Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems

Joshua Hildred
Cheriton School of Computer Science
University of Waterloo
jthildred@uwaterloo.ca

Michael Abebe
Cheriton School of Computer Science
University of Waterloo
michael.abebe@uwaterloo.ca

Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
khuzaima.daudjee@uwaterloo.ca

ABSTRACT

Distributed deterministic database systems achieve high transaction throughput for geographically replicated data. Supporting transactions with ACID guarantees requires deterministic databases to order transactions globally to dictate execution order. In a geographically distributed environment, ordering transactions globally can take multiple wide-area network (WAN) round trips of messaging, which adds significant latency to transaction response times, leading to poor user experiences. To improve the response time of transactions in deterministic databases, we propose an ordering protocol that can include a transaction in the global order in a single WAN round trip to the primary regions of the data items within the transaction's read and write set. The protocol reduces the cost of determining the global order for all transactions by leveraging deterministic merging of partial sequences of transactions per geographic region. We implement the protocol in Caerus, our geo-replicated deterministic database system that serializably commits and replicates transactions after a delay of only a single WAN round trip of messaging. Using popular workload benchmarks over geographically replicated data in Azure, we show that Caerus outperforms state-of-the-art comparison systems to deliver low-latency transaction execution.

PVLDB Reference Format:
Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. PVLDB, 17(3): 469 - 482, 2023.
doi:10.14778/3632093.3632109

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/josh-hildred/Caerus.

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: advertising platforms [15], banking [28], global travel operations [4] and online games [17].

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 [22]. Companies can leverage geo-replicated databases to place data closer to clients for low latency access of their global services [52]. For example, a Yahoo trace revealed 85% regional locality for user data accesses [14].

Second, 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 only against 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 [5]. For example, Amazon reportedly lost 99 million dollars of revenue when their e-commerce site experienced an hour of downtime during peak shopping time [21]. To mitigate such serious revenue losses due to failures, companies have started implementing engineering policies stating a service must be able to survive some number of geographically distributed availability zone failures plus one machine failure, placing greater importance on having performant geo-replicated database systems [18, 20].

Distributing work across data replicas requires coordination and communication that can pose significant challenges for geographically 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 [33, 47]. 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 (Table 1). Unless a geo-distributed system can mitigate these large WAN latencies, they will translate into high transaction response times and lower throughput, leading to poor overall distributed system performance [47].

Early work on geo-replicated database systems that support strong consistency for transactions uses a combination of Paxos [31] and Two-Phase Commit (2PC) to provide ACID transactional guarantees. For example, Spanner [15] uses Paxos and 2PC which can take blue at least 4 WAN round trips to coordinate distributed transaction commit and consistent replica maintenance [33]. The number of WAN round trips for a transaction to commit can be reduced to 2 by running 2PC within each region while using Paxos to agree on the outcome of 2PC [33]. A geo-replicated database system can use a primary site architecture in which a designated region (primary) is responsible for enforcing ordered access to data

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632109

Table 1: Azure inter- and intra-continent latency [35]

WAN RTT			
Continent	NA	AP	EU
NA	< 6 ms	> 200 ms	> 82 ms
AP	> 200 ms	< 34 ms	> 159 ms
EU	> 82 ms	> 159 ms	< 12 ms

items. This architecture can reduce transaction latency within the primary's region in which communication will happen over a LAN. However, transactions outside the primary region will still incur high latency when communicating with the primary over a WAN [2, 43, 47]. Thus, the challenge of ameliorating the high cost of transaction latencies in geo-distributed database systems remains.

Systems that rely on consensus approaches such as Paxos for coordination and fault tolerance over large WAN latencies generally do not perform well. 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 East Asia [35]. Based on these regions, 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 a worst case of 191 ms (East Asia, France Central, West EU) resulting in high latencies for geo-replicated transactions.

Deterministic database systems have improved upon the performance of geo-replicated database systems that use distributed commit protocols, such as 2PC, by predetermining a global order for transaction execution [43, 47]. An observation for distributed deterministic database systems is that much of a transaction's latency is from the WAN communication required to include the transaction in the global order, as once the transaction is in the global order, no further coordination is needed [42, 43, 47]. For example, Paxos is used by Calvin to facilitate the creation of the global order, resulting in 2 round trips over the replicas to reach an agreement to add a transaction to the global order. We discuss deterministic databases in more detail in Section 2.

1.1 Contributions

In this paper, we present Caerus, our geo-replicated deterministic database system that significantly reduces transaction latencies by exploiting locality, and determining regional transaction orders that are consistent with global transaction execution schedules. We provide a transaction ordering protocol that requires, at most, a single WAN round trip to order any transaction, allowing Caerus to commit and replicate transactions within a single WAN round trip. The ordering protocol deterministically merges partial transaction orders into a globally consistent merged order that preserves correctness for serializable transaction isolation and replica consistency. The deterministic merging of partial transaction orders allows any replica to begin executing a transaction 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).

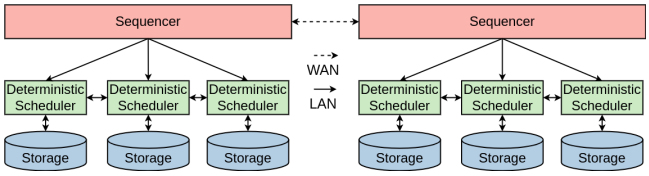


Figure 1: Architecture of a replicated deterministic database

We evaluate Caerus' performance using the TPC-C [49] and MovR [50] benchmarks. We demonstrate that Caerus outperforms the comparison systems SLOG [43] by 6× and Calvin [47] by up to 38× on transaction latency. Finally, we demonstrate that Caerus provides a lightweight mechanism for region-level fault tolerance and high-availability.

2 BACKGROUND

Deterministic database systems enforce determinism for transaction execution to eliminate communication during execution and commit [1, 47]. When transaction execution is deterministic, as long as a transaction executes on a consistent database state, the transaction's result will always be the same. This means replicas can independently commit transactions without coordination as long as they can guarantee that the transactions execute on consistent database states. Thus, deterministic databases generally create a predetermined *global order* for transaction execution using transactions' read and write sets [47] to ensure that transactions execute on consistent states. A deterministic database system will execute transactions according to the global order to ensure transactions are *globally serializable*[1] across replicas.

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 [47]. Deterministic database systems generally assume that transactions' read and write sets are known a priori. In practice, static analysis of transaction code can be used to deduce a transaction's read and write set [44]. Protocols such as OLLP can used when static analysis cannot [44, 46, 47]. Distributed deterministic database systems typically have three core components; *sequencer, deterministic scheduler*, and *storage layer* (Figure 1).

The *sequencer* is responsible for generating the global transaction order, thus 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[2] [42, 43, 47]. Typically, the global order is created through agreement by a consensus protocol such as Paxos [37, 47]. The consensus protocol would provide 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

[1]Formally, concurrent transactions executing on multiple copies of a data item appear as if they have executed on a single copy of the data item in a serial order [9].
[2]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 200 ms.

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 [47]. However, primary-based ordering schemes are more susceptible to failure than consensus-based approaches. Consensus protocols replicate both the transaction execution schedule and 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 nonoperational for possibly long time periods.

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 [47].

The Calvin system [47] 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 and 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.

SLOG [43] attempts to mitigate high transaction latency in geo-replicated deployments through the use of a dual approach to transaction sequencing. A replica is designated as the primary replica for each data item. A transaction is single-region if all data items in the transaction's read and write sets are located at a single (primary) replica. Otherwise, the transaction is multi-region. Single-region transactions can be executed immediately and committed at the region's replica through the ordering in a per replica local log. Multi-region transactions must fall back to being totally ordered against all other multi-region transactions that have an added complexity: they must be broken up into pieces and ordered against all other conflicting transactions in the appropriate local logs. These transaction pieces represent lock requests for a partition of the transaction's read and write set at a given primary replica. Furthermore, the position of a multi-region transaction's pieces in a local log shows the ordering among conflicting operations of multi-region and single-region transactions for data at the primary replica for which the local log belongs. The local logs must be synchronized at all replicas, up to the position of each transaction piece for the multi-region transactions, before a transaction's execution can be completed [43]. This means that a multi-region transaction in SLOG incurs an extra half round trip of WAN communication compared to the same transaction in Calvin (Figure 2).

Our Caerus system 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.

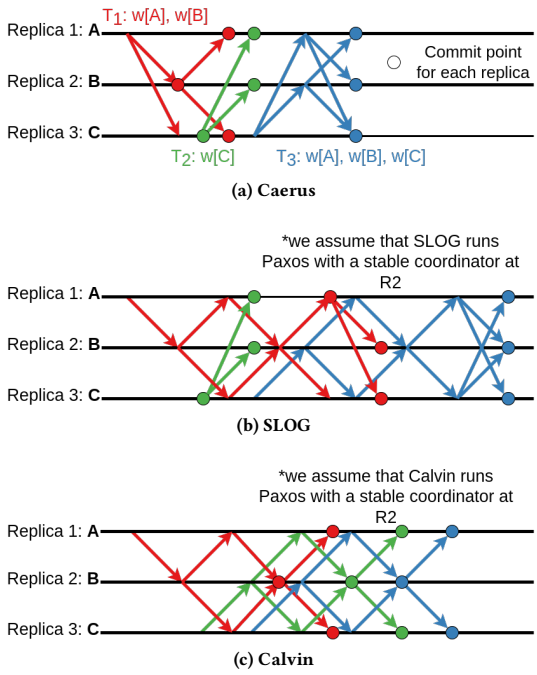


Figure 2: Example of Calvin, SLOG and Caerus sequencer message passing

3 CAERUS OVERVIEW

This section presents an overview of Caerus' system model and transaction ordering. We also present some basic terminology used in the rest of the paper followed by an example that shows how Caerus delivers transaction execution latency savings over its competitors.

3.1 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*. Furthermore, we assume that each transaction T has a globally unique ID.

When presenting Caerus' sequencer protocol, we consider 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, we refer to R 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). We refer to a transaction T as being a single-region (SR) transaction when the data items that it accesses (reads and/or writes) have

their primary copies located at a single region. Otherwise, T is a multi-region (MR) transaction.

A Caerus 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. We refer to this order as R's *partial sequence*, and for a Transaction T, we say R *sequences* T, or R is a *sequencer* for T. 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 *merged 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. The merging means that each region's merged order of transactions may differ from the merged orders in other regions. However, importantly, if any two transactions conflict, then their relative (conflict equivalent) order is preserved in the merged order at each region (Definition 4.1).

3.2 Transaction Ordering

Per region deterministic merging creates a consistent merged transaction order for serializable transaction execution across regions. Deterministic merging can result in transactions being added to the merged order and executed in different orders at different regions. In particular, as long as a region has transaction T's *dependency information* defined as T's position in all relevant partial sequences, along with any preceding conflicting transaction's positions, 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 to create a total global order. For example, a region R_1 can add T to its transaction order and execute T before another region R_2 knows T exists. We detail the sequencer in Section 4.1 and discuss deterministic merging in Section 4.2.

A fundamental beneficial property of Caerus' ordering protocol is that it incurs *at most* a single round trip of communication to include a transaction into a globally consistent merged transaction order; thus, the transaction can execute and commit with a single RTT delay. As mentioned above, a region can add a transaction T to the merged order once it has T's position in all partial sequences. The transaction requires half a round trip to send it to each region, and another half round trip to propagate partial sequences to all regions.

Furthermore, the latency can be reduced significantly if locality exists among regions that 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. If T is an SR transaction, no communication will be performed before T can executed at R. If R holds the primary copy of only part of T's read and write set with another region R' containing one or more primary copies for the rest, R must wait on communication with only R', which significantly reduces transaction latency if these regions are close to each other.

3.2.1 Example. We exemplify the performance advantages of the above-mentioned properties using Figure 2a. In this example, data is fully replicated at all regions, i.e., each region holds copies of all 3 data items while being the primary for 1 of the 3 data items. Region R_1 holds the primary copy of A, region R_2 holds the primary copy of B, and region R_3 holds the primary for C. For simplicity, we assume that the round trip latency between any two of these 3 regions is the same.

Multi-region Transaction $T1$, 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 merged order of transactions. Once the transaction is part of the merged 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 merged orders (again without any communication).

Single-region 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. As T_2 is singe-region, T_2 needs to be sequenced by only R_3's partial sequencer, T_2 can be added to the merged 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 merged transaction order at the respective regions.

As multi-region 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 2b 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_1 and T_3 are MR and must be totally ordered against all 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 ½ round trip. Thus, in SLOG, MR transactions require 2½ WAN round trips before they can be committed, compared to only 1 round trip in Caerus.

Figure 2c 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.

As the (Figure 2) example demonstrates, Caerus achieves significant latency savings over both SLOG and Calvin. SLOG and Calvin wait 2× longer than Caerus for T_1 to commit. Calvin waits 2× longer and SLOG waits 2.5× longer than Caerus for T_3 to commit. Caerus' performance gains over SLOG and Calvin result from Caerus needing at most a single round trip to order transactions.

4 THE CAERUS SYSTEM

In this section, we describe the design of the Caerus system with a focus on its sequencer and related components including the provision of fault tolerance. To provide distributed low-latency geo-replicated transactions, we present a novel protocol that the sequencer in Caerus utilizes to generate a serializable global transaction order *without* needing to know the total order of all transactions. Caerus is implemented into the Calvin codebase [12]. Caerus uses the Calvin deterministic scheduler and storage engine. The Caerus sequencer was implemented in C++ from scratch.

4.1 Sequencer Architecture

Caerus' sequencer has 3 components; the transaction batcher, the partial sequencer, and the sequence merger (Figure 3). The transaction batcher receives transactions from clients, creates batches of transactions and sends the batches to all partial sequencers (① in Figure 3). 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 (② in Figure 3). Formally, given the read set $\text{rs}(T_i)$ and write set $\text{ws}(T_i)$ of transaction T_i and the set $\text{ps}(R)$ 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. $(\text{ws}(T_i) \cup \text{rs}(T_i)) \cap \text{ps}(R) \neq \emptyset$. As an optimization, the partial sequencer orders batches of transactions rather than individual transactions.

The sequence merger creates a globally consistent merged transaction order by performing the deterministic merging of all partial sequences from each regions' partial sequencer (③ in Figure 3). The merged ordering of transactions is *conflict equivalent* [9] to all other merged orderings of the transactions at other regions (created by the regions' sequence mergers).

Definition 4.1. 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* [9] if for all pairs of transactions T_i and T_j s.t. $[\text{ws}(T_i) \cap (\text{ws}(T_j) \cup \text{rs}(T_j))] \cup [\text{ws}(T_j) \cap (\text{rs}(T_i) \cup \text{ws}(T_i))] \neq \emptyset$, T_i and T_j appear in the same relative order in each of the transaction orderings O_1 and O_2.

A key idea behind the sequencer in Caerus 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 region's sequence merger needs to wait for only 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 merged transaction order with no messaging delay, and if the regions are geographically close then the messaging delay will be small. In contrast, systems such as Calvin [47] must incur the WAN messaging delays of Paxos before a transaction can be executed at any region.

Next, we describe how the merging of partial sequences creates a globally consistent merged transaction order. Furthermore, we show that the conflict equivalent orderings of transactions in the merged orders at each region guarantee global serializability for transaction execution.

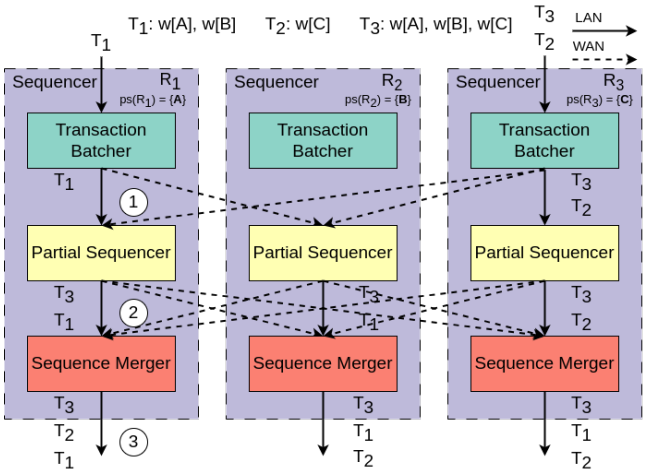


Figure 3: Caerus sequencer architecture with three regions (follows the running example from Figure 2).

4.2 Sequence Merging

Sequence merging in Caerus allows transactions to be added to a region's globally consistent merged transaction order without incurring extra communication with other regions. Caerus takes advantage of this property to exploit any existing geographic locality to reduce transaction latency. To create a globally consistent merged 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 conflicts between transactions. Sequence mergers use the ordering provided by the partial sequences to dictate the direction of the edges in this conflict graph. As all sequence mergers will see identical orders of transactions within each partial sequence (and thus the same order of transaction conflicts), 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.

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 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, sequence mergers continuously add vertices and edges to the directed conflict graph. When a sequence merger has received all of a transaction's dependency information, the transaction's vertex is removed from the graph, and the transaction is added to the merged transaction order.

When a sequence merger at region R receives a partial sequence (batch) of transactions, the sequence merger runs Caerus' 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 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 (Section 4.3).

By running Algorithm 1 on transactions in the partial sequence orders at each sequence merger, conflicts will be discovered in the same order at all regions. Thus, the order of conflicting transactions will also be the same at each region, meaning each region will independently construct the same global directed conflict graph, as each region will add the same edges to their respective copies.

Sequence mergers continuously remove vertices from their copy of the directed conflict graph in topologically sorted order and add the corresponding transactions to the globally consistent merged order. We discuss how a sequence merger can independently know when a transaction can be added to the topologically sorted merged order in Section 4.5. In creating the merged 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, sequence mergers must remove the cycles from their copies of the directed conflict graph. These cycles are removed deterministically to preserve the equivalence of global directed conflict at each region. Sequence mergers monitor for strongly connected components (SCC) [16] and replace them with a single representative vertex, creating the *condensation* [16] of the directed conflict graph to ensure a topological order exists. We describe next the three main steps of deterministic sequence merging – inserting transactions, resolving cyclic conflicts, and removing transactions in topologically sorted order. Subsequently, we show the correctness of the above approach.

Algorithm 1: Insert Algorithm for transactions in the partial sequence of region R, being run by region R'

```
1  T ⟵ next transaction in the partial sequence from R;
2  ps(R) ⟵ primary set for R;
3  G ⟵ the global conflict graph copy at R';
4  if vertex(T) ∉ G then
5  |  addVertex(T, G);
6  end
7  for D in T.ReadSet ∩ ps(R) do
8  |  T* ⟵ MostRecentWriter(D);
9  |  if !T*.removed then
10 |  |  addEdge(T, T*, G);
11 |  end
12 end
13 for D ∈ T.WriteSet ∩ ps(R) do
14 |  T* ⟵ MostRecentWriter(D);
15 |  Readers ⟵ GetReaders(D) ;
16 |  if T*.removed then
17 |  |  addEdge(T, T*, G);
18 |  end
19 |  if Readers ∩ ps(R) ≠ ∅ then
20 |  |  for T** in Readers ∩ ps(R) do
21 |  |  |  if !T**.removed then
22 |  |  |  |  addEdge(T, T**, G);
23 |  |  |  end
24 |  |  end
25 |  end
26 end
```

4.3 Conflict Graph

When a sequence merger receives transactions as part of a region's partial sequence, the sequence merger updates its copy of the global conflict graph (Algorithm 1) to reflect the new conflict information in the partial sequence. When a sequence merger receives a transaction T, any conflicts among data items in T's read and write sets are discovered (lines 8 and 14 in Algorithm 1) and the corresponding edges are added to the graph (lines 10, 17, and 22, Algorithm 1). This gives Property 4.2, which ensures that sequence mergers will eventually build identical copies of the global conflict graph:

PROPERTY 4.2. *All sequence mergers will identify the same set of transaction conflicts.*

There are three types of conflicts between transactions that must be considered: read follows write (RW), write follows read (WR), and write follows write (WW). Edges are added for only the most recent conflicts, whereas older conflicts are captured implicitly through a directed path in the conflict graph. Sequence mergers handle 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, (lines 10 and 17, Algorithm 1). 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, (line 22 in Algorithm 1). These edges ensure that a transaction's write operations on D will come after 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 with the same setup as in Figure 3 – three regions with each data item's primary copy at a distinct region. Consider Figure 4a. 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).

If transactions T_1 and T_2 are submitted (at different regions) concurrently, 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 Figure 4b. 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, Figure 4c 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.

4.4 Resolving Cyclic Conflicts

We describe how cycles in the directed conflict graph of Figure 4b are eliminated by the sequence merger at each region independently and with no communication with other regions. The challenging task is to deterministically choose which cycle to reorder at each region. Caerus 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, sequence mergers choose an SCC (containing one or more cycles) to reorder. We leverage two properties of graphs that allow the sequence merger to reorder conflicts deterministically:

PROPERTY 4.3. *Any set of strongly connected components in a graph is unique [16].*

PROPERTY 4.4. *The condensation of a directed graph is a directed acyclic graph (DAG) [16].*

Property 4.2 together with Property 4.3 ensure that the set of SCCs will be the same at each region, and therefore the condensation graph will also be the same at each region. Property 4.4 shows a topological order can exist when SCCs form. Thus, if the sequence mergers identify SCCs, they can create a transaction order at each region that corresponds to a topological order of the condensation of the global conflict graph. Sequence mergers use Tarjan's algorithm [45] 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 further in Section 4.5.

If Caerus detects that the global conflict graph has grown past a (tunable) threshold value, the transaction batchers will throttle back sending to the partial sequencers. This throttling allows sequence mergers to prevent formation of large SCCs.

4.5 Generating Merged Transaction Orders

The sequencers implement a modified version of Khan's algorithm [26] to create a topologically sorted order from the condensation of the global conflict graph (Algorithm 2). The algorithm assesses whether a sequence merger can add a transaction to its globally consistent merged order independently of the other regions. Since the number of partial sequences a transaction will appear in is extracted from a transaction's read/write sets, a sequence merger knows when no more outgoing edges will be added to its copy of the global conflict graph for a transaction T. Since Caerus 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 4.5. *If a transaction T exists in each region's partial sequence for which the region holds T's primary data items, then no new outgoing edges from T will be added to any region's copy of the global conflict graph. We refer to T's vertex in the graph as being complete.*

PROPERTY 4.6. *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. We call such an SCC complete.*

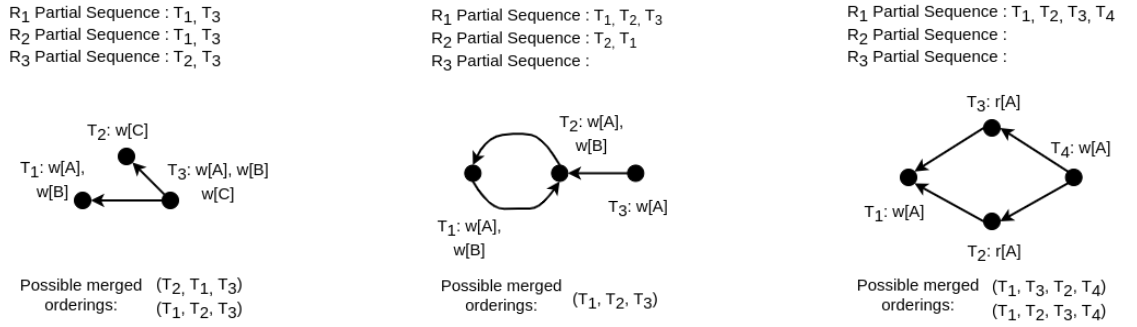
If an SCC is complete, transactions in the SCC can be added to the global order deterministically by the sequence merger. The transactions are added in sorted order based on their globally unique transaction ID. Furthermore, if the vertex for a transaction has no outgoing edges and is complete, meaning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

Figure 4 provides an example. Both T_1 and T_2 in Figure 4a 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, sequence mergers can add both transactions in any order to their globally consistent merged 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 merged transaction order. Hence, T_3 will appear after T_1 and T_2 in all merged orders, which gives (T_1, T_2, T_3) and (T_2, T_1, T_3) as valid conflict equivalent merged transaction orderings. A cycle exists in Figure 4b, resulting in the SCC containing T_1 and T_2. T_1 and T_2 will be added to the merged 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 all merged orders after all transactions in the SCC, which results in a single valid merged ordering (T_1, T_2, T_3) for transactions. Finally, in Figure 4c, as T_3 and T_2 are read operations, they can be added to the merged 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 merged 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, merged orderings of transactions.

As each sequence merger independently runs Algorithm 2 to create the merged sequence from the conflict graph, transactions may be added in different orders at each replica due to the merging algorithm being able to select any transaction (or SCC) with no outgoing edges. These merged orders will result in a globally serializable transaction schedule at each region, as proved in the next section.

4.6 Correctness

We prove the correctness of the aforementioned Caerus protocols. We show that each merged ordering of transactions is conflict equivalent to all possible merged orderings at other replicas. We also show that this property, coupled with the Calvin deterministic scheduler, produces conflict equivalent serializable transaction schedules at all regions.



(a) Running example **(b) Cyclic conflict** **(c) Reads interleaved with writes**

Figure 4: Global conflict graph examples

Algorithm 2: Algorithm run continuously for creating a merged order over replica R's copy of the evolving conflict graph

```
1  Q ⟵ Queue for transactions that may be ready to be added
     to globally consistent merged transaction order;
2  S ⟵ Globally consistent merged transaction order;
3  while True do
4      T ⟵ Q.pop();
5      if T.SCC.size < 2 then
6          if T.complete and T.getOutNeighbours() = ∅ then
7              S.append(T);
8              Q.push(T.getInNeighbours());
9          end
10     else
11         SCC ⟵ T.SCC;
12         if SCC.complete and SCC.outEdges = ∅ then
13             for T in Sort(SCC.transactions) do
14                 S.append(T);
15             end
16             Q.push(SCC.getInNeighbours()) ;
17         end
18     end
19 end
```

Lemma 4.7. *Each region will identify the same set of strongly connected components in their copy of the global directed conflict graph.*

Proof. Each sequence merger will see the same transaction conflicts (Property 4.2). 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 4.3), these SCCs will converge to be the same at each region's sequencer. □

Theorem 4.8. *Each ordering of transaction at a replica is conflict equivalent to all orders at the other replicas.*

Proof. Consider two conflicting transactions in the global 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 merged 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 SCC, 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 merged 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 4.7, 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 merged orderings. □

Theorem 4.9. *The merged orders result in conflict equivalent transaction execution schedules across all regions.*

Proof. By Theorem 4.8, the orderings of transactions at each region are conflict equivalent to all other regions. Given that the deterministic scheduler executes transactions in these orders, the resulting execution schedule will be equivalent at each region. □

Thus, as each merged ordering of transactions is a serial ordering of transactions, and as each of these orders are conflict equivalent by Theorem 4.9, each transaction execution schedule is conflict equivalent to a serial order.

4.7 Fault Tolerance

We discuss how Caerus tolerates failures while still committing transactions in a single WAN round trip. A two-level approach is used in Caerus for fault tolerance: one handles failures within a region, and another handles region failures.

Caerus handles machine-level failures with no WAN RTTs by replicating each partial sequence within the region from which the partial sequence originates. This scheme is similar to how SLOG handles machine-level failures [43].

In handling region-level failures, Caerus utilizes a key observation from Flexible Paxos [25] for consensus: the election and replication quorums do not need to be the same. Rather, they need to only intersect, thus, systems can trade-off election quorum size for lower replication latency under WAN communication by reducing the size of the replication quorum [3, 25, 38].

Caerus' sequencers replicate each partial sequence to K other regions. With N total number of regions, $(N - K)$ alive regions are required to recover from a failure (for intersection of replication and election quorums [25]). Since sequencers already propagate partial sequences to all regions, replication requires a region to wait 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. Note that the fault-tolerance scheme does not change the 1 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 1 WAN round 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).

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, we assume 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.

4.7.1 Recovery protocol. To tolerate region failures, if a region stops receiving the partial sequence from another region, it initiates the recovery process. The recovery process needs at least $(N - K)$ alive regions, 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:

(1) $(N - K)$ regions agree to elect a new primary region R_n for data items for which R_f is the primary.
(2) Each region stops accepting the partial sequence from R_f.
(3) 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.
(4) The new partial sequencer at R_n includes transactions to the most up-to-date partial sequence.
(5) Each region resends all missing transactions from the partial sequence to R_n.

If a region R_f fails, other regions will no longer receive transactions as part of the partial sequence from R_f, and will therefore not be able to execute any transactions on data items for which R_f was the primary. Thus, 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 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. Therefore, the execution order will not change at any region.

Lemma 4.10. *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 4.11. *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 will be preserved since at least one surviving region will still have T in its copy of R_f's partial sequence. □

5 PERFORMANCE EVALUATION

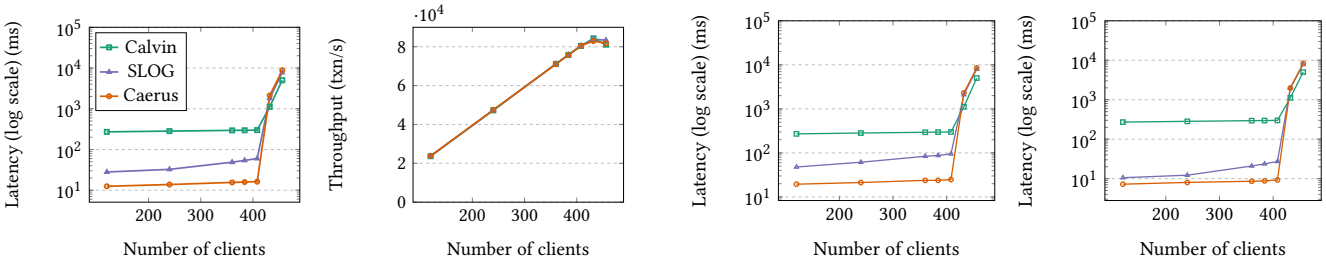
This section evaluates the performance of Caerus. We primarily compare against two geo-replicated systems: (i) Calvin [47], a principal system that incorporates deterministic geo-replication, and (ii) SLOG [43], a state-of-the-art system for low latency transactions in geo-replicated deterministic databases. Both Caerus and SLOG build on the Calvin code base [12] that implements the scheduler and storage components.

5.1 Methodology

Our experiments were conducted on Microsoft's Azure Cloud using 24 Standard_D48_v5 virtual machines each with 48 vCPUs and 192 GiB RAM. Our 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 region: US East, US East 2, France Central, EU West, Southeast Asia and East Asia. Each region contains a full replica. We call regions within the same Continent *close* regions, e.g., US East and US East 2, and regions in different Continents *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 an open loop to the respective system for execution.

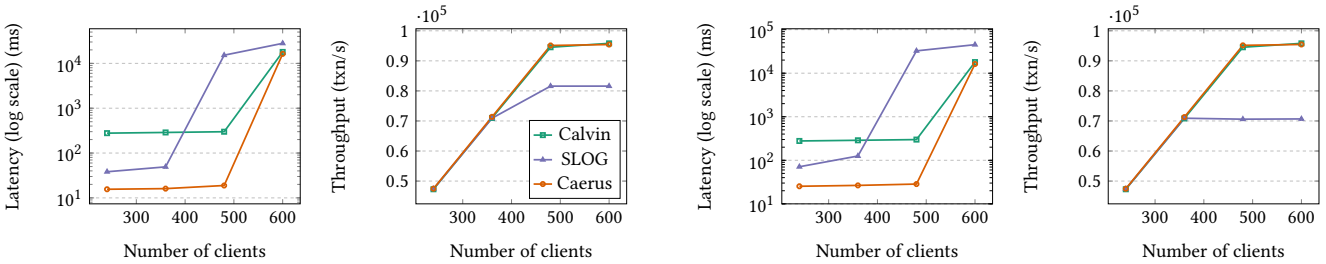
5.2 Benchmarks

We use the popular TPC-C [49] and MovR [29, 50] benchmark workloads to evaluate the systems. TPC-C represents a business application that processes orders. Every warehouse has a primary in a region with all supporting data (district records, customer records, and so on). We focus 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. Each MW NewOrder transaction has a probability to involve two warehouses' records with primary copies in



(a) 50% of MW Transactions are MR (b) 100% of MW Transactions are MR (c) 10% of MW Transactions are MR

Figure 5: TPCC Latency & Throughput



(a) 15%MR transactions (P_f = .5, P_c = .10) (b) 30%MR transactions (P_f = .10, P_c = .20)

Figure 6: MovR Latency & Throughout

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. We use 960 warehouses.

MovR is a carsharing application benchmark [29, 50]. With locality added to MovR, users live in a region but travel to close regions with probability P_c and far regions with probability P_f. Each car belongs to a region. Every user data item has its primary copy located in the user's region and every vehicle data item has its primary copy located 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. We measure the throughput and latency of the BeginRide transactions. Our experiments use 4 million MovR users and 40,000 vehicles.

5.3 Results

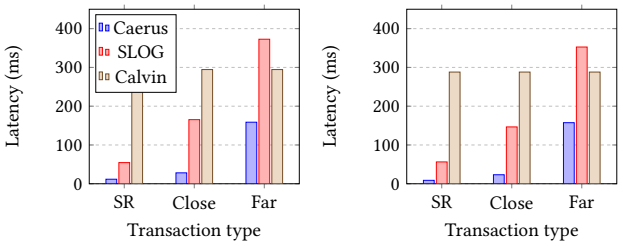
We study the performance of Caerus, SLOG, and Calvin on the TPC-C benchmark as the percentage of MW transactions that are MR transactions increases – 10% (Figure 5c), 50% (Figure 5a), and 100% (Figure 5b). We show the throughput graph for 50% MW transactions that are MR (we omit graphs for 10% and 100% as all 3 systems are bottlenecked by the scheduler resulting in similar throughput graphs). Caerus has significantly lower latency than Calvin and SLOG until saturation point (408 clients), while transaction throughput is comparable for all systems. As the number of clients increases, the gap resulting from Caerus' (lower) transaction latency and SLOG's latency also increases. This increased latency difference is due to SLOG becoming overloaded from the higher load on the deterministic scheduler placed by SLOG's MR transaction pieces.

Under all load conditions, Caerus incurs significantly lower transaction latency than the other systems. 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.

In Figure 6, we measure latency and throughput of Calvin, SLOG, and Caerus with the MovR benchmark as the client load on the systems increases. We run MovR twice: with P_f = .05 and P_c = .10 (Figure 6a), and also with P_f = .10 and P_c = .20 (Figure 6b) for 15 and 30 percent of MR transactions, respectively. While Calvin and Caerus 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 put on SLOG. After SLOG's throughput plateaus, its latency increases above the other systems. Calvin and Caerus become saturated at similar points, thereby showing similar latency and throughput behaviour.

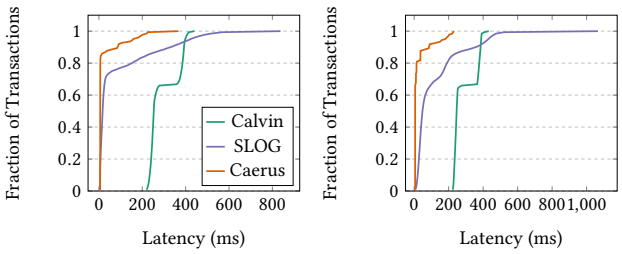
With 15% MR transactions, Caerus has 3.1× lower latency than SLOG and almost 18× lower latency than Calvin (Figure 6a). The latency wins by Caerus over SLOG increase to 4.7× with 30% MR transactions and to almost 11× over Calvin (Figure 6b).

SLOG has two drawbacks that limit its performance compared to Caerus. First, SLOG totally orders all multi-region (MR) transactions. This total ordering requirement means SLOG cannot exploit



(a) TPC-C **(b) MovR**

Figure 7: Latency breakdown by transaction type



(a) TPC-C **(b) MovR**

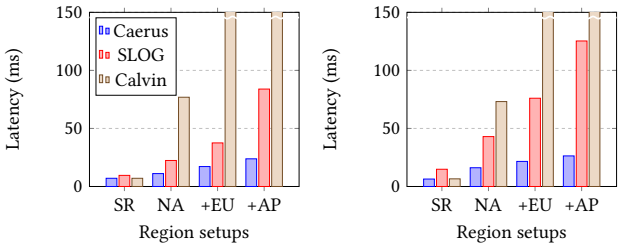
Figure 8: CDF of transaction latency

data locality when a transaction is MR, resulting in higher latency. Second, as SLOG breaks MR transactions into transaction pieces, these pieces add additional load on the deterministic scheduler, thereby increasing latency and lowering throughput. That is, when an MR transaction access 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 Caerus.

Caerus' across-the-board latency gains allow it to outperform SLOG and Calvin. Caerus' performance advantage is from not needing to totally order any transactions globally. Calvin's high latency comes from totally ordering *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 Caerus.

5.3.1 Latency Analysis. We study the latency of SR transactions, MR transactions between close regions, and MR transactions between far regions in Figure 7. We use 10% total MR transactions for TPC-C (Figure 7a) and 30% MR for MovR (Figure 7b). As each transaction is globally ordered through Paxos, Calvin's average transaction latency is the same for SR, close, and far transactions. Caerus 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 Caerus and SLOG is for MR transactions between close regions where Caerus outperforms SLOG by about 6× on TPC-C and 6.3× on MovR. These results demonstrate Caerus' ability to take advantage of the locality of data placement. Per Calvin's performance in Figure 7b, totally ordering transactions in geo-replicated systems can add large latency overheads.

To provide a more complete picture, we include empirical cumulative distribution functions (CDFs) of transaction latency for the 3 systems using TPC-C (Figure 8a) and MovR (Figure 8b) with 10% and 30% MR transactions, respectively, with 360 clients. We see that the CDFs of Caerus, for both TPC-C and MovR, are steeper than the CDF of SLOG. The divergence of the CDFs for the slowest 10% for TPC-C and 30% 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.



(a) TPC-C **(b) MovR**

Figure 9: Latency for different geo-replicated settings

5.3.2 Geo-Replication Analysis. In Figure 9, we look at the average latency of transactions when the geographical location of the replicas is changed. SR is a replicated deployment with all six replicas contained in a single region. NA represents the deployment of a single replica per region within North America. +EU adds to NA by holding a single replica per region with three regions in EU and three in NA. Finally, +AP adds to EU and NA by having a single replica per region with two regions in NA, two in EU, and two in AP (used in all other experiments). We use 360 clients, with 10% of all transactions being MR for TPC-C and 30% being MR for MovR.

As distance between regions decreases (and thus the WAN latency), the difference in average latency between the systems also decreases. Caerus continues to outperform SLOG and Calvin when the systems are geo-replicated. On TPCC, Caerus outperforms SLOG by 2.1× and Calvin by 13.1× when replicas are in EU and NA, and SLOG and Calvin by 2.1× and 6.9×, respectively, when replicas are in NA. For MovR, Caerus outperforms SLOG by 3.5× and Calvin by 10.0× when replicas are in EU and NA, and SLOG by 2.7× and Calvin by 4.5× when replicas are in NA. When data is not geo-replicated but instead replicated withing a single region, the performance of Calvin, SLOG, and Caerus is similar as the communication is performed over LAN, reducing much of the overhead from network round trips performed by Calvin and SLOG. Caerus' ordering protocol reduces the overhead for ordering transactions in the geo-replicated setting. Moreover, the similar transaction latency of Caerus, Calvin and SLOG for SR shows that the Caerus ordering protocol does not introduce extra overhead even when communication latency is small.

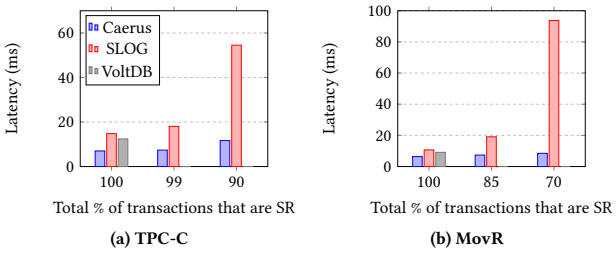


Figure 10: Single-region transaction latency

5.3.3 Single-Region Transactions. In Figure 10, we compare the latency of the commercial in-memory deterministic database VoltDB [51] deployed within a single Azure region with single-region transactions in geo-replicated Caerus and SLOG. VoltDB becomes saturated at a lower load than Caerus and SLOG, so we advantage VoltDB with a load set right before its saturation point. TPC-C is run with 10% multi-partition transactions, and MovR is run with 20% multi-partition transactions. When transactions are not distributed, the similar performance of VoltDB and Caerus shows the low overhead of the Caerus ordering protocol for single-region transactions. In particular, for TPC-C, even when 10% of transactions are cross-region (90% single-region), the average latency of single-region transactions in Caerus is the same as transactions in VoltDB. Furthermore, Caerus' ordering protocol incurs lower latency than VoltDB when the percentage of MR transactions is low (1.77× for 100% single-region and 1.68× for 99%). SLOG does not remain competitive with VoltDB due to transaction overheads. The general performance trends continue with MovR – single-region transactions in Caerus either outperform or perform similarly to transactions in VoltDB. When all transactions are single-region, SLOG also performs similarly to Caerus and VoltDB; however, single-region transactions in SLOG perform much worse when 30% and 15% of transactions are multi-region.

6 RELATED WORK

SLOG and Calvin have already been discussed so we do not include them in this section. Systems such as [36, 37, 55] use graph-based conflict tracking while leveraging SCCs to handle circular dependencies. These systems depend on at least one round of communication to agree on dependencies, with a second round when a quorum does not hold. Caerus does not need to reach a consensus on transaction conflicts/dependencies but instead uses a primary architecture to avoid unneeded WAN trips and thus reduce transaction latency.

There are several systems that use logs that are shared, deterministic, perform merging of multiple logs, or do graph-based dependency checking to resolve conflicts and sequence transactions [10, 11, 13]. However, none of them are geo-distributed/replicated.

Geo-replicated database systems such as [27, 37, 41, 53–55] can commit transactions with a single round of WAN communication only under certain conditions. However, these systems may require a second round of communication if these conditions do not hold. Similarly, consensus systems such as [32, 34, 36] can sequence transactions in as little as one WAN round trip but again may require a second round of WAN communication. In contrast, Caerus commits all transactions in a single WAN round trip.

Primo [30] coordinates distributed transaction execution within a single network round trip among partition leaders. Primo acquires exclusive locks for reads and relies on group commit among participating sites. When replicas and partition leaders are geo-distributed, transactions have to wait for at least 2 WAN round trips before transaction results are durable. Unlike Primo, Caerus performs transaction coordination and replication to ensure durability within a single WAN round trip.

WPaxos [3] considers low-latency ordering over geo-replicas by reducing the size of the replication quorum to include only close replicas. However, WPaxos must perform an object-stealing phase if all objects are not local to the leader while Caerus has no such requirement when sequencing transactions and replicating them.

Geo-distributed byzantine-tolerant systems [6–8, 19, 23] create groups in a hierarchical structure to reduce the complexity of coordination performed over WAN. Unlike Caerus, [6–8, 19, 23] do not perform coordination within a single WAN round trip. The Blockplane communication framework [39] does not provide cross-group coordination. ResilientDB [23] will wait for the maximum pair-wise latency between all regions before executing transactions.

Between the original work [24] of this paper and this publication, Detock [40] appeared. We include a comparison of these two works that were done independently and concurrently. Like Caerus, Detock uses a single WAN round trip of messaging to execute geo-distributed multi-region transactions using graph-based techniques for resolving transaction conflicts. While Caerus outputs a sequence of transactions that can be executed by any deterministic scheduler (e.g., Calvin [47], QStore [42]), Detock integrates sequencing, concurrency control and execution, limiting scheduler implementation choices. Caerus processes partial sequences of transactions in parallel, as opposed to Detock that merges regional logs into a global log. Caerus continuously executes its sequencing algorithm as operations arrive, minimizing processing delays. By contrast, Detock executes periodically based on heuristic estimates of network delays, similar to [48]. Unlike Detock, Caerus includes a recovery protocol for fault tolerance and high availability for region failure.

7 CONCLUSION

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

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We are grateful to University of Waterloo and Microsoft for providing Azure cloud credits in support of this research. This project was supported by funding from NSERC, WHJIL, CFI, and ORF.

REFERENCES

[1] Daniel J. Abadi and Jose M. Faleiro. 2018. An Overview of Deterministic Database Systems. *Commun. ACM* 61, 9 (aug 2018), 78–88. https://doi.org/10.1145/3181853

[2] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive Dynamic Mastering for Replicated Systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1381–1392. https://doi.org/10.1109/ICDE48307.2020.00123

[3] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Trans. Parallel Distrib. Syst.* 31, 1 (jan 2020), 211–223. https://doi.org/10.1109/TPDS.2019.2929793

[4] American Airlines. 2022. *American Airlines and Microsoft Partnership Takes Flight to Create a Smoother Travel Experience for Customers and Better Technology Tools for Team Members.* Retrieved November 17, 2023 from 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

[5] Amazon. 2021. *Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region.* Retrieved November 17, 2023 from https://aws.amazon.com/message/12721/

[6] Yair Amir, Claudiu B. Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2006. Scaling Byzantine Fault-Tolerant Replication toWide Area Networks. *International Conference on Dependable Systems and Networks (DSN'06)* (2006), 105–114. https://api.semanticscholar.org/CorpusID:4625677

[7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 76–88. https://doi.org/10.1145/3448016.3452807

[8] Mohammad Javad Amiri, Daniel Shu, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. 2023. Ziziphus: Scalable Data Management Across Byzantine Edge Servers. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 490–502. https://doi.org/10.1109/ICDE55515.2023.00044

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[10] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *Conference on Innovative Data Systems Research*.

[11] Philip A. Bernstein, Colin W. Reid, Ming Wu, and Xinhao Yuan. 2011. Optimistic Concurrency Control by Melding Trees. *Proc. VLDB Endow.* 4, 11 (aug 2011), 944–955. https://doi.org/10.14778/3402707.3402732

[12] CalvinDB. 2019. GitHub. Retrieved Febuary 20, 2022 from https://github.com/kunrenyale/calvindb

[13] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. 2014. ConfluxDB. *Proceedings of the VLDB Endowment* 7 (07 2014), 947–958. https://doi.org/10.14778/2732967.2732970

[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug 2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[15] 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. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.

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

[17] Charlie Custer. 2022. *How Devsisters made uptime a core requirement for their games.* Retrieved November 17, 2023 from www.cockroachlabs.com/blog/customer-devsisters-10-million-downloads-cockroachdb/

[18] Charlie Custer. 2023. *Performance goals for mission-critical workloads.* Retrieved November 17, 2023 from https://www.cockroachlabs.com/blog/performance-goals-for-mission-critical-workloads/

[19] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 123–140. https://doi.org/10.1145/3299869.3319889

[20] Murat Demirbas. 2022. *Amazon Aurora: Design Considerations + On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes.* Retrieved November 17, 2023 from http://muratbuffalo.blogspot.com/2022/03/amazon-aurora-design-considerations-and.html?m=1

[21] Marisa Fernandez. 2018. *Prime Day woes might have cost Amazon* $72m-99m$ *in sales.* Retrieved November 17, 2023 from https://www.axios.com/2018/07/18/prime-day-woes-might-have-cost-amazon-from-72-99-million

[22] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. 2020. ChronoCache: Predictive and Adaptive Mid-Tier Query Result Caching. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2391–2406. https://doi.org/10.1145/3318464.3380593

[23] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (feb 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[24] Joshua Thomas Hildred. 2023 (May). *Efficient Geo-Distributed Transaction Processing.* Master's thesis. University of Waterloo. https://uwspace.uwaterloo.ca/handle/10012/19516

[25] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain (LIPIcs)*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.), Vol. 70. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:14. https://doi.org/10.4230/LIPICS.OPODIS.2016.25

[26] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

[27] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[28] Cockroach Labs. [n.d.]. *Expanding across the globe: Banking and payments at hyperscale.* Retrieved November 14, 2023 from https://www.cockroachlabs.com/customers/pismo/

[29] Cockroach Labs. [n.d.]. *MovR.* Retrieved November 17, 2023 from https://www.cockroachlabs.com/docs/stable/movr.html

[30] Z. Lai, H. Fan, W. Zhou, Z. Ma, X. Peng, F. Li, and E. Lo. 2023. Knock Out 2PC with Practicality Intact: a High-performance and General Distributed Transaction Protocol. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2317–2331. https://doi.org/10.1109/ICDE55515.2023.00179

[31] Leslie Lamport. 2001. Paxos Made Simple.

[32] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (2006), 79–103.

[33] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-Latency Multi-Datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.* 6, 9 (Jul 2013), 661–672. https://doi.org/10.14778/2536360.2536366

[34] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 369–384.

[35] Hussein Al Kazwini Diana Richards Kumud Dwivedi Mahesh Nayak Cheryl McGuire Michael Bender, Allen Sudbring. 2023. *Azure network round-trip latency statistics.* Retrieved November 17, 2023 from https://learn.microsoft.com/en-us/azure/networking/azure-network-latency

[36] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[37] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 517–532. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

[38] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1221–1236. https://doi.org/10.1145/3183713.3196928

[39] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 124–135. https://doi.org/10.1109/ICDE.2019.00020

[40] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (20 June 2023), 1–27.

[41] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 47–64. https://www.usenix.org/conference/nsdi19/presentation/park

[42] Thamir M. Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *International Conference on Extending Database Technology*.

[43] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (Jul 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[44] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (jun 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[45] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[46] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1–2 (Sep 2010), 70–80. https://doi.org/10.14778/1920841.1920855

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

[48] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 613–632. https://www.usenix.org/conference/nsdi21/presentation/tollman

[49] TPC. [n.d.]. *TPC-C*. Retrieved November 17, 2023 from https://www.tpc.org/tpcc/

[50] 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. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2312–2325. https://doi.org/10.1145/3514221.3526053

[51] VoltDB. 2023. GitHub. Retrieved July 12, 2023 from https://github.com/VoltDB/voltdb

[52] Wyatt Wenzel. 2023. *Achieving low latencies and low emissions at the edge for Climatiq's carbon calculation API.* Retrieved November 17, 2023 from https://fauna.com/blog/low-latencies-and-emissions-at-the-edge-for-climatiqs-carbon-calculation-api#about-climatiq

[53] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 231–243. https://doi.org/10.1145/3183713.3196912

[54] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[55] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. 2022. Starry: Multi-Master Transaction Processing on Semi-Leader Architecture. *Proc. VLDB Endow.* 16, 1 (Sep 2022), 77–89. https://doi.org/10.14778/3561261.3561268