SLOG: Serializable, Low-latency, Geo-replicated Transactions

Kun Ren
eBay Inc
kuren@ebay.com

Dennis Li
UMD College Park
dli12348@umd.edu

Daniel J. Abadi
UMD College Park
abadi@cs.umd.edu

ABSTRACT

For decades, applications deployed on a world-wide scale have been forced to give up at least one of (1) strict serializability (2) low latency writes (3) high transactional throughput. In this paper we discuss SLOG: a system that avoids this tradeoff for workloads which contain physical region locality in data access. SLOG achieves high-throughput, strictly serializable ACID transactions at geo-replicated distance and scale for all transactions submitted across the world, all the while achieving low latency for transactions that initiate from a location close to the home region for data they access. Experiments find that SLOG can reduce latency by more than an order of magnitude relative to state-of-the-art strictly serializable geo-replicated database systems such as Spanner and Calvin, while maintaining high throughput under contention.

PVLDB Reference Format:
Kun Ren, Dennis Li, and Daniel J. Abadi. SLOG: Serializable, Low-latency, Geo-replicated Transactions. *PVLDB*, 12(11): 1747-1761, 2019.
DOI: https://doi.org/10.14778/3342263.3342647

1. INTRODUCTION

Many modern applications replicate data across geographic regions in order to (a) achieve high availability in the event of region failure and (b) serve low-latency reads to clients spread across the world. Existing database systems that support geographic replication force the user to give up one at least one of the following essential features: (1) strict serializability (2) low-latency writes (3) high throughput multi-region transactions — even under contention.

(1) Strict serializability [16, 17, 41, 64], in the context of distributed database systems, implies both strong isolation (one-copy serializable [10]) and real-time ordering guarantees. More precisely, concurrent transaction processing must be equivalent to executing transactions in a one-copy serial order, S, such that for every pair of transactions X and Y, if X starts after Y completes, then X follows Y in S. This implies that all reads within a transaction must see the value of any writes that committed before the transaction began, no matter where that write was performed world-wide. Furthermore, if a transaction, A, begins after (in real time) transaction B completes, no client can see the effect of A without the effect of B. Strict serializability reduces application code complexity and bugs, since it behaves like a system that is running on a single machine processing transactions sequentially [4, 5].

By giving up strict serializability, it is straightforward to achieve the other two properties mentioned above (low-latency writes and high multi-region transactional throughput). If reads are allowed to access stale data, then reads can be served by any replica (which improves read latency) and replication can be entirely asynchronous (which improves write latency) . Furthermore, avoiding the multi-region coordination necessary to enforce strict serializability facilitates throughput scalability of distributed database systems [11].

Much research effort has been spent in designing systems that reduce the consistency level below strict serializability, while still providing useful guarantees to the application. For example, Dynamo [26], Cassandra [47], and Riak [3] use eventual consistency; PNUTs [20] supports timeline consistency; COPS [51], Eiger [52], and Contrarian [27] support a variation of causal consistency; Lynx (Transaction Chains) [95] supports non-strict serializability with read-your-writes consistency [81]; Walter [77], Jessy [8], and Blotter [56] support variations of snapshot consistency.

(2) Low latency writes. Despite the suitability of these weaker consistency models for numerous classes of applications, they often expose applications to potential race condition bugs, and typically require skilled application programmers. Thus, the demand for systems that support strict serializability has only increased. Many recent geo-replicated data stores, including from two of the three major cloud vendors (Google with several systems [12, 22, 76] and Microsoft with Cosmos DB) support strong consistency models at least under some configurations. Spanner [22] is widely used throughout Google, and is now available in the Google Cloud for anybody to use. Other examples in industry include comdb2, FaunaDB, and (with a few caveats) CockroachDB and YugaByte. Examples from the research community include Helios [59], MDCC [46], Calvin [87], Carousel [93], and TAPIR [94].

Every single one of these above cited strictly serializable systems pays at least one cross-region round trip (coordination) message to commit a write transaction. This type of coordination enables strict serializability, but increases the latency of **every** write. The farther apart the regions, the longer it takes to complete a write.

(3) High transaction throughput Cross-region coordination on every write is not necessary to guarantee strict serializability. If every read of a data item is served from the location of the most recent write to that data item, then there is no need to synchronously replicate writes across regions. For example, if one region is declared as the master region for a data item ([20, 23, 38, 61, 66, 67]), and all writes and consistent reads of that data item are directed to this region, such as done by NuoDB [66, 67] and G-Store [23], then it is possible to achieve strict serializability along with low latency

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342647

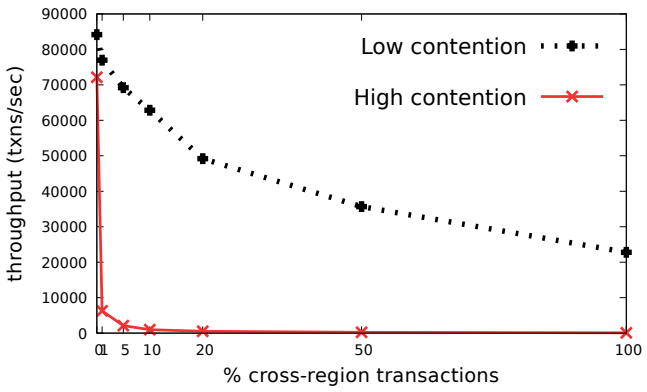


Figure 1: Single-master throughput under contention.

reads and writes — as long as those reads and writes initiate from a region near the master region for the data being accessed.

Many workloads have a locality in their access patterns. For example, for applications that revolve around users, the data associated with a user is heavily skewed to being accessed from the physical location of that user. Thus, for these workloads, it is possible to achieve strict serializability and low latency writes. However, arbitrary transactions may access multiple data items, where each data item is mastered at a different region. For such transactions, achieving strict serializability requires coordination across regions. Existing approaches prevent conflicting transactions from running during this coordination. Cross-region coordination is time consuming since messages must be sent over a WAN. Thus, the time window that conflicting transactions cannot run is large, which reduces system throughput. For example, we implemented a version of NuoDB[1], and ran some benchmarks (see Section 4) where we varied the contention level and percentage of multi-region transactions. As shown in Figure 1, under high contention, the throughput of the system dropped dramatically at even low percentages of multi-region transactions, and also fell (although less dramatically) at low contention. For this reason, many systems that allow different data to be controlled by different regions do not support multi-region transactions, such as PNUTS [20] and DPaxos [58].

In this paper, we present the design of a system, SLOG, that is the first (to the best of our knowledge) to achieve all three: (1) strict serializability (2) low-latency reads and writes (on at least some transactions) and (3) high throughput. SLOG uses locality in access patterns to assign a *home region* to each data granule. Reads and writes to nearby data occur rapidly, without cross-region communication. However, reads and writes to remote data, along with transactions that access data from multiple regions, must pay cross-region communication costs. Nonetheless, SLOG uses a deterministic architecture to move most of this communication outside of conflict boundaries, thereby enabling these transactions to be processed at high throughput, even for high contention workloads.

SLOG supports two availability levels: one in which the only synchronous replication is internally within a home region (which is susceptible to unavailability in the event of an entire region failure), and one in which data is synchronously replicated to one or more nearby regions (or availability zones) to achieve an availability similar to Amazon Aurora [89, 90] where the system remains available even in the event of a failure of an entire region. Unlike Spanner, Cosmos DB, Aurora, or the other previously cited systems that support synchronous cross-region replication, SLOG's deterministic architecture ensures that its throughput is unaffected by the presence of synchronous cross-region replication.

[1]NuoDB declined to give us their software, so we implemented our own version. NuoDB uses a MVCC protocol that is susceptible to write skew anomalies. Our version used locking instead in order to guarantee strict serializability.

2. BACKGROUND

SLOG supports strictly serializable transactions that access data mastered in multiple regions. Unlike systems such as L-Store [49] and G-Store [23], which remaster data on the fly so that all data accessed by a transaction becomes mastered at the same physical location, SLOG does not remaster data as part of executing a transaction. Instead, it utilizes a coordination protocol across regions to avoid data remastership. One important technique used by SLOG to overcome the scalability and throughput limitations caused by coordination across partitions is to leverage a *deterministic execution framework*. Our technique is inspired by the work on Lazy Transactions [30], Calvin [87, 88], T-Part [92], PWV [31], and FaunaDB [1] which use determinism to move coordination outside of transactional boundaries, thereby enabling conflicting transactions to run during this coordination process.

In the above-cited deterministic systems, all nodes involved in processing a transaction — all replicas and all partitions within a replica — run an agreement protocol prior to processing a batch of transactions that plans out how to process the batch. This plan is deterministic in the sense that all replicas that see the same plan must have only one possible final state after processing transactions according to this plan. Once all parties agree to the plan, processing occurs (mostly) independently on each node, with the system relying on the plan's determinism in order to avoid replica divergence. This approach prevents the dramatic reductions in throughput under data contention that are observed in systems such as NuoDB [66, 67] and the strictly serializable transaction implementation on top of FuzzyLog [53] that perform coordination inside transactional boundaries, and G-Store and L-Store which prevent contended data access during the remastering operations.

Determinism also reduces latency and improves throughput by eliminating any possibility of distributed and local deadlock [7, 70, 78, 79, 87], and reducing (or eliminating) distributed commit protocols such as two-phase commit [7, 83, 84, 86, 87].

Unfortunately, in order to create a deterministic plan of execution, more knowledge about the transaction is needed prior to processing it relative to traditional nondeterministic systems. Most importantly, the entire transaction must be present during this planning process. This makes deterministic database systems a poor fit for ORM tools and other applications that submit transactions to the database in pieces.

Second, advanced planning usually requires knowledge regarding which data will be accessed by a transaction [33, 34, 83, 87]. Most deterministic systems do not require the client to specify this information when the transaction is submitted. Instead they attempt to statically derive this knowledge from inspecting the transaction code [87], make conservative estimates [31], and/or speculatively execute parts of the transaction, such as the OLLP protocol in Calvin or the multi-stage execution process in FaunaDB.

SLOG's use of determinism causes it to inherit both of these requirements. As we will describe in Section 4, we implement SLOG inside the open source version of Calvin's codebase. Since Calvin uses a combination of static analysis and OLLP to determine read and write sets prior to transaction execution, our implementation also uses these techniques.

One important challenge in the design of SLOG is the lack of information about all transactions running in the system during the planning process. In Calvin, every transaction, no matter where it originates from, is sequenced by a global Paxos process. This enables Calvin to have complete knowledge of the input to the system while planning how a batch of transactions will be executed. However, this comes at the cost of requiring every transaction to pay the cross-region latency to run Paxos across regions. SLOG removes the global Paxos process in order to reduce latency, but this causes unawareness of transactions submitted to replicas located in different regions during the planning of transaction processing.

3. SLOG

In order to achieve low latency writes, replication across geographic regions must be performed either entirely asynchronously, or synchronously only to nearby regions; otherwise every write must pay a round trip network cost across the geographic diameter of the system [6]. In order to maintain strong consistency in the face of asynchronous replication, SLOG uses a master-oriented architecture, where every data item is mastered at a single "home" replica. Writes and linearizable reads of a data item must be directed to its home replica.

A region in SLOG is a set of servers that are connected via a low-latency network, usually within the same data center. Transactions are classified into two categories: if all data that will be accessed have their master replica in the same region, it is **single-home**. Otherwise, it is **multi-home**. Single-home transactions are sent to its home and are confirmed as completed either (a) after the region completes the transaction (SLOG-B) or (b) after the region completes the transaction AND it has been replicated to a configurable number of nearby regions (SLOG-HA). Either way, transactions that initiate near the home region of the data being accessed will complete without global coordination. In contrast, multi-home transactions, and even single-home transactions that initiate physically far from that home, will experience larger latency. SLOG does not require any remastering of data to process multi-home transactions, but does perform dynamic data remastering as access frequency from particular locations changes over time (Section 3.4).

The most technically challenging problem in the design of SLOG is the execution of multi-home transactions. The goal is to maintain strict serializability guarantees and high throughput in the presence of potentially large numbers of multi-home transactions, even though they may access contended data, and even though they require coordination across physically distant regions.

3.1 High-level overview

Each SLOG region contains a number of servers over which data stored at that region is partitioned (and replicated). Some of this data is *mastered* by that region (it is the home region for that data) and the rest is a replica of data from a different home region. The partitioning of the data across servers within a region is independent of the home status of data — each partition will generally have a mix of locally-mastered and remotely-mastered data.

Each region maintains a local input log which is implemented via Paxos across its servers. This local log only contains transactions that are expected to modify data mastered at that region. This input log is sent to all other regions in batches. Each batch is labeled with a sequence number, and the other regions use this sequence number to ensure that they have not missed any batches from that region. Thus, each region is eventually able to reconstruct the complete local log from every other region, potentially with some delay in the event of a network partition. Regions use deterministic transaction processing (see Section 2) to replay the local log from other regions in parallel (at the same speed as the original processing), while ensuring that their copy of the data ends up in the same final state as the original region that sent this log.

Data replication across regions is not strictly necessary in SLOG since the master region for a data item must oversee all writes and linearizable reads to it. However, by continuously replaying local logs from other regions, the system is able to support local snapshot reads of data mastered at other regions at any desired point in the versioned history. Furthermore, by keeping the replicas close to up to date, the process of dynamically remastering a data item is cheaper (see Section 3.4).

If there are n regions in the system, each region must process its own input log, in addition to the $(n-1)$ input logs of the other regions[2]. If all transactions are single-home, then it is impossible for transactions in different logs to conflict with each other, and the processing of these input logs can be interleaved arbitrarily without risking violations of strict serializability or replica non-divergence guarantees. However, achieving strict serializability in the presence of multi-home transactions is not straightforward, since coordination is required across regions.

In Sections 3.2 and 3.3, we explain how SLOG processes transactions in more detail. We start by discussing single-home transactions, and then discuss multi-home transactions.

3.2 (Assumed) Single-home transactions

Data is assigned home regions in *granules* which can either be individual records, or a sorted range of records. SLOG associates two pieces of metadata with each granule: the identifier of its master region at the time it was written, and a count of the total number of times its master has (recently) changed. This metadata is stored in the granule header and replicated with the data granule. The counter is used to ensure correctness of remastering (Section 3.4) and never needs to exceed the number of regions in the deployment. The counter cycles back to 0 after it reaches a max count, and an extra bit is used to detect wrap-around, so the number of bits needed for the counter is always one more than the number of bits needed for the region identifier. Thus, these two pieces of metadata can be combined into a single 8 bit integer for deployments across 8 regions or fewer. Each region contains a distributed index called the "Lookup Master" which maintains a mapping of granule ids to this 8-bit value in the granule header. This index is asynchronously updated after a metadata change and thus may return stale metadata.

Clients can send transactions to the nearest region whether or not it is the home for the data accessed by the transaction. As described in Section 2, we assume that a region can determine the precise set of granules that will be accessed by a transaction, before processing it, either directly or through the advanced techniques we cited.

When a region receives a transaction to process, it sends a message to its Lookup Master to obtain its cached value for the home of each granule accessed by the transaction. The returned locations are stored inside the transaction code. If every granule is currently mapped to the same home region, the transaction becomes initially assumed to be a single-home transaction, and is shipped to that region. Otherwise, it is assumed to be multi-home, and handled according to the algorithm described in Section 3.3. This functionality is shown in the ReceiveNewTxn pseudocode in Figure 2.

Once the (assumed) single-home transaction reaches its home region, it is appended into an in-memory batch of transactional input on the server at which it arrives, and this server appends this batch to that region's input log via Paxos. This corresponds to the

[2]For ease of exposition, we will assume that each region contains a complete copy of all data stored across the system. SLOG can also work correctly with incomplete replication.

```
function ReceiveNewTxn(txn):
masterInfo = ∅
foreach granule in txn.readSet ⋃ txn.writeSet
    masterInfo = masterInfo ⋃ LookupMaster.Lookup(granule)
txn.masterInfo = masterInfo
if (num of unique regions in txn.masterInfo == 1)
    txn.singleHome = true
    send txn to InsertIntoLocalLog(txn) of that home region
else
    txn.singleHome = false
    send txn to the multi-home txn ordering module
    //ordering module orders all multi-home txns and calls
    //InsertIntoLocalLog(txn) at every region in this order

function InsertIntoLocalLog(txn):
if (txn.singleHome == true)
    append txn at end of localLogBatch
else
    accessSet = ∅
    foreach <granule,granule.homeRegionID> in txn.masterInfo
        if (granule.homeRegionID == this.regionID)
            accessSet = accessSet ⋃ granule
    if (accessSet ≠ ∅)
        t = new LockOnlyTxn(txn.ID,accessSet)
        append t at end of localLogBatch
if (isComplete(localLogBatch))
    batchID = insert localLogBatch into Paxos-managed local log
    call ReceiveBatch(localLogBatch, batchID) at each region
    init new localLogBatch

function ReceiveBatch(batch, batchID):
localLogs[batch.regionID][batchID] = batch
prevID = largest batch ID in global log from batch.regionID
while (localLogs[batch.regionID][prevID+1] ≠ null)
    append localLogs[batch.regionID][prevID+1] into global log
    prevID = prevID + 1

Lock manager thread code that continuously runs:
txn = getNextTxnFromGlobalLog() //block until exists
if (txn.isSingleHome == false and txn.lockOnlyTxn == false)
    ExecutionEngine.process(txn) //don't request locks yet
else
    Call RequestLocks(txn) //pseudocode in Figure 6

Code called after all locks for single-home txn acquired:
ExecutionEngine.process(txn)

Execution engine code called prior to read/write of granule:
if (don't have lock on granule) //can happen if txn is multi-home
    Block until have lock //wait for needed lockOnlyTxn
    //will always eventually get lock since SLOG is deadlock-free
if (txn.masterInfo[granule] ≠ granule.header.masterInfo)
    RELEASE LOCKS AND ABORT TRANSACTION

Execution engine code run at end of every transaction:
ReleaseLocks(txn)
```

Figure 2: Pseudocode for core SLOG functionality

beginning and end of InsertIntoLocalLog in Figure 2. A separate Paxos process interleaves batches from the local log with batches from the local logs that are received from other regions in order to create that region's view of the *global log*. This is shown in the ReceiveBatch function. The pseudocode for InsertIntoLocalLog and ReceiveBatch in Figure 2 are written as if they are being called on the region as a whole. The details of how these functions are mapped to individual machines at a region are not shown, to make the pseudocode easier to read.

Processing of transactions by the different servers within that region works similarly to distributed deterministic database systems

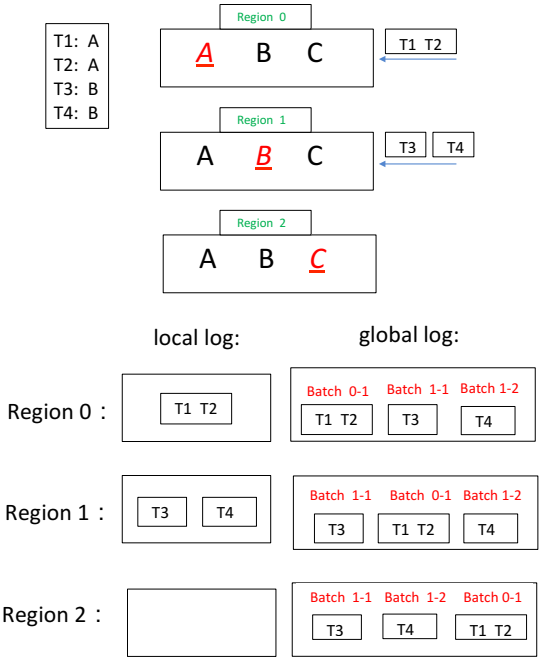


Figure 3: Single-home transactions in SLOG.

such as Calvin: data is partitioned across servers, and each server reads transactions from the same global log and requests locks on any data present in its local partition in the order that the transactions making those requests appear in the log [69, 71, 87] (see the lock manager code in Figure 2). Locks are granted in the order they are requested, which, when also ordering lock requests by the order in which transactions appear in the global log, makes deadlock impossible [68]. The only difference relative to traditional deterministic database systems is that prior to reading or writing a data granule, the metadata associated with that granule must be checked for the current home and counter values. If the counter that is found is different than the counter that was returned from the Lookup Master when the transaction was originally submitted to the system, then the transaction is aborted and restarted.

Deterministic systems ensure that all data progress through the same sequence of updates at every replica, without runtime coordination. Since home metadata is part of the data granule, the metadata is updated at the same point within the sequence of updates of that granule at every region. Therefore, any assumed single-home transaction that is not actually single-home will be exposed as non-single-home independently at each region (i.e., each region will independently, without coordination, observe that it is not single-home, and will all agree to abort the transaction without any cross-region communication). The transaction is then restarted as a multi-home transaction. Similarly, if an assumed single-home transaction is indeed single-home, but the assumed home is incorrect, all regions will see the incorrect home and counter annotation and independently agree to abort the transaction.

In SLOG-B, the client is notified that a transaction has committed as soon as the first region commits it. For SLOG-HA, the client cannot be notified of the commit until the batch of the input log that contains this transaction has been replicated to a configurable number of nearby regions. Since only the input is replicated, replication latency can be overlapped with transaction processing.

Figure 3 shows an example of this process. Each region contains a complete copy of the database, with its "home" granule shown

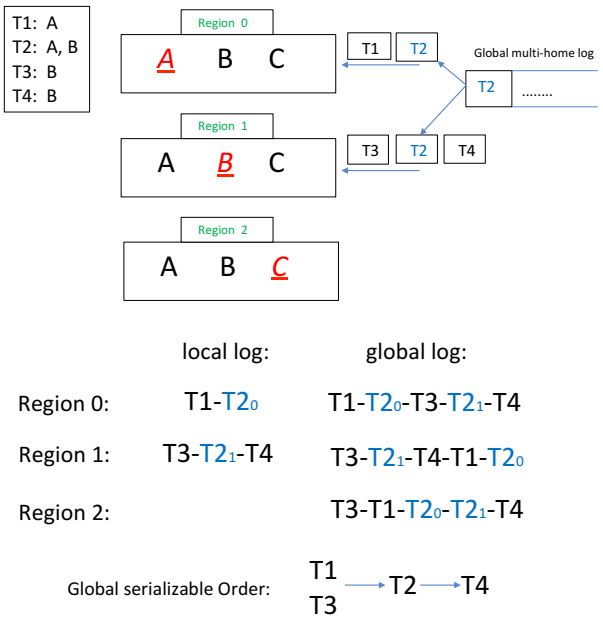


Figure 4: Multi-home transactions in SLOG.

in red and underlined. T1 and T2 update granule A and therefore are sent to region 0 (batch 0-1 is created that contains T1 and T2); T3 and T4 update granule B and are sent to region 1 (which places them in separate batches — T3 is in batch 1-1, and T4 in batch 1-2). Region 0 sends batch 0-1 to the other two regions, and region 1 sends batch 1-1 and batch 1-2 to the other two regions. Thus all three batches appear in the global log of all three regions. However, the order in which these batches appear in the three respective global logs is different. The only guarantee is that 1-2 will appear after 1-1, since they originated from the same region.

If all transactions are single-home, then it is guaranteed that each region's local log will access a disjoint set of database system granules (i.e., transactions across local logs do not conflict with each other). Therefore, the limited degree to which the global logs are ordered differently across different regions will not cause replica divergence. However, if there exist multi-home transactions in the workload, these differences may result in replica divergence. For example, if T2 accessed both A and B (instead of just A in our example above), the order of processing T2 and T3 now makes a difference since they both access B. Since T2 and T3 may appear in different orders in the global logs, there is a risk of region divergence. We will now discuss how SLOG deals with this problem.

3.3 Multi-home transactions

All multi-home transactions, no matter where they originate, must be ordered with respect to each other. A variety of global ordering algorithms could be used, such as (1) Using a single ordering server [19], (2) Running Paxos across all regions [87], or (3) Sending all multi-home transactions to the same region to be ordered by the local log there. Our current implementation uses the third option for simplicity; however, the second option is more robust to failure or unavailability of an entire region. The last line of ReceiveNewTxn in Figure 2 is vague enough to allow for either option. Either way, all regions receive the complete set of multi-home transactions in their proper order via calls to InsertIntoLocalLog.

Recall that the assumed home information for each granule is stored inside the transaction by ReceiveNewTxn. InsertIntoLocalLog checks to see if the region at which the code is running is the assumed home for any of the granules that will be accessed. If so, it generates a special kind of transaction, called a LockOnlyTxn, that consists only[3] of locking the local reads and local writes of the multi-home transaction.

A LockOnlyTxn is similar to a single-home transaction – it only involves local granules, and it is placed in the local log of its home region (alongside regular single-home transactions) which is replicated (eventually) into the global log of every other region. The only difference is that they do not have to include executable code. The code for the multi-home transaction can arrive separately — as part of the local log from the multi-home transaction ordering module, which eventually gets integrated into the global log at every region. LockOnlyTxns exist to specify how the multi-home transaction should be ordered relative to single-home transactions at that region. Depending on where the LockOnlyTxn gets placed in the local log of that region, it will ensure that the multi-home transaction will observe the writes of all transactions earlier than it in the local log, and none of the writes from transactions after it.

This leads to a difference in execution of single-home vs. multi-home transactions. Single-home transactions appear once in a region's global log. The lock manager thread reads the single-home transaction from the global log and acquires all necessary locks before handing over the transaction to an execution thread for processing. In contrast, a multi-home transaction exists in several different locations in a region's global log. There could be an entry containing the code for that transaction, and then separately (usually later in the log), several LockOnlyTxns — one from each region that houses data expected to be accessed by that transaction. The code can start to be processed as soon as it is reached in the global log. However, the code will block whenever it tries to access data for which the corresponding LockOnlyTxn has yet to complete. The bottom half of Figure 2 shows the pseudocode for this process.

Figure 4 shows an example of multi-home transaction processing. The home information and set of transactions submitted to the system are the same as in Figure 3. The only difference is that T2 now accesses A and B and is thus multi-home. Whichever region receives T2 from the client annotates it with the assumed home information (which, in this example, is correct) and sends it to the multi-home ordering module. At Region 0, InsertIntoLocalLog(T2) is called after it has placed single-home T1 into its local log. It therefore places its generated LockOnlyTxn for T2 after T1. LockOnlyTxns for a given multi-home transaction are shown in the figure with the subscript of the region that generated it[4]. InsertIntoLocalLog(T2) is called at Region 1 between placing single home transactions T3 and T4 into its local log and thus places the LockOnlyTxn for T2 it generates there. InsertIntoLocalLog(T2) is also called at Region 2 but no LockOnlyTxn is generated since it was not listed as the home for any data accessed by that transaction. The local logs are replicated to every other region and are interleaved differently, which results in T2's LockOnlyTxns being ordered differently at each region. This is not problematic since LockOnlyTxns always access disjoint data and thus commute.

Regions do not diverge since (1) the global log at each region maintains the original order of the local logs that it interleaves, (2) local logs from different regions access disjoint data (unless the

[3] In some cases, a LockOnlyTxn may include a limited amount of code, such as non-dependent reads/writes of the locked data.

[4] The global log entries that contain T2's code are not shown in the figure, since the code blocks anyway until the LockOnlyTxns are reached. Furthermore, in some cases, code can be included in LockOnlyTxns, which makes the separate shipping of transaction code unnecessary.

home for a data granule has moved – see Section 3.4), and (3) the deterministic processing engine ensures that transactions are processed such that the final result is equivalent to the result of processing them in the serial order present in that region's global log.

3.3.1 Proof of strict serializability

Serializability theory analyzes the order in which reads and writes occur to data in a database. This order is usually called a schedule [29], history [63], or log [16]. Serializability proofs take the form of proving that a given schedule, in which read and writes from different transactions may be interleaved with each other, is equivalent to a serial schedule. In replicated systems, the equivalence must be to a 1-copy serial schedule [10].

Define the relationship T_i reads-x-from T_j as done by Bernstein and Goodman [14]: T_i reads-x-from T_j holds if transaction T_j writes to x, and no other transaction writes to it before T_i reads from it. Two schedules are equivalent if they have the same set of reads-from relationships — for all i, j, and x, T_i reads-x-from T_j in one schedule iff this relationship holds in the other [14].

A serial schedule is one-copy equivalent (1-serial) if for all i, j, and x, if T_i reads-x-from T_j then T_j is the last transaction prior to T_i that writes to any copy of x [10, 14].

SLOG has one global log per region. Even though each global log contains the same set of transactions, each region may process its global log according to a different schedule.

LEMMA 3.1. **All region schedules are equivalent**
A SLOG region requests and acquires locks in the order that transactions appear in its global log. Therefore, when a transaction T_i reads x, the T_j for which T_i reads-x-from T_j will hold is the closest previous transaction to T_i in the global log that writes x. Since there is only one master region for a given x, and all reads and writes to x are found in the local log for its master region, this T_j will come from the same local log as T_i. Thus, the T_j for which T_i reads-x-from T_j will hold will be the same in all global logs, since each global log orders batches from any given local log identically.

LEMMA 3.2. **All serializable schedules in SLOG are 1-copy serializable**. *We already explained in Lemma 3.1 that if T_i reads-x-from T_j holds, then T_i and T_j are in the same local log and T_j is the closest previous transaction to T_i in that local log that writes x. It is impossible for any other local log to contain a transaction that writes to x since only the master region for a data item can include transactions that write to x in its local log.*

Less formally: a 1-serial schedule represents a serial execution of transactions in which the replicated copies of each data item behave like a single logical copy of that data item [10]. In SLOG, this is enabled by ensuring that all writes and reads (excluding read-only snapshot transactions running at a reduced consistency level) to a data item (granule) are managed by its master.

Since all region schedules are equivalent and all serializable schedules in SLOG are 1-copy serializable, all we have to prove is that the schedule from any region is strictly serializable. This is straightforward since any schedule produced by a 2PL implementation, if locks are held until after the transaction commits, and all reads read the value of the most recent write to that data item, then the resulting schedule is strictly serializable [16, 64]. SLOG's deterministic locking scheme acquires all locks prior to processing a transaction and releases all locks after commit. Thus it is a form of 2PL.

SLOG actually guarantees external consistency [37] which is slightly stronger than strict serializability. External consistency enforces an ordering of all transactions — even concurrent ones — such that the execution is equivalent to executing the transactions serially in commit time order. SLOG commits conflicting transactions in the order they appear in each local log (but does not order commit times across local logs). Since all conflicting transactions appear in the same local log, the order in the log specifies their equivalent serial order.

3.4 Dynamic remastering

As access patterns change, it may become desirable to move the home of a data granule, which we refer to as a "remaster" operation. Previous work has shown that simple algorithms are sufficient for deciding when to remaster data. For example, PNUTS changes a data item's master region if the last 3 accesses to it were from a region other than its master [20]. Tuba similarly uses a simple cost model based on recent accesses to decide when to remaster [9]. The challenge in SLOG is not *when* to remaster (SLOG uses the PNUTS heuristic). Rather the challenge is *how* to remaster data without bringing the system offline, while maintaining strict serializability and region non-divergence guarantees and high throughput.

As specified in Section 3.2, SLOG stores two metadata values with each data granule: the identifier of its master region at the time it was written, and a count of the number of times its master has changed. A remaster request modifies both parts of this metadata, and can, for the most part, be treated similarly to any other write request. The request is sent to the home region for that granule, which will insert the request in its local log, which will eventually cause the request to appear in the global logs of all regions. When each region processes this request in its global log, it updates the granule metadata. The Lookup Master of that region is also asynchronously updated to reflect the new mastership information.

A remaster request writes to a single data granule and is thus a single-home transaction. It is sent to its current home region potentially concurrently with other requests to access data within the granule. The order that these requests are placed in the local log of this home region determines which ones will be successful. Transactions that are placed in the local log after the remaster request will see the new granule metadata when they access it, and will abort and resubmit to the new master.

Remastering causes a particular danger to the non-divergence guarantees of SLOG. As mentioned in Section 3.2, the first SLOG region to receive a transaction annotates it with the (home,counter) metadata that was returned from its LookupMaster cache for every granule that it accesses, and stores this cached metadata inside the transaction code. Regions use this cached information to decide whether they are responsible for handling it entirely (if it is single-home) or creating a LockOnlyTxn for it (multi-home). Once the LookupMaster at a region is updated after the region has processed the remaster request, it will start annotating transactions with the new master information, even though other regions may not have processed the remaster request yet. These (up-to-date) annotations will cause the transaction to be placed into the local log of the *new* home region for that granule. However, the remaster request itself is processed by the local log of the *old* home region. This leads to a potential race condition across the local logs. For example, assume a transaction, T, accesses the remastered granule and is sent to the new master region. Some regions may place the log batch that contains T before the log batch that contains the remaster request, while other regions may place it afterwards. This is possible because the local logs from different regions can be interleaved differently in the global log at different regions. As long as T and the remaster request are in different local logs, no assumptions can be made about their relative order in each region's global log. It is true that the remaster request was submitted to SLOG first, and must be completed by at least one region before T can possibly come into

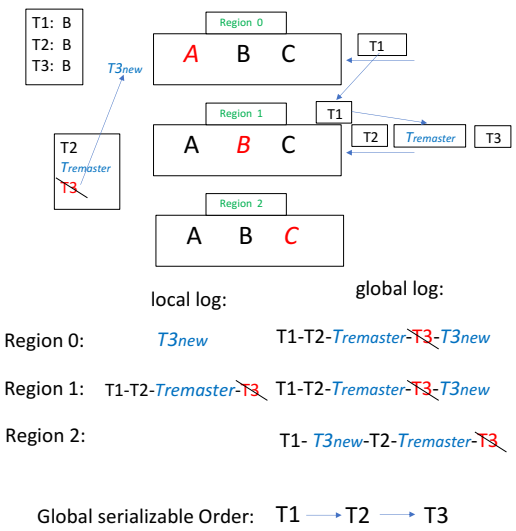


Figure 5: Data remastering in SLOG.

```
Lock manager thread: RequestLocks(txn):
bool can_execute_now = true;
foreach granule in txn.readSet ⋃ txn.writeSet
    if (inLocalPartition(granule)) //a region may have >1 partition
        if (granule.counter > storage.getCounter(granule))
            can_execute_now = false;
        else if (granule.counter < storage.getCounter(granule))
            RELEASE LOCKS AND ABORT TRANSACTION
        else //counters are the same
            can_execute_now = can_execute_now & Lock(granule)
            //Lock(granule) returns true if lock acquired
if (can_execute_now == true)  Send txn to execution thread
else  Queue txn until locks are acquired and counters are correct

Lock manager thread: after releasing locks for txn:
if (txn was a remaster request)
    Wake up txns that are waiting for this remaster request;
else
    Wake up txns that are waiting on locks that were released;
```

Figure 6: Pseudocode for dynamic remastering

existence, so most of the time T will appear after the remaster request in every global log. However, this cannot be guaranteed. If the commit or abort decision of T is dependent on how a region interleaves local logs, replica divergence may occur.

An example of this divergence danger is shown in Figure 5. In this example, replica 0 receives transaction T1, and annotates the transaction with the current home for the accessed granule B — region 1 — and sends it there. After replica 1 executes T1, the threshold necessary to remaster granule B to replica 0 is reached, and a remaster transaction $T_{remaster}$ is generated and sent to replica 1. $T_{remaster}$ is thus inserted into replica 1's local log. Meanwhile, replica 1 receives two other transactions that also access B: T2 and T3. The relative location of the $T_{remaster}$ transaction in replica 1's local log is after T2 but before T3. After execution of $T_{remaster}$, the home for B will have changed, and T3 is now in the wrong local log. Since all regions see transactions from the same local log in the same order, all regions will see T3 after $T_{remaster}$ and independently decide to abort it. The end result is that T3 gets aborted and resubmitted to SLOG as $T3_{new}$, which gets placed into the local log of B's new home: region 0. The problem is: local logs from different regions may be interleaved differently. Regions 0 and 1 place the local log from region 0 that contains $T3_{new}$ after the local log entry from region 1 that contains T2. Thus, they see the serial order: T1, T2, and then T3. However, region 2 places the local log from region 0 that contains $T3_{new}$ **before** the local log entry from region 1 that contains T2. Thus, it sees a different serial order: T1, T3, T2. This leads to potential replica divergence.

The counter part of the metadata is used to circumvent this danger. Prior to requesting a lock on a granule, SLOG compares the counter that was written into the transaction metadata by the LookupMaster at the region the transaction was submitted to with the current counter in storage at that region. If the counter is too low, it can be certain that the LookupMaster that annotated the transaction had out of date mastership information, and the transaction is immediately aborted (every region will independently come to this same conclusion). If the counter is too high, the transaction blocks until the region has processed a number of remaster requests for that data granule equal to the difference between the counters.

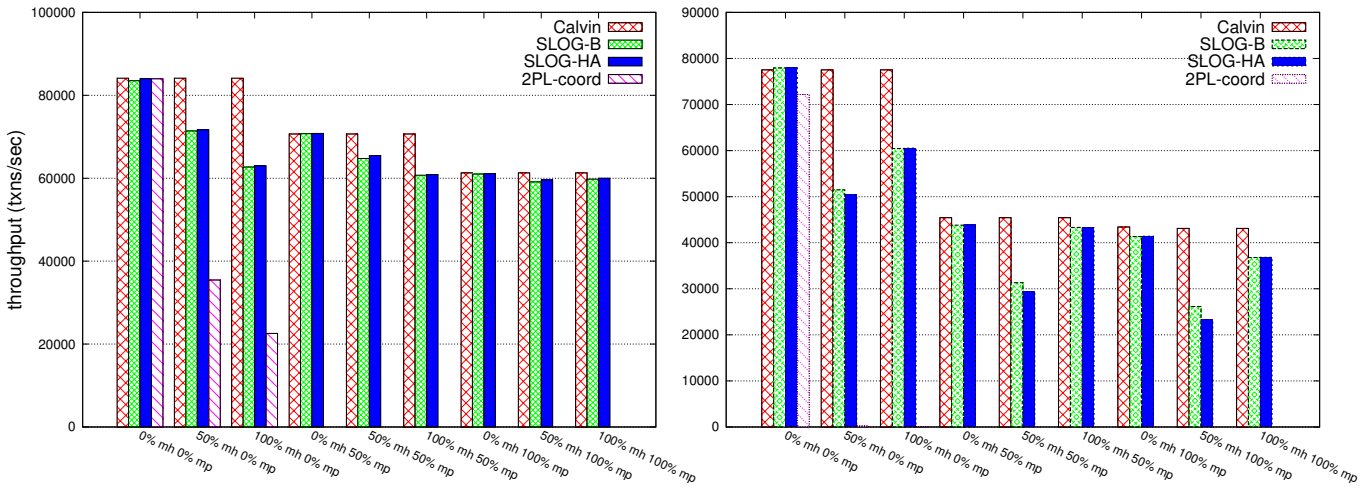
Performing counter comparisons prior to lock acquisition instead of afterwards ensures that the lock is available for the remaster request, which needs the lock to update the header. However, the checks must be repeated after lock acquisition. The pseudocode presented in Figure 6 outlines how counter checks are performed prior to requesting locks, and the end of the pseudocode in Figure 2 shows how these checks are performed again during execution.

4. EXPERIMENTAL RESULTS

Since SLOG relies on deterministic processing to avoid two phase commit and achieve high throughput even when there are many multi-home transactions in a workload, we implemented SLOG via starting with the open source Calvin codebase [2], adding the two Paxos logs per region and the Lookup Master index, and processing transactions according to the specification outlined in Section 3.

As described in Section 1, SLOG is the first system to achieve (1) strict serializability (2) low latency writes (on at least some transactions and (3) high throughput transactions even under high conflict workloads. Throughput and latency are performance measures, which we investigate in this section. We only compare SLOG to other systems that achieve strict serializability. Our two primary comparison points are architectures that achieve low latency writes or high throughput, but not both. More specifically, we built a non-deterministic system that is based on the design of NuoDB: different data is mastered in different regions (NuoDB calls this "chairmanship") and all writes and reads to data are controlled by the chairman. Transactions that access data chaired by different machines require coordination across these machines. NuoDB uses an MVCC algorithm that is susceptible to write skew anomalies (and thus is not serializable). In order to guarantee strict serializability, we use instead a traditional distributed 2PL algorithm where the lock manager for a granule is located in its home partition. We will call this implementation 2PL-coord. We expect 2PL-coord to achieve low-latency writes (reads and writes to data that are mastered nearby can complete without coordination), but poor throughput (as explained in Section 1).

Since we started with the original Calvin codebase, we use Calvin as our second comparison point. By moving almost all commit and replication coordination outside of transactional boundaries, Calvin achieves high throughput even for high conflict workloads [70, 87]. However, the latency of every transaction is at least equal to the latency of multi-region Paxos that is run prior to every transaction processed by the system. We do not expect SLOG to achieve the



(a) Low contention(HOT = 10,000)

(b) High contention(HOT = 30)

Figure 7: YCSB throughput experiments.

same throughput as Calvin for two reasons: (1) Multi-home transactions cause the conflicting intermediate transactions between two of its LockOnlyTxns to potentially block. (2) The ability of the system to remaster granules dynamically causes more checks and potential aborts for each transaction. On the other hand, SLOG is expected to yield much better latency than Calvin for workloads containing many single-home transactions. We thus ran some experiments to more precisely explore these differences in throughput and latency between Calvin and SLOG.

As a third comparison point, we use Cloud Spanner. Cloud Spanner is an entirely separate code base than Calvin, and is far more production-ready and feature-complete. Thus, this is not an apples-to-apples comparison, and raw magnitude performance differences mean little. Nonetheless, information can be gleaned from the performance trends of each system relative to itself as experimental variables are varied, which shed light on the performance consequences of the architectural differences between the systems.

For these experiments, we used system deployments that replicated all data three ways across data-centers. All experiments (except for the Cloud Spanner experiments which we will discuss later) were run on EC2 x1e.2xlarge instances (each instance contains 244 GB of memory, and 8 CPU cores) across six geographical regions: US-East (Virginia), US-East (Ohio), US-West (Oregon), US-West (California), EU (Ireland) and EU (London). The database system was partitioned across 4 machines within each region. For SLOG-HA (the HA mode from Section 3 that is tolerant to failure of an entire region), the US-East(Virginia) region is synchronously replicated to US-East (Ohio); US-West (Oregon) to US-West (California); and EU (Ireland) to EU (London).

We use a version of the Yahoo Cloud Serving Benchmark (YCSB) [21] adapted for transactions, and also the TPC-C New Order transaction to benchmark these systems. In both cases, each record (tuple) is stored in a separate SLOG granule, and we therefore use the terms "record" and "granule" interchangeably in this section. We vary the important parameters that can affect system performance relative to each other: likelihood of transactions to lock the same records (i.e., conflict ratio between transactions), percentage of transactions that are multi-partition within a region, and the ratio of single-home to multi-home transactions.

For YCSB, we use a table of 8.8 billion records, each 100 bytes. Since we have 4 EC2 instances per region, each instance contains a partition consisting of 2.2 billion records. Each region contains a complete replica of the data. Each transaction reads and updates 10 records. In order to carefully vary the contention of the workload, we divide the data set into "hot records" and "cold records". Each transaction reads and writes at (uniformly) random two hot records, and all remaining reads and writes are to cold records. Cold record accesses have negligible contention, so the contention of an experiment can be finely tuned by varying the size of the hot record set.

For the TPC-C dataset, each of the 4 EC2 instances per region contain 240 warehouses (960 warehouses in total), and each warehouse contains 10 districts. In addition to serving as a natural partitioning attribute, warehouses also make for a natural "housing" attribute — the home for a record associated with a particular warehouse should be at the region closest to the physical location of that warehouse (we spread the TPC-C warehouses evenly across the regions in our deployment). Thus, unlike our YCSB experiments, for TPC-C, only multi-partition transactions can be multi-home.

4.1 Throughput experiments

Throughout this section we use the abbreviations sp and mp to refer to single- and multi-partition transactions, and sh and mh for single-home and multi-home. For YCSB data, it is possible for a single partition to contain records that are mastered at different regions. Therefore, it is possible for a transaction to be both single-partition (sp) and multi-home (mh).

We compare SLOG to our primary comparison points: 2PL-coord and Calvin in this section, and to Spanner in Section 4.3. Our first experiment, shown in Figure 7, varies the % mp and % mh parameters. Calvin is unaffected by % mh since it has no concept of a "home" for data. The performance of 2PL-coord is only presented at 0% mp (which is its best case scenario).

Figure 7(a) shows the results of our YCSB experiment under low contention (each partition has 10,000 hot records). When running SLOG with 0% mh, the overhead of multi-home transactions is not present in the workload, and, as expected, all systems perform the same. As more multi-home transactions are added to the workload, the throughput of SLOG and 2PL-coord deteriorates relative to Calvin. In SLOG, the reason is two-fold: first, extra computational resources are consumed by generating the LockOnlyTxns and processing their associated log entries. Second, whenever a LockOnlyTxn appears in the global log of a region prior to a differ-

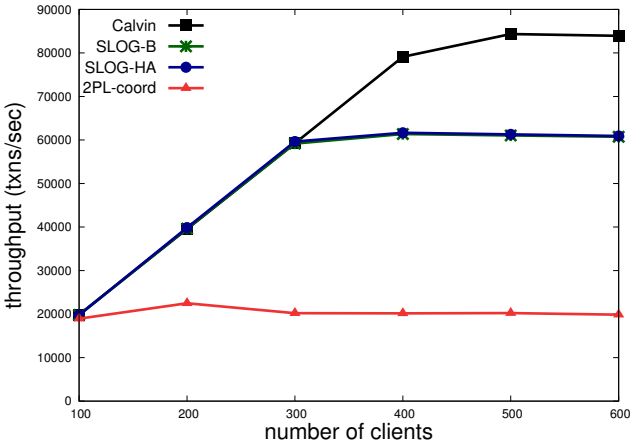


Figure 8: Throughput while varying client count

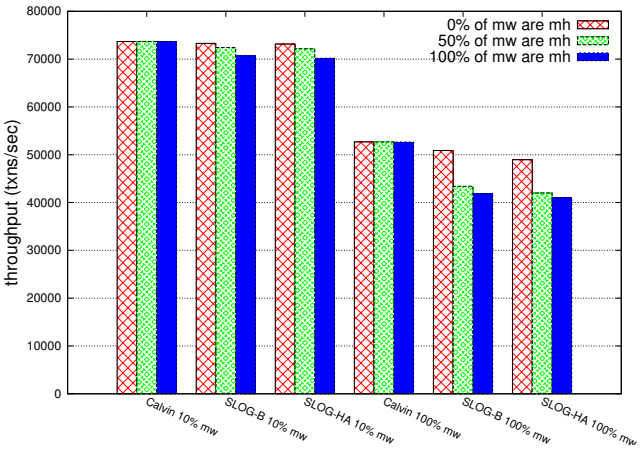


Figure 9: TPC-C New Order throughput experiments

ent LockOnlyTxn from the same original multi-home transaction, that LockOnlyTxn and all transactions that conflict with it (including single-home transactions) must block until the later LockOnlyTxn is processed. Under low contention, the percentage of transactions that are blocked is low; however, under high contention (Figure 7(b)), the contention and reduction of throughput caused by multi-home transactions is higher.

The throughput drop for multi-home transactions is much larger for 2PL-coord than for SLOG. This is because 2PL-coord has a longer window during which contending transactions cannot run. Once a transaction acquires a lock on a remote master, this lock is held for the entire duration of transaction processing, including the time to acquire other locks, execute transaction logic, and perform the commit protocol. In contrast, SLOG only prevents conflicting transactions from running during the window between placing the first and last LockOnlyTxn into its global log. The difference between 2PL-coord and SLOG is more extreme at high contention (Figure 7(b)) where 2PL-coord's throughput is so low at 50% and 100% multi-home that the bars are not visible on the graph. [See Figure 1 for a line graph version of 2PL-coord for this experiment.]

Multi-partition transactions require coordination across the partitions in any ACID-compliant distributed database system. Traditional database systems use 2PC to ensure that all partitions commit. In contrast, deterministic database systems like Calvin reduce this to a single phase that can be overlapped with transaction processing. Nonetheless, this coordination is not zero-cost in Calvin, and throughput decreases with more multi-partition transactions. In SLOG, part of the blocking that is imposed by the multi-home transaction processing code of SLOG is overlapped with the waiting that is necessary anyway because of multi-partition coordination. Therefore, the relative cost of multi-home transactions in SLOG is reduced as the % mp parameter increases.

This also explains why the throughput drop from Calvin to SLOG is more significant when the entire workload is single-partition. However, when there are multi-partition transactions in the workload, the difference between Calvin and SLOG is much smaller, since Calvin also must pay the extra coordination cost for multi-partition transactions. In general, we have found that in practice, when every transaction is multi-partition, then no matter the contention level, and no matter the % multi-home level, the performance of Calvin and SLOG is similar.

Under high contention (Figure 7(b)), SLOG increases in throughput from 50% mh to 100% mh as a result of how the high contention workload is generated. We have 6 regions in this experiment, and each region is the master of 1/6 of all database records, which includes 1/6 of the hot records, and 1/6 of the cold records. For a single-home transaction, both hot records in the transaction come from the same pool of one sixth of all the hot records in the table. However, for a multi-home transaction, the two hot records from the transaction come from different pools — 5/6 of all the hot records. By expanding the pool of hot records that the transaction can draw from, this actually reduces the contention.

The throughput of SLOG-HA is similar to SLOG since deterministic database systems do not need to hold locks during replication, so the synchronous replication to a different region does not cause additional contention, and there is no reduction in concurrency. Furthermore, the computing overhead of managing the replication itself on a per batch basis is negligible.

Figure 8 shows the results of the same experiment as Figure 7(a) at the worst case scenario of SLOG relative to Calvin: 100% multi-home and 0% multi-partition transactions. Each client continuously sends 200 transactions per second to the system. 2PL-coord saturates much earlier than the other systems.

The results of the experiment on New Order transactions on the TPC-C dataset is shown in Figure 9. In TPC-C, warehouse and district records are "hot records", and each New Order transaction needs to read one or two warehouse records, and update one district record. TPC-C specification requires that 10% of New Order transactions need to access two separate warehouses, which may become multi-partition and/or multi-home transactions if those two warehouses are located in separate partitions (which is greater than 75% probability in our 4-partition set-up) or have different home regions. We ran this experiment using both this default TPC-C requirement, and also with 100% of transactions touching multiple warehouses (in order to increase the number of multi-partition and multi-home transactions in the workload). Since only transactions that touch multiple warehouses can be multi-home for this dataset, we have less flexibility in our ability to experiment with multi-home transactions. Thus, for the default TPC-C specification of 10% multi-warehouse transactions, the lines in the figure labeled as "0%/50%/100% of mw are mh" correspond to 0%, 5%, and 10% of all total transactions that are multi-home. However, when we make 100% of transactions multi-warehouse, 0%, 50%, and 100% mh correspond to 0%, 50%, and 100% of all total transactions.

For the 10% multi-warehouse experiment (TPC-C's required configuration), the performance of Calvin and SLOG are similar. These

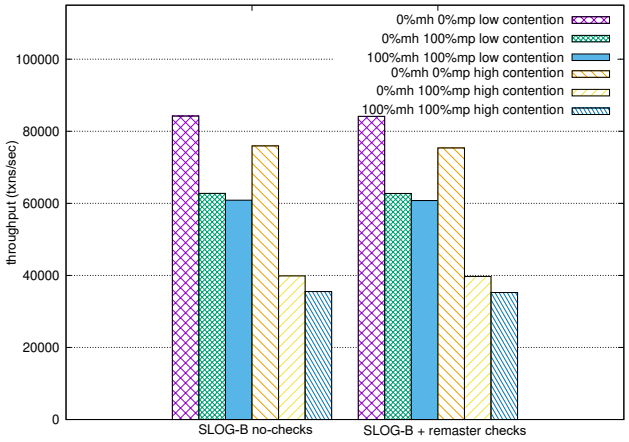


Figure 10: Overhead of dynamic remastering checks

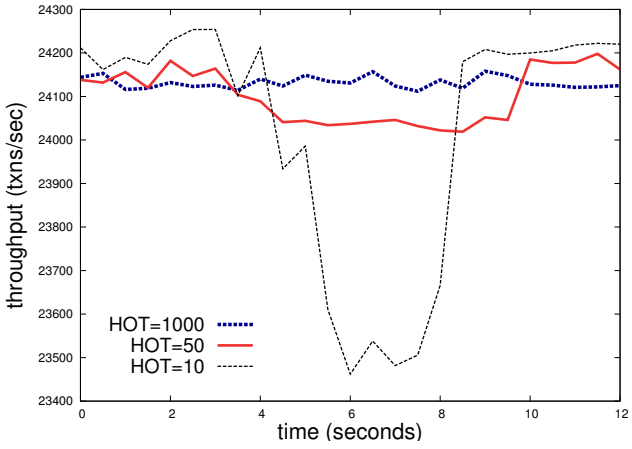


Figure 11: Throughput reduction during remastering

results resemble the YCSB experiment: when there are small numbers of multi-home transactions (a maximum of 10% in this case), the extra overhead in SLOG to process multi-home transactions is less noticeable. At 100% multi-warehouse transactions, Calvin performs slightly better than SLOG when there were 50% or 100% multi-home transactions. These results again resemble the YCSB experiment when there are large numbers of multi-partition **and** multi-home transactions. Once again, the reason why the difference between Calvin and SLOG is not larger is because SLOG is able to overlap the coordination required for multi-partition transactions with the coordination required for multi-home transactions.

4.1.1 Dynamic Remastering

We now investigate the overhead of dynamic remastering of data in SLOG. We first ran an experiment where there are guaranteed to be no actual remastering operations in the workload, so SLOG does not have to perform the checks to see if mastership annotations at transaction submit time remain correct at runtime. As shown in Figure 10, we found that the savings achieved by avoiding these checks is negligible. This is because the home and counter information are stored inside the granule header. Since the granule has to be brought into cache anyway in order to be accessed, the extra CPU overhead of the two extra if-statements is not noticeable.

We also ran an experiment to investigate the actual overhead of record remastering. In this experiment, we measured the temporary throughput reduction during the remastering process of a single record. In order to isolate the throughput effects of just the remastering operation itself, and ignore the throughput benefits that arise from the reduction of multi-home transactions as a result of this remastering, the access-set of each transaction was reduced from 10 to 1. In this experiment, each region has one machine, and we vary the HOT set size as 10, 50 and 1000 records. The remaster operation is initiated approximately five seconds into the experiment.

The experimental results are in Figure 11. Once the remaster transaction is reached and processed in the local log of the region at which it was appended, throughput begins to drop because all subsequent transactions that access the same data are annotated with the old master information and will be aborted and resubmitted. At lower contention, this drop is small because there are fewer transactions that update the same record and must be aborted. Only at very high levels of contention is the drop noticeable. We magnified the size of the drop in the figure by having the y-axis labels not start at 0, but the actual percentage drop is no more than 3% even when the HOT set size is 10. The reason why the overhead is small is because transactions are aborted and resubmitted prior to their execution, so the amount of resources wasted on aborted transactions is small. Throughput returns to normal once the Lookup Master cache at each region has been updated to reflect the correct new master, and no new transactions submitted to the system have incorrect annotations and have to be aborted.

4.1.2 Throughput Experiments Conclusions

Of the two expected sources of throughput reduction of SLOG relative to Calvin, we found that only one is significant: the presence of multi-home transactions in the workload. However, the throughput reduction caused by multi-home transactions is softened by the presence of multi-partition transactions in a workload.

There are two categories of workloads: those that are easy to partition by data access and those that are hard to partition. Workloads that are easy to partition also tend to have location locality using the same partitioning function (as we found in our TPC-C experiment). For example, a partitioning scheme based on a group of users, group of products, or group of accounts usually have associated locations with each of these groups. Thus, different partitions will be mastered at different regions based on the access affinity of that partition to a location. Therefore, we expect most workloads to be in one of two categories: either it is mostly single-partition and mostly-single-home, or otherwise both multi-partition and multi-home transactions are common. For both these categories, our experiments resulted in Calvin and SLOG achieving similar throughput even though Calvin has access to all transactions submitted to the system during pre-planning, while SLOG does not. Meanwhile, under high contention, SLOG's throughput is an order of magnitude higher than nondeterministic systems such as 2PL-coord (and, as will be discussed in Section 4.3, also Spanner).

4.2 Latency experiments

Figure 12 shows the cumulative distribution function (CDF) of transaction latency for SLOG-B, SLOG-HA, and Calvin with 0%, 1%, 10% and 100% multi-home transactions for the same experimental setup as Figure 7(a). Figure 13 looks deeper at the 10% multi-home case, showing 50%, 90%, and 99% latency as the number of clients sending 200 transactions per second to the system is varied. In these experiments, single-home transactions originate from a location near their home.

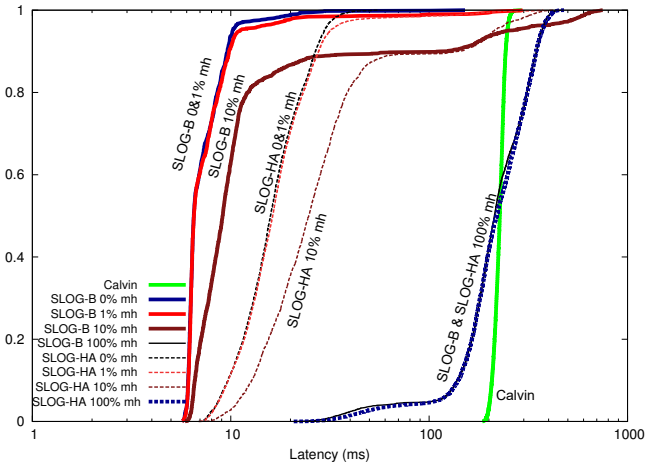


Figure 12: CDF of transaction latency

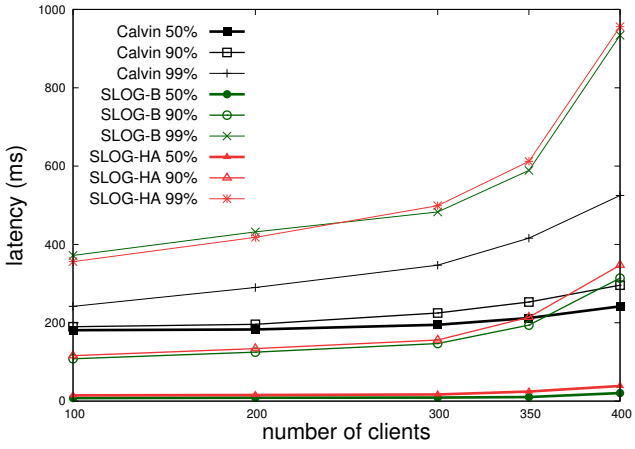


Figure 13: Latency (10% mh) while varying client count

As described above, Calvin places every strictly serializable transaction into a multi-region Paxos log, and thus its latency is dominated by the WAN round-trip times necessary for insertion into this log [32, 87]. To reduce the size of this log, Calvin only stores the batch identifiers in the log. The contents of each batch are replicated separately. Thus, Calvin generally incurs more than 200 ms latency for each transaction.

As expected, SLOG's ability to commit single-home transactions as soon as they are processed at their home region allows SLOG to dramatically reduce its latency relative to Calvin. When the entire workload consists of such transactions (0% mh), almost every transaction achieves less than 10 ms latency — more than an order of magnitude faster than Calvin. The latency of SLOG is dominated by the batch delay prior to transaction processing, as each SLOG machine batches received transactions for 5ms prior to inserting the batch into the Paxos-maintained intra-region local log. In addition, communication with the distributed Lookup Master of a region usually requires network communication, and adds several milliseconds to the transaction latency.

A multi-home transaction being processed at a region cannot complete until all the component LockOnlyTxns arrive from the relevant regions. The farther away the relevant regions, the longer the latency. This delay is potentially longer than the global Paxos delay of Calvin, since Paxos only requires a majority of messages to be received, whereas multi-home transactions require messages from specific (potentially remote) regions. If any region runs behind in its log processing module, this delay gets magnified in the latency of other regions that wait for specific LockOnlyTxn log records from it. This effect is best observed in the 99% latency lines in Figure 13. In addition to the message delay, multi-home transactions also experience log queuing delay at both the remote and local regions, which further increases their latency.

SLOG-HA has a higher latency than SLOG-B since a region's input log is synchronously replicated to another region prior to commit. However, this additional latency is less noticeable for higher % mh since replication of log records in deterministic systems can begin prior to transaction execution (since only the input is replicated) and thus the replication latency is overlapped with the other latency caused by multi-home transactions discussed above.

Section 3.1 mentioned that SLOG also supports snapshot read-only transactions (which run at serializable isolation instead of strict serializable). The latency of such queries are similar to the 0% mh line in Figure 12 since they are processed at a single region.

4.3 Comparison to Spanner

Spanner is similar to SLOG in that it also supports strictly serializable transactions on top of geo-replicated storage. However, the performance profile of Spanner is substantially different. To investigate these differences, we ran the same YCSB benchmark on Cloud Spanner. As mentioned above, Spanner is a different (and more production-ready) codebase relative to SLOG, and runs on a different cloud platform than our SLOG codebase, so comparing the absolute performance numbers has little meaning. However, the differences in performance trends can yield insight into the consequences of the architectural differences between the systems.

Since Google's cloud does not have equivalent regions as AWS, we experimented with two different region-sets for Spanner: (1) nam3 where replicas are in Northern Virginia and South Carolina (and thus closer together than our Amazon regions) and (2) nam-eur-asia1 where replicas are more globally distributed (US, Belgium, and Taiwan). However, Cloud Spanner currently refuses to allow remote regions to be able to accept writes and participate in the Paxos protocol (due to the latency increase and throughput reduction this entails). Thus, the regions in Europe and Asia in nam-eur-asia1 are read-only and stale, and writes require communication only within a 1000 mile radius in the US. Therefore, we found nam3 and nam-eur-asia1 to have similar performance, and we report only nam3 in the figures (they are slightly more favorable to Spanner). We generated the workload for Spanner from 6 n1-standard-8 machines in Northern Virginia, each machine with 8 vCPUs and 30GB of RAM. We deployed SLOG on the most similar type of instance available on EC2: c4.2xlarge instances with 15GB RAM and 8 CPU cores. Spanner performed poorly on the large YCSB dataset from the previous experiments in a 4-node configuration, so we reduced the size of the YCSB dataset for both Spanner and SLOG to 40,000,000 records.

Cloud Spanner did not allow us to control how data was partitioned across the four nodes in each region. However, since our workload accessed 10 random tuples from the dataset, it must be assumed that virtually all transactions are multi-partition (since the chance that all 10 tuples come from the same 1 out of 4 partitions is very low). Thus, we compare Spanner against the 100% mp versions of SLOG. Figure 14 presents normalized results where the throughput of SLOG and Spanner at different contention levels is displayed as a fraction of their throughput at the lowest contention.

Figure 14 shows that Spanner's throughput decreases rapidly as contention increases. At the highest contention, Spanner's through-

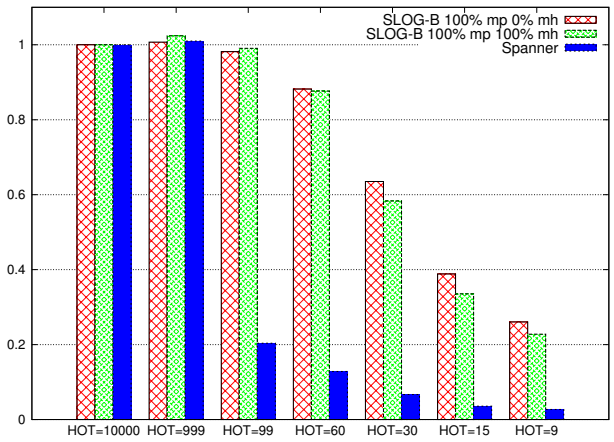


Figure 14: SLOG/Spanner normalized throughput

put decreased by a factor of 37 relative to its throughput at low contention. This is because Spanner does not allow conflicting transactions to run during two phase commit and Paxos-implemented geo-replication. As contention increases, the number of transactions that can run concurrently decreases. If Cloud Spanner allowed synchronous replication over a larger diameter than their current 1000 mile limit, Paxos would take much longer, and Spanner's throughput reduction would be even more severe. In contrast, SLOG maintains strict serializability despite doing replication asynchronously, and does not require two phase commit. Thus, SLOG blocks contending transactions for a much shorter period relative to Spanner[5]. It still experiences decreasing throughput as contention increases, but this decrease is less than a factor of 5 at the highest contention, as opposed to Spanner's factor of 37.

Spanner's latency was similar to Calvin's for the same experiment from Figure 12. This is because write transactions in both Calvin and Spanner run Paxos over the geographic diameter of the deployment. Thus, SLOG has a significant latency advantage over Spanner for write transactions. However, read-only transactions do not require Paxos in Spanner and, if they originate near the Paxos leader, can complete in the same order of magnitude as single-home transactions in SLOG.

5. RELATED WORK

Asynchronous replication is widely used in geo-replicated distributed systems [3, 20, 26, 47, 48, 51, 52, 65, 82], and single-master replication [25, 60] is a popular implementation of asynchronous replication. Most of these systems use weak consistency models which are able to achieve low latency and high throughput, but do not support strictly serializable transactions. In order to support strict serializability, reads and writes must only be processed by the master copy in such "active-passive" architectures. We discussed examples of systems that do this in Section 1 and Section 4. Existing work (prior to SLOG) either do not support transactions that access data at multiple masters [18, 20, 38, 58, 61, 72], rely on physical clock synchronization [62], or otherwise yield poor latency for such transactions (see Section 4).

The challenges of supporting general serializable ACID transactions over a globally-replicated and distributed system has been discussed in detail in prior work [38, 40, 42, 91]. The MDCC [46], TAPIR [94], Replicated Commit [54], and Carousel [93] protocols reduce WAN round-trips; however, they still all incur cross-region latency to commit transactions.

SLOG supports online data remastering which is related to a large body of work in this area [9, 28, 55, 73, 80]. Unlike G-Store [23] and L-Store [49], SLOG's core protocol does not require remastering. Rather, it is an optimization that handles situations where data locality access patterns change. Nonetheless, it is an important feature that is inspired by previous work. SLOG applies these ideas within the particular deterministic system environment in which SLOG is built.

SLOG decomposes transactions into multiple LockOnlyTxns in order to handle multi-region transactions. This approach is related to other research on transaction decomposition [13, 17, 31, 35, 36, 39, 57, 74, 75]. However, LockOnlyTxns in SLOG do not necessarily include transaction code. Rather, their primary purpose is to order deterministic processing of a particular transaction relative to other transactions that may access overlapping datasets. LockOnlyTxns in SLOG only include code when it is straightforward to automatically generate this code from the original transaction. Unlike other transaction decomposition approaches, the LockOnlyTxn approach does not result in SLOG placing any kind of restrictions on the transactions that are being decomposed, and SLOG fully supports ad-hoc transactions.

Transaction Chains [95] achieves low latency geo-replication by chopping transactions into pieces, and using static analysis to determine if each hop can execute separately. In contrast, SLOG does not require any static analysis or independence requirements of transactions. Furthermore, SLOG supports strict serializability, whereas Transaction Chains supports only serializability with read-your-writes consistency.

Previous work explores merging or synchronizing multiple logs to generate a global serialization order for partitioned systems [15, 24, 50]; however, this work does not support geo-replicated configurations. ConfluxDB [18] uses log merging to implement a multi-replica scheme, but does not support multi-master transactions, and only supports snapshot isolation.

There have been several proposals for deterministic database system designs [43, 44, 45, 69, 78, 79, 83, 84, 85, 87]. These papers also make the observation that deterministic database systems facilitate replication since the same input can be independently sent to two different replicas without concern for replica divergence. However, none of those deterministic database systems are able to achieve low-latency geo-replicated transactions.

6. CONCLUSION

Current state-of-the-art geo-replicated systems force their users to give up one of: (1) strict serializability (2) low latency writes (3) high transactional throughput. Some widely-used systems force their users to give up two of them. For example, Spanner experiences poor throughput under data contention, while also paying high-latency cross-region Paxos for each write. SLOG leverages physical region locality in an application workload in order to achieve all three, while also supporting online consistent dynamic "re-mastering" of data as application patterns change over time.

7. ACKNOWLEDGMENTS

We are grateful to Cuong Nguyen, Tasnim Kabir, Jianjun Chen, Sanket Purandare, Jose Faleiro, Tarikul Papon, and the anonymous reviewers for their feedback on this paper. We also thank the paper's shepherd: Phil Bernstein. This work was sponsored by the NSF under grants IIS-1718581 and IIS-1763797.

[5]Spanner's absolute throughput numbers are also significantly lower than SLOG's.

8. REFERENCES

[1] https://fauna.com/.

[2] https://github.com/kunrenyale/calvindb.

[3] Riak. http://wiki.basho.com/riak.html.

[4] D. Abadi. Correctness Anomalies Under Serializable Isolation. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html.

[5] D. Abadi and M. Freels. Serializability vs Strict Serializability: The Dirty Secret of Database Isolation Levels. https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty-secret-of-database-isolation-levels.

[6] D. J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. IEEE Computer, 45(2), 2012.

[7] D. J. Abadi and J. M. Faleiro. An Overview of Deterministic Database Systems. *Communications of the ACM (CACM)*, 61(9):78–88, September 2018.

[8] M. S. Ardekani, P. Sutra, and M. Shapiro. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172, 2013.

[9] M. S. Ardekani and D. B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–381, 2014.

[10] R. Attar, P. A. Bernstein, and N. Goodman. Site Initialization, Recovery, and Backup in a Distributed Database System. *IEEE Transactions on Software Engineering*, 10(6):645–650, Nov. 1984.

[11] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. *PVLDB*, 8(3):185–196, 2014.

[12] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[13] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency Control for Step-decomposed Transactions. *Information Systems*, 24(9):673–698, December 1999.

[14] P. Bernstein and N. Goodman. A Proof Technique for Concurrency Control and Recovery Algorithms for Replicated Databases. *Distributed Computing*, 2(1):32–44, Mar 1987.

[15] P. A. Bernstein and S. Das. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Engineering Bulletin*, Jan-38(1):32–49, March 2015.

[16] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(3):203–216, May 1979.

[17] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–240, Oct. 1992.

[18] P. Chairunnanda, K. Daudjee, and M. T. Ozsu. ConfluxDB: Multi-Master Replication for Partitioned Snapshot Isolation Databases. *PVLDB*, 7(11):947–958, 2014.

[19] J. Chen, Y. Chen, Z. Chen, A. Ghazal, G. Li, S. Li, W. Ou, Y. Sun, M. Zhang, and M. Zhou. Data Management at Huawei: Recent Accomplishments and Future Challenges. In *IEEE International Conference on Data Engineering (ICDE)*, pages 13–24, 2019.

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8:1–8:22, Aug. 2013.

[23] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 163–174, 2010.

[24] K. Daudjee and K. Salem. Inferring a Serialization Order for Distributed Transactions. In *IEEE International Conference on Data Engineering (ICDE)*, 2006.

[25] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[27] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal Consistency and Latency Optimality: Friend or Foe? *PVLDB*, 11(11):1618–1632, 2018.

[28] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, 2015.

[29] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM (CACM)*, 19(11):624–633, Nov. 1976.

[30] J. Faleiro, A. Thomson, and D. J. Abadi. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '14, 2014.

[31] J. M. Faleiro, D. Abadi, and J. M. Hellerstein. High Performance Transactions via Early Write Visibility. *PVLDB*, 10(5):613–624, 2017.

[32] J. M. Faleiro and D. J. Abadi. FIT: A Distributed Database Performance Tradeoff. IEEE Data Engineering Bulletin, 38(1): 10-17, 2015.

[33] J. M. Faleiro and D. J. Abadi. Rethinking Serializable Multiversion Concurrency Control. *PVLDB*, 8(11):1190–1201, 2015.

[34] J. M. Faleiro and D. J. Abadi. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold? In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2017.

[35] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, June 1983.

[36] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, 1987.

[37] D. K. Gifford. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[38] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 International Conference on Management of Data*, SIGMOD '96, 1996.

[39] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. In *Morgan Kaufmann Publishers Inc.*, 1993.

[40] P. Helland. Life Beyond Distributed Transactions: An Apostate's Opinion. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2007.

[41] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[42] C. Humble and F. Marinescu. Trading consistency for scalability in distributed architectures. http://www.infoq.com/news/2008/03/ebaybase, 2008.

[43] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *IEEE SRDS*, 2000.

[44] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, 2010.

[45] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2000.

[46] T. Kraska, G. Pang, M. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems*, 2013.

[47] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[48] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Operating Systems Design and Implementation (OSDI)*, 2012.

[49] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1659–1674, 2016.

[50] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing Global Transactions from Multiple Recovery Log Files in a Partitioned Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[51] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '11, 2011.

[52] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, 2013.

[53] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaram, D. J. Abadi, J. Aspnes, S. Siddhartha, and M. Balakrishnan. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 357–372, Oct. 2018.

[54] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency Multi-datacenter Databases using Replicated Commit. *PVLDB*, 6(9):661–672, 2013.

[55] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic Scale-Out for Partition-Based Database Systems. In *IEEE International Conference on Data Engineering Workshops*, ICDEW '12, pages 281–288, 2012.

[56] H. Moniz, J. a. Leitão, R. J. Dias, J. Gehrke, N. Preguiça, and R. Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, 2017.

[57] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting More Concurrency from Distributed Transactions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[58] F. Nawab, D. Agrawal, and A. El Abbadi. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1221–1236, 2018.

[59] F. Nawab, V. Arora, D. Agrawal, and A. E. Abbadi. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 2015 International Conference on Management of Data*, SIGMOD '15, 2015.

[60] E. Pacitti, P. Minet, , and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1999.

[61] E. Pacitti, P. Minet, and E. Simon. Replica Consistency in Lazy Master Replicated Databases. In *Kluwer Academic, 9(3)*, May 2001.

[62] E. Pacitti, M. T. zsu, and C. Coulon. Preventive Multi-Master Replication in a Cluster of Autonomous Databases. In *Proceedings of Euro-Par*, 2003.

[63] C. Papadimitriou, P. Bernstein, and J. Ronthie. Some Computational Problems Related to Database Concurrency Control. In *Proceedings of the Conference on Theoretical Computer Science*, pages 275–282, 1977.

[64] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[65] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '97, 1997.

[66] S. Proctor. Exploring the Architecture of the NuoDB Database, Part 1. Blog Post. https://www.infoq.com/articles/nuodb-architecture-1.

[67] S. Proctor. Exploring the Architecture of the NuoDB Database, Part 2. Blog Post. https://www.infoq.com/articles/nuodb-architecture-2.

[68] K. Ren, J. Faleiro, and D. J. Abadi. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1583–1598, 2016.

[69] K. Ren, A. Thomson, and D. J. Abadi. Lightweight Locking for Main Memory Database Systems. *PVLDB*, 6(2):145–156, 2012.

[70] K. Ren, A. Thomson, and D. J. Abadi. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *PVLDB*, 7(10):821–832, 2014.

[71] K. Ren, A. Thomson, and D. J. Abadi. VLL: A Lock Manager Redesign for Main Memory Database Systems. VLDB Journal 24(5): 681-705, October 2015.

[72] J. Runkel. Active-Active Application Architectures with MongoDB. https://www.mongodb.com/blog/post/active-active-application-architectures-with-mongodb.

[73] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *PVLDB*, 7(12):1035–1046, 2014.

[74] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, September 1995.

[75] D. Shasha, E. Simon, and P. Valduriez. Simple Rational Guidance for Chopping up Transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 298–307, 1992.

[76] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A Distributed SQL Database That Scales. *PVLDB*, 6(11):1068–1079, 2013.

[77] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[78] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[79] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, June 2013.

[80] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *PVLDB*, 8(3):245–256, 2014.

[81] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, 1994.

[82] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '95, 1995.

[83] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[84] A. Thomson and D. J. Abadi. Modularity and Scalability in Calvin. In *IEEE Data Engineering Bulletin, 36(2): 48-55*, 2013.

[85] A. Thomson and D. J. Abadi. Deterministic database systems. US Patent 8700563, 2014.

[86] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *FAST*, pages 1–14, 2015.

[87] A. Thomson, T. Diamond, P. Shao, K. Ren, S.-C. Weng, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 International Conference on Management of Data*, SIGMOD '12, 2012.

[88] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Transactions on Database Systems (TODS)*, 39(2):11:1–11:39, 2014.

[89] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, 2017.

[90] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, and X. Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 789–796, 2018.

[91] W. Vogels. Eventually Consistent. *Queue*, 6(6):14–19, Oct. 2008.

[92] S.-H. Wu, T.-Y. Feng, M.-K. Liao, S.-K. Pi, and Y.-S. Lin. T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1553–1565, 2016.

[93] X. Yan, L. Yang, H. Zhang, X. C. Lin, B. Wong, K. Salem, and T. Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 231–243, 2018.

[94] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4), Dec. 2018.

[95] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '13, 2013.