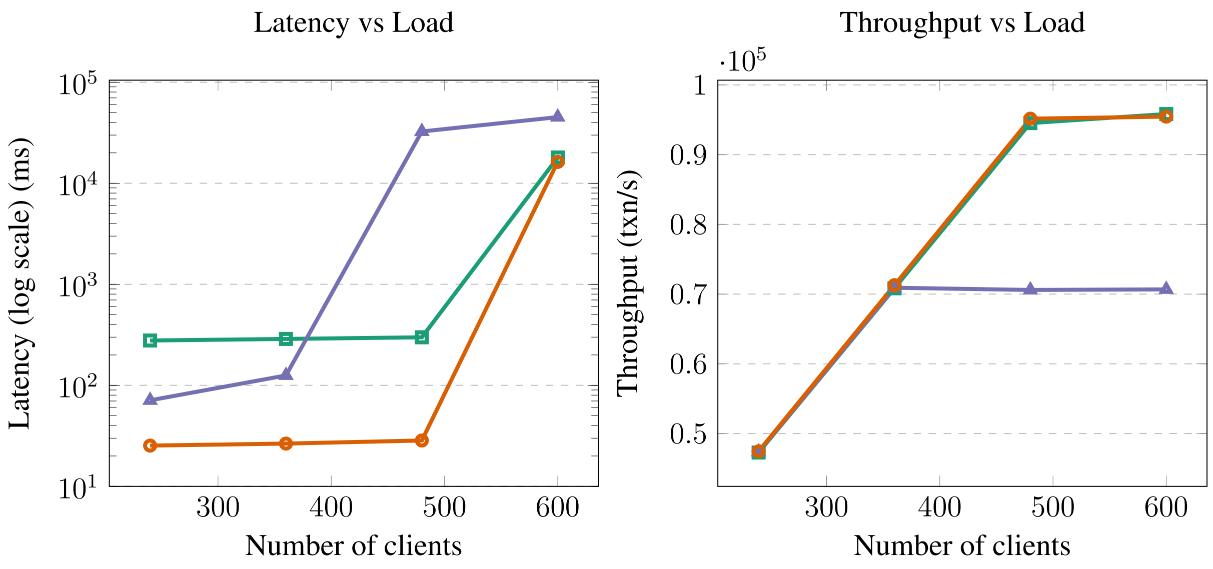
Figure 5.2a and Figure 5.2b, look at the latency and throughput of Calvin, SLOG, and Sloth with the MovR benchmark as the client load on the systems increases. 10000 vehicles per region is used for the MovR experiments. Furthermore, MovR is run twice: with $P_f = .05$ and $P_c = .10$ (Figure 5.2a), and also with $P_f = .10$ and $P_c = .20$ (Figure 5.2b) for 15 and 30 percent of MR transactions, respectively. While Calvin and Sloth have comparable throughput, SLOG's peak throughput is lower, and plateaus earlier, due to the increased load on the schedulers that a higher percentage of MR transactions incur in SLOG. After SLOG's throughput plateaus, its latency increases above the other systems. Calvin and Sloth become saturated at similar points, thereby showing similar latency and throughput behaviour.

With 15%MR transactions, Sloth has 3.1× lower latency than SLOG and almost 18× lower latency than Calvin (Figure 5.2a). The latency advantages by Sloth over SLOG increase to 4.7× when 30% of transactions are MR with an almost 11× improvement over Calvin (Figure 5.2b).

SLOG has two drawbacks that limit its performance compared to Sloth: First, SLOG totally orders all multi-region (MR) transactions. The total ordering requirement means SLOG cannot exploit data locality when a transaction is MR, resulting in higher latency. Second, as SLOG



(a) 15%MR Transactions ($P_f = .5$, $P_c = .10$)



(b) 30%MR transactions ($P_f = .10$, $P_c = .20$)

Figure 5.2: Azure MovR Latency and Throughout

breaks MR transactions into transaction pieces, these pieces add additional load on the deterministic scheduler, increasing latency and lowering throughput. That is, when an MR transaction accesses k regions, k transaction pieces are sent to the scheduler (MR transactions place k times more load on the scheduler in SLOG than in Calvin or Sloth).

Sloth's across-the-board latency gains allow it to generally outperform SLOG and Calvin. Sloth's performance advantage is from not needing to totally order any transactions globally. Calvin's high latency comes from the need to totally order *all* transactions globally . SLOG needs to enforce a total order on MR transactions globally, which results in lower latency when compared to Calvin but much higher latency than Sloth (Chapter 5.3.1, Figure 5.4).

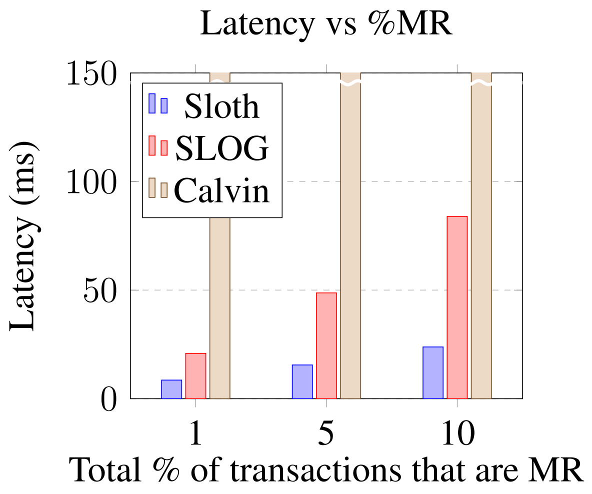
5.3.1 Latency

The rest of this chapter looks more closely at the latency of Sloth, SLOG, and Calvin on both the MovR and TPC-C benchmarks using varying percentages of MR transactions.

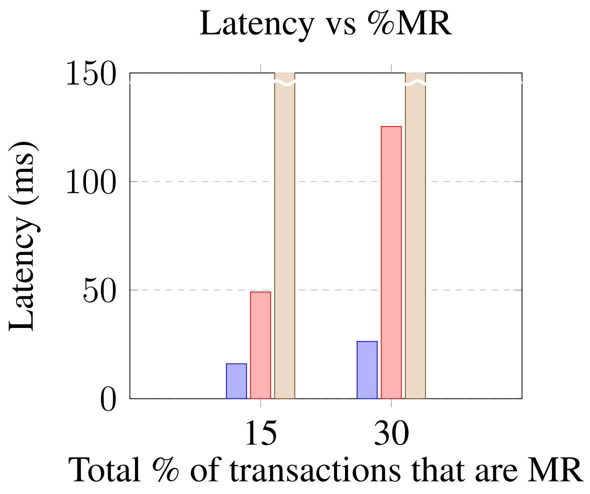
Sloth, Calvin and SLOG's latency performance is compared when the percentage of MR transactions is varied. For both benchmarks the number of clients is set to 320, which is a high enough value to place significant load on the systems but without putting them into overload. Sloth outperforms SLOG by 4.7× on MoVR (Figure 5.3b) and 3.5× on TPC-C (Figure 5.3a) with 10% and 30% total MR transactions, respectively. Figure 5.3a and Figure 5.3b show that as the percent of MR transactions increases, the average transaction latency of SLOG increases faster than that of Sloth due to SLOG's need to totally order all multi-region transactions globally that is not done by Sloth.

A comparison between the latency of local transactions, MR transactions between close regions, and MR transactions between far regions is given in Figure 5.4a and Figure 5.4b. 10% total MR transactions is used for TPC-C (Figure 5.4a) and 30% MR transactions is used for MovR (Figure 5.4b). As each transaction is globally ordered through Paxos, Calvin's average transaction latency is the same for local, close, and far transactions. Sloth outperforms both Calvin and SLOG for every type of transaction. SLOG performs worse than Calvin for MR transactions between far regions due to the propagation of the positions of transaction pieces in local logs after the global total ordering is done. The largest latency disparity between Sloth and SLOG is for multi-region transactions between close regions where Sloth outperforms SLOG by about 6× on TPC-C and 6.3× on MovR. These results demonstrate Sloth's ability to take advantage of the locality of data placement. Furthermore, per Calvin's performance in Figure 5.4b, totally ordering transactions in geo-replicated systems can add a very large latency overhead.

To provide a more complete picture of latency performance, this thesis provides the empirical cumulative distribution functions (CDFs) of transaction latency for Sloth, SLOG, and Calvin for

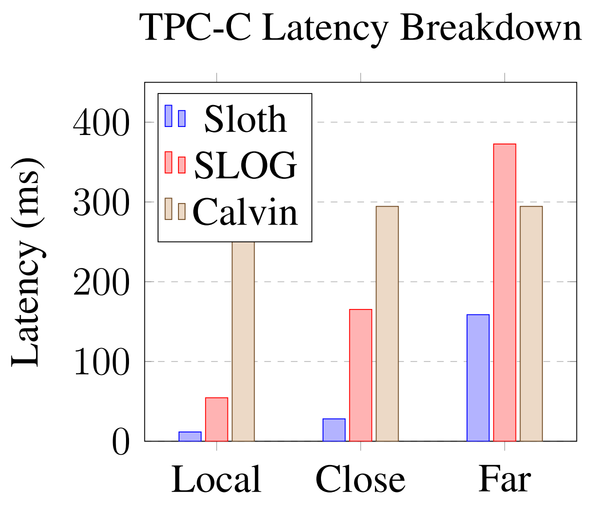


(a) TPC-C

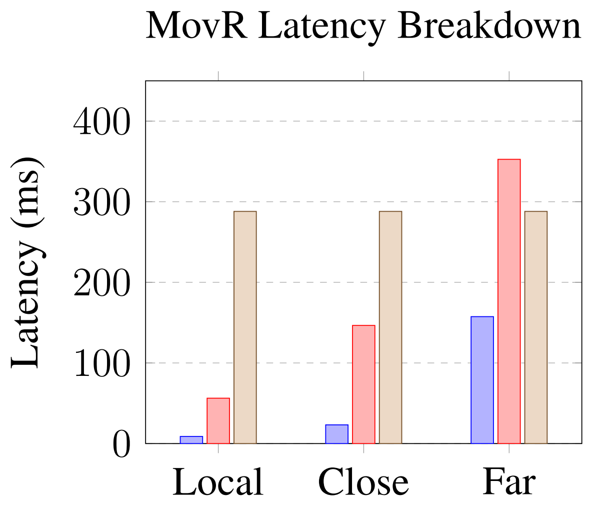


(b) MovR

Figure 5.3: Calvin, SLOG & Sloth Latency



(a) TPC-C

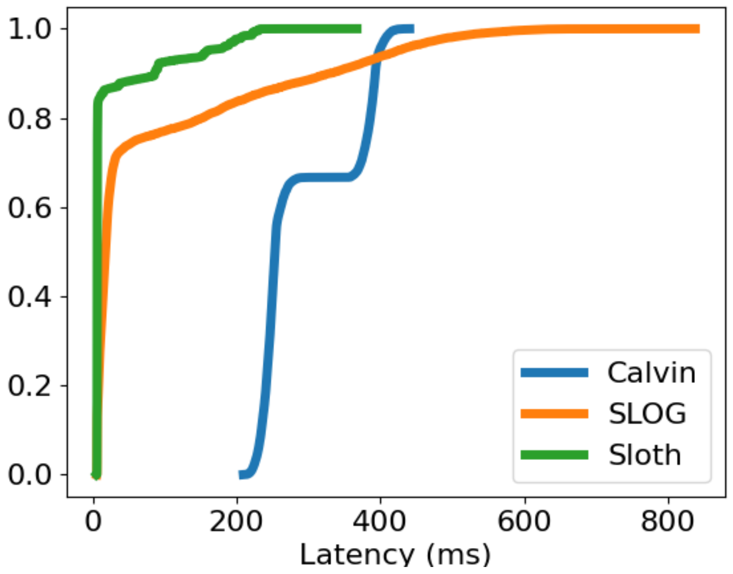


(b) MovR

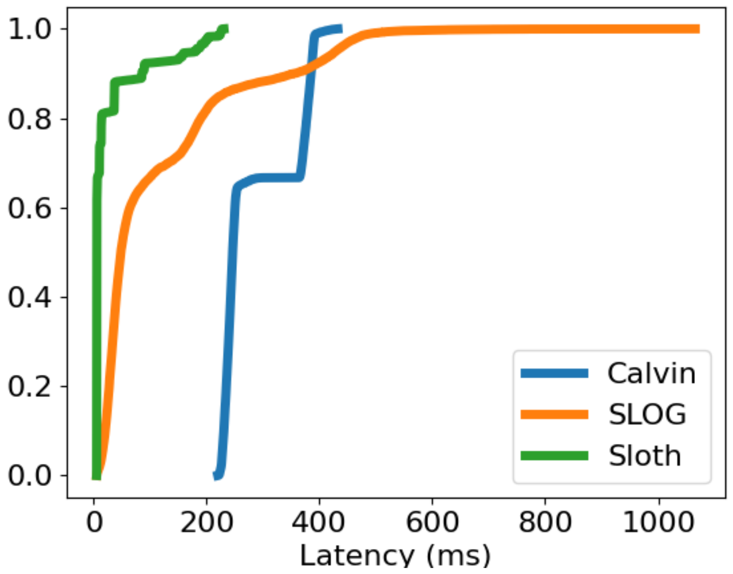
Figure 5.4: Latency breakdown by transaction type

TPC-C (Figure 5.5a) and MovR (Figure 5.5b) using the same setup as for the experimental results in Figure 5.3. It is seen that the CDFs of Sloth, for both TPC-C (Figure 5.5a) and MovR (Figure 5.5b), are steeper than the CDF of SLOG. The divergence of the CDFs when the fraction of the data is above 0.9 for TPC-C and 0.7 for MovR is expected due to the percentage of MR transactions. However, the divergence of the curves below these points show the increased load SLOG places on its deterministic scheduler due to the MR transaction pieces it generates. SLOG beats Calvin on average latency. However, SLOG has larger latency at the tail. Sloth provides the best of both worlds with lower average latency and lower latency at the tail.

The SLOG CDF shows the effect of waiting transactions. For TPC-C, 90% of transactions are single-region, meaning that in SLOG and Sloth these single-region transactions can execute without waiting on WAN communication, which would result in low latency for these transactions. However, for SLOG, the curve starts to plateau much earlier than 90%, meaning that the single region transactions are experiencing higher latency. The increased latency for single-region transactions shows the effect of the load on the scheduler delaying transaction execution (transaction latency increases as transactions must wait to execute).



(a) TPC-C CDF



(b) MovR CDF

Figure 5.5: CDF of transaction latency

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presented Sloth, a geo-replicated deterministic database system that commits transactions serializably after a single WAN round trip of messaging delay. Sloth performs deterministic merging of partial sequences of transactions per region to totally order transactions globally rather than relying on a cross-region total ordering of transactions. Moreover, Sloth exploits locality in workloads allowing transactions to execute without waiting on WAN messaging for non-conflicting transactions. The evaluation shows that Sloth outperforms state-of-the-art deterministic database systems Calvin and SLOG by up to 38$\times$ and 6$\times$, respectively, for transaction latency.

6.2 Future Work

I now discuss two ways that Sloth can be extended for future research.

6.2.1 Snapshot Isolation

An interesting avenue to explore is reducing the isolation level of transactions in Sloth from serializability to snapshot isolation. This change would lower latency for transactions that do not perform writes, but do perform reads, on data with the primary copy at other regions. In particular, with the proper timestamp management, only a transaction's write set (and not read set) would need to be sequenced in partial sequences. For a transaction's reads, a begin timestamp would need to be deterministically chosen by all replicas. A simple scheme is to have each primary for the transaction propose a timestamp and choose the largest. This change should further reduce transaction latency while also reducing the sequencer load as less conflicts would need to be tracked (reads would no longer be sequenced).

6.2.2 Migrating the Primary

SLOG [37] can use a heuristic scheme to move the primary location to increase locality. Exploring machine learning-based techniques to increase locality in Sloth would be interesting. Furthermore, it would be interesting to see the effect of machine learning-based primary movement on transaction latency in Sloth.

References

[1] American Airlines and Microsoft Partnership Takes Flight to Create a Smoother Travel Experience for Customers and Better Technology Tools for Team Members. https://news.aa.com/news/news-details/2022/American-Airlines-and-Microsoft-Partnership-Takes-Flight-to-Create-a-Smoother-Travel-Experience-for-Customers-and-Better-Technology-Tools-for-Team-Members-MKG-OTH-05/default.aspx. Accessed: 2023-05-09.

[2] Azure network round-trip latency statistics. https://learn.microsoft.com/en-us/azure/networking/azure-network-latency. Accessed: 2023-05-09.

[3] CalvinDB. https://github.com/kunrenyale/calvindb. Accessed: 2023-05-09.

[4] MovR. https://www.cockroachlabs.com/docs/stable/movr.html. Accessed: 2023-05-09.

[5] Pismo. https://www.cockroachlabs.com/customers/pismo/. Accessed: 2023-05-09.

[6] Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/12721/. Accessed: 2023-05-09.

[7] TPC-C. https://www.tpc.org/tpcc/. Accessed: 2023-05-09.

[8] Daniel J. Abadi and Jose M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, aug 2018.

[9] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Dynamast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1381–1392, 2020.

[10] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223, jan 2020.

[11] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, 2008.

[12] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.

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

[14] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - a transactional record manager for shared flash. In *Conference on Innovative Data Systems Research*, 2011.

[15] Philip A. Bernstein, Colin W. Reid, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. *Proc. VLDB Endow.*, 4(11):944–955, aug 2011.

[16] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. Confluxdb. *Proceedings of the VLDB Endowment*, 7:947–958, 07 2014.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[19] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2391–2406. ACM, 2020.

[20] F. Harary, R.Z. Norman, and D. Cartwright. *Structural Models: An Introduction to the Theory of Directed Graphs*. Wiley, 1965.

[21] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited, 2016.

[22] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, nov 1962.

[23] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, nov 1962.

[24] 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*, EuroSys '13, page 113–126, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Leslie Lamport. Paxos made simple. 2001.

[26] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[27] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[28] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[29] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, jul 2013.

[30] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 369–384, USA, 2008. USENIX Association.

[31] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[32] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[33] 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, Savannah, GA, November 2016. USENIX Association.

[34] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1221–1236, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 47–64, Boston, MA, February 2019. USENIX Association.

[36] Thamir M. Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *International Conference on Extending Database Technology*, 2020.

[37] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, jul 2019.

[38] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple rational guidance for chopping up transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, page 298–307, New York, NY, USA, 1992. Association for Computing Machinery.

[39] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple rational guidance for chopping up transactions. *SIGMOD Rec.*, 21(2):298–307, jun 1992.

[40] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[41] 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.

[42] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2312–2325, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 231–243, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. Starry: Multi-master transaction processing on semi-leader architecture. *Proc. VLDB Endow.*, 16(1):77–89, sep 2022.