An Evaluation of Distributed Concurrency Control

Rachael Harding
MIT CSAIL
rhardin@mit.edu

Dana Van Aken
Carnegie Mellon University
dvanaken@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Increasing transaction volumes have led to a resurgence of interest in distributed transaction processing. In particular, partitioning data across several servers can improve throughput by allowing servers to process transactions in parallel. But executing transactions across servers limits the scalability and performance of these systems.

In this paper, we quantify the effects of distribution on concurrency control protocols in a distributed environment. We evaluate six classic and modern protocols in an in-memory distributed database evaluation framework called Deneva, providing an apples-to-apples comparison between each. Our results expose severe limitations of distributed transaction processing engines. Moreover, in our analysis, we identify several protocol-specific scalability bottlenecks. We conclude that to achieve truly scalable operation, distributed concurrency control solutions must seek a tighter coupling with either novel network hardware (in the local area) or applications (via data modeling and semantically-aware execution), or both.

1. INTRODUCTION

Data generation and query volumes are outpacing the capacity of single-server database management systems (DBMS) [20, 47, 17]. As a result, organizations are increasingly partitioning data across several servers, where each partition contains only a subset of the database. These distributed DBMSs have the potential to alleviate contention and achieve high throughput when queries only need to access data at a single partition [33, 49]. For many on-line transaction processing (OLTP) applications, however, it is challenging (if not impossible) to partition data in a way that guarantees that all queries will only need to access a single partition [22, 43]. Invariably some queries will need to access data on multiple partitions.

Unfortunately, multi-partition serializable concurrency control protocols incur significant performance penalties [54, 49]. When a transaction accesses multiple servers over the network, any other transactions it contends with may have to wait for it to complete [6]—a potentially disastrous proposition for system scalability.

In this paper, we investigate this phenomenon: that is, when does distributing concurrency control benefit performance, and when is distribution strictly worse for a given workload? Although the costs of distributed transaction processing are well known [9, 52], there is little understanding of the trade-offs in a modern cloud computing environment offering high scalability and elasticity. Few of the recent publications that propose new distributed protocols compare more than one other approach. For example, none of the papers published since 2012 in Table 1 compare against timestamp-based or multi-version protocols, and seven of them do not compare to any other serializable protocol. As a result, it is difficult to compare proposed protocols, especially as hardware and workload configurations vary across publications.

Our aim is to quantify and compare existing distributed concurrency control protocols for in-memory DBMSs. We develop an empirical understanding of the behavior of distributed transactions on modern cloud computing infrastructure using a mix of both classic and newly proposed concurrency control protocols, including some from Table 1. We developed a lightweight distributed main-memory DBMS evaluation framework, called **Deneva**, to assess the performance and trade-offs of multiple distributed serializable concurrency control protocols. A unified platform ensuring a fair comparison between protocols and a quantitative lens on the behavior of each across a range of workload conditions. To the best of our knowledge, this is the most comprehensive performance evaluation of concurrency control protocols on cloud computing infrastructure. Deneva is also available as an open source and extensible framework for the transaction processing research community. [1]

Using Deneva, we analyze the behavior of six concurrency control protocols on public cloud infrastructure with a combination of microbenchmarks and a standard OLTP workload. We find that the scalability of all of the protocols is limited. Our results show that certain workload characteristics lend themselves to different existing protocols in a distributed setting. Under low contention and low update workloads, all of the protocols perform well. But for workloads with high update rates, two-phase locking with no waiting outperforms other non-deterministic protocols by up to 54%, and for workloads with high contention, it outperforms them by up to 78%. Deterministic protocols outperform all other protocols under the highest contention levels and update rates for simple transactions, but when workloads include transactions with foreign key lookups, the deterministic protocol is the only one whose performance does not scale as the cluster size increases.

The remainder of this paper proceeds as follows: in Section 2, we present the design and implementation of Deneva, our lightweight framework for evaluating the behavior of distributed concurrency control protocols. In Section 3, we provide an overview of the concurrency control protocols and optimizations we study in this paper. In Section 4, we evaluate each of our protocols and identify their scalability bottlenecks and present possible solutions in Section 5.

Proceedings of the VLDB Endowment, Vol. 10, No. 5

[1] http://www.github.com/mitdbg/deneva

Table 1: A comparison of experimental evaluations from recently published protocols for serializable distributed transactions (**Lock**: two-phase locking, **TS**: timestamp-based protocols, **MV**: multi-version concurrency control, **OCC**: optimistic concurrency control, **Det**: deterministic methods).

	Experimental Comparisons Performed					
Publication	Lock	TS	MV	OCC	Det	None
Tango [7]	✓					✗
Spanner [20]						✗
Granola [21]	✓					
Centiman [25]						✗
FaRM [26]						✗
Warp [27]						✗
MaaT [39]	✓					
Rococo [41]	✓			✓		
Ren et al. [45]	✓			✓		
F1 [47]						✗
Calvin [54]						✗
Wei et al. [58]					✓	
TaPiR [61]	✓			✓		
Lynx [62]						✗
Deneva (this study)	✓×2	✓	✓	✓	✓	

We conclude with a discussion of the related work in Section 6 and our future work plans in Section 7.

2. SYSTEM OVERVIEW

OLTP applications today are ubiquitous, and include banking, e-commerce, and order fulfillment [52]. OLTP DBMSs provide these applications with transactional support: the ability to process transactions, or sequences of multiple operations over a set of multiple, shared data records. In practice, transactions in OLTP applications are (1) short-lived, (2) access a small number of records at a time, and (3) are repeatedly executed with different input parameters [33].

In this study, we investigate the highest ideal in transaction processing: serializable execution. Under this model, transactions behave as if they were executed one-at-a-time against a single copy of database state [9]. Insofar as each transaction maintains application correctness (or integrity) criteria (e.g., usernames are unique), then a serializable execution guarantees application integrity. It is the job of the DBMS's concurrency control protocol to extract the maximum amount of parallelism despite the exacting demands of serializable semantics. In practice, a large number of OLTP databases implement transactions using non-serializable semantics, such as Read Committed or Snapshot Isolation [4]. But since these isolation levels can compromise application integrity, we restrict our primary focus to studying serializable executions.

Since its introduction in the 1970s, there have been a large number of techniques developed for enforcing serializability both in a single-node and a distributed environment. Several decades later, we continue to see new variants of distributed serializable concurrency control in both academic [33, 53] and commercial DBMSs [20, 35] (see Section 6). This diversity makes it difficult to determine when one strategy dominates another. As we show in Table 1, many recent studies restrict themselves to comparing against one or two alternatives (often, two-phase locking). Therefore, the goal of our work is a quantitative evaluation of these protocols on the same framework, both from the classic literature as well as the past decade spanning both pessimistic and timestamp-ordering strategies [11].

To conduct a proper comparison of these protocols, we implemented a lightweight, distributed testing framework for in-memory DBMS, which we discuss in the remainder of this section.

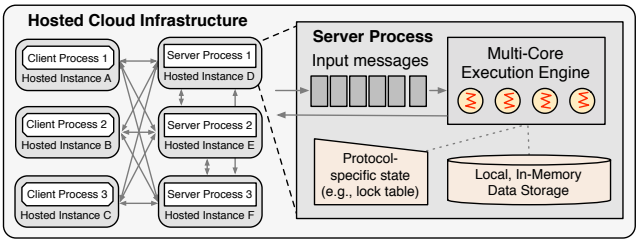


Figure 1: Deneva Framework Architecture – A set of client and server processes deployed on a set of hosted instances (virtual machines or bare-metal servers) on public cloud infrastructure. The multi-threaded, multi-core execution engine maintains the shared-nothing in-memory data store, with auxiliary in-memory data structures for protocol-specific metadata.

2.1 Principles & Architecture

Since we seek to measure the scalability bottlenecks in distributed protocols, we wanted an architecture that allows us to isolate the effect of concurrency control and study it in a controlled environment. Accordingly, we created a framework, called Deneva, that allows us to implement and deploy multiple protocols in a single platform. Deneva uses a custom DBMS engine (as opposed to adapting an existing system) to avoid the overhead of unrelated functionalities. It also ensures that we evaluate the protocols in a distributed environment without the presence of bottlenecks that are endemic to existing implementations.

Figure 1 shows the high-level architecture of Deneva. Deneva provides a set of client and server processes that execute a range of transactional workloads using a variety of pre-defined but extensible protocols. It is a shared-nothing system where each server is responsible for one or more partitions of the data, and no partition is managed by more than one server. Deneva supports distributed transactions across partitions but it does not provide replication or fault tolerance; thus, this investigation is limited to failure-free scenarios. We discuss potential extensions in Section 7.

Before describing Deneva's transaction model, execution model, and server architecture, we note that we designed Deneva for extensible, modular implementations of distributed concurrency control protocols. Adding a protocol requires implementing new transaction coordinator logic, remote procedure calls, and server event handler routines. The system's data storage, networking, and execution components remain the same. In our experience, each protocol takes approximately one week of development time.

2.2 Transaction Model

All transactions in Deneva execute as stored procedures that run on the servers. Each procedure contains program logic intermixed with queries that read or update records in the database. Some protocols—namely those dependent on deterministic execution (e.g., Calvin [54], VoltDB [1], H-Store [33])—require that transactions' read and write sets be known in advance or otherwise calculated on-line via an expensive "reconnaissance" step. Hence, to ensure a more fair and realistic comparison, we perform this reconnaissance step to compute this information for the protocols that need it.

2.3 Execution Model

We arrange clients and servers in a fully connected topology over a set of deployed cloud computing instances. We use `nanomsg` [51], a scalable and thread-safe socket library to communicate between instances using TCP/IP. Unless otherwise specified, the client and server processes are located on different instances. Servers are arranged using consistent hashing [34], and clients are aware of partition mappings to servers, which do not change during execution.

Deneva provides workloads that each exercise different aspects of the protocols (Section 4.1). The framework allows multiple outstanding transactions per client process; we refer to the total number of open client requests as the *offered system load*. When a client wants to execute a transaction, it first generates the input parameters for the target stored procedure according to workload specification. It then sends the request to the server that manages the data partition first accessed by the transaction, which we call the *coordinator*. If a transaction accesses multiple partitions, we call it a *multi-partition transaction* (MPT). When multiple servers are involved in a MPT, we call them *participants*.

2.4 Server-Side Execution

Each server executes requests on behalf of the clients and other servers. When a server receives a new transaction request, it invokes the stored procedure, which will then generate queries that access data either on its local partition or a remote partition managed by another server. If an active transaction aborts due to protocol behavior, the coordinator sends a message to the other participating servers to abort. They will each roll back any changes that the transaction made to its local partition. The coordinator then puts the request back into its work queue. To reduce the likelihood that the request will fail again due to the same conflict when it restarts, coordinator applies an exponential back-off penalty (starting at 10 ms) that specifies how long it should wait before re-executing that aborted request. Once a transaction is complete, the coordinator sends an acknowledgement back to the client and any internal data structures used for that transaction are reclaimed.

Priority Work Queue: Remote procedure calls are handled by a set of I/O threads (in our study, eight per server) that are responsible for marshalling and unmarshalling transactions, operations, and their return values. When a transaction or operation arrives at a server, it is placed in a work queue in which operations from transactions that have already begun execution are prioritized over new transactions from clients. Otherwise, the DBMS processes transactions and operations on a first-come, first-served basis.

Execution Engine: A pool of worker threads (in our study, four per server, each on a dedicated core) poll the work queue and execute concurrent operations in a non-blocking manner. The DBMS executes transactions until they are forced to block while waiting for a shared resource (e.g., access to a record lock) or they need to access data from a remote partition. In the latter case, the DBMS ships transaction parameters necessary to execute remote operations (e.g., which records to read) to the remote server. The worker thread can then return to the worker pool and accept more work. This means that although transactions may block, threads do not. When a transaction is ready to resume execution at a server, its next set of operations are added to the priority queue and executed by the first available worker thread. We opted for this non-blocking event model as it provided better scalability and less thrashing under heavy load [59], albeit at the cost of higher implementation complexity.

Storage Engine: Each server stores data from its partitions using an in-memory hash table that supports efficient primary-key lookups with minimal storage overhead. The engine does not provide logging for durability to avoid the unnecessary overhead. Investigating the impact of recovery and checkpointing mechanisms in a distributed DBMS is an interesting area for future work.To support multiple concurrency control protocols in Deneva, we also implemented a diverse set of protocol-specific data structures, such as local lock tables and validation metadata. We discuss these in detail in the next section but note that we separate the database storage mechanism (i.e., hash index) from these data structures. We chose this system design because it provides modularity in the implementation at a relatively small price to efficiency.

Timestamp Generation: Several of the protocols depend on the use of wall-clock timestamps. We provide timestamps as a basic primitive in the Deneva framework. Each server generates its own timestamps by reading the local system clock, and Deneva ensures that timestamps are unique by appending the server and thread ids to each timestamp. We manage skew between server clocks by syncing the servers using `ntpdate`, which we found does not cause performances variations among the throughput in our evaluation.

3. TRANSACTION PROTOCOLS

We next describe the protocols we study as well as optimizations we apply to improve their performance in a distributed environment. These include classic protocols currently used in real DBMSs and state-of-the-art protocols proposed within the last few years.

3.1 Two-Phase Locking

The first provably serializable concurrency control was *two-phase locking* (2PL) [28]. In the first phase, known as the *growing phase*, is where a transaction acquires a lock for any record that it needs to access. Locks are acquired in either shared or exclusive mode, depending on whether the transaction needs to read or write the data, respectively. Locks in shared modes are compatible, meaning that two transactions reading the same data can acquire the lock for the data simultaneously. Exclusive locks are not compatible with either shared or other exclusive locks. If a piece of data's lock is already held in a non-compatible mode, then the requesting transaction must wait for the lock to become available.

The transaction enters the second phase of 2PL, called the *shrinking phase*, once it releases one of its locks. Once the transaction enters this phase, it is not allowed to acquire new locks, but it can still perform read or write operations on any object for which it still holds the lock. In our implementation, the DBMS holds locks until the transaction commits or aborts (i.e., strict 2PL). We use record-level locks to maximize concurrency [30]. Each server only records which transaction holds the locks for the records in its partition.

2PL implementations differ on how to handle deadlocks by altering the behavior of transactions that attempt to acquire data with conflicting lock types. We now describe the two variants that we study in this paper.

In **`NO_WAIT`**, if a transaction tries to access a locked record and the lock mode is not compatible with the requested mode, then the DBMS aborts the transaction that is requesting the lock. Any locks held by the aborted transaction are released, thereby allowing other conflicting transactions to access those records. By aborting transactions whenever they encounter a conflict, `NO_WAIT` prevents deadlocks. But not every transaction would have resulted in a lock dependency cycle, and thus many aborts may be unnecessary.

The **`WAIT_DIE`** protocol is similar except it attempts to avoid aborts by ordering transactions based on the timestamps that the DBMS assigned them when they started. The DBMS queues a conflicting transaction as long as its timestamp is smaller (older) than any of the transactions that currently own the lock [11]. If a transaction attempts to acquire a shared lock on a record that is already locked in shared mode, it can bypass the queue and immediately add itself to the lock owners; while this may penalize writing transactions, we found this optimization to be beneficial.

One could also implement deadlock detection by checking for cycles within transaction accesses [30]. This process requires substantial network communication between servers to detect cycles [60].

Hence, this deadlock detection mechanism is cost prohibitive in a distributed environment and we did not include it in our evaluation.

3.2 Timestamp Ordering

Another family of concurrency control protocols relies on timestamps. In these protocols, unique timestamps are used to order the transactions and prevent deadlock. As with all protocols in this paper that require timestamps for operation, timestamps are generated using Deneva's mechanism described in Section 2.4.

In the most basic algorithm of this class (**TIMESTAMP**), transaction's operations are ordered by their assigned timestamp [11]. The transaction's timestamp dictates access to a record. Unlike WAIT_DIE, transactions cannot bypass a waiting queue of transactions that the DBMS stores per-server in the protocol-specific data structures. This protocol avoids deadlock by aborting transactions with a smaller (older) timestamp than the transaction that currently has an exclusive hold on a record. When a transaction restarts, the DBMS assigns it a new, unique timestamp based on the system clock at the time that it was restarted.

In contrast, multi-version concurrency control (**MVCC**) maintains several timestamped copies of each record [12]. This enables reads and writes to proceed with minimal conflict, since reads can access older copies of the data if a write is not committed. In Deneva, we store multiple copies of each record in the in-memory data structure and limit the number of copies stored, aborting transactions that attempt to access records that have been garbage collected.

3.3 Optimistic

Optimistic concurrency control (**OCC**) executes transactions concurrently and determines whether the result of transaction execution was in fact serializable at the time of commit [38]. That is, before committing, the DBMS validates the transaction against all the transactions that committed since the transaction started or are currently in the validation phase. If a transaction can commit, the DBMS copies the local writes to the database and results are returned to the client. Otherwise, it aborts the transaction aborts and destroys any local copies of the data.

We base our version of OCC on the MaaT protocol [39]. The main advantage of MaaT over that of traditional OCC is it that it reduces the number of conflicts that lead to unnecessary aborts. Deneva's implementation requires three protocol-specific data structures: (1) a private workspace that tracks the transaction's write set, (2) a per-server table, called the *timetable*, that contains the range (i.e., the upper and lower bounds) of each active transaction's potential commit timestamp, and (3) per-record metadata that stores two sets—a set of reader IDs and a set of writer IDs—for transactions that intend to read or write the record, and the commit timestamps of the last transactions that accessed the record.

Before starting, each transaction acquires a unique transaction ID and then adds it to the server's local timetable with its commit timestamp range initialized to a lower bound of 0 and an upper bound of ∞. During transaction execution, the DBMS copies each updated record into a private workspace. This allows transactions to proceed without blocking or spending time checking for conflicts while executing. The DBMS updates the per-record metadata each time a record is accessed by the transaction. When a read occurs, the DBMS adds the transaction ID to the set of reader IDs, and copies both the reader ID set and the most recent read commit timestamp into the private workspace for reference during validation. For write operations, the system adds the transaction's ID to the record's writer ID set, and the IDs from both the reader and writer sets are copied, as well as the most recent read commit timestamp. The first time a transaction reads or writes a record on a remote server, the DBMS creates an entry in the remote server's timetable for that transaction.

OCC's validation phase occurs when the transaction finishes and invokes the atomic commitment protocol (see Section 3.5). Starting with the initial timestamp ranges stored in the timetable, each participating server adjusts the timestamp range of the validating transaction and the ranges of those transactions present in its reader/writer ID sets such that the ranges of conflicting transactions do not overlap. The DBMS updates any modified ranges in the timetable, and the transaction enters a validated state in which its range can no longer be modified by other transactions. If at the end of this process the timestamp range of the validating transaction is still valid (i.e., the timestamp's upper bound is greater than its lower bound), then the server sends the coordinator a COMMIT validation decision. Otherwise, it sends an ABORT decision. The coordinator collects the validation votes from all participating servers and commits the transaction if every server voted to commit. The coordinator then notifies the other servers of the final decision. At this point the transaction's ID can be removed from the reader and writer sets of all its record accesses, and, if the transaction commits, the record's read or write timestamp can be modified.

3.4 Deterministic

Deterministic scheduling is a recent proposal as an alternative to traditional concurrency control protocols [45]. Centralized coordinators that decide on a deterministic order of the transactions can eliminate the need for coordination between servers required in other concurrency control protocols. Thus, deterministic protocols have two benefits. First, they do not need to use an atomic commitment protocol (Section 3.5) to determine the fate of a transaction. Second, they support simpler replication strategies.

We based Deneva's deterministic lock coordinator (**CALVIN**) implementation on the Calvin framework [54]. All clients send their queries to a distributed coordination layer comprised of *sequencers* that order the transactions and assign them a unique transaction id. Time is divided into 5 ms epochs. At the end of each epoch, the sequencers batch all of the transactions that they collected in the last epoch and forward them to the servers that manage the partitions that contain the records that the transaction wants to access. At each server, another component called the *scheduler* acquires record-level locks from each sequencer in a predetermined order. That is, each scheduler processes an entire batch of transactions from the same sequencer in the transaction order predetermined by that sequencer before moving to the next batch from the another sequencer. If the transaction cannot acquire a lock, then the DBMS queues it for that lock and the scheduler continues processing.

The DBMS can execute a transaction once it acquired all the record-level locks that it needs. Execution proceeds in phases. First, the read/write set of the transaction is analyzed to determine a set of *participating servers* (i.e., all servers that read or update records) and *active servers* (i.e., all servers that perform updates). Next, the system performs all of the local reads. If data from these local reads are needed during transaction execution at other servers (e.g., to determine whether a transaction should abort based on values in the database), then the system forwards them to the active servers. At this point, non-active servers that perform no updates are finished executing and can release their data locks. Once an active server receives messages from all participants it expects to receive data from, it applies the writes to its local partition. At this stage, active servers deterministically decide to commit or abort the transaction, and the locks are released. The servers send their responses to the sequencer, which sends an acknowledgement to the client once all responses have arrived.

Read-only transactions in CALVIN contain no active servers and therefore complete after the local read phase. This means that read-only transactions only require messages between the sequencers and servers, but not between the servers themselves.

Because CALVIN is a deterministic algorithm, it requires a transaction's read/write sets to be known *a priori*. If the read/write sets are unknown, then the framework must calculate them at runtime. This causes some transactions to execute twice: once speculatively to determine the read/write sets, then again to execute under the deterministic protocol. During this *reconnaissance step*, the transaction can proceed without acquiring locks. But if one of the records changes between the reconnaissance and execution steps, the transaction must abort and restart the entire process.

In our version of CALVIN in Deneva, we implement the sequencer, scheduler, and work layers as separate threads that are co-located on the same servers. We replace two of the worker threads with a sequencer thread and a scheduler thread.

3.5 Two-Phase Commit

To ensure that either all servers commit or none do (i.e., the atomic commitment problem [52, 9]), all of the protocols that we implemented in Deneva (except for CALVIN) employ the two-phase commit (2PC) protocol. The system only requires 2PC for transactions that perform updates on multiple partitions. Read-only transactions and single-partition transactions skip this step and send responses immediately back to the client. OCC is an exception to the read-only transaction rule, since it must validate its reads as well.

Once a multi-partition update transaction completes execution, the coordinator server sends a prepare message to each participant server. During this *prepare* phase, participants vote on whether to commit or abort the transaction. Since we do not consider failures in this paper, in all protocols except OCC all transactions that reach the prepare phase commit. In OCC, which performs validation during this phase and may cause the transaction to abort. Participants reply to the coordinator server with a response to commit or abort the transaction. At this point, the protocol enters the second and final *commit* phase. If any participant (including the coordinator) votes to abort, the coordinator server broadcasts an abort message. Otherwise, the coordinator broadcasts a commit message. Once participants receive the final message, they commit or abort and perform any necessary clean up, such as releasing any locks held by the transaction. The participants then send an acknowledgment back to the coordinator server. The coordinator also performs clean up during this phase but can only respond to the client after it receives all of the final acknowledgments from the participants.

4. EVALUATION

We now present our evaluation and analysis of the six concurrency control protocols described in Section 3. Unless otherwise stated, we deployed the Deneva framework on Amazon EC2, using the m4.2xlarge instance type in the US East region. We use a varying number of client and server instances. Each instance contains a set of eight virtualized CPU cores and 32 GB of memory with a rating of "high" network performance (1 ms average RTT).

Before each experiment, table partitions are loaded on each server. During the experiment, we apply a load of 10,000 open client connections per server. The first 60 s is a warm-up period followed by another 60 s of measurements. We measure throughput as the number of transactions successfully completed after the warm-up period. When a transaction aborts due to conflicts determined by the concurrency control protocol, it restarts after a penalization period.

For our first experiments in Sections 4.2 to 4.4, we use a microbenchmark that allows us to tweak specific aspects of an OLTP workload to measure how the protocols perform. We then measure their scalability in Section 4.5 and provide a breakdown of where the DBMS spends time when executing transactions. We then compare the protocols in an operating environment that simulates a wide-area network deployment in Section 4.6. Finally, we finish in Sections 4.7 to 4.9 with experiments that model different application scenarios.

4.1 Workloads

We next describe the benchmarks used in our evaluation.

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) [19] is designed to evaluate large-scale Internet applications. It has a single table with a primary key and 10 additional columns with 100 B of random characters. For all our experiments, we use a YCSB table of ∼16 million records per partition, which represents a database size of ∼16 GB per node. The table is partitioned by the primary key using hash partitioning. Each transaction in YCSB accesses 10 records (unless otherwise stated) that are a combination of independent read and update operations that occur in random order. Data access follows a Zipfian distribution, where the frequency of access to sets of hot records is tuned using a skew parameter (*theta*). When *theta* is 0, data is accessed with uniform frequency, and when it is 0.9 it is extremely skewed.

TPC-C: This benchmark models a warehouse order processing application and is the industry standard for evaluating OLTP databases [55]. It contains nine tables that are each partitioned by warehouse ID, except for the item table that is read-only and replicated at every server [50]. We support the two transactions in TPC-C that comprise 88% of the default workload mix: Payment and NewOrder. The other transactions require functionality, such as scans, that are currently unsupported in Deneva, so we omit them. We also do not include "think time" or simulate user data errors that cause 1% of the NewOrder transactions to abort.

The Payment transaction accesses at most two partitions. The first step in the transaction is to update payment amounts for the associated warehouse and district. Every transaction requests exclusive access to its home warehouse. The customer's information is then updated in the second part of the transaction. The customer belongs to remote warehouse with a 15% probability.

The first part of the NewOrder transaction reads its home warehouse and district records, then it updates the district record. In the second part, the transaction updates 5–15 items in the stock table. Overall, 99% of all items updated in a transaction are local to its home partition, while 1% are at a remote partition. This means that ∼10% of all NewOrder transactions are multi-partition.

PPS: The Product-Parts-Supplier workload (PPS) is another OLTP benchmark that contains transactions that execute foreign key lookups. It contains five tables: one each for products, parts, and suppliers, partitioned by their respective primary key IDs, a table that maps products to parts that they use and a table that maps suppliers to parts they supply. The benchmark assigns parts to products and suppliers randomly with a uniform distribution.

The benchmark's workload is comprised of a mix of single-partition and multi-partition transactions. The multi-partition OrderProduct transaction first retrieves the parts for a product and then decrements their stock quantities. The LookupProduct transaction is similar in that it retrieves the parts for a product and their stock quantities, but without updating any record. Both transaction types execute one or more foreign key look-ups that each may span multiple partitions. The last transaction (UpdateProductPart) updates a random product-to-part mapping in the database.]

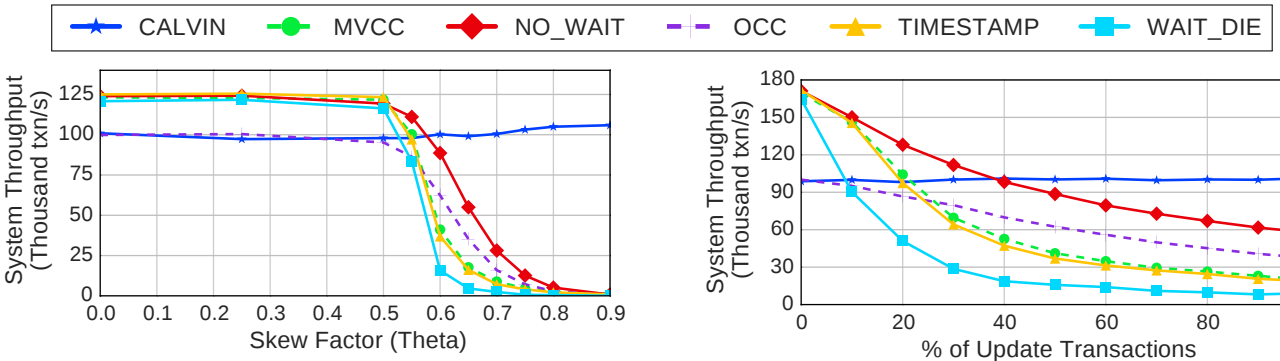


Figure 2: Contention – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.

Figure 3: Update Rate – The measured throughput of the protocols on 16 servers when varying the number of update transactions (5 reads / 5 updates) versus read-only transactions (10 reads) in the workload mixture for YCSB with medium contention (*theta*=0.6).

4.2 Contention

We begin with measuring the effect of increasing amount of contention in the system on the protocols, since it is often one of the most important factors that affect the performance of an OLTP database application. Contention occurs when transactions attempt to read or write to the same record. For this experiment, we use the YCSB workload and vary its skew parameter for the access patterns of transactions. The cluster is configured with 16 servers.

Figure 2 shows that the protocols' throughput is relatively unaffected by skew for *theta* values up to ∼0.5. After this point, most of them decline in performance but at different rates. Once *theta* reaches ∼0.8, all but one of them converge to the same low performance. CALVIN is the only protocol to maintain good performance despite high skew. First, the scheduling layer is a bottleneck in CALVIN. As soon as a transaction acquires all of its locks, a worker thread executes it right away and then releases the locks immediately afterward. This means that locks are not held long enough for the scheduler to process conflicting transactions. Second, since all of the transactions' data accesses are independent, CALVIN does not need to send messages between the read and execution phases. Thus, unlike the other protocols, it does not hold locks while waiting for messages from remote servers.

OCC performs worse than the other non-deterministic protocols under low contention due to the overheads of validation and copying during transaction execution [60]. At higher levels of contention, however, the benefit of tolerating more conflicts and thus avoiding unnecessary aborts outweighs these overheads.

MVCC and TIMESTAMP have a steep performance degradation when *theta* reaches ∼0.5 because they block newer transactions that conflict until the older ones commit. Although some transactions avoid this in MVCC by reading older versions, this requires that the reading transaction is older than all transactions with non-committed writes to the data, which is often not the case in this workload.

4.3 Update Rate

We next compare the ways that the protocols apply update operations from transactions to the database. For this experiment, we vary the percentage of YCSB transactions that invoke updates while keeping the server count constant at 16 nodes. Each transaction accesses 10 different records. We designate some transactions as "read-only" meaning that they only read records. The "update" transactions read five records and modify five records. These update operations are executed at random points in the transaction and never modify a record that is read by that same transaction. We use a medium skew setting (*theta*=0.6) since our previous experiment showed that this provides noticeable contention without overwhelming the protocols.

The results in Figure 3 show that the performance of most of the protocols declines as the percentage of update transactions increases. Introducing a small number of update transactions results in a throughput drop for WAIT_DIE. This is due to more transactions aborting when trying to access a hot record before it becomes old enough to wait for its lock (at a 10% update rate, there is an average of 1.4 aborts per transaction). Further, transactions in WAIT_DIE often spend time acquiring locks only to be aborted at a later time. NO_WAIT is not susceptible to this because transactions abort immediately if the desired lock is not available.

The performances of NO_WAIT, MVCC, and TIMESTAMP are nearly identical until the percentage of update transactions increases past 10%, at which point MVCC and TIMESTAMP diverge, and by 100% update transactions perform at 33% of the throughput of NO_WAIT. MVCC and TIMESTAMP benefit from being able to overlap transactions when there are more reads. But as the percentage of updates increases, MVCC and TIMESTAMP must block more transactions. For MVCC, the percentage of completed transactions that block varies from 3% to 16% for a 10% and 100% update rate, respectively. This has a negative effect on performance, as transactions that are blocked may in turn be causing other transactions to block on other records. NO_WAIT is not affected as severely, since read accesses proceed if the record-level lock is already in a shared state. When 100% of the transactions are updates, NO_WAIT performs 54% better than the next-best non-deterministic algorithm (OCC).

Figure 3 also shows that OCC and CALVIN do not perform as well as the other protocols when the percentage of update transactions in the workload is low. For OCC, this is due to the overhead of copying and validation. Although it also declines in performance as the percentage of updates increases because more validation steps are necessary if a transaction performs writes. But since OCC is often able to reconcile concurrent updates to the same record by being flexible about what commit timestamp it can assign, the drop is not as steep as with the other protocols.

Lastly, once again CALVIN's performance has a different trend than the other protocols. There is almost no performance difference between a workload with read-only transactions and one in which all transactions perform updates. Instead, CALVIN's performance is limited by the single-threaded scheduler that is unable to process transactions at the same rate as the multiple worker threads in other protocols. Each transaction is processed immediately once it acquires all of the locks that it needs. This means that locks are released as soon as the transaction finishes execution, so the DBMS does not hold locks long enough to cause contention in the scheduler. CALVIN does not need to hold locks during any message passing, unlike the other protocols, due to its deterministic properties. All of the reads in YCSB are independent so CALVIN does not need to pass any messages between servers whatsoever.

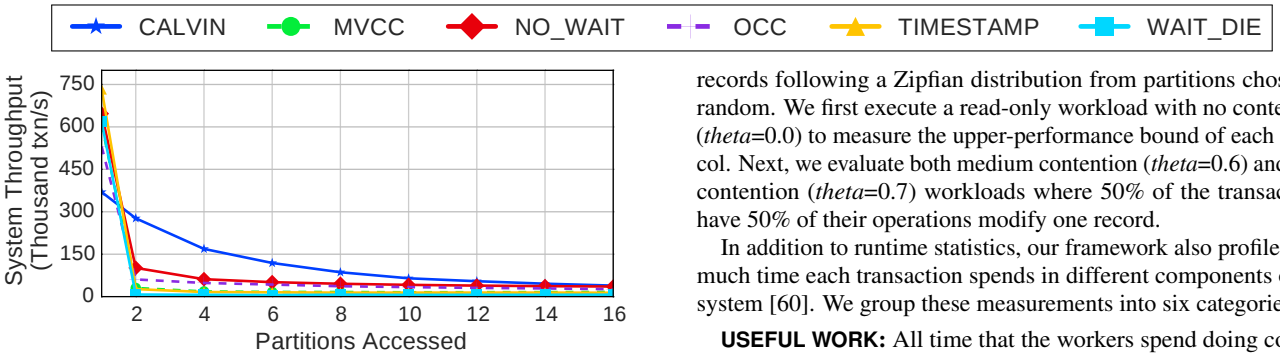


Figure 4: Multi-Partition Transactions – Throughput with a varying number of partitions accessed by each YCSB transaction.

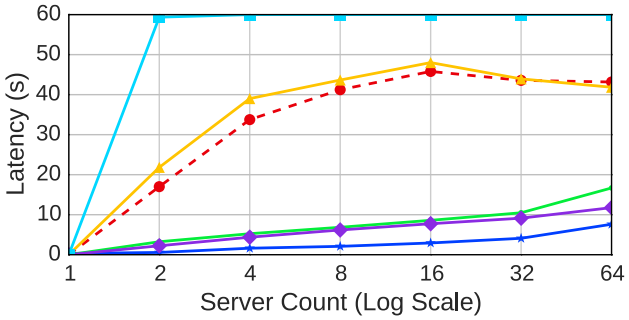


Figure 7: 99%ile Latency – Latency from a transaction's first start to its final commit for varying cluster size.

4.4 Multi-Partition Transactions

Our next experiment measures how the coordination overhead of multi-partition transactions affects performance. For each transaction, we vary the number of unique partitions each transaction accesses while keeping the total number of servers in the DBMS's cluster fixed at 16. We set the number of operations in each transaction is set at 16, with 50% reads and 50% writes. We configure the number of partitions accessed by each transaction is set per trial and assign them partitions at random.

Our results in Figure 4 show the performance of the protocols as we vary the number of partitions each transaction accesses. With the exception of CALVIN, the protocols' throughput plummets when transactions touch more than one partition. From two to four partitions, performance drops by 12–40%. This degradation is due to two reasons: (1) the overhead of sending remote requests and resuming transactions during execution, (2) the overhead of 2PC and the impact of holding locks for multiple round trip times between transaction execution and 2PC. CALVIN's performance drop from single- to multi-partition transactions is not as extreme as the other protocols. CALVIN's servers synchronize at every epoch. Even no multi-partition transactions arrive from remote sequencers, a scheduler must wait until it receives an acknowledgement from each sequencer before proceeding to ensure deterministic transaction ordering. If some sequencers lag behind other node's, this can cause slowdown in the system. Thus CALVIN does not scale as well as other protocols to 16 nodes when Deneva is only executing single-partition transactions.

4.5 Scalability

The previous three experiments examined the effects of various workload configuration in a fixed cluster size. In this section, we fix the workload and vary the cluster size to evaluate how the protocols scale with more servers. We again use YCSB and scale the size of the table with the number of servers. Each transaction accesses 10 records following a Zipfian distribution from partitions chosen at random. We first execute a read-only workload with no contention (*theta*=0.0) to measure the upper-performance bound of each protocol. Next, we evaluate both medium contention (*theta*=0.6) and high contention (*theta*=0.7) workloads where 50% of the transactions have 50% of their operations modify one record.

In addition to runtime statistics, our framework also profiles how much time each transaction spends in different components of the system [60]. We group these measurements into six categories:

USEFUL WORK: All time that the workers spend doing computation on behalf of read or update operations.

TXN MANAGER: The time spent updating transaction metadata and cleaning up committed transactions.

CC MANAGER: The time spent acquiring locks or validating as part of the protocol. For CALVIN, this includes time spent by the sequencer and scheduler to compute execution orders.

2PC: The overhead from two-phase commit.

ABORT: The time spent cleaning up aborted transactions.

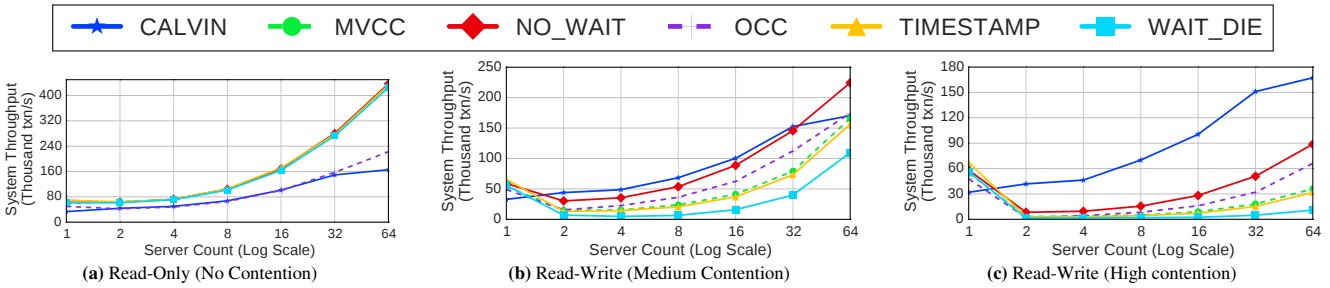
IDLE: The time worker threads spend waiting for work.

Read-Only Workload: The results in Figure 5a show that all protocols allow reads to proceed without blocking, so these results are close to the throughput that is achieved without any concurrency control. The throughputs of the protocols are nearly identical except for OCC and CALVIN. OCC's throughput is limited by the overhead of copying items for reference during the validation phase and the cost of the validation phase itself. CALVIN has the lowest throughput due to its bottleneck at the scheduler.

Medium Contention Workload: In Figure 5b, we see that with a mixed read/write workload, the landscape begins to change. Contention become an issue once the workload contains updates. Though all protocols improved throughput at 64 nodes over a single server, the gains are limited. At 64 servers, the protocols all improve their single-server performance by 1.7–3.8×. Although CALVIN shows the best improvement over its single-node performance, NO_WAIT has the best overall performance for all numbers of servers. In contrast to the read-only results, OCC outperforms all of the protocols with the exception of NO_WAIT and CALVIN when there is more than one server as the benefit of being able to tolerate more conflicts exceeds the costs of copying and validation.

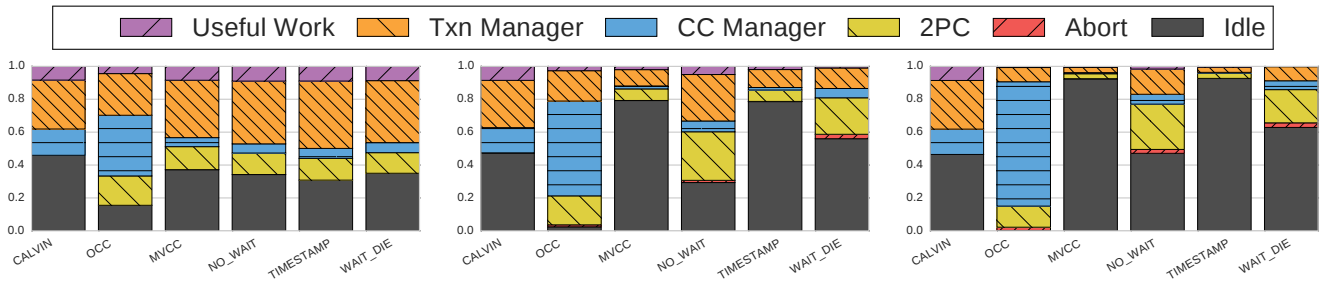
In comparing the breakdown in Figure 6b with the read-only results in Figure 6a, we see that the IDLE time increases for both MVCC and TIMESTAMP. This is because these protocols buffer more transactions while waiting for older transactions to complete.

In Figure 7, we observe the 99%ile latency increases for larger cluster sizes. OCC's 99%ile latency appears to drop because the longest transaction latencies would exceed the duration of the experiment. In Figure 8, we break down the average latency of successful transactions (from the most recent start or restart to final commit) in a cluster of 16 nodes to better understand where each transaction spends its time. Transaction processing and concurrency control manager time are eclipsed by other contributions to latency, such as time spent blocking as part of the protocol, time spent waiting on the work queue, and network latency. We see that transactions in OCC spend most of their time in the work queue. While this may be counterintuitive at first, note that the worker threads spend over 50% of their execution time in the validation phase of OCC. The work queue is where transactions naturally backup while waiting for the next worker thread to become available. Thus, in the case of OCC,



(a) Read-Only (No Contention) **(b)** Read-Write (Medium Contention) **(c)** Read-Write (High contention)

Figure 5: Scalability (Throughput) – Performance measurements for the protocols using variations of the YCSB workload and different cluster sizes.



(a) Read-Only (No Contention) **(b)** Read-Write (Medium Contention) **(c)** Read-Write (High Contention)

Figure 6: Scalability (Breakdown) – The percentage of time spent in Deneva's components for the concurrency control protocols using the same variations of the YCSB workload with 16 servers from Figure 5.

validation is the bottleneck in terms of throughput and latency. In the timestamp protocols MVCC and TIMESTAMP, most latency can be contributed to blocking as part of the protocol, which reinforces the hypothesis that these protocols inherently do poorly when records are highly contended, even when transactions do not abort frequently. NO_WAIT, when successful, only suffers from waiting for available worker threads. Successful WAIT_DIE transactions may also block.

High Contention Workload: Raising the contention even further continues to degrade the performance of all protocols except for CALVIN, as shown in Figure 5c. With the largest cluster configuration, the non-deterministic protocols achieve only 0.2–1.5× better throughout. This means these protocols are performing at less than 10% of its ideal capacity had it been able to scale linearly with the number of servers added. CALVIN performs up to 5.2× better on the largest cluster when the contention is high. Its performance remains nearly constant for all results in Figures 5 and 6 regardless of the contention setting and workload mixture.

We found that the protocols are sensitive to the transaction model. Since we limit the number of requests per transaction to 10 and maintain one partition per server, we see different behaviors for cluster configurations with more than 10 servers than those with less. The protocols are also sensitive to testbed configurations. For example, 2PL is sensitive to the parameters of the back-off penalty applied to aborts. The amount of load also affects throughput, particularly for protocols such as MVCC and TIMESTAMP as greater load allows them to process and buffer more transactions.

4.6 Network Speed

In this section, we isolate the effect of wide-area network (WAN) latency on distributed database performance. We deployed two servers and two clients on a single machine with a dual-socket Intel Xeon CPU E7-4830 (16 cores per CPU, 32 with hyper-threading). We insert an artificial network delay between server instances at the sender by buffering each message for the target delay amount before sending it to its destination. Communication between clients and servers was not affected by network delay.

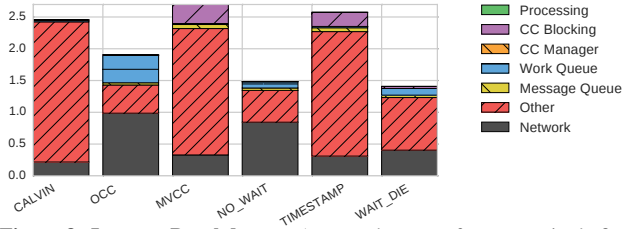


Figure 8: Latency Breakdown – Average latency of a transaction's final execution before commit.

The results in Figure 9 show how performance decreases as we increase network latency. The workload presented here is YCSB with medium contention and 50% update transactions. Due to higher variability between results in this compute environment, the results presented in this section are the average of five experiments with the lowest and highest throughput removed.

Additional network latency lengthens the 2PC protocol, which is the primary reason that the DBMS's throughput declines for network delays greater than 1 ms. Since CALVIN does not use 2PC with YCSB, it does not exchange any messages between servers and thus does not degrade when network latency increases. In the other protocols, however, the 2PC protocol contributes to longer lock hand-offs between transactions. 2PL locks owned for more extended periods of time cause new transactions to abort or block. In WAIT_DIE, since locks in distributed transactions are held longer, the number of conflicting transactions that can be buffered drops from ∼45% to ∼12% after 1 ms of delay, resulting in a much higher abort rate. A transaction may also be buffered longer before it aborts on a subsequent data access when the network latency increases. In TIMESTAMP and MVCC, reads and writes may block behind older transactions that have not yet committed. However, they are more resilient to changes in network latency than WAIT_DIE because they buffer most transactions and have a low abort rate. NO_WAIT has a higher abort rate than the timestamp protocols, but since the protocol does not block, locks are held for shorter time periods, allowing more transactions to make forward progress.

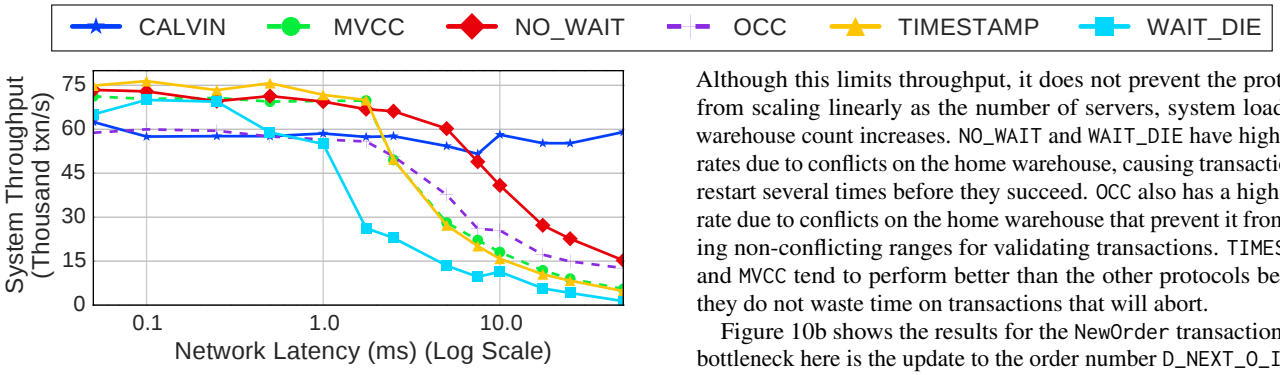


Figure 9: Network Speed – The sustained throughput measured for the concurrency protocols for YCSB with artificial network delays.

Table 2: Multi-Region Cluster – Throughput of a 2-node cluster with servers in AWS US East and US West regions.

Algorithm	CALVIN	OCC	MVCC
Throughput	8,412	11,572	5,486

Algorithm	NO_WAIT	TIMESTAMP	WAIT_DIE
Throughput	15,921	4,635	4,736

In Table 2, we measure the performance of YCSB under real WAN speeds. In this experiment, we used a nodes in the AWS east coast region (Virginia) and west coast region (California) to form a 2-node cluster communicating using a virtual private network (VPN). When compared to Figure 5b, we see that the protocols perform worse than their LAN-counterparts. We discuss the implications of network speed in Section 5.2.

4.7 Data-Dependent Aborts

In our previous experiments, CALVIN has an advantage because it does not need to use 2PC or other message passing beyond communicating with the sequencer. Since YCSB reads and writes are independent, and transactions do not conditionally abort based on values read during the transaction, CALVIN does not need to exchange any messages between servers between its read and write phase. But a workload with these properties under CALVIN requires servers that perform reads to send, and servers performing writes to receive, a round of messages before the transaction can complete.

To measure the effect of data-dependent aborts, we added a conditional statement to YCSB to model the execution logic associated with making an abort decision. We then re-ran the experiments using the modified YCSB workload. Though most protocols' throughput were only affected by 2–10%, CALVIN experienced a 36% decrease in throughput when the workload was run on 16 servers with medium contention (*theta*=0.6, 50% update transactions) compared to the original YCSB workload. We also found that as contention increases from *theta*=0.8 to *theta*=0.9, CALVIN's throughput declines from 73k to 19k transactions per second.

4.8 TPC-C

We next test the protocols using a more realistic workload. The TPC-C benchmark demonstrates how the protocols perform on a workload whose transactions are mostly single-partition, and whose distributed transactions are mostly limited to two partitions. We use a database size of 128 warehouses per server. Since there are multiple warehouses at each server, a distributed transaction may touch two partitions located on the same server.

Our first experiment in Figure 10a shows how the Payment transaction scales. The home warehouse is a bottleneck since exclusive access is required to update the warehouse's payment information. Although this limits throughput, it does not prevent the protocols from scaling linearly as the number of servers, system load, and warehouse count increases. NO_WAIT and WAIT_DIE have high abort rates due to conflicts on the home warehouse, causing transactions to restart several times before they succeed. OCC also has a high abort rate due to conflicts on the home warehouse that prevent it from finding non-conflicting ranges for validating transactions. TIMESTAMP and MVCC tend to perform better than the other protocols because they do not waste time on transactions that will abort.

Figure 10b shows the results for the NewOrder transaction. The bottleneck here is the update to the order number D_NEXT_O_ID in a transaction's district. But since there are 10 districts per warehouse, there is slightly less contention with than Payment transactions. Thus, with the exception of OCC, the non-deterministic protocols perform better than CALVIN in this workload.

4.9 Product-Parts-Supplier

Lastly, we examine the scalability of the protocols using the PPS benchmark. Unlike the other workloads, PPS contains transactions that update tables through foreign key lookups. This aspect of the workload stresses CALVIN because it must perform reconnaissance to look up the foreign keys and the transactions may abort.

The results in Section 4.8 show that most protocols scale. CALVIN has a flat-lined throughput of about 7k transactions per second. This is caused by several factors. First, two of the transactions in the workload require reconnaissance queries to determine the full read and write set. Second, random updates to the table that maps products to parts causes the other transactions to fail because their read and write sets are invalidated when the transaction executes. The other protocols do not suffer from aborts due to changing parts numbers. OCC's abort rate is 15%, which is as high as NO_WAIT. Unlike NO_WAIT, however, OCC must roll back the entire completed transaction, resulting in wasted computation and other resources.

5. DISCUSSION

The experimental results in the previous section provide a mixed outlook on the current and state-of-the-art support mechanisms for distributed OLTP transactions. On the one hand, workloads with distributed transactions can scale, but depending on the properties of the workload it may take a large cluster to beat the performance of a single machine. In this section, we discuss the scalability issues facing distributed DBMSs, as well as some of their ramifications.

5.1 Distributed DBMS Bottlenecks

Our results show that all of the protocols that we evaluated faced scalability challenges. We summarize the bottlenecks we observed in Table 3. Foremost is that the commit protocol is one of the primary factors that affects throughput. Most of the protocols require two round trips before committing. CALVIN was designed specifically to try to mitigate the affects of 2PC, but if there is the possibility that a transaction will abort, a round of messages must be broadcast and collected before transactions can commit at their local node and release their locks. In all cases, locks held during 2PC or while waiting for messages prevent forward progress in the application.

Another major bottleneck for distributed DBMSs is data contention due to two major factors. First, there is no contention without record updates. When we varied the update rate in the YCSB workload, we found that when there were few updates using a protocol with minimal overhead, such as 2PL or a timestamp protocol, made the most sense. These protocols quickly degraded as the number of writes increased and past a certain threshold using an optimistic or

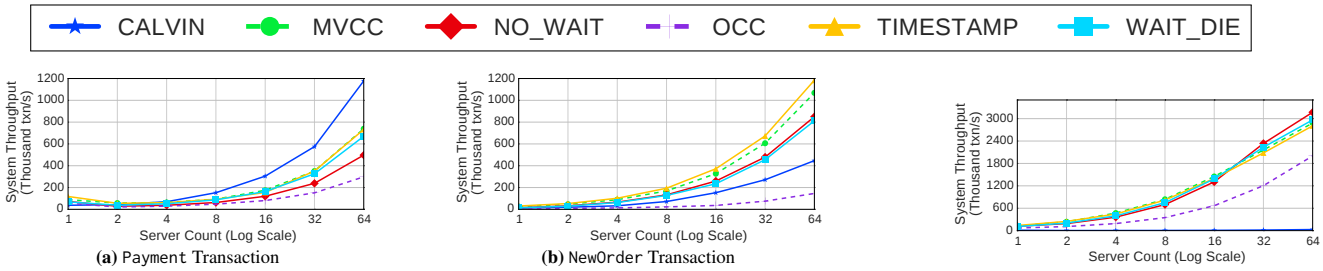


Figure 10: TPC-C – The measured throughput of the protocols when scaling out the cluster with 128 warehouses per server.

Figure 11: Product-Parts-Supplier – The throughput of a PPS workload with 80% transactions foreign key lookups as the number of servers increases.

Table 3: Results Summary – An overview of the results of our experiments on concurrency control protocols evaluated in this study, summarized by whether they are bottlenecks (▼) or advantages compared to the other protocols (▲), or have minimal effect on relative performance (–).

Class	Algorithms	Two-Phase Commit Delay	Multi-Partition Transactions	Low Contention	High Contention
Locking	NO_WAIT, WAIT_DIE	▼	▼	▲	▼
Timestamp	TIMESTAMP, MVCC	▼	▼	▲	▼
Optimistic	OCC	▼	▼	▼	▲
Deterministic	CALVIN	–	▼	▼	▲

deterministic protocol was a better choice. Second, the frequency of reads and updates to hot records determines how often contention occurs. When we examined in the YCSB workload, the 2PL and timestamp protocols perform best when there is low skew. Once there is enough skew to cause concurrent transactions to access the same records, these protocols experience a sharp performance drop. Optimistic concurrency control is slightly more resilient to high skew, but once the skew reaches a critical point, CALVIN emerges as the most consistent protocol.

5.2 Potential Solutions

Given this bleak outlook, we now discuss potential solutions to the challenges of distributed concurrency control protocols.

Improve the Network: Ultimately, the use of the network—the defining property of distributed transactions—impedes scalability. Accordingly, a natural direction for improving scalability is to improve network performance. The public cloud infrastructure available today provides reasonable performance, but it is far from the cutting edge. Best-of-class private datacenter networks today enjoy full bisection bandwidth between servers and considerably lower latency. With formerly exotic but increasingly common hardware including RDMA and RoCE capability, the performance overhead of distributed transactions can similarly drop precipitously: a 5 μs message delay admits considerably more parallelism than the $\sim$500 μs delays we experience on cloud infrastructure today. Indeed, several recent protocols leverage these hardware capabilities (Section 6).

Although improvements in networking hardware present substantial benefits to today's transaction processing systems within a *single* datacenter, multi-datacenter operation remains challenging. Despite recent excitement demonstrating the possibility of quantum entanglement, the speed of light presents a formidable lower bound on network communication time. Given a lack of line-of-sight communication, wide-area communication is much more expensive [4]. This implies serializable transactions over the WAN are likely to remain expensive indefinitely.

As a result, we perceive an opportunity for improvement within a single datacenter—subject to the adoption, availability, and hardening of previously non-commodity hardware—but it is non-trivial across datacenters. A related deployment scenario that we believe deserves further consideration is increasingly mobile and ubiquitous sensing; while the "Internet of Things" remains amorphous, its future degree of distribution and parallelism will pose scalability challenges due to networked operation limitations.

Adapt the Data Model: While distributed transactions are expensive, transactions that execute within a single node are relatively inexpensive. As a result, applications that express their transactions in a manner that is amenable to single-node execution are not subject to the penalties we investigate here.

A primary strategy for achieving single-node operation is to perform partitioning within the data model. For example, Helland's entity group approach [31] over data stored within a single, possibly hierarchical, set of data that can fit on a single server. This necessarily restricts the class of applications that can be expressed (e.g., the DBMS cannot enforce foreign key constraints across entity groups) and effectively shifts the burden to the application developer. Nevertheless, for applications that are easily partitionable (one of the authors called these applications "delightful" [49] in mid-1980s), the entity group model solves the distributed transaction problem.

Simultaneously, the research community has developed techniques for automatically partitioning applications, both statically and dynamically [22, 43, 23]. These techniques are promising, although they have yet to make it to mainstream deployments.

Seek Alternative Programming Models: Given the cost of serializability, a reasonable alternative is to seek alternatives. Often, discussions regarding non-serializable behavior present a false dichotomy between serializability and application consistency versus difficult to understand and error-prone alternatives. It is helpful to remember that serializability is a means towards achieving application-level consistency, but it is not strictly necessary; despite the fact that many existing formulations of non-serializable isolation (e.g., Read Committed isolation [2] and eventual consistency [10]) are unintuitive, this does not necessarily mean that all non-serializable programmer interfaces need be unintuitive. For example, the recently proposed Homeostasis Protocol [46] analyzes programs for opportunities for non-serializable execution and automatically executes selected portions of user code in a non-serializable manner without the programmer or user detecting it. Invariant confluence analysis [6] shows that many common database constraints can be enforced without distributed coordination, permitting scalable but compliant TPC-C execution without relying on distributed trans-

actions. Related results from the systems [8] and programming languages [48] communities show similar promise.

In effect, this alternative prompts a re-investigation of so-called semantics-based concurrency control methods [52]. Given a lack of information about the semantics of applications, serializability is, in a sense, the "optimal" strategy for guaranteeing application consistency [37]. But given the scalability challenges we have observed, we believe it is worth investigating techniques for pushing down additional semantics into the database engine. In fact, one study observed that application developers for Web frameworks (e.g., Ruby on Rails) already use non-transactional interfaces for expressing their application consistency criteria [5]. This offers an exciting opportunity to rethink the transaction concept and pursue alternative programming models.

Summary: There are several directions in which to pursue solutions, ranging from novel hardware to advanced data modeling to program analysis. We believe each is potentially fruitful, with many opportunities for both innovative research and meaningful improvements in performance and scalability.

6. RELATED WORK

There have been several efforts to compare transaction processing architectures since the 1980s. Many of the earlier studies used modeling techniques [42, 3, 14, 13, 16], but they sometimes performed comparative experimental analyses as we perform here [32]. More recently, the OLTP-Bench project developed a standardized set of OLTP workloads [24], while BigBench [29], BigDataBench [57], and Chen et al. [18] have presented a range of benchmarks and workloads for Big Data analytics frameworks. A related set of studies [15, 56, 36] examines cloud computing platforms and multi-tenant workloads, while Rabl et al. [44] evaluate a range of distributed databases, only one of which is transactional. We continue this tradition of empirical systems analysis by developing the Deneva framework and evaluating the performance of six protocols on cloud computing infrastructure. This is the most comprehensive study on modern hardware of which we are aware. Perhaps closest to our work here is a recent survey of concurrency control protocols in a non-distributed, single server with 1000 CPU cores [60].

Our findings corroborate previous conclusions yet have important differences due to the scale and operating environment of cloud infrastructure. For example, Agrawal et al. [3] find that optimistic methods are remarkably expensive in the event of aborts and that, under load, pessimistic methods may provide more robust behavior. Our results show that optimistic methods perform well, but only under idealized conditions. Furthermore, message delay has important implications in a distributed environment. For example, while Carey and Livny examine the CPU cost of message sending and receiving instead of network delay [16]. We have shown that network delay has a considerable impact on efficiency, leading to our conclusion that faster network hardware is indeed a promising means of improving performance. Especially compared to the 1000-core setting [60], the workload with the best performance is often different in a cloud environment than on a single server or simulated network.

In this work, we have studied six concurrency control protocols for implementing serializability in a distributed environment. As we hinted in Table 1, there is are plethora of alternatives, and new protocols are proposed every year [7, 20, 21, 25, 26, 27, 39, 41, 47, 54, 58, 61, 62]. Many of these use variants of the approaches we have investigated here: for example, Spanner [20] and Wei et al. [58] implement two-phase locking, Granola [21], VoltDB [1], H-Store [33], and Calvin [54] (which we evaluate) implement deterministic methods, and Centiman [25], FaRM [26], Warp [27], MaaT [39], Rococo [41], F1 [47], and Tapir [61] implement variants of OCC. Given this panoply of new protocols, we believe it is an especially ripe opportunity to perform an analysis similar to Bernstein and Goodman's 1981 seminal deconstruction of then-modern concurrency control protocols [11], in which they showed how to express almost all proposed as either variants of locking or timestamp methods. We believe Deneva provides a quantitative framework for understanding the performance trade-offs made by these systems as well, and, as we will discuss in Section 7, are interested in performing this analysis in the future.

7. FUTURE WORK

We see several promising avenues for future work, both in terms of our evaluation framework and novel distributed concurrency control protocols.

First, our study has focused on concurrency control over a partitioned database. This decision allowed us to isolate the scalability bottlenecks to the concurrency control subsystem. This is reflected in our framework design, which supports multi-partition serializable transactions. We are interested in investigating the effect of both replication (e.g., active-active versus active-passive protocols [40]) and failure recovery (e.g., failover, behavior during network partitions). Each has non-trivial implications for scalability, which we intend to explore in the future.

Second, our study has focused on six specific concurrency control protocols. While this study is more comprehensive than any other in the recent literature, there are recent protocol proposals whose performance has been quantitatively compared—at best—to a handful of other schemes. We are interested in quantitatively evaluating the most promising of these newer schemes. We hope that open sourcing Deneva will provide an incentive for the research community to integrate their own protocols as well. To extend the number of workloads supported, we are considering integrating the Deneva framework with the OLTP-Bench benchmarking suite [24]. This integration will require care for certain protocols, like `CALVIN`, which require read-write sets to be pre-declared but is a worthwhile engineering effort.

Third, we wish to investigate the effect of the potential solutions described in Section 5.2. For example, integrating RDMA and alternative networking technologies into Deneva would allow a fair evaluation of their costs and benefits. It would also be interesting to perform a head-to-head comparison with recent proposals for semantics-based concurrency control to quantitatively verify the potential for speedups as promised in the literature.

8. CONCLUSION

We investigated the behavior of serializable distributed transactions in a modern cloud computing environment. We studied the behavior of six classic and modern concurrency control protocols and demonstrated that, for many workloads, distributed transactions on a cluster often only exceed the throughput of non-distributed transactions on a single machine by small amounts. The exact scalability bottleneck is protocol-dependent: two-phase locking performs poorly under high contention due to aborts, timestamp-ordered concurrency control does not perform well under high contention due to buffering, optimistic concurrency control has validation overhead, and deterministic protocol maintains performance across a range of adverse load and data skew but has limited performance due to transaction scheduling. Ultimately, these results point to a serious scalability problem for distributed transactions. We believe the solution lies in a tighter coupling of concurrency control engines with both hardware and applications, via a combination of network

improvements on cloud infrastructure (at least within a single datacenter), data modeling, and semantics-based concurrency control techniques. Moreover, we intend for the Deneva framework to provide an open platform for others to perform a rigorous assessment of novel and alternative concurrency control techniques and to bring clarity to an often confusing space of concurrency control protocols.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous VLDB reviewers for their useful feedback. We also would like to thank Peter Bailis for his help with early drafts of this work and Xiangyao Yu, Daniel Abadi, Alexander Thomson, and Jose Faleiro for their valuable discussions.

10. REFERENCES

[1] VoltDB. `http://voltdb.com`.

[2] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[3] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *TODS*, 12(4):609–654, 1987.

[4] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[5] P. Bailis et al. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. In *VLDB*, 2015.

[7] M. Balakrishnan et al. Tango: Distributed data structures over a shared log. In *SOSP*, 2013.

[8] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys*, 2015.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley, 1987.

[10] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, pages 923–928. ACM, 2013.

[11] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[12] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[13] A. Bhide, F. Bancilhon, and D. Dewitt. An analysis of three transaction processing architectures. In *VLDB*, 1988.

[14] A. Bhide and M. Stonebraker. A performance comparison of two architectures for fast transaction processing. In *ICDE*, 1988.

[15] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow? towards a benchmark for the cloud. In *DBTest*, 2009.

[16] M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In *VLDB*, pages 13–25, 1988.

[17] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[18] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *VLDB*, 2012.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. SoCC, pages 143–154, 2010.

[20] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[21] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, 2012.

[22] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3(1-2):48–57, 2010.

[23] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[24] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *VLDB*, 2014.

[25] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.

[26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[27] R. Escriva, B. Wong, and E. G. Sirer. Warp: Multi-key transactions for keyvalue stores. *United Networks, LLC, Tech. Rep*, 5, 2013.

[28] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[29] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208, 2013.

[30] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[31] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[32] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, 1991.

[33] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.

[34] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[35] S. Kimball. Living without atomic clocks. `https://www.cockroachlabs.com/blog/living-without-atomic-clocks/`, February 2016.

[36] R. Krebs, A. Wert, and S. Kounev. Multi-tenancy performance benchmark for web application platforms. In *Web Engineering*, pages 424–438. Springer, 2013.

[37] H.-T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, 1979.

[38] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[39] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. In *VLDB*, 2014.

[40] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *ICDE*, pages 604–615. IEEE, 2014.

[41] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[42] M. Nicola and M. Jarke. Performance modeling of distributed and replicated databases. *TKDE*, 12(4):645–672, 2000.

[43] A. Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, pages 61–72, 2012.

[44] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. In *VLDB*, 2012.

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

[46] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[47] J. Shute et al. F1: A distributed SQL database that scales. In *VLDB*, 2013.

[48] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI*, 2015.

[49] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9, 1986.

[50] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). VLDB, pages 1150–1160, 2007.

[51] M. Sustrik. nanomsg. `http://nanomsg.org`.

[52] M. Tamer Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[53] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3:70–80, September 2010.

[54] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[55] Transaction Processing Performance Council. TPC Benchmark C (Revision 5.11), February 2010.

[56] A. Turner, A. Fox, J. Payne, and H. S. Kim. C-mart: Benchmarking the cloud. *IEEE TPDS*, 24(6):1256–1266, 2013.

[57] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. BigDataBench: A big data benchmark suite from internet services. In *HPCA*, pages 488–499. IEEE, 2014.

[58] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, 2015.

[59] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.

[60] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *VLDB*, 2014.

[61] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *SOSP*, 2015.

[62] Y. Zhang et al. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.