Detock: High Performance Multi-region Transactions at Scale

CUONG D. T. NGUYEN, University of Maryland, USA
JOHANN K. MILLER, University of Maryland, USA
DANIEL J. ABADI, University of Maryland, USA

Many globally distributed data stores need to replicate data across large geographic distances. Since synchronously replicating data across such distances is slow, those systems with high consistency requirements often geo-partition data and direct all linearizable requests to the primary region of the accessed data. This significantly improves performance for workloads where most transactions access data close to where they originate from. However, supporting serializable multi-geo-partition transactions is a challenge, and they often degrade the performance of the whole system. This becomes even more challenging when they conflict with single-partition requests, where optimistic protocols lead to high numbers of aborts, and pessimistic protocols lead to high numbers of distributed deadlocks. In this paper, we describe the design of concurrency control and deadlock resolution protocols, built within a practical, complete implementation of a geographically replicated database system called Detock, that enables processing strictly-serializable multi-region transactions with near-zero performance degradation at extremely high conflict and order of magnitude higher throughput relative to state-of-the art geo-replication approaches, while improving latency by up to a factor of 5.

CCS Concepts: • **Information systems** → **Distributed database transactions**; **Deadlocks**; *Database transaction processing*.

Additional Key Words and Phrases: multi-region database, deterministic database, deadlock resolution

ACM Reference Format:
Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proc. ACM Manag. Data* 1, 2, Article 148 (June 2023), 27 pages. https://doi.org/10.1145/3589293

1 INTRODUCTION

Modern data stores typically replicate data for improved availability, durability, read throughput, and/or latency. Data stores designed for global applications typically replicate data across large geographic distances, which further improves robustness to region failure, and can allow reads to occur locally to an application client.

Data stores that allow replicas to temporarily diverge in a manner visible to the client are termed *weakly consistent*, and those for which such divergence does not exist or is kept invisible to the client are termed *strongly consistent*. The gold standard for strongly consistent guarantees in the context of transactional systems is strict serializability [11, 12, 24, 42], which ensures that transactions submitted after earlier transactions complete never observe a state prior to those completed transactions, even if they are processed on a different replica.

There are two common approaches for implementing strict serializability in practice. The simplest approach is to have a primary copy of each data item. All writes are performed by that primary

Authors' addresses: Cuong D. T. Nguyen, ctring@umd.edu, University of Maryland, College Park, MD, USA; Johann K. Miller, jkmiller@umd.edu, University of Maryland, College Park, MD, USA; Daniel J. Abadi, abadi@umd.edu, University of Maryland, College Park, MD, USA.

2836-6573/2023/6-ART148
https://doi.org/10.1145/3589293

copy, and strongly consistent reads are directed either to that primary copy or other copies that are replicated to synchronously from that primary copy [18, 33, 45, 46]. The second approach allows writes to occur at any replica, but performs a consensus protocol to avoid replica divergence [9, 17, 52, 54, 58, 60].

Both of these approaches can support geographic partitioning of data for improved performance. In the first approach, the primary copy of different data items can be stored in different regions [15, 18, 33]. In the second approach, separate consensus groups can be formed in different regions [49, 52]. Either way, geo-partitioning decreases latency of transactions that initiate near the region of their accessed data and can therefore increase the overall performance of a workload if such transactions are prevalent.

Unfortunately, even for workloads where such transactions are common, there still exist some transactions which must access data in more than one partition. Such transactions necessarily require coordination across partitions (and therefore across geographic regions) to ensure strict serializability, and this increases latency. The bigger problem, however, is that when they conflict with single-partition transactions, they become hard to complete: optimistic concurrency control approaches result in extremely high abort rates under high contention, and pessimistic approaches result in high amounts of deadlock. Even single-region transactions can end up getting involved in deadlocks or OCC aborting because of the presence of these slow multi-region transactions. Therefore, it is extraordinarily difficult to achieve high throughput under high contention and a non-trivial number of multi-region transactions.

One approach to avoiding these issues is to use deadlock avoidance techniques to enable pessimistic concurrency, providing high-throughput transaction processing under high contention. For example, the work on SLOG creates separate consensus groups per region, geographically partitions data across these regions, and runs every multi-region transaction through a global ordering mechanism to completely avoid deadlock [49]. However, this approach adds the latency of the global ordering layer **in addition** to the latency required for coordination during normal processing of strictly serializable multi-region transactions, further impacting the performance of conflicting single-region transactions.

Instead, in this paper, we present a new graph-based concurrency control protocol that enables multi-region transactions (in addition to single-region transactions) to be scheduled deterministically at each region such that all regions involved in processing a transaction will construct the same graph independently and process transactions completely without cross-region coordination after receiving all parts of the transaction. The graphs constructed by each region are formed based on conflicting accesses by different transactions, and indeed may contain deadlocks. However, since each region constructs the same graph, deadlocks can be resolved by dynamically reordering accesses by deadlocked transactions to resolve the deadlock **deterministically** without ever having to resort to aborting transactions and without having to communicate this reordering with other regions.

Nevertheless, high network delays between regions can cause the size and number of deadlocks to grow unbounded. We therefore implement a practical version of this algorithm within a new system called Detock that annotates transactions with real-time based timestamps, which are used to strategically schedule transactions to reduce the probability of deadlock. Detock also implements a novel protocol for migrating data to other geo-partitions using a simpler approach than used in previous work [49].

When comparing Detock to several alternative state-of-the-art systems that support geographic partitioning such as SLOG and CockroachDB, we find that Detock can lessen throughput reductions caused by multi-region transactions under high conflict by several orders of magnitude, while simultaneously reducing latency by avoiding unnecessarily cross-region coordination.

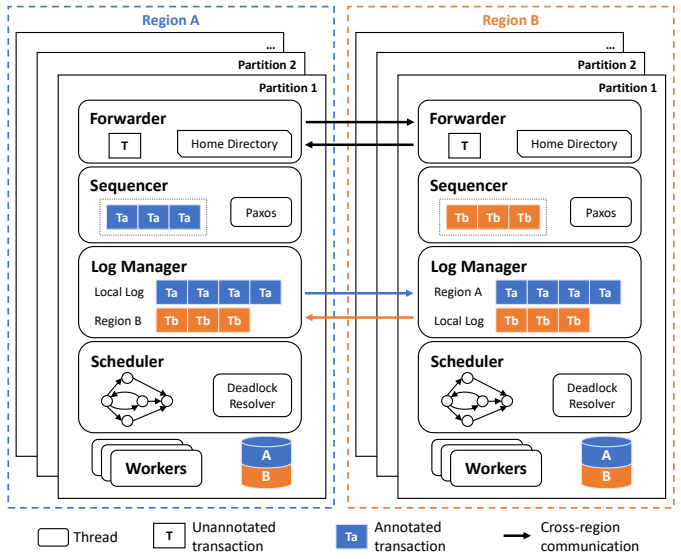


Fig. 1. Architecture of Detock

This paper thus makes the following contributions:

- A concurrency control protocol for geo-distributed transactions.
- An abort-free deterministic deadlock resolution protocol that enables replicas to resolve deadlocks independently.
- A practical implementation of these protocols within an open source system that leverages real-time-based timestamps.
- Experiments that investigate the impact of these practical optimizations and compares to two state of the art systems.
- A novel, simple, data migration protocol.

2 SYSTEM ARCHITECTURE

Detock is a geo-partitioned database system, and uses a similar high-level architecture to recent state-of-the-art geo-partitioned database systems, while introducing novel approaches to concurrency control, deadlock resolution, and data migration. This section overviews the basic architectural approach that Detock shares with other geo-partitioned database systems — most notably SLOG and CockroachDB — and we defer the discussion of the unique aspects of Detock's design to the following section.

The system is deployed across multiple geographic regions. A *region* consists of servers connected via a low-latency network. These servers typically reside within a single data-center or multiple data-centers that are in close proximity with each other. Each item is assigned to exactly one geographic region. This is called a *home region* in both SLOG and CockroachDB. The identifier of the home region for a data item is stored in the header of that data item.

A region may store *local data* for which it is designated as the home region, and *remote data* which are materialized by replaying logs asynchronously replicated from other regions. Data is also partitioned locally within a region independently of home status: each partition might contain a mix of local and remote data.

Similar to SLOG, Detock relies on deterministic transaction execution to substantially reduce cross-region coordination. Fig. 1 shows the architecture of this style of deterministic system in a deployment over two regions A and B. Clients send transactions to their closest region. The first server that receives a transaction becomes its *coordinator*, which first resolves non-deterministic commands in the transaction (e.g. `random()` and `time()`), then attempts to extract its read/write set. When this is not possible via static analysis [25], the OLLP protocol is used, which obtains an initial estimate of the transaction's read/write set via a reconnaissance query [54]. Each region maintains a distributed index called a *Home Directory* that contains the cached value of the current home for each known data item. The Forwarder of the coordinator uses this index to augment the read/write set with the home information of every data item. It then annotates the transaction with this augmented read/write set and forwards the transaction to its home region(s).

We denote in Fig. 1 annotated transactions housed by region A as Ta, and by region B as Tb. Once these transactions reach their home regions, they are put into batches and inserted into a Paxos-maintained *local log* by the Sequencers. This log is synchronously replicated within a region to tolerate failure of individual servers, and optionally to nearby regions to increase availability during (rare) failures of an entire region.

A region can deterministically replay a local log from any other region R to obtain the state of R's local data. Therefore, persisting the local logs is sufficient for durability, and replication is performed by shipping these local logs. While it is not required for a region to hold any remote data, having a possibly stale copy of the remote data allows local snapshot reads of data from other regions and makes executing multi-home transactions (including the home-movement transactions in Section 4) faster. To this end, regions in Detock and SLOG **asynchronously** exchange their local logs to each other, so each region eventually receives and replays the complete local log from every other region, as can be seen in the Log Managers of both regions in Fig. 1.

To replay the logs, a Scheduler constructs a dependency graph for the transactions in the logs with the help of the Deadlock Resolver (Section 3.2), and schedules them to be executed by the Workers.

If all data accessed by a transaction belong to a single region, it is called a *single-home* transaction; otherwise, it is *multi-home*. Multi-home transactions insert records into the local logs of each home region for the data accessed by that transaction. As mentioned above, SLOG globally orders multi-home transactions to avoid inconsistently ordering them across regions (e.g. T1 before T2 at region 1, but T2 before T1 at region 2) which could result in serializability violations, OCC aborts, or deadlock [49]. Detock eliminates this global ordering, but must therefore deal with the problems that arise from inconsistent ordering (discussed in the next section). By eliminating multi-home transaction ordering, Detock is able to guarantee that each transaction, regardless of its type, only needs **a single-round trip** from the initiating region to the participating regions: the initiating region sends the multi-home transaction to each region which houses data that it accesses, waits to receive the local log records back from those regions through which it can derive the state at those regions over which that transaction must run, and can then process that transaction to completion locally. Every other region, including the ones that write local data, see the same local logs and also process that transaction locally.

3 TRANSACTION PROCESSING

When a new transaction arrives at the system, its coordinator invokes the function **`StartTxn`** shown in Algorithm 1. Although most deterministic systems such as SLOG and Calvin do not require assigning a globally unique identifier to a transaction upon arrival, most other highly consistent ACID-compliant distributed systems — including CockroachDB and Spanner — give transactions globally unique identifiers. Detock takes the latter approach despite being a deterministic system

since the identifier will be used in the concurrency control protocol. The globally unique ID is generated by concatenating (in binary) a local transaction counter with a globally unique ID statically assigned to the coordinator's server (Line 2). The Home Directory is used to find the cached values of the home regions for all data items in the read/write set (Line 3-10).

Algorithm 1: Starting a new transaction

```
1  function StartTxn(txn)
2      txn.id = new globally unique ID
3      if txn.isHomeMovement then                                  /* see Section 4 */
4          key = txn.movedKey
5          txn.oldHome = HomeDirectory(key)
6          txn.homeInfo = {(key, txn.oldHome), (key, txn.newHome)}
7      else
8          txn.homeInfo = ∅                                        /* set of (key, region) pairs */
9          foreach key in txn.readSet ⋃ txn.writeSet do
10             Add HomeDirectory(key) to txn.homeinfo
11     regions = unique regions in txn.homeInfo
12     txn.isMultiHome = size of regions is larger than 1
13     if txn.isMultiHome then
14         /* oneway[r] is estimated one-way network delay to region r */
15         maxOneWay = Max(oneway[r] : ∀r in regions)
16         txn.timestamp = Now() + maxOneWay + overshoot
17     Call AppendLocalLog(txn) for every region in regions
```

The number of unique home regions retrieved is used to determine whether the transaction is single-home or not (Line 11-12). Incorrect read/write sets (from the OLLP protocol) or home information (from stale values in the Home Directory) are deterministically detected later during execution and cause the transaction to abort and restart. However, these restarts are expected to be uncommon in practice: OLLP aborts only occur when the access set of data depends on the current state of the database [50, 54], while home information aborts only occur for a short period of time after data is rehoused in a different region. The transaction is then forwarded to the participating regions (Line 17). [The timestamp assigned in Line 16 is an optimization that is described in Section 3.2.]

3.1 Single-home Transactions

The initial steps of transaction processing of single-home transactions Detock are identical to those of SLOG: When a single-home transaction reaches a node at its presumed home region, the Sequencer of that node runs the code in Algorithm 2, which puts the transaction into an in-memory batch along with other concurrent transactions that arrive at the same node (Line 8). The size of the batch window is configurable, and defaults to 5ms. The Sequencer then appends the batch to the region's local input log via Paxos (Line 10). After this point, the transaction is durably logged for recovery. The position in the Paxos log, along with the contents of the batch is asynchronously replicated from that region to every other region, such that every region eventually receives the complete set of ordered batches from every other region (Line 11). Detock and SLOG also support synchronous replication to near-by regions for improved robustness to region-failure.

At each region, the Paxos logs from all regions (including its own) are interleaved arbitrarily by the Log Managers to form that region's view of the *global log* (Line 14-19). Each region may

Algorithm 2: Appending transactions to the logs

```
1  function AppendLocalLog(txn)
2      localTxn = txn
3      if txn.isMultiHome then
4          Sleep until Now() ≥ txn.timestamp
5          exclude = keys in txn.homeInfo where region ≠ curRegion
6          localTxn.readSet = localTxn.readSet \ exclude
7          localTxn.writeSet = localTxn.writeSet \ exclude
8      Append localTxn to batch
9  upon batch is ready do
10     pos = Append batch to local Paxos log and get its position
11     Asynchronously call AppendGlobalLog(curRegion, pos, batch) for every region
12     batch = ∅
13 function AppendGlobalLog(reg, pos, batch)
14     localLogs[reg, pos] = batch
15     lastPos = position of the last batch in localLogs[reg] that was added to globalLog
16     while localLogs[reg, lastPos + 1] ≠ null do
17         foreach txn in localLogs[reg, lastPos + 1] do
18             Append txn to globalLog
19         lastPos = lastPos + 1
```

interleave transactions from different local logs into the global log in different ways; however, if transaction B is after A in any region's local log, B will be after A in every region's global log, since logs records from the same region are numbered by that region and never reordered.

From this point forward, Detock's processing of single-home transactions differs from SLOG. In both systems, each region executes all transactions found in its global log in parallel, but in a manner equivalent to if they had been executed sequentially in log order. However, SLOG uses a locking based mechanism to achieve this, while Detock uses an approach based on dependency graphs in order to facilitate deadlock detection and resolution. Algorithm 3 presents the pseudocode of the Scheduler where the dependency graph is constructed (Line 2-6).

Definition 3.1. Two transactions T_i and T_j are said to *conflict* on a tuple (d, r), where d is a data item and r is a region, if both transactions access d, at least one of the transactions writes to d, and both of them expect r to be d's home region.

A dependency graph is a directed graph where vertices correspond to transactions, and an edge (T_i, T_j) exists if and only if:

- T_i is at a position earlier than T_j in the global log[1], and
- There exists a tuple (d, r) such that both T_i and T_j conflict on (d, r), and there does NOT exist a transaction T_k such that T_k is between T_i and T_j in the global log and T_k conflicts with both T_i and T_j on (d, r).

Despite each region having slightly different versions of the global log, they are all guaranteed to (eventually) construct the same dependency graph, since the definition of conflict prevents conflicts across regions, and thus the order of interleaving logs from different regions does not impact the ultimate structure of the dependency graph. Therefore, each region can process the transactions

[1]Traditionally, a wait-for graph is constructed with directed edges pointing away from the waiting transaction. However, reversing the edges simplifies our implementation.

Algorithm 3: Scheduling transactions for execution

```
1  upon a new transaction txn is appended to globalLog do
2      newEdges = ∅
3      foreach (k, r) in txn.homeInfo do
4          prev = latest transaction that conflicts with txn on (k, r)
5          if prev ≠ null then  Add (prev.id, txn.id) to newEdges
6      Add txn.id and newEdges to dependency graph G
7      Broadcast txn.id and newEdges to all local partitions
8  upon a transaction txn becomes ready; in a Worker thread do
9      if not txn.isHomeMovement then
10         foreach (key, region) in txn.homeInfo do
11             if region ≠ storage.getHome(key) then  Abort txn
12     else if txn.oldHome ≠ storage.getHome(txn.movedKey) then
13         Abort txn
14     Execute the code in txn
15     Remove txn.id and its associated edges from G
16 /* Called periodically in a background thread */
17 function FindAndResolveDeadlocks()
18     G' = FindStableSubgraph(G)                    /* see Section 3.2 */
19     foreach scc in FindSCCs(G') do
20         Deterministically serialize scc in G
```

in its global log independently, without any communication with other regions, and arrive at the same final state.

If all transactions are single-home, the dependency graph constructed from the global log will be a directed acyclic graph (DAG). This is because edges can only arise among transactions within a region, which are strictly ordered. Therefore, processing of the transactions follows the topology order of that graph. A finished transaction is removed from the graph along with its outgoing edges. A transaction is executed only when there are no more incoming edges pointing to it. Different partitions in the same replica may only see partial views of the DAG. However, since there is no cycle in a DAG (because all transactions are single-home), the transactions can be processed without a distributed deadlock detection mechanism.

Transactions accessing multiple partitions in the same replica follow a deterministic execution protocol similar to Calvin [54] and thus do not require two-phase commit. Unlike Calvin, before accessing a data item, a transaction needs to check whether the home region identifier stored alongside the data item matches with the expected home region retrieved previously from the Home Directory. If they do not match, the transaction must be aborted and restarted (Line 9-13). Since all regions eventually receive the same set of logs and the transactions are processed deterministically, the regions apply the same sequence of updates to each data item. The home region identifier, being part of a data item, is thus updated at the same point in that update sequence. Consequently, all regions make the same decision as to whether to abort a transaction or not based on the comparison between the actual home region identifier stored alongside the data item and the assumed home region identifier stored in the transaction. Once a transaction finishes its execution, the scheduler removes it from the graph and schedules transactions that become ready as a result of this removal (Line 14-15).

Fig. 2 shows an example of single-home transaction processing. There are 3 regions: US, EU, and AP, each of which holds a complete copy of the database. In this example, there is only one partition

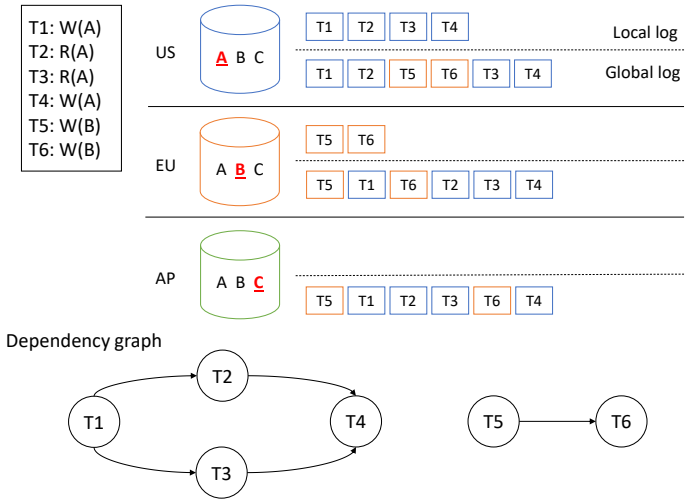


Fig. 2. Single-home transaction processing

and one replica in each region. The local data of each region is shown in red and underlined. The 4 transactions T1-T4 access data item A so they are ordered in the local log of the US region. Transactions T5 and T6 access data item B so they are ordered in the EU region. Each region eventually obtains the local log from every other region and interleaves them to form its view of the global log. As noted above, the generated dependency graphs for the regions eventually become identical, despite each region having a different version of the global log. The associated home regions of the data items are not changed in this example, so conflict of two transactions is determined based solely on their read and write operations on the data items, shown on the top left of the figure.

3.2 Multi-home Transactions

When a multi-home transaction reaches a participating region, it follows a different protocol than that of a single-home transaction. Each region uses the home information stored in the transaction to generate a special kind of transaction called a GraphPlacementTxn that contains a list of the keys from the original multi-home transaction that are local to the current region (Algorithm 2, Line 3-7). One GraphPlacementTxn per transaction, designated by the coordinator, also contains the original code for that transaction.

GraphPlacementTxns are initially treated like single-home transactions: they are put into the local logs at their home regions, make their way to the global logs through local log replication, and finally get added to the dependency graphs at every region.

For each transaction, T, each region will eventually receive GraphPlacementTxns from each region that the coordinator expected to house relevant data. The first GraphPlacementTxn for T that is placed in a region's global log causes a new vertex representing T to be created in that region's graph. Subsequent GraphPlacementTxns of T share this same vertex. Edges created by these GraphPlacementTxns are added to that vertex.

GraphPlacementTxns establish an order between multi-home and single-home transactions at the region that generated the GraphPlacementTxn. However, they do not globally order multi-home transactions, since two different regions may generate GraphPlacementTxns for a set of multi-home

transactions in different orders. There is thus a concern that the generated graph may contain cycles, which would lead to deadlock during processing.

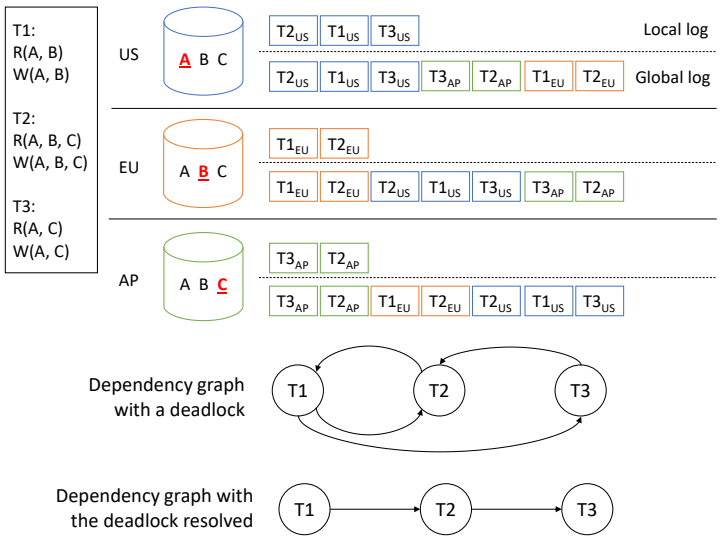


Fig. 3. Deadlocks from multi-home transactions

Fig. 3 shows an example scenario that would lead to deadlock, using the same setup as Fig. 2. Three multi-home transactions arrive at the system: T1 and T2 both access data item A and B, so they are sent to the US and EU regions. T3 accesses data item A and C, so it is sent to the US and AP regions. All accesses are read-modify-write operations. Each region generates the GraphPlacementTxns for the multi-home transactions and places them in its local log; however, the order they are placed differs across regions. Every region eventually receives all local logs and constructs the dependency graph. The deadlocks manifest themselves as cycles in the dependency graph. In this case, all three transactions are in a deadlock.

In order for the transactions to progress, the deadlocks must be eliminated, either by aborting transactions or modifying the dependencies such that the graph is free of cycles. We chose the latter approach because aborting and restarting transactions increase latency of those restarted transactions. However, a key constraint is that this modification must be deterministic: every region must independently make the same decision on how to resolve the deadlock without runtime communication across regions.

One dependency modification strategy is to serialize the transactions following the order of their IDs (see Section 3.1 for how IDs are generated). For example, the dependencies in the graph in Fig. 3 can be changed such that the processing order of the transactions is T1, T2, and T3, as shown at the bottom of the figure.

However, making this change deterministically is more complicated than it initially appears: performing the deadlock resolution as soon as a cycle is detected may cause divergence. For example, if a server in the EU region in Fig. 3 resolved the deadlock between T1 and T2 as soon as it saw the first four log entries in the global log, only a single edge (T1, T2) would remain between T1 and T2. When the rest of the log arrived, the edges (T1, T3) and (T3, T2) were added to the graph. This final graph is a DAG whose topological order is T1, T3, and T2, which is different from what

the order would have been if the deadlock was resolved only after receiving all the log entries. Therefore, both the timing for when to run deadlock resolution along with the resolution itself must be deterministic. This is complicated by the fact that each region interleaves local log records into its global log differently and waiting too long to resolve deadlocks increases latency.

Deterministic deadlock resolution (DDR). We give an intuition for our deadlock resolution algorithm by considering its naïve version over a finite set of transactions. Each region waits until all transactions arrive then constructs a *condensation* of the dependency graph. A condensation of a directed graph $\mathcal{G}$ is formed by contracting each strongly connected components (SCC) into a super vertex, and adding a directed edge between two super vertices U and V if there is a directed edge in $\mathcal{G}$ that starts in U and ends in V (e.g. Fig. 4a). A condensation is a DAG since it does not have any SCC over its super vertices with a size larger than one. Therefore, we can find a topological order on the condensation. We additionally impose an order agreed upon by every region on the vertices within each super vertex (e.g. by their IDs). Determinism across regions can then be achieved by executing the transactions following both of the these orders.

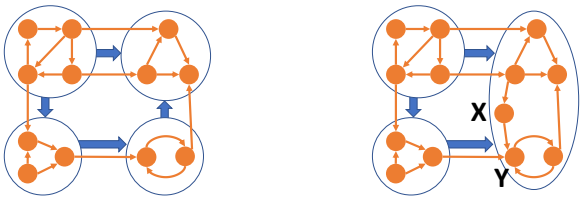


Fig. 4. Example dependency graphs (in orange) and their condensations (in blue)

In reality, it is extremely inefficient or impossible to wait for all transactions to arrive as the set of transactions may not be finite. On the other hand, running the algorithm at arbitrary time may cause the regions to see different condensations. For example, another region will observe a different condensation as shown in Fig. 4b if it runs the algorithm after the transaction X arrives, merging two of the SCCs together. To avoid this problem, the DDR algorithm identifies a *stable subgraph* then finds and executes the SCCs only within this subgraph. These SCCs are guaranteed to never mutate as new transactions arrive, thus ensuring convergence across the regions.

For clarity, we initially assume that the data at each region is not partitioned and will remove this assumption shortly. For every vertex corresponding a multi-home transaction T in the dependency graph, let $GP_{total}(T)$ be the total number of GraphPlacementTxns generated for T, a counter $GP(T)$ is associated with T to keep track of the number of GraphPlacementTxns of T that have been added to the graph so far. We define two types of vertices:

- A *complete vertex* T is either a single-home transaction or a multi-home transaction with $GP(T)$ equal to $GP_{total}(T)$.
- A *stable vertex* T is a *complete vertex* and there does not exist a path going from an incomplete vertex to T.

For example, before X is added to the graph in Fig. 4b, Y was not a complete vertex because the edge (X, Y) being missing implies at least one GraphPlacementTxn of Y was not added to the graph. As a result, any vertex that Y had a path to was not stable.

Let $\mathcal{G}(\mathcal{T}, \mathcal{E})$ be a dependency graph comprised of a vertex set $\mathcal{T}$ and an edge set $\mathcal{E}$. The *stable subgraph* of $\mathcal{G}$ is the graph $\mathcal{G}'(\mathcal{T}', \mathcal{E}')$ such that $\mathcal{T}'$ is the subset of $\mathcal{T}$ that contains all stable vertices and $\mathcal{E}'$ is the subset of $\mathcal{E}$ that contains all edges whose both incident vertices are in $\mathcal{T}'$. The stable

vertices set $\mathcal{T}'$ can be found with breadth-first search, starting from the set of incomplete vertices; any traversed vertices (including the starting vertices) are marked as unstable; the remaining untraversed vertices are stable vertices.

The DDR algorithm finds all SCCs in the stable subgraph $\mathcal{G}'$; for each found SCC, it removes all its edges and adds a new simple chain of edges between the vertices found within that SCC ordered by their transaction IDs. After running this algorithm, the stable subgraph $\mathcal{G}'$ will become a DAG and, as an optimization, can be ignored in subsequent runs of the algorithm. We run this algorithm in a background thread (Algorithm 3, Line 17-20) at a configurable interval, which we set to 40 ms. This thread builds its own copy of the graph and communicates with the Scheduler via a message queue to avoid access conflict on an otherwise shared graph.

When the data is partitioned within a replica, each partition may only have a partial view of the dependency graph. Therefore, we make two modifications to the above algorithm. First, each partition will periodically broadcast its view of the graph to all other partitions (Algorithm 3, Line 7). Second, for every vertex T, let $Part_{total}(T)$ be the number of partitions participating in T and $Part(T)$ be the number of partitions participating in T that have sent their views that included T to the current server; for the vertex T to be considered complete, regardless of whether it is single-home or multi-home, it must satisfy that $Part(T)$ is equal to $Part_{total}(T)$, in addition to the conditions stated above.

3.3 Proof of Correctness

We now prove that Detock achieves determinism and strict serializability. We use "component" as short for "strongly connected component". To simplify the proof, we slightly modify the DDR and transaction execution algorithms. After reordering the vertices within a component C, the deadlock resolver adds a *virtual edge* visible only to the deadlock resolver from the last vertex to the first vertex of the series, so that the vertices in C continue to form a strongly connected component after reordering. The transactions are executed following the non-virtual edges. After a transaction's execution, it is marked as executed instead of being removed from the graph, and its outgoing edges become virtual edges.

Definition 3.2. A region R *determines* a vertex T to be in a component C at a prefix p of the global log in R if and only if the DDR algorithm computes T to be in C in the stable subgraph of the graph constructed from p.

Lemma 3.3. *For any two regions R_A and R_B and two conflicting transactions T_i and T_j, if R_A determines T_i and T_j to be in the same component at some prefix p_A of the global log in R_A, then if R_B determines T_i and T_j to be in some component(s) at some prefix p_B of the global log in R_B, it also determines that T_i and T_j to be in the same component.*

Proof. (By contradiction) Assume that R_B determines T_i and T_j to be in two different components C_i and C_j, respectively, at p_B. In R_B, since T_i and T_j are determined to be in two different components, there does not exist a path either from T_i to T_j or from T_j to T_i in the graph constructed from p_B. Without loss of generality, we assume that the path from T_i to T_j does not exist.

In R_A, since T_i and T_j are determined to be in the same component, there exists a path ρ from T_i to T_j in the graph at p_A.

While ρ does not exist in R_B at p_B, there always exists in R_B at p_B a subpath of ρ ending in T_j (the subpath containing only T_j is one such subpath). Let T_k be the starting vertex of the longest such subpath. T_k cannot be T_i because the whole path ρ does not exists in R_B at p_B. Hence, there exists a vertex T_h that is immediately precedes T_k on ρ.

The edge (T_h, T_k) does not exist in R_B at p_B because the subpath from T_k to T_j is already the longest. Per Detock's protocol, R_B will eventually construct the edge (T_h, T_k) at some extension

of p_B. This means T_k is an an incomplete vertex at p_B, which makes T_j an unstable vertex at p_B because there is a path from T_k to T_j. This is a contradiction because R_B cannot determine T_j to be in the component C_j at p_B if T_j is still not stable. Therefore, R_B must determine T_i and T_j to be in the same component at p_B. □

Lemma 3.4. *If a region R determines a vertex T to be in a component C at some global log prefix p in R, then R determines T to be in C at any extension of p.*

Proof. Let $\mathcal{A}_p$ be the set of vertices each of which has a path leading to T ($\mathcal{A}_p$ contains T) in the graph constructed from p. Let p' be some extension of p.

$|\mathcal{A}_{p'}| \geq |\mathcal{A}_p|$ because we don't remove vertices. $|\mathcal{A}_{p'}| > |\mathcal{A}_p|$ only if there exists a vertex S at p' such that $S \notin \mathcal{A}_p$ and S has an outgoing edge pointing to a vertex in $\mathcal{A}_p$. An edge pointing to some vertex V can only come from a transaction preceding V in the global log, but all vertices in $\mathcal{A}_p$ are already complete because T is a stable vertex at p. Therefore, such a vertex S does not exist. Hence, $|\mathcal{A}_{p'}| = |\mathcal{A}_p|$.

Consequently, the size of C does not grow as the global log extends from p to p'. C also does not shrink because we don't remove edges and vertices as viewed by the deadlock resolver. As a result, R still determines T to be in C at p'. □

Lemma 3.3 implies that two regions eventually agree on which vertices constitute a component. Lemma 3.4 implies that once a determination is made, it stays true forever. Together these lemmas assert that the DDR algorithm is deterministic. Next, we show that Detock guarantees strict serializability.

Lemma 3.5. *The execution order of any two conflicting transactions T_i and T_j is the same in every region.*

Proof. It follows from lemmas 3.3 and 3.4 that eventually all regions agree on one of the following cases:

T_i and T_j are in the same component. The DDR algorithm deterministically reorders them by their IDs, and they are executed following this order in every region.

T_i and T_j are in different components. Since they conflict with each other, without loss of generality, there is a path from T_i to T_j in the dependency graph in every region. By the execution algorithm, T_i is executed before T_j, and this order is the same in every region. □

Proposition 3.6. *Transaction schedules are strictly serializable.*

Proof. Detock eliminates cycles in the dependency graph using the DDR algorithm, thus its execution schedule follows the topological order of a DAG, which can be written as a serial schedule. Because of Lemma 3.5, all regions follow the same execution order, hence Detock guarantees one-copy serializability.

Furthermore, non-concurrent transactions are executed according to their temporal order in this serializable schedule: Let T_i and T_j be two conflicting transactions such that T_j is sent to Detock after T_i is executed and returned to a client. T_i getting executed means that the component containing T_i (which may include only T_i) is part of a stable subgraph. Therefore, T_j cannot be in the same component as T_i, thus must be executed strictly after T_i. As a result, execution of transactions in Detock is strictly serializable. □

3.4 Avoiding Livelock

The DDR algorithm finds and resolves **stable** SCCs that will never grow as new transactions are added to the graph. These stable SCCs correspond to deadlocks which it deterministically

resolves. However, unstable SCCs also correspond to deadlocks, which the DDR algorithm cannot immediately resolve. In theory, it is possible for an SCC to grow indefinitely and never become stable, which results in livelock. Although such livelock is easy to prevent by forcing coordinators to run an admission control algorithm that temporarily holds back transactions that conflict with transactions tied up in a large unstable SCC until that SCC becomes stable, such admission control increases transaction latency, and should only be used as a last resort. Preferably, large unstable SCCs should be prevented in the first place. This is equivalent to taking measures to limit situations where deadlock is likely to occur.

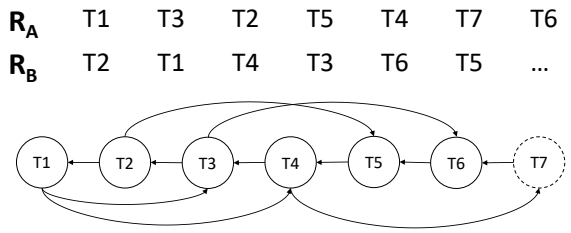


Fig. 5. An example where an SCC can grow indefinitely.

Deadlock occurs when different regions insert multi-home transactions into their respective local logs in different orders. High network delay between two regions increases the probability of this occurring. For example, assume there are 2 regions R_A and R_B that are hundreds of milliseconds apart in terms of network delay. The whole database consists of two data items A and B, local to R_A and R_B, respectively. All transactions in this example access both of these keys and thus are multi-home transactions. The first transaction T1 starts in R_A and enters the local log of R_A almost immediately, while it takes hundreds of milliseconds for T1 to reach R_B. In the meantime, another transaction T2 starts in R_B and enters the local log of R_B before T1 reaches R_B. It takes some time for T2 to reach R_A, but before that happens, T3 starts in R_A and enters the local log of R_A, and so on. Fig. 5 shows the local logs and dependency graph of up to 7 transactions that got into this scenario. At any time, the SCC cannot be resolved because the last transaction on this chain is always an incomplete transaction and has a path leading to every other transaction (T7 in Fig. 5).

The best way to avoid deadlock and livelock is to attempt to have multi-home transactions be inserted into every relevant region's local log in the same order. However, Detock's performance requirements prevents it from being able to globally order multi-home transactions before they begin, as done in other systems (such as SLOG, Fauna [4], and Calvin [54]). Instead Detock uses a best-effort scheme called *opportunistic ordering* that merely reduces the probability of conflicting orders of multi-home transactions (and thus deadlock), but does not eliminate it entirely.

When a transaction T first enters Detock, its coordinator assigns it a future (real time) timestamp based on its local clock (Algorithm 1, Line 16). Each participating region inserts T into its local log as soon as possible after its local time exceeds this timestamp (Algorithm 2, Line 4). Thus, if two transactions reach a region before their designated start times, they can be inserted into the local log at that region by this timestamp order. This is true, even if the clock at that region is not synchronized with the clocks at the regions which originally generated the timestamps of these transactions. Therefore, if these two transactions arrive **everywhere** such that their timestamps are later than the local clock at the location of their arrival, they are guaranteed to be consistently ordered everywhere.

However, if a region receives T at a local time later than the timestamp assigned to T, all it can do is to immediately insert T to its local log. Therefore, two transactions can be placed into the local log at a region out of (timestamp) order if at least one of them arrives after its designated time. To reduce the probability of this occurring, the coordinator attempts to assign a timestamp far enough into the future so that it will arrive everywhere prior to its designated start time. To accomplish this, the future timestamp is computed by adding to the coordinator's local time the one-way delay to the farthest participating region (delay-wise) plus a small overshoot (2 ms in our implementation) (Algorithm 1, Line 16).

The one-way delay from a region R_A to another region R_B is estimated by periodically sending a message from R_A to R_B containing the sending time t_A, then R_B responses with the time offset $t_B - t_A$ where t_B is the time when R_B receives the message. The servers at R_A compute a moving average of this offset to smooth out noise and use this value as the one-way delay from R_A to R_B. This offset also incorporates the clock difference between the two regions (and thus can be negative) so this scheme does not require highly synchronized clocks. Inaccurate estimation does not affect the correctness of the system but potentially degrades its performance due to increased number of deadlocks.

4 HOME-MOVEMENT TRANSACTIONS

When the locality of a workload changes (e.g. a user moves to a new continent), data migration between regions is needed to keep up with access pattern changes. Detock carries out such data migration by using *home-movement transactions*. It performs home movement within a transaction to ensure strict serializability and avoid down time. Our description focuses on home-movement transactions that involves only one data item at a time, but it is straightforward to extend this to multiple data items.

As mentioned previously, the identifier of a data item's home region is physically stored next to the data itself. A home-movement transaction's only action is to modify this identifier. In Algorithm 1, a home-movement transaction T_{hm} has three self-explanatory fields: *movedKey*, *oldHome*, and *newHome*. The values of *movedKey* and *newHome* are determined by the Detock system component (or system administrator) that decides that the data item should be located at a particular region and submits the home-movement transaction. The value of *oldHome* is retrieved from the **`HomeDirectory`** index at the transaction's coordinator (Algorithm 1, Line 5). Unlike ordinary transactions, a home-movement transaction does not only store the current home of a data item in the *homeInfo* map but also its soon-to-be new home (Line 6). Consequently, a home-movement transaction is always a multi-home transaction.

The home-movement transaction T_{hm} is treated exactly like an ordinary multi-home transaction from this point onward. Each region processes the transaction after receiving its two component GraphPlacementTxns, T_{hm}^{old} and T_{hm}^{new}, and then updates its **`HomeDirectory`** accordingly. Concurrent transactions which access the moved data item may see the old or new home location when entering the system. If they see the old location, they will be sent there and inserted into the old region's local log. If it gets inserted into that local log after T_{hm}^{old}, it will abort and restart at runtime during home validation. If it gets inserted prior to T_{hm}^{old}, it logically occurs before the data item moved, and will succeed.

According to Definition 3.1, two transactions conflict not only on the key they access but also on the expected home region. If T is a transaction that sees the old location and is placed before T_{hm}^{old} but after T_{hm}^{new} in the global log, this definition prevents T from being blocked by T_{hm}^{new} because although they access the same key, T expects the key's home region to be *oldHome* whereas T_{hm}^{new} expects that to be *newHome*.

Discussion. SLOG also introduces an algorithm for data home-movement [49]. However, the algorithm described here is both easier to reason about and simpler to implement. Additionally, SLOG's home-movement algorithm requires storing a counter in the header of every data item, thus increasing the size of the database, while this new algorithm does not require such a counter. The key difference that enables these advantages is that SLOG's algorithm makes the home-movement transactions single-home, whereas Detock's algorithm constructs them as multi-home transactions.

5 EVALUATION

We implemented Detock in C++ with ZeroMQ [1] for message passing between processes on different nodes and between threads in the same process. Detock supports pluggable storage layers, and currently defaults to an in-memory key-value store. Transactions are implemented as stored procedures containing read and write operations over a set of keys[2]

The goal of Detock is to achieve high throughput and low latency for strictly serializable transactions over a geo-replicated and geo-partitioned database. Hence, we compare Detock to four other systems that also support globally distributed transactions: Calvin [54], SLOG [49], Janus [40], and CockroachDB [52]. To reduce performance artifacts that are unrelated to the architectural designs discussed in this paper, we re-implemented Calvin, SLOG, and Janus inside the Detock codebase so that all four systems can use the same storage layer, communication library, local consensus code, and logging infrastructure. Unlike Detock, Calvin globally orders all transactions, and SLOG globally orders all multi-home transactions. The SLOG paper discusses two ways to do this: (1) sending them all to the same region/ordering service and (2) performing global consensus via Paxos or Raft. Option (2) increases the latency of every multi-home transaction by at least the latency of the global consensus protocol (hundreds of milliseconds for a truly global deployment). Option (1) only increases the latency of multi-home transactions that initiate far from the ordering service. We experiment with both versions in Section 5.2, but since option (1) yields better latency, we use it for both Calvin and SLOG in the other experimental sections, in order to present their latency in the best possible light, even though it is less robust to region failure than option (2). Janus generalizes the EPaxos protocol [38] to process distributed transactions, which (similar to Detock) includes a reordering technique over a dependency graph and execution of transactions across all replicas and shards deterministically following the graph order. However, unlike Detock's asynchronous protocol, Janus synchronously replicates data to every region so it needs at least one WAN round-trip to all regions to commit.

[2]The source code is available at https://github.com/umd-dslam/Detock

Table 1. Round-trip time for all pairs of regions (ms)

	apse2	apse1	apne2	apne1	euw2	euw1	usw2*	usw1*	use2
use1	197	211	173	148	75	67	66	61	12
use2	187	197	160	132	85	77	52	50	
usw1*	137	169	134	107	145	136	20		
usw2*	139	174	124	95	128	127			
euw1	254	183	229	202	11				
euw2	263	171	236	209					
apne1	128	71	32						
apne2	148	72							
apse1	91								

* Regions used only in the CockroachDB experiment.

We choose CockroachDB as another comparison point because it has support for state-of-the-art geo-partitioning that uses a non-deterministic approach. CockroachDB inherits many of its architectural principles from Spanner [17]. Both CockroachDB and Spanner are widely used; however CockroachDB is the better comparison point since it is available as independent downloadable code and can be deployed on the same cluster as our other experimental systems, and supports geo-partitioning. Nonetheless, it is far more production-ready than the research prototype of Detock, uses a more robust and fully-featured storage layer, and any raw performance numbers with Detock would be oranges to apples. Therefore, we only report relative measurements and observe the performance trends of each system.

Unless stated otherwise, we ran our experiments on Amazon EC2 using r5.4xlarge instances. Each machine has 16 vCPU and 128GB of memory. We deployed the systems over 8 AWS regions: us-east-1 (N. Virginia), us-east-2 (Ohio), eu-west-1 (Ireland), eu-west-2 (London), ap-northeast-1 (Tokyo), ap-northeast-2 (Seoul), ap-southeast-1 (Singapore), and ap-southeast-2 (Sydney). Table 1 contains the round-trip time for every pair of regions. In each region, a replica of the database is partitioned across 4 machines.

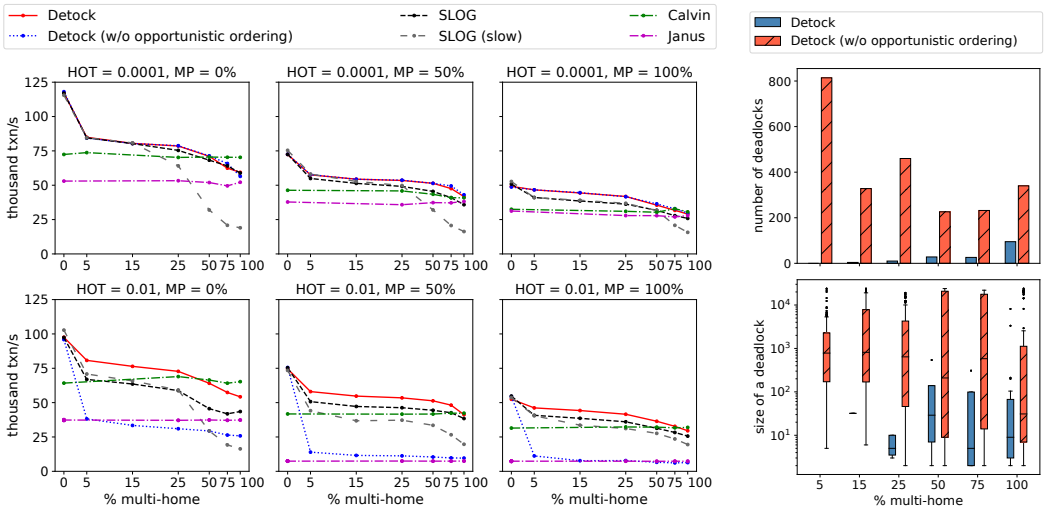
The clients generating the workloads were deployed on separate machines spread evenly across all regions, and had enough capacity to avoid being bottlenecks. Each client thread issued one transaction at a time. Transactions are either single-home (SH) or multi-home (MH); separately, as an unrelated consideration, they can be either single-partition (SP) or multi-partition (MP). SH transactions access data in the region closest to the client that generates it. MH transactions access data from two regions. Calvin and Janus do not assign home regions to data items, and do not support geo-partitioning so their transactions can only be SP or MP. Each client weights the regions following a Zipfian distribution such that regions closer to the client are more likely to be selected for a MH transaction. We vary the percentage of MH and MP transactions.

5.1 Microbenchmark experiments

In our first set of experiments, we use a version of the Yahoo! Cloud Serving Benchmark (YCSB) [16] adapted for transactions. The data consists of a single table containing a billion rows and two columns: a 64-bit integer key and a value consisting of 100 random bytes. Identically to previous work running experiments on this same dataset [49], contention of the workload is varied by dividing the table into "hot records" and "cold records". A transaction performed read-modify-write on 2 hot records and 8 cold records; all records were uniformly selected at random. Contention is varied by changing the size of the hot record set. We define HOT to be the reciprocal of the size of the hot re cord set per partition. Thus, contention increases with the value of HOT. Our initial set of experiments places all machines in the us-east-2 (Ohio) region, and uses tc [5] to simulate the network round-trip time of the 8-region deployment with symmetric network paths and a jitter uniformly distributed within 1 ms. This will allow us to directly vary network conditions in Section 5.1.3. In Section 5.2 we remove the simulation and run over the real full 8-region deployment.

5.1.1 Throughput. Fig. 6a shows the peak throughput of Calvin, Janus, SLOG and Detock, with and without opportunistic ordering. We varied the % MP and % MH parameters at two HOT settings corresponding to a low contention workload (HOT = 0.0001) and a high contention workload (HOT = 0.01).

Calvin and Janus do not distinguish between SH and MH transactions (since they do not support geo-partitioning); thus their throughput stays constant as % MH is varied. In contrast, Detock and SLOG are able to benefit from the presence of SH transactions in workload by processing them at different regions in parallel, while Calvin and Janus have to order all transactions within a single global log. Therefore, for workloads which geo-partition well (e.g. 15% or fewer MH transactions),



(a) Throughput (b) Deadlocks

Fig. 6. Microbenchmark results

Detock and SLOG significantly outperform Calvin and Janus. However, as MH% increases past 30%, the geo-partitioning advantage of Detock and SLOG disappears and all systems perform similarly. At extremely high levels of MH transactions, they even perform slightly worse than Calvin, since they incur additional overhead to process MH transactions (i.e. dividing the transaction into its region-local components, and inserting in the local log of each region). However, this lower throughput relative to Calvin only occurs when the MP% is 0. When MP transactions are common, all systems have to pay overhead processing the transaction across the different local machines that own the partitions of data accessed. Therefore, Calvin only outperforms Detock in the scenario of high MH% and low MP% — an unlikely scenario since if a workload is partitionable, it is generally locally geo-partitionable as well.

Janus requires traversal of the dependency graph, including communication with other shards for missing information, on the critical path of every transaction to determine when it is ready to be executed. This overhead causes Janus to perform worse than other systems. Conversely, Detock traverses the dependency graph and communicates with other partitions in a background thread that wakes up periodically, thus its cost is amortized across the periodic runs. The throughput of Janus drops further under high contention, especially when there are MP transactions, because the graph grows faster and more cross-shard messages are needed.

Under low contention (top row), the best versions of Detock and SLOG have similar throughput. However, under high contention (bottom row), Detock outperforms SLOG when there are MH transactions in the workload. This is because SLOG must globally order all MH transactions. Different regions involved in the transaction find out the order (and then insert the transaction into their local log) at different points in time. The closer they are to the region that determines the order, the faster they get started with the transaction. However, a transaction cannot release locks until they receive local logs from all regions involved in the transaction, even remote regions. The slowest transactions in SLOG therefore must hold locks for longer than the slowest transactions in Detock. Under low contention, this does not affect performance. But under high contention, this longer hold time reduces throughput. In contrast, Detock uses opportunistic ordering to insert the

GraphPlacementTxn to the local log at roughly the same time, thus distributing the blocking time evenly across the regions.

SLOG's performance depends on resources allocated for its ordering service. To show this, we ran a version of SLOG where we cut down the number of threads used for deserializing and batching the transactions in the ordering service from 8 to 1. The throughput of this version, shown as SLOG (slow) in Fig. 6a, quickly drops as the amount of MH transactions increases. Detock is not bounded by this constraint because it does not need an ordering service.

Surprisingly, even at large numbers of multi-home transactions, Detock's performance is almost unchanged relative to its performance at low contention. Such independence from performance degradation for strictly serializable multi-region transactions at extremely high contention is rare amongst current state-of-the-art systems and is an important advantage of Detock's approach. The comparison to CockroachDB's state-of-the-art geo-partitioning system in Section 5.4 will further highlight the Detock's exceptional resilience to high contention workloads.

To understand the drop of throughput of the Detock version without opportunistic ordering, we plotted the number and size of deadlocks (SCCs) of the two Detock versions for HOT = 0.01 and MP = 100% in Fig. 6b. The reason for the performance drop is thus due to the growth of the number and size of deadlocks. When the contention is high, the probability that new incomplete dependencies emerging while a deadlock is forming increases, preventing the DDR algorithm to immediately resolve the deadlock.

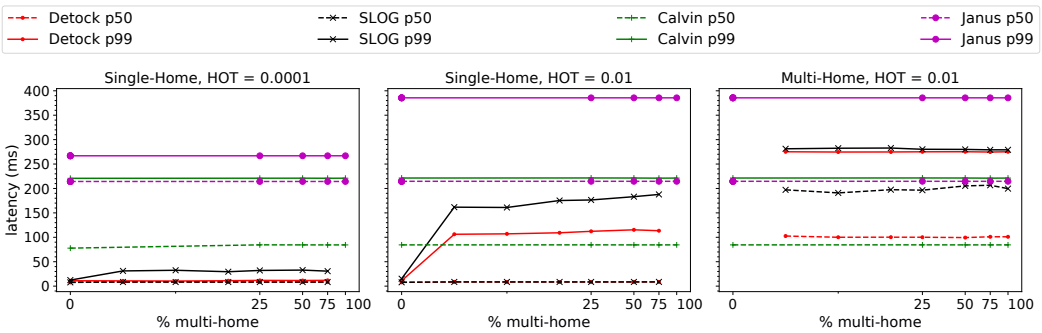


Fig. 7. Microbenchmark latency (MP = 100%)

5.1.2 Latency. We measured the end-to-end latency using a smaller number of clients to avoid including queuing time. Fig. 7 presents the p50 and p99 latency at 100% MP. Under low contention, SLOG and Detock[3] achieve low latency for SH transactions as expected. Meanwhile, Calvin must globally order **all** transactions so it results in an order of magnitude worse latency. Under high contention, it becomes more likely for SH transactions to conflict with the longer-running MH transactions, so SLOG and Detock have higher p99 latency for SH transactions in the presence of MH transactions. However, the impact of this on Detock is much lower than SLOG.

For MH transactions, SLOG has a latency higher than Calvin's and Detock's because every MH transaction in SLOG needs (1) a round-trip to the ordering region and (2) additional communication to exchange the local logs across regions. Calvin also must pay cost (1) but avoids cost (2). Detock must pay cost (2) but avoids cost (1). Therefore they have similar latency at p50. However, cost (2)

[3]The latency difference between Detock with and without opportunistic ordering is the amount of the overshoot (2ms). All other delays caused by opportunistic ordering is overlapped with transaction processing and do not cause a latency increase. To avoid cluttering the graph, we only show Detock **with** opportunistic ordering.

has a longer tail latency when locks are held during this communication; therefore, Calvin achieves better p99 latency for MH transactions.

Janus has the highest latency because its fast quorums contain all replicas, thus every transaction coordinator generally has to wait for the response from the slowest region.

5.1.3 Performance under different network conditions. The effectiveness of opportunistic ordering and consequently the overall performance of Detock is affected by the accuracy of its estimation of the one-way delay between two regions, and irregular network conditions may affect such estimation. Therefore, we now study the effect of asymmetric network delay and jitter on Detock performance. In the following experiments, we ran a workload with HOT = 0.01 (high contention), MP = 100% and MH = 10%.

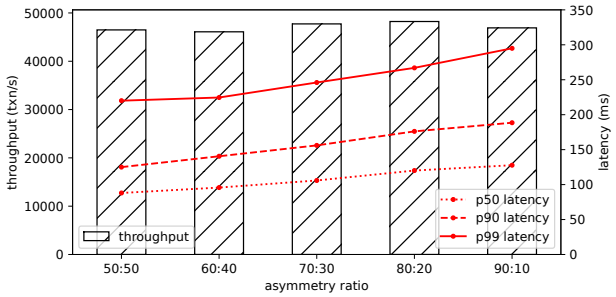


Fig. 8. Detock's performance under asymmetric delay

A network path is asymmetric if the forward and backward one-way delays differ. We varied the ratio between the one-way delays on the same network path. This ratio was applied to all paths of every pair of servers across two different regions, with the direction of asymmetry randomly chosen. Fig. 8 shows that delay asymmetry does not affect throughput. This is because the opportunistic ordering scheme constantly monitors the one-way latency between regions and adjusts its predictions accordingly. Accurate one-way predictions are sufficient to avoid deadlock.

In contrast to throughput, latency increases as the ratio becomes more extreme. This is because in an asymmetric network, the region with the largest one-way delay from the coordinator might not be the one with the largest round-trip time and vice-versa. Therefore, opportunistic ordering scheme might cause the farthest region from the coordinator to hold off the transactions as if it is a closer region, increasing the overall latency of the transaction.

Nonetheless, the impact on overall latency is insignificant except at extreme asymmetries, yet studies have shown that 90% of the measured one-way delay on the Internet are within 40%–60% of the round-trip time [43]. Furthermore, consistently severe asymmetry is unlikely in real-world data center environments [57].

Network jitter is variance in network delay. We simulated different uniform jitter values in all inter-region paths. Fig. 9a shows peak throughput, p99 latency at peak throughput (blue line), and p99 latency at a lower load (red line). Increase in jitter impacts peak throughput more than latency at an unsaturated load. Fig. 9b shows the reason for this result: at 15ms or more jitter, opportunistic ordering's delay estimations become inaccurate, and it becomes increasingly ineffective at preventing deadlocks, which reduces throughput. Detock deployments with high network jitter need higher overshoots. Fig. 9b showed that, with the default overshoot of 2 ms, Detock is robust to jitter of up to 15 ms. Fig. 9c and 9d show that increasing to a 10 ms overshoot significantly reduces the number of deadlocks, thereby raising throughput.

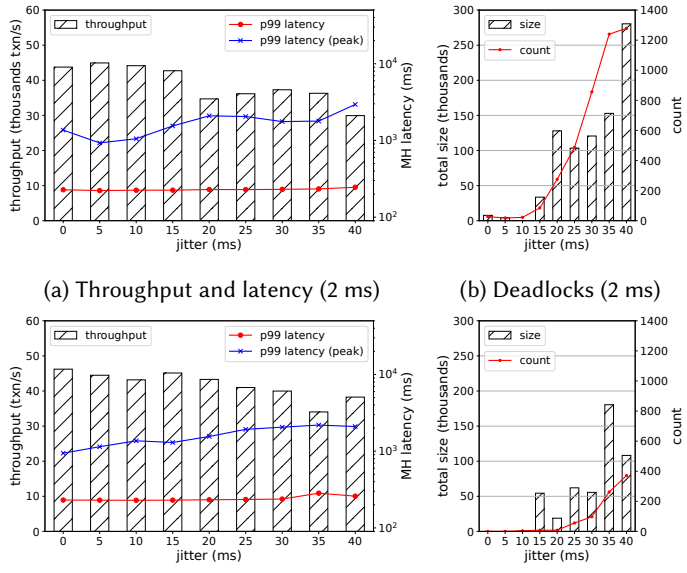


Fig. 9. Network delay jitter experiments (numbers in parentheses are the opportunistic ordering overshoots)

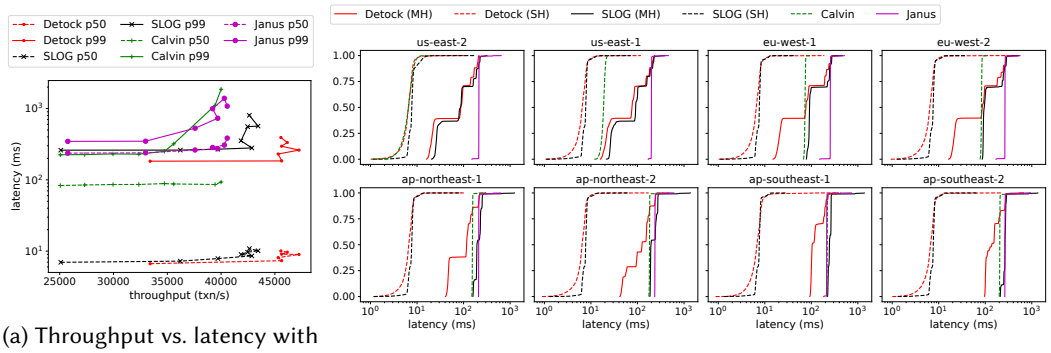
5.2 TPC-C

Next, we evaluate Detock on the TPC-C benchmark [3] that is designed based on the activities of a wholesale supplier with 9 tables and 5 types of transactions. TPC-C data is typically partitioned by the warehouse table and we follow this partitioning by assigning different warehouses across the eight physical regions in our deployment. We initialized the database with 1200 warehouses and 10 districts per warehouse. We followed the specification for the transaction mix ratio, displayed in Table 2. However, transactions whose access set are dependent on a read were modified to remove these dependencies, since the Detock codebase does not currently support dependent transactions. For example, payment transactions select customers only by their IDs (instead of combination of IDs and last names). Some new-order and payment transactions may access "remote" warehouses in addition to their default warehouses. The specification only requires a remote warehouse to be any warehouse other than the default one. However, we redefine a remote warehouse to specifically be a warehouse that resides in a remote region; hence, these transactions become multi-home transactions.

Table 2. TPC-C transactions mix ratio

	new-order	payment	order-status	delivery	stock-level
SH	40.7%	42.3%	4.1%	4.1%	4.0%
MH	4.4%	0.4%	0%	0%	0%
total	45.1%	42.7%	4.1%	4.1%	4.0%

We ran the TPC-C workload while increasing the number of clients until reaching peak throughput and plotted the p50 and p99 latency at different throughputs in Fig. 10a. Detock and SLOG



(a) Throughput vs. latency with increasing number of clients

(b) CDF of latency per region

Fig. 10. TPC-C results

have the same median latency because the majority of transactions in the TPC-C workload are single-home. Calvin and Janus have much higher latency since every transaction has to be globally ordered. Detock's 99% latency is 66ms lower than SLOG's because of its ability to avoid the global ordering step for multi-home transactions which constitute 4.8% of the workload (see above, Section 5.1.2).

Detock reaches a higher peak throughput than SLOG, Calvin, and Janus do for the same reasons discussed in Section 5.1.1.

To further explore the advantage of Detock for MH transactions, we plotted in Fig. 10b the CDF of SH and MH transaction latency in every region. Note that the x-axis is log scaled. In SLOG, the ordering service was in us-east-2. Therefore, the farther a region is from us-east-2 (US East Coast), the more benefit is accrued from Detock's ability to serve MH transactions without making a round-trip to the ordering service. This results in many transactions having a factor of 5 better latency than SLOG's. Us-east-1 and us-east-2 also have improvement in their transaction latency because the ordering service incurs queuing and processing delays, which are not present in Detock. The Calvin version in this experiment uses one region us-east-2 for ordering, thus the latency of transactions increases as they originate farther away from the ordering region. On the other hand, every transaction in Janus generally has to wait for responses from all regions for its fast quorum, hence every region experiences high latency.

5.3 Scalability

We evaluate the scalability of Detock by running the microbenchmark as the number of machines per region increases from 3 to 21. Fig. 11 shows the results of Detock in comparison to SLOG under different settings of the parameters HOT, % MP, and% MH.

When MP and MH are 0, SLOG scales better than Detock due to the overhead of the background thread in Detock which periodically scans the dependency graph for deadlocks. This overhead can be mitigated by dynamically adjusting the activity of the background thread based on the frequency of deadlocks. The cost of dependency graph management in Detock grows under high contention when there are MH transactions in the workload and even more so when there are also MP transactions. Thus, Detock reaches scalability limitations earlier in the lower graph where contention is high. Figure 12 shows the reason for this is that the unstable part of the graph takes longer to be resolved and thus grows large in the presence of MP transactions, as each partition only has a partial view of the graph and needs to wait for more information from other partitions

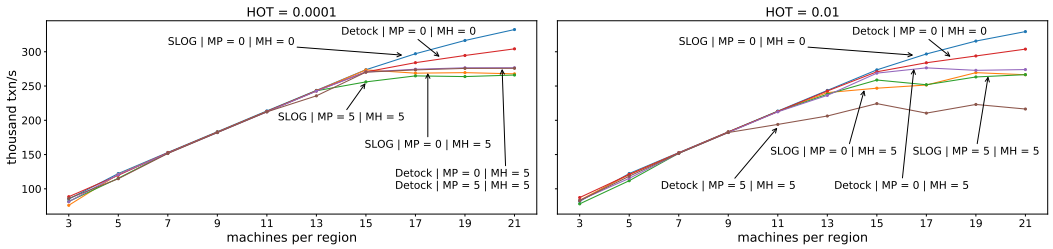


Fig. 11. Scalability of Detock and SLOG

to proceed. Either way, when there are multi-home transactions in the workload, the throughput of both Detock and SLOG cease to increase after 15 machines per region. This is because multi-home transactions generate extra GraphPlacementTxns in Detock and LockOnlyTxns in SLOG, the routing of which becomes more complex as the cluster size increases. An improved routing layer would increase the scalability of both systems.

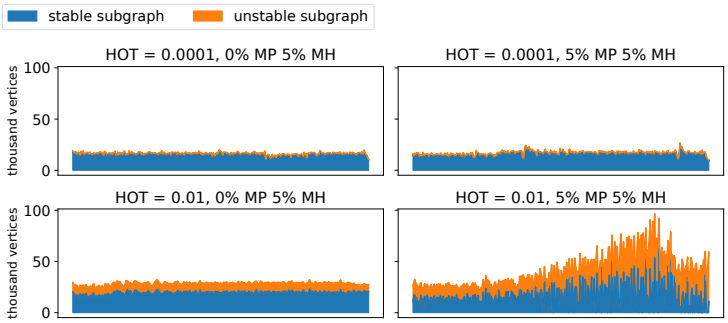
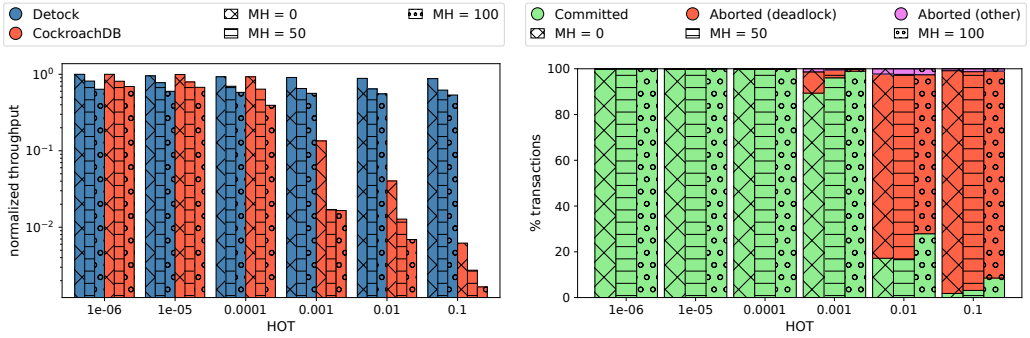


Fig. 12. Graph size over time at 19 machines per region

5.4 Comparison to CockroachDB

CockroachDB is a distributed transactional database system that allows users to control the locality of individual rows, and thus compares directly with Detock. Although, CockroachDB does not guarantee strict serializability because it is susceptible to the causal reverse anomaly [27], it guarantees strict serializability for the vast majority of practical workloads and its architecture is based on Spanner which does guarantee strict serializability for all workloads. As discussed above, the absolute performance numbers between Detock and CockroachDB are incomparable because the two systems come from separate codebases. Nonetheless, information about the consequences of the architectural differences can still be gleaned from their relative performance trends.

We deployed the two systems in 6 regions: us-east-1 (N. Virginia), us-east-2 (Ohio), us-west-1 (N. California), us-west-2 (Oregon), eu-west-1 (Ireland), and eu-west-2 (London); the inter-region latency is included in Table 1. Each region had 3 c5.4xlarge EC2 instances (16 vCPU and 32GB memory), as recommended by CockroachDB [6]. We used CockroachDB v21.1, which was the latest version when we ran the experiment. To eliminate as many differences between the two systems as possible, we configured CockroachDB such that it used the in-memory storage engine, had only one copy per replica within a region, and parsed the SQL queries only once using prepared statements. We sent every transaction in one shot, with automatic retry turned off so that we could



(a) Normalized throughput (b) % committed/aborted of CockroachDB

Fig. 13. Comparison to CockroachDB

collect abort information. We ran the YCSB workload while varying the % MH and HOT parameters. Since CockroachDB distributed data uniformly across servers in each region, most transactions are multi-partition. Therefore, we compared against 100% MP for Detock.

The results of this experiment is normalized in Fig. 13a such that the throughput of Detock and CockroachDB at different contention levels is relative to their throughput at the lowest contention level. CockroachDB's throughput plunges at high contention, and even more at high MH%. In the worst case, the throughput of CockroachDB drops to less than 1% relative to its throughput at the lowest contention. Conversely, Detock's throughput decreases more gradually and slowly. It is able to retain at least 76% of the original throughput at the highest contention level.

CockroachDB partitions the database into multiple consensus groups. Its geo-partitioning feature places each group within a single region so that nearby reads and writes have low latency. CockroachDB uses a form of locking to handle write-write conflicts and thus is susceptible to deadlocks. It breaks a deadlock by randomly aborting one of the transactions. Additionally, its use of two-phase commit exacerbates the time a transaction needs to hold locks. Fig. 13b shows the percentage of CockroachDB's transactions that are committed, aborted due to deadlocks, and aborted due to other reasons. When contention increases, CockroachDB aborts more transactions because of deadlocks, causing wasted work. In contrast, Detock does not abort transactions due to deadlocks.

6 RELATED WORK

Graph-based concurrency control. Previous work has proposed analyzing transaction dependencies to improve the performance of concurrency control protocols [22, 39, 40, 56, 59]. Furthermore, dependency graphs have been used in practice for decades for deadlock detection [10]. Unlike traditional deadlock detection algorithms, DDR guarantees deterministic deadlock resolution, and therefore must generate a more complete dependency graph than traditional algorithms that only need to identify and destroy simple cycles. DDR minimizes the cost of this extra work by using opportunistic ordering to reduce the probability of deadlock.

Geo-replication. To achieve good latency and throughput, many geo-replicated systems use asynchronous replication and opt for weak consistency models such as eventual consistency [2, 13, 20, 29, 32, 44, 53], strong session serializability [19], timeline consistency [15], or causal consistency [21, 34, 35]. For stronger consistency (e.g. linearizability) over wide area network (WAN), the Paxos [30, 31] and Raft [41] consensus protocols are commonly implemented [4, 17, 37, 39, 49, 52, 54, 58].

These protocols require clients to send commands to a stable leader, causing a remote client to pay for extra WAN round-trip time. EPaxos [38] is a leaderless consensus protocol which involves tracking dependencies between commands and reordering strongly connected components, and has similarities to Detock's DDR algorithm. However, EPaxos has to rate limit to reduce the effect of livelock by prioritizing executing old commands over starting new commands. In contrast, Detock targets high conflict transactional workloads in which livelock would be common, and therefore uses opportunistic ordering instead.

Distributed database systems. MDCC [28], Replicated Commit [37], TAPIR [60], Carousel [58], and Ocean Vista [23] are globally distributed database systems that aim to cut down the number of WAN round-trips but still incur cross-region latency for every transaction. In contrast, Detock processes transactions with strict serializability without incurring cross-region latency (except for multi-home transactions).

CockroachDB [52, 55], Spanner [17], Ocean Vista [23], and Dast [14] order transactions strictly by their timestamps. In contrast, Detock does not require global ordering, and only generates timestamps to reduce the probability of livelock, and can tolerate much large error bounds clock accuracy. Also, unlike Spanner, inaccurate clocks never affect the correctness of Detock.

RingBFT [48] uses a deadlock avoidance technique where it passes cross-shard transactions around the shards in a predetermined ring order, hence avoiding deadlocks and achieving high throughput. Detock instead allows distributed deadlocks to occur and resolves them on the fly, providing a new point in the tradeoff space when considering deadlock avoidance vs. detection for geo-partitioned systems. RingBFT can tolerate Byzantine failures, which comes with high latency, while Detock focuses on applications that benefit from low latency in non-Byzantine environments.

G-Store [18], L-Store [33], DynaMast [7], and MorphoSys [8] co-locate data in a single node using data migration or dynamic remastering to guarantee single-partition transactions. In contrast, home movement in Detock is a rare event, and is never required during transaction processing. Instead, it supports efficient multi-partition transactions.

Most proposed deterministic database systems cannot provide low-latency geo-distributed transactions due to reliance on a centralized global sequencing layer [26, 36, 47, 51, 54].

7 CONCLUSION

While the related work described above must trade off consistency for latency, Detock is able to completely side-step this trade-off for geographically partitionable workloads. Furthermore, even for non-partitionable workloads, Detock is able to process strictly-serializable multi-partition transactions with single round-trip latency and high throughput. Even under extremely high contention we observed near-zero performance degradation, and orders of magnitude better throughput robustness than CockroachDB.

ACKNOWLEDGMENTS

We would like to thank Pooja Nilangekar and the anonymous reviewers for their insightful comments and suggestions that helped us improve the quality of this paper. This work is supported by the National Science Foundation under grants DGE-1840340 and IIS-1910613.

REFERENCES

[1] 2007. ZeroMQ. https://zeromq.org/.
[2] 2009. MongoDB. https://mongodb.com.
[3] 2010. TPC Benchmark C. http://www.tpc.org/tpcc/.
[4] 2012. Fauna. https://fauna.com.

[5] 2021. tc(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.
[6] 2022. Production checklist | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/recommended-production-settings.htm.
[7] 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
[8] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *Proc. VLDB Endow.* 13, 13 (sep 2020), 3573–3587. https://doi.org/10.14778/3424573.3424578
[9] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf
[10] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA.
[11] P. A. Bernstein, D. W. Shipman, and W. S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Softw. Eng.* 5, 3 (may 1979), 203–216. https://doi.org/10.1109/TSE.1979.234182
[12] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. 1992. Overview of Multidatabase Transaction Management. *The VLDB Journal* 1, 2 (oct 1992), 181–240. https://doi.org/10.1007/BF01231700
[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 49–60.
[14] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving Low Tail-Latency and High Scalability for Serializable Transactions in Edge Computing. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 210–227. https://doi.org/10.1145/3447786.3456238
[15] 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
[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152
[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245
[18] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/1807128.1807157
[19] K. Daudjee and K. Salem. 2004. Lazy database replication with ordering guarantees. In *Proceedings. 20th International Conference on Data Engineering*. 424–435. https://doi.org/10.1109/ICDE.2004.1320016
[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281
[21] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (jul 2018), 1618–1632. https://doi.org/10.14778/3236187.3236210
[22] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2588555.2610529
[23] Hua Fan and Wojciech Golab. 2019. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1471–1484. https://doi.org/10.14778/3342263.3342627
[24] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[25] Shady Issa, Miguel Viegas, Pedro Raminhas, Nuno Machado, Miguel Matos, and Paolo Romano. 2020. Exploiting Symbolic Execution to Accelerate Deterministic Databases. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 678–688. https://doi.org/10.1109/ICDCS47774.2020.00040

[26] Bettina Kemme and Gustavo Alonso. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 134–143.

[27] Spencer Kimball and Irfan Sharif. 2022. Living Without Atomic Clocks. https://www.cockroachlabs.com/blog/living-without-atomic-clocks/.

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

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[30] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

[31] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[32] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 265–278.

[33] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[34] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[35] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 313–328.

[36] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[37] 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

[38] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[39] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 479–494.

[40] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 517–532.

[41] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[42] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (oct 1979), 631–653. https://doi.org/10.1145/322154.322158

[43] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. 2008. A Measurement Study of Internet Delay Asymmetry. In *Proceedings of the 9th International Conference on Passive and Active Network Measurement* (Cleveland, OH, USA) *(PAM'08)*. Springer-Verlag, Berlin, Heidelberg, 182–191.

[44] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 288–301. https://doi.org/10.1145/268998.266711

[45] Seth Proctor. 2013. Exploring the Architecture of the NuoDB Database, Part 1. https://www.infoq.com/articles/nuodb-architecture-1.

[46] Seth Proctor. 2013. Exploring the Architecture of the NuoDB Database, Part 2. https://www.infoq.com/articles/nuodb-architecture-2.

[47] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT* (Copenhagen, Denmark). 73–84.

[48] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2022. RingBFT: Resilient Consensus over Sharded Ring Topology. In *EDBT* (Edinburgh, UK). 298–311.

[49] 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

[50] 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

[51] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1150–1160.

[52] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[53] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 172–182. https://doi.org/10.1145/224056.224070

[54] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[55] 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

[56] Arthur Whitney, Dennis Shasha, and Stevan Apter. 1997. High volume transaction processing without currency control, two phase commit, SQL or C++. In *Seventh international workshop on high performance transaction systems, September 1997, Asimolar, California.* 211–217.

[57] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/3386367.3431291

[58] 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

[59] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Trans. on Knowl. and Data Eng.* 28, 10 (oct 2016), 2635–2650. https://doi.org/10.1109/TKDE.2016.2578319

[60] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4, Article 12 (dec 2018), 37 pages. https://doi.org/10.1145/3269981

Received October 2022; revised January 2023; accepted February 2023