Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems

ALEXANDER THOMSON, THADDEUS DIAMOND, SHU-CHUN WENG, KUN REN, PHILIP SHAO, and DANIEL J. ABADI, Yale University

As more data management software is designed for deployment in public and private clouds, or on a cluster of commodity servers, new distributed storage systems increasingly achieve high data access throughput via partitioning and replication. In order to achieve high scalability, however, today's systems generally reduce transactional support, disallowing single transactions from spanning multiple partitions.

This article describes Calvin, a practical transaction scheduling and data replication layer that uses a deterministic ordering guarantee to significantly reduce the normally prohibitive contention costs associated with distributed transactions. This allows near-linear scalability on a cluster of commodity machines, without eliminating traditional transactional guarantees, introducing a single point of failure, or requiring application developers to reason about data partitioning. By replicating transaction inputs instead of transactional actions, Calvin is able to support multiple consistency levels—including Paxos-based strong consistency across geographically distant replicas—at no cost to transactional throughput.

Furthermore, Calvin introduces a set of tools that will allow application developers to gain the full performance benefit of Calvin's server-side transaction scheduling mechanisms without introducing the additional code complexity and inconvenience normally associated with using DBMS stored procedures in place of ad hoc client-side transactions.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Distributed Databases; H.2.4 [**Database Management**]: Systems—*Concurrency, distributed databases, transaction processing*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Determinism, distributed database systems, replication, transaction processing

ACM Reference Format:
Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast distributed transactions and strongly consistent replication for OLTP database systems. ACM Trans. Datab. Syst. 39, 2, Article 11 (May 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2556685

1. BACKGROUND AND INTRODUCTION

Many recently introduced distributed database system designs move away from supporting traditional ACID database transactions. Some systems, such as Amazon's

This work was done while A. Thomson is currently affiliated with Google; T. Diamond is currently affiliated with Hadapt; K. Ren is currently affiliated with Northwestern Polytechnical University, China; and P. Shao is currently affiliated with Microsoft.
This work was sponsored by the NSF under grants IIS-0845643 and IIS-1249722. K. Ren is supported by National Natural Science Foundation of China under grant 61033007 and National 973 Project under grant 2012CB316203.
Authors' addresses: A. Thomson (corresponding author), Google; email: thomson@cs.yale.edu; T. Diamond, Hadapt; S.-C. Weng, Yale University; K. Ren, Northwestern Polytechnical University, China; P. Shao, Microsoft; D. J. Abadi, Yale University.

DOI: http://dx.doi.org/10.1145/2556685

Dynamo [DeCandia et al. 2007], MongoDB [Plugge et al. 2010], CouchDB [Anderson et al. 2010], and Cassandra [Lakshman and Malik 2009] provide no transactional support whatsoever (at of the time of this writing). Others provide only limited transactionality, such as single-row transactional updates (e.g., Bigtable [Chang et al. 2006]) or transactions whose accesses are limited to small subsets of a database (e.g., Azure [Campbell et al. 2010], Megastore [Baker et al. 2011], and the Oracle NoSQL Database [Seltzer 2011]). The primary reason that each of these systems does not support fully ACID transactions is to provide linear outward scalability. Other systems (e.g., VoltDB [Stonebraker et al. 2007; Jones et al. 2010]) support full ACID, but cease (or limit) concurrent transaction execution when processing a transaction that accesses data spanning multiple partitions.

Reducing transactional support greatly simplifies the task of building linearly scalable distributed database solutions. In particular, if there are no distributed transactions, then there is no need for distributed coordination and commit protocols that limit system scalability by extending the amount of time that locks must be held (potentially clogging the system) and by introducing network bandwidth bottlenecks. Some applications are naturally "embarrassingly partitionable", and have no need for a system that supports distributed transactions. For example, if each user of an application accesses only his or her own private account data (e.g., a private calendar or a photo uploading service), then it is straightforward to partition the application's data across many machines.

Many applications, however, are not naturally embarrassingly partitionable and are forced into partitioned access patterns due to the limits of the underlying system. Examples of this are as follows.

—*A customer placing an order from an e-commerce Web site*. Since customers can buy any arbitrary combination of items, it is impossible to guarantee that inventory information about the items being bought will all be located on the same machine. Thus updating all the inventories of all the items being bought transactionally is impossible to scale arbitrarily without support of distributed transactions. Consequently, large retail companies such as Amazon are forced into nontransactional updates of inventory information (and potentially unhappy customers) as a result of system limitations of distributed transactions.

—*A credit-debit transaction between an arbitrary pair of users*. As soon as the system scales to the point where different users are stored on different machines, the transfer of money (or other widgets) between users located on different machines requires support for distributed transactions. In practice, banks typically end up doing complicated workflows where the debit and the credit occur in separate transactions (with compensatory actions if one them fails) due to the lack of system support for distributed transactions.

—*Adding an edge to a social graph to connect two users*. As soon as the system scales to the point where different users are stored on different machines, adding this edge requires the participation of both machines. In practice, social networks are forced to perform these actions nontransctionally, and users can sometimes see the effects of the temporary inconsistency that results.

—Other examples include a user placing a bid in an online auction, a player in an MMO game interacting with a virtual game object or with another player, or a message being sent and received in a messaging application. Each of these naturally includes multiple different machines naturally being involved in the transaction as the application scales, and the application code is forced to work around system limitations of distributed transactional support.

While these examples span a wide range of application domains (we use the example of a `BalanceTransfer` throughout the rest of this article), they do have certain things in common.

—The world's largest deployments of these types of applications process extremely high throughput (often hundreds of thousands to millions of operations per second) for these particular operation types.
—When data is partitioned across many machines, the records accessed by one of these transactions cannot always be expected to be co-located on the same partition. Each operation represents an interaction between a pair (or small group) of logical entities, and a given operation likely to span multiple data partitions (two is the most common).
—Correctly implementing these operations is vastly simplified by database support for arbitrary ACID transactions. Otherwise, complicated workarounds must be implemented in application logic.

Distributed storage systems with reduced transactional guarantees leave the burden of ensuring atomicity and isolation to the application programmer for the preceding class of transactions—resulting in increased code complexity, slower application development, low-performance client-side transaction scheduling, and user-facing inconsistencies in application behavior.

Calvin is designed to run alongside a nontransactional storage system, transforming it into a shared-nothing (near-)linearly scalable database system that provides high availability[1] and full ACID transactions. These transactions can potentially span multiple partitions spread across the shared-nothing cluster. Calvin accomplishes this by providing a layer above the storage system that handles the scheduling of distributed transactions, as well as replication and network communication in the system. The key technical feature that allows for scalability in the face of distributed transactions is a deterministic locking mechanism that enables the elimination of distributed commit protocols.

1.1. The Cost of Distributed Transactions

Distributed transactions have historically been implemented by the database community in the manner pioneered by the architects of System R* [Mohan et al. 1986] in the 1980s. The primary mechanism by which System R*-style distributed transactions impede throughput and extend latency is the requirement of an agreement protocol between all participating machines at commit time to ensure atomicity and durability. To ensure isolation, all of a transaction's locks must be held for the full duration of this agreement protocol, which is typically two-phase commit.

The problem with holding locks during the agreement protocol is that two-phase commit requires multiple network round-trips between all participating machines, and therefore the time required to run the protocol can often be considerably greater than the time required to execute all local transaction logic. If a few popularly accessed records are frequently involved in distributed transactions, the resulting extra time that locks are held on these records can have an extremely deleterious effect on overall transactional throughput. We refer to the total duration that a transaction holds

[1]In this article we use the term *high availability* in the common colloquial sense found in the database community where a database is highly available if it can fail over to an active replica on-the-fly with no downtime, rather than the definition of high availability used in the CAP theorem which requires that even minority replicas remain available during a network partition.

its locks—which includes the duration of any required commit protocol—as the transaction's *contention footprint*. Although most of the discussion in this article assumes pessimistic concurrency control mechanisms, the costs of extending a transaction's contention footprint are equally applicable—and often even worse due to the possibility of cascading aborts—in optimistic schemes.

Certain optimizations to two-phase commit, such as combining multiple concurrent transactions' commit decisions into a single round of the protocol, can reduce the CPU and network overhead of two-phase commit, but do not ameliorate its contention cost.

Allowing distributed transactions may also introduce the possibility of distributed deadlock in systems implementing pessimistic concurrency control schemes. While detecting and correcting deadlocks does not typically incur prohibitive system overhead, it can cause transactions to be aborted and restarted, increasing latency and reducing throughput to some extent.

1.2. Consistent Replication

A second trend in distributed database system design has been towards reduced consistency guarantees with respect to replication. Systems such as Dynamo, SimpleDB, Cassandra, Voldemort, Riak, and PNUTS all lessen the consistency guarantees for replicated data [DeCandia et al. 2007; Lakshman and Malik 2009; Cooper et al. 2008]. The typical reason given for reducing the replication consistency of these systems is the CAP theorem [Gilbert and Lynch 2002]—in order for the system to achieve 24/7 global availability and remain available even in the event of a network partition, the system must provide lower consistency guarantees [Abadi 2012]. However, in the last year, this trend has been starting to reverse—perhaps in part due to ever-improving global information infrastructure that makes nontrivial network partitions increasingly rare—with several new systems supporting strongly consistent replication. Google's Megastore and Spanner [Baker et al. 2011; Corbett et al. 2012] and IBM's Spinnaker [Rao et al. 2011], for example, are synchronously replicated via Paxos [Lamport 1998, 2001].

Synchronous updates come with a latency cost fundamental to the agreement protocol, which is dependent on network latency between replicas. This cost can be significant, since replicas are often geographically separated to reduce correlated failures. However, this is intrinsically a latency cost only, and need not necessarily affect contention or throughput.

1.3. Achieving Agreement without Increasing Contention

Calvin's approach to achieving inexpensive distributed transactions and synchronous replication is the following: when multiple machines need to agree on how to handle a particular transaction, they do it outside of transactional boundaries, that is, before they acquire locks and begin executing the transaction.

Once an agreement about how to handle the transaction has been reached, it must be executed to completion according to the plan—node failure and related problems cannot cause the transaction to abort. If a node fails, it can recover from a replica that had been executing the same plan in parallel, or alternatively, it can replay the history of planned activity for that node. Both parallel plan execution and replay of plan history require activity plans to be deterministic—otherwise replicas might diverge or history might be repeated incorrectly.

To support this determinism guarantee while maximizing concurrency in transaction execution, Calvin uses a deterministic locking protocol based on one we introduced in previous work [Thomson and Abadi 2010].

Since all Calvin nodes reach an agreement regarding what transactions to attempt and in what order, it is able to completely eschew distributed commit protocols, reducing the contention footprints of distributed transactions, thereby allowing throughput to scale out nearly linearly despite the presence of multipartition transactions. Our experiments show that Calvin significantly outperforms traditional distributed database designs under high-contention workloads. We find that it is possible to run half a million TPC-C transactions per second on a cluster of commodity machines in the Amazon cloud, which is immediately competitive with the world-record results currently published on the TPC-C Web site that were obtained on much higher-end hardware.

1.4. Improving Calvin's Usability

Calvin's determinism guarantee requires all transaction logic to be declared in advance of transaction execution and then executed entirely by the database server. Stored procedures are a well-known method of achieving exactly this, and are widely supported by commercial database systems. However, implementing transactions as stored procedures rather than as ad hoc client-side transactions often introduces significant code complexity and inconvenience for application developers.

We therefore explore more developer-friendly options than Calvin's native transaction definition format of C++ stored procedures. In particular, we introduce a novel set of techniques that allow transactions that are defined as client-side functions to be executed instead by the database server with little to no additional instrumentation required of the application developer. We describe a Python module that implements a collection of these techniques.

1.5. Contributions

This article's primary contributions are the following.

—We design a transaction scheduling and data replication layer that transforms a nontransactional storage system into a (near-)linearly scalable shared-nothing database system that provides high availability, strong consistency, and full ACID transactions.
—We give a practical implementation of a deterministic concurrency control protocol that is more scalable than previous approaches, and does not introduce a potential single point of failure.
—We presents a data prefetching mechanism that leverages the planning phase performed prior to transaction execution to allow transactions to operate on disk-resident data without extending transactions' contention footprints for the full duration of disk lookups.
—We provide a novel set of tools that allows application developers to define transactions as simple, ad hoc client-side functions, while gaining the performance benefits of server-side execution of transaction code.
—We give a fast checkpointing scheme that, together with Calvin's determinism guarantee, completely removes the need for physical REDO logging and its associated overhead.

The following section discusses further background on deterministic database systems. In Section 3 we present Calvin's design and architecture. Section 4 discusses developer-friendly extensions to Calvin's native transaction API. In Section 5 we address how Calvin handles transactions that access disk-resident data. Section 6 covers Calvin's mechanism for periodically taking full database snapshots. In Section 7 we present a series of experiments that explore the throughput and latency of Calvin under different workloads. We present related work in Section 8, discuss future work in Section 9, and conclude in Section 10.

2. DETERMINISTIC DATABASE SYSTEMS

In traditional (System R*-style) distributed database systems, the primary reason that an agreement protocol is needed when committing a distributed transaction is to ensure that all effects of a transaction have successfully made it to durable storage in an atomic fashion—either all nodes involved the transaction agree to "commit" their local changes or none of them do. Events that prevent a node from committing its local changes (and therefore cause the entire transaction to abort) fall into two categories: nondeterministic events (such as node failures) and deterministic events (such as transaction logic that forces an abort if, say, an inventory stock level would fall below zero).

There is no fundamental reason that a transaction must abort as a result of any nondeterministic event; when systems do choose to abort transactions due to outside events, it is due to practical consideration. After all, forcing all other nodes in a system to wait for the node that experienced a nondeterministic event (such as a hardware failure) to recover could bring a system to a painfully long stand-still.

If there is a replica node performing the exact same operations in parallel to a failed node, however, then other nodes that depend on communication with the afflicted node to execute a transaction need not wait for the failed node to recover back to its original state—rather they can make requests to the replica node for any data needed for the current or future transactions. Furthermore, the transaction can be committed since the replica node was able to complete the transaction, and the failed node will eventually be able to complete the transaction upon recovery.[2]

Therefore, if there exists a replica that is processing the same transactions in parallel to the node that experiences the nondeterministic failure, the requirement to abort transactions upon such failures is eliminated. The only problem is that replicas need to be going through the same sequence of database states in order for a replica to immediately replace a failed node in the middle of a transaction. Synchronously replicating every database state change would have far too high of an overhead to be feasible. Instead, deterministic database systems synchronously replicate batches of transaction *requests*. In a traditional database implementation, simply replicating transactional input is not generally sufficient to ensure that replicas do not diverge, since databases guarantee that they will process transactions in a manner that is logically equivalent to some serial ordering of transactional input—but two replicas may choose to process the input in manners equivalent to different serial orders, for example due to different thread scheduling, network latencies, or other hardware constraints. However, if the concurrency control layer of the database is modified to acquire locks in the order of the agreed-upon transactional input (and several other minor modifications to the database are made [Thomson and Abadi 2010]), all replicas can be made to emulate the same serial execution order, and database state can be guaranteed not to diverge.[3]

Such deterministic databases allow two replicas to stay consistent simply by replicating database input and, as described before, the presence of these actively replicated nodes enables distributed transactions to commit their work in the presence of nondeterministic failures (which can potentially occur in the middle of a transaction). This eliminates the primary justification for an agreement protocol at the end of distributed

[2]Even in the unlikely event that all replicas experience the same nondeterministic failure, the transaction can still be committed if there was no deterministic code in the part of the transaction assigned to the failed nodes that could cause the transaction to abort, since the transactional input can replayed upon recovery (see the following).

[3]More precisely, the replica states are guaranteed not to appear divergent to outside requests for data, even though their physical states are typically not identical at any particular snapshot of the system.

transactions (the need to check for a node failure which could cause the transaction to abort). The other potential cause of an abort mentioned earlier, namely deterministic logic in the transaction (e.g., a transaction should be aborted if inventory is zero), does not necessarily have to be performed as part of an agreement protocol at the end of a transaction. Rather, each node involved in a transaction waits for a one-way message from each node that could potentially deterministically abort the transaction, and only commits once it receives these messages. These messages can be sent as soon as a node knows it has finished with any transactional logic that could possibly cause the transaction to abort (instead of at the end of the transaction during a commit protocol).

3. SYSTEM DESIGN

We designed Calvin for deployment across large clusters of inexpensive commodity machines. For fault tolerance, Calvin relies heavily on replication. Although it is possible to deploy unreplicated instances of Calvin, in such a configuration system downtime will be incurred as a result of any node failure in the system, as a recovery process replays input transactions from a checkpoint in order to repair the failed node.

Calvin is designed to serve as a scalable transactional layer above any storage system that implements a basic CRUD interface (Create/insert, Read, Update, and Delete). Although it is possible to run Calvin on top of distributed nontransactional storage systems such as SimpleDB or Cassandra, it is more straightforward to explain the architecture of Calvin assuming that the storage system is not distributed out of the box. In this section, we describe the architecture of a standard Calvin deployment in which data is partitioned across local storage systems distributed across many machines (“nodes”)—and then the full database is replicated across multiple clusters (“replicas”). Calvin orchestrates all network communication that must occur between replicas (and between nodes within each replica) in the course of transaction execution.

Figure 1 shows an example Calvin deployment that replicates the database three times (replicas a, b, and c), and partitions each replica across six machines—all in a completely shared-nothing cluster. We refer to each machine using an identifier M_{ij}, where i indicates the partition stored at the machine, and j specifies the replica to which the machine belongs.

The essence of Calvin lies in separating the system into three separate layers of processing, each distributed across every machine M_{ij} participating in the deployment.

—The *sequencing layer* (or “sequencer”) intercepts transactional inputs and places them into a global transactional input sequence—this sequence will be the order of transactions to which all replicas will ensure serial equivalence during their execution. The sequencer therefore also handles the replication and logging of this input sequence.

—The *scheduling layer* (or “scheduler”) orchestrates transaction execution using a deterministic locking scheme to guarantee equivalence to the serial order specified by the sequencing layer while allowing transactions to be executed concurrently by a pool of transaction execution threads. (Although they are shown below the scheduler components in Figure 1, these execution threads conceptually belong to the scheduling layer.)

—The *storage layer* handles all physical data layout. Calvin transactions access data using a simple CRUD interface; any storage engine supporting a similar interface can be plugged into Calvin fairly easily.

By separating the replication mechanism, transactional functionality, and concurrency control (in the sequencing and scheduling layers) from the storage system, the design

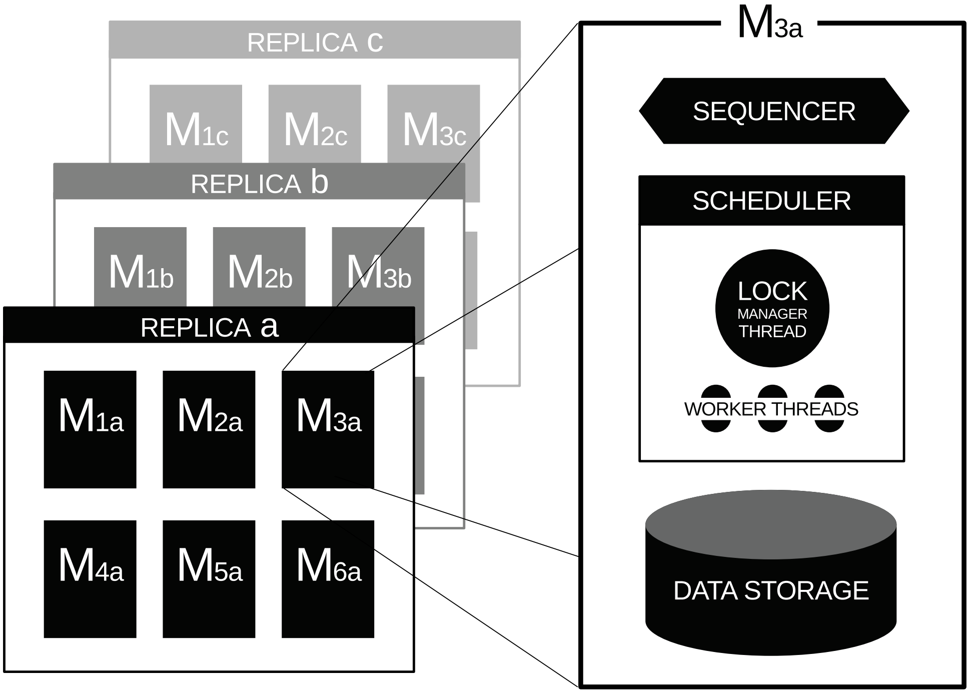


Fig. 1. A Calvin deployment is distributed across a shared-nothing cluster of machines and consists of multiple system replicas, each partitioned across many machines. Each machine consists of a sequencer component, a scheduler component, and a data storage component.

of Calvin deviates significantly from traditional database design which is highly monolithic, with physical access methods, buffer manager, lock manager, and log manager highly integrated and cross-reliant. This decoupling makes it impossible to implement certain popular recovery and concurrency control techniques such as the physiological logging in ARIES and next-key locking technique to handle phantoms (i.e., using physical surrogates for logical properties in concurrency control). Calvin is not the only attempt to separate the transactional components of a database system from the data components—thanks to cloud computing and its highly modular services, there has been a renewed interest within the database community in separating these functionalities into distinct and modular system components [Lomet et al. 2009].

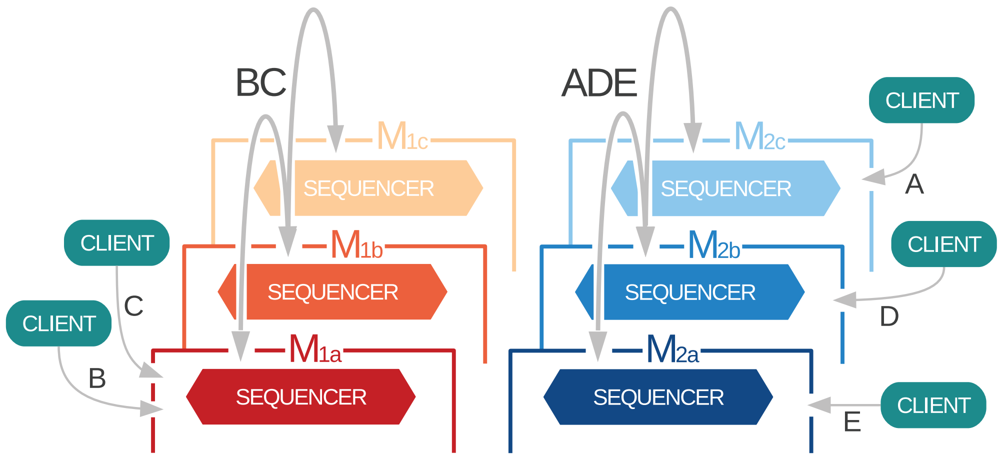
3.1. Sequencer and Replication

In previous work with deterministic database systems, we implemented the sequencing layer's functionality as a simple echo server—a single node which accepted transaction requests, logged them to disk, and forwarded them in timestamp order to the appropriate database nodes within each replica [Thomson and Abadi 2010]. The problems with single-node sequencers are: (a) that they represent potential single points of failure and (b) that as systems grow the constant throughput bound of a single-node sequencer brings overall system scalability to a quick halt. Calvin's sequencing layer is distributed across all system replicas, and also partitioned across every machine within each replica.

Calvin divides time into 10-millisecond epochs during which every machine's sequencer component collects transaction requests from clients. At the end of each epoch, all requests that have arrived at a sequencer node are compiled into a batch. This is the point at which replication of transactional inputs (discussed shortly) occurs.

In each epoch, once every sequencer has successfully replicated its batch, all the batches are logically considered to have been appended (in order by sequencer ID) to the global transaction request sequence. Note, however, that this interleaving doesn't physically occur at any machine. Instead, after replicating a batch, each sequencer sends a message to the scheduler on every partition within its replica containing: (1) the sequencer's unique node ID, (2) the epoch number (which is synchronously incremented across the entire system once every 10 ms), and (3) all transaction inputs collected in which the recipient will need to participate. This allows every scheduler to piece together its own view of a global transaction order by interleaving (in a deterministic, round-robin manner) all sequencers' batches for that epoch.

To illustrate this process, we will consider a six-machine Calvin deployment consisting of three replicas, each partitioned across two nodes. Suppose that clients submit five transactions (let's call them **A**, **B**, **C**, **D**, and **E**) to the sequencer layer during a particular epoch (specifically, to nodes M_{2c}, M_{1a}, M_{1a}, M_{2b}, and M_{2a}, respectively). At the end of that epoch, each partition's set of replicas forges an agreement about what should appear in the batch: sequencers at M_{1a}, M_{1b}, and M_{1c} agree on a batch containing **B** then **C**, and sequencers at M_{2a}, M_{2b}, and M_{2c} agree on a batch containing **A** then **D** then **E**.



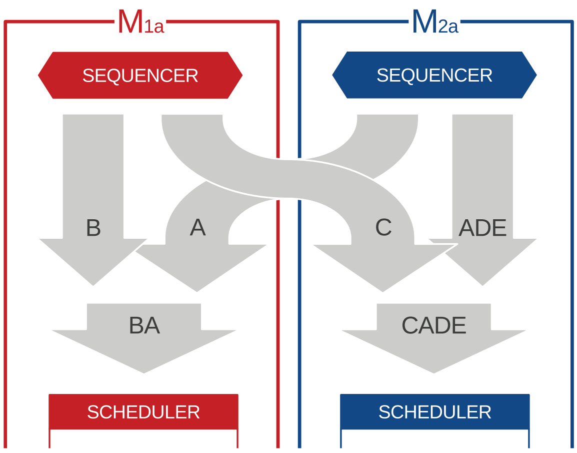
Once these agreements are reached, the global sequence is considered to include these batches, appended in partition order.

B C A D E

At this point, for each transaction in a batch, each sequencer node derives which partitions contain data that will be accessed by that transaction (see Section 3.2.1), and sends to each scheduler inside the same replica as the sequencer a list of transaction requests for all transactions in that batch that access data local to that scheduler. For example, suppose that records are partitioned such that partition 1 (machines M_{1a}, M_{1b}, and M_{1c}) stores records in the alphabetical range [Aaron, Buffy], and partition 2 (machines M_{2a}, M_{2b}, and M_{2c}) stores records in the range [Caleb, Zeus]; and suppose that the four transactions are balance transfers as follows.

A	Alice	$\rightarrow$	Dale
B	Bob	$\rightarrow$	Alice
C	Charles	$\rightarrow$	Elise
D	Dale	$\rightarrow$	Elise
E	Elise	$\rightarrow$	Frances

Since transaction **B** only accesses records on partition 1, it is only forwarded to schedulers at M_{1a}, M_{1b}, and M_{1c}. Similarly, transactions **C**, **D**, and **E**, which only access records on partition 2, are only forwarded to partition 2 of each replica. Transaction **A**, however, accesses one record on partition 1 and one record on partition 2, so that request is forwarded to schedulers at *both* partitions in every replica.



Here, we see how the sequencers in replica a therefore forward the epoch's transaction request batches to scheduler components. The same process takes place independently at replicas b and c as well.

3.1.1. Synchronous and Asynchronous Replication. Calvin currently supports two modes for replicating transactional input: asynchronous replication and Paxos-based synchronous replication. In both modes, nodes are organized into *replication groups*, each of which contains all replicas of a particular partition. In the deployment in Figure 1, for example, partition 1 in replica A and partition 1 in replica B would together form one replication group.

In asynchronous replication mode, one replica is designated as a master replica, and all transaction requests are forwarded immediately to sequencers located at nodes of this replica. After compiling each batch, the sequencer component on each master node forwards the batch to all other (slave) sequencers in its replication group. This has the advantage of extremely low latency before a transaction can begin being executed at the master replica, at the cost of significant complexity in failover. On the failure of a master sequencer, agreement has to be reached between all nodes in the same replica *and* all members of the failed node's replication group regarding: (a) which batch was the last valid batch sent out by the failed sequencer and (b) exactly what transactions

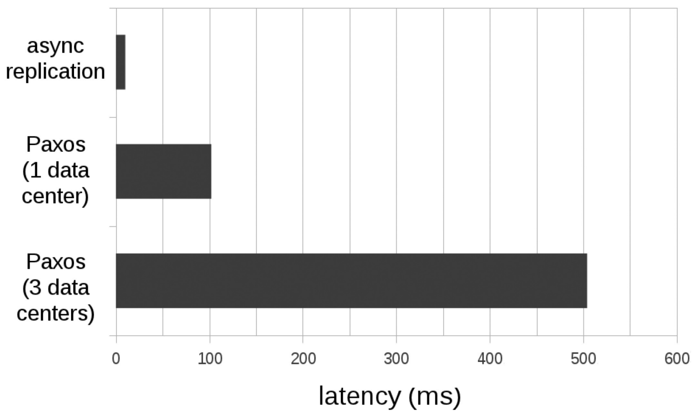


Fig. 2. Average transaction latency under Calvin's different replication modes.

that batch contained, since each scheduler is only sent the partial view of each batch that it actually needs in order to execute.

Calvin also supports Paxos-based synchronous replication of transactional inputs. In this mode, all sequencers within a replication group use Paxos to agree on a combined batch of transaction requests for each epoch.

Calvin's current implementation uses ZooKeeper, a highly reliable distributed coordination service often used by distributed database systems for heartbeats, configuration synchronization, and naming [Hunt et al. 2010]. ZooKeeper is not optimized for storing high data volumes, and may incur higher total latencies than the most efficient possible Paxos implementations. However, ZooKeeper handles the necessary *throughput* to replicate Calvin's transactional inputs for all the experiments run in this article and, since this synchronization step does not extend contention footprints, transactional throughput is completely unaffected by this preprocessing step. Improving the Calvin codebase by implementing a more streamlined Paxos agreement protocol between Calvin sequencers than what comes out of the box with ZooKeeper could be useful for latency-sensitive applications, but would not improve Calvin's transactional throughput.

Figure 2 presents average transaction latencies for the current Calvin codebase under different replication modes. The preceding data was collected using 4 EC2 High-CPU machines per replica, running 40000 microbenchmark transactions per second (10000 per node), 10% of which were multipartition (see Section 7 for additional details on our experimental setup). Both Paxos latencies reported used three replicas (12 total nodes). When all replicas were run on one datacenter, ping time between replicas was approximately 1 ms. When replicating across datacenters, one replica was run on Amazon's East U.S. (Virginia) datacenter, one was run on Amazon's West U.S. (Northern California) datacenter, and one was run on Amazon's EU (Ireland) datacenter. Ping times between replicas ranged from 100 ms to 170 ms. Total transactional throughput was not affected by changing Calvin's replication mode.

3.1.2. Low-Isolation Reads and Single-Partition Snapshot Reads. Some read-only database operations need not be processed by the sequencer and inserted into the global ordering prior to execution. In particular, when a read operation is guaranteed to access only data on a single partition, no synchronization is required between multiple replicas or multiple partitions to make sure the client sees a consistent snapshot of data—instead, the scheduler (discussed in Section 3.2 shortly) may choose an arbitrary point at in the sequencer order at which to execute the read with ACID (snapshot) semantics.

Furthermore, for clients whose latency requirements are more stringent than their consistency concerns, read-only operations that span multiple data partitions may be sent directly to the corresponding schedulers. Each scheduler will execute its local part of the operation with snapshot consistency, but since the schedulers have not communicated, each one may choose to perform the snapshot read as of a different version. The resulting isolation level the client will observe in this case is *repeatable read*.

Note that if the data storage backend provides strong multiversioning semantics (i.e., exposes arbitrary historical snapshots of data using a transaction sequence number for timestamp), clients may explicitly specify the sequence number at which multipartition snapshot reads and send these directly to the schedulers, which perform them on the specified snapshot of the database. If a client specifies a *future* sequence number, schedulers will wait until all transactions that appear earlier in the transaction sequence have committed to perform the reads.

3.2. Scheduler and Concurrency Control

Every machine in a Calvin deployment includes a scheduler component. Each scheduler accepts the transaction requests forwarded by the sequencing layer and executes them in a manner that is equivalent to serial execution in the order specified by the sequencer. This section describes the locking protocol used to accomplish this while achieving high concurrency.

When the transactional component of a database system is unbundled from the storage component, it can no longer make any assumptions about the physical implementation of the data layer, and cannot refer to physical data structures like pages and indexes, nor can it be aware of side-effects of a transaction on the physical layout of the data in the database. Both the logging and concurrency protocols have to be completely logical, referring only to record keys rather than physical data structures. Fortunately, the inability to perform physiological logging is not at all a problem in deterministic database systems; since the state of a database can be completely determined from the input to the database, logical logging is straightforward (the input is logged by the sequencing layer, and occasional checkpoints are taken by the storage layer—see Section 6 for further discussion of checkpointing in Calvin).

However, only having access to logical records is slightly more problematic for concurrency control, since locking ranges of keys and being robust to phantom updates typically require physical access to the data. To handle this case, Calvin could use an approach proposed recently for another unbundled database system by creating virtual resources that can be logically locked in the transactional layer [Lomet and Mokbel 2009], although implementation of this feature remains future work.

Calvin's deterministic lock manager is partitioned across the entire scheduling layer, and each node's scheduler is only responsible for locking records that are stored at that node's storage component—even for transactions that access records stored on other nodes. The locking protocol resembles strict two-phase locking, but with two added invariants.

—For any pair of transactions α and β that both request exclusive locks on some local record R, if transaction α appears before β in the serial order provided by the sequencing layer then α must request its lock on R before β does. In practice, Calvin implements this by serializing all lock requests in a single thread. The thread scans the serial transaction order sent by the sequencing layer; for each entry, it requests all locks that the transaction will need in its lifetime. (All transactions are therefore required to declare their full read/write sets in advance; Section 3.2.1 discusses the limitations entailed.)

—The lock manager must grant each lock to requesting transactions strictly in the order in which those transactions requested the lock. So in the previous example, $\boldsymbol{\beta}$ could not be granted its lock on R until after $\boldsymbol{\alpha}$ has acquired the lock on R, executed to completion, and released the lock.

Clients specify transaction logic as C++ functions that may access any data using a basic CRUD interface. Transaction code does not need to be at all aware of partitioning (although the user may specify elsewhere how keys should be partitioned across machines), since Calvin intercepts all data accesses that appear in transaction code and performs all remote read result forwarding automatically.

Once a transaction has acquired all of its locks under this protocol (and can therefore be safely executed in its entirety) it is handed off to a worker thread to be executed. Note that, since each scheduler only manages locks for its own local data, each distributed transaction actually goes through the full scheduling process independently at each participating node, and all participants independently execute all logic in the transaction.

Each actual transaction execution by a worker thread proceeds in five phases.

(1) *Read/write set analysis.* The first thing a transaction execution thread does when handed a transaction request is analyze the transaction's read and write sets, noting: (a) the elements of the read and write sets that are stored locally (i.e., at the node on which the thread is executing), and (b) the set of participating nodes at which elements of the write set are stored. These nodes are called *active participants* in the transaction; participating nodes at which only elements of the read set are stored are called *passive participants*.
(2) *Perform local reads.* Next, the worker thread looks up the values of all records in the read set that are stored locally. Depending on the storage interface, this may mean making a copy of the record to a local buffer, or just saving a pointer to the location in memory at which the record can be found.
(3) *Serve remote reads.* All results from the local read phase are forwarded to counterpart worker threads on every *actively* participating node. Since passive participants do not modify any data, they need not execute the actual transaction code, and therefore do not have to collect any remote read results. If the worker thread is executing at a passively participating node, then it is finished after this phase.
(4) *Collect remote read results.* If the worker thread is executing at an actively participating node, then it must execute transaction code, and thus it must first acquire all read results—both the results of local reads (acquired in the second phase) and the results of remote reads (forwarded appropriately by every participating node during the third phase). In this phase, the worker thread collects the latter set of read results.
(5) *Transaction logic execution and applying writes.* Once the worker thread has collected all read results, it proceeds to execute all transaction logic, applying any *local* writes. Nonlocal writes can be ignored, since they will be viewed as local writes by the counterpart transaction execution thread at the appropriate node, and applied there.

Assuming a distributed transaction begins executing at approximately the same time at every participating node (which is not always the case—this is discussed in greater length in Section 7), all reads occur in parallel, and all remote read results are delivered in parallel as well, with no need for worker threads at different nodes to request data from one another at transaction execution time.

To illustrate this, let us revisit the example we introduced in Section 3.1 before.[4] Recall that five balance transfer transactions were submitted and assigned the following sequence.

sequence number	txn id	read/write set	participating partitions
1	**B**	Bob, Alice	1
2	**C**	Charles, Elise	2
3	**A**	Alice, Dale	1, 2
4	**D**	Dale, Elise	2
5	**E**	Elise, Frances	2

Partition 1's scheduler has therefore received the request batch

B A

for this epoch, and partition 2's scheduler has received the following batch.

C A D E

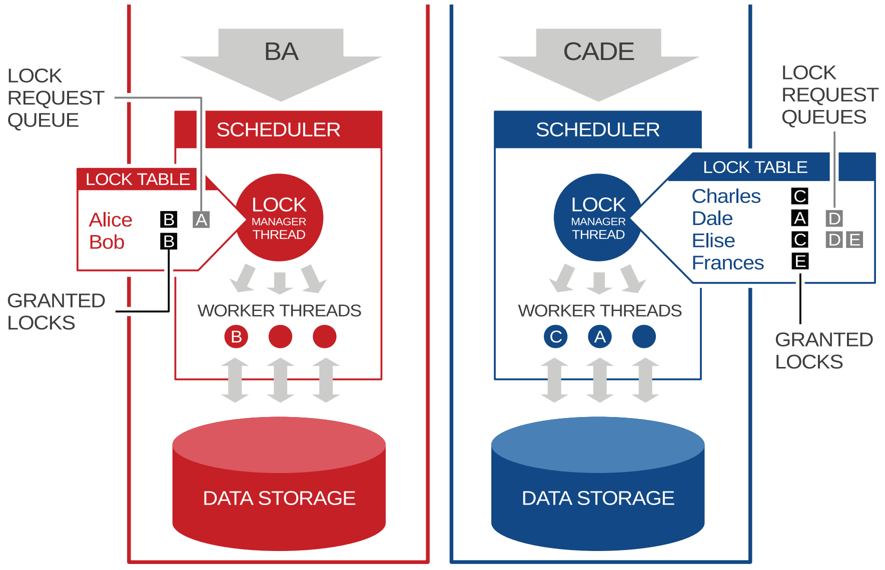
Each scheduler's lock manager processes these requests in order, adding all lock requests that each transaction will need on *locally stored* records. At partition 1, therefore, the lock manager first adds lock request for transaction **B** on records Alice and Bob, which are immediately granted. Transaction **B** is handed to a worker thread for execution. The lock manager then adds transaction **A**'s lock request for record Alice—but not for record Dale, also accessed by **A**, since each node's lock manager only handles locking of *local* records. Since **B** already holds a lock on Alice, **A** is not granted its lock, and therefore does not yet begin execution.

Meanwhile, partition 2's lock manager adds lock requests for **C**, **A**, **D**, and **E** as follows.

—**C**'s lock requests for Charles and Elise are immediately granted, so **C** begins execution in a worker thread.
—**A**'s *local* lock request for Dale is immediately granted, so a worker thread begins running **A** as well.
—**D**'s lock requests for Dale and Elise both fail, since these locks are already held by **A** and **C**, respectively. **D** is not yet handed off to a worker thread to run.
—**E**'s lock request for Frances is granted, while **E**'s lock request for Dale fails. Since it has not acquired all locks, **E** also does not begin execution in a worker thread.

At this point, **B** has started running on partition 1, and **C** and **A** have started running on partition 2.

[4]Since all replicas have already agreed on transaction request ordering for the epoch in question, the remainder of the execution pipeline does not require inter-replica communication. We therefore examine here how execution proceeds within a single replica.



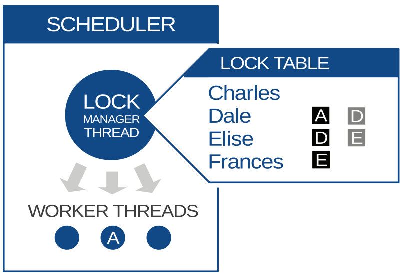
Let us first examine what happens at partition 2 during the five-phase executions of **C** and **A**.

—*Transaction* ***C***. Phase 1 (read/write set analysis) determines that partition 2 is the only active participant in the transaction, and there are no passive participants. In phase 2 (*local reads*), **C**'s worker thread reads records Charles and Elise. Phases 3 and 4 (serve remote reads, and collect remote read results) are no-ops, since partition 2 is **C**'s only participant. In phase 5, **C**'s transaction logic is executed (using the read results from phase 2), and its writes are installed to the database.

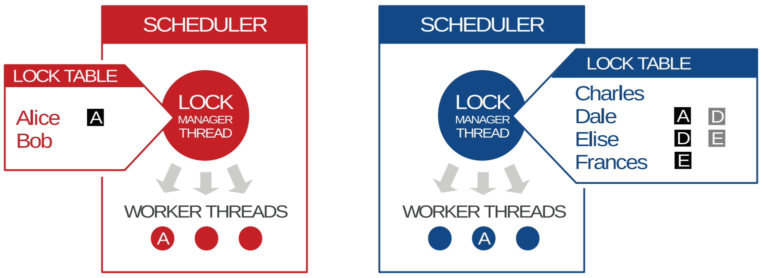
—*Transaction* ***A***. Phase 1 determines that partitions 1 and 2 are both active participants in **A**. In phase 2, **A**'s worker thread reads the *local* record Dale, and in phase 3, it broadcasts the value of record Dale to all other participants—in this case partition 1. In phase 4, it collects the results of partition 1's read of record Alice. But recall that **A** hasn't started executing yet at partition 1, because **B** held the lock it needed. Partition 2's worker thread blocks at this point, waiting for that result.[5]

At this point, **C** has executed to completion at partition 2. It therefore releases its locks on records Charles and Elise. Note that both **D** and **E** have outstanding lock requests on record Elise. Note also that **D** is also blocked waiting for record Dale, while **E** has already acquired all of its other locks and would be able to execute immediately if granted a lock on Elise. A traditional (nondeterministic) lock manager might therefore grant the lock on Elise to **E** in order to maximize concurrency. However, this behavior would not emulate serial execution in the sequencer-determined order **BCADE**, since **E** would explicitly read, modify, and write record Elise before **D** did. Calvin's deterministic locking mechanism therefore grants the lock on Elise to the next transaction to have requested it, namely **D**.

[5]If the remote read result is not been received within a specified time interval, the partition will actively request it. Partition 1 will examine its recent network log and resend the result. If it continues to not arrive (for example, because the local replica's partition 1 is crashed), partition 2 will request it from another replica's partition 1.



Now let's revisit partition 1. While **C** executed to completion at partition 2 (and **A** went through its first three execution phases), **B** began executing at partition 1. **B** executes to completion without having to coordinate with any other participants (so phases 2 and 3 are no-ops) and releases its locks. **A** is then granted its lock on Alice and allowed to start executing on partition 1 as well.



At partition 1, **A**'s worker thread discovers in phase 1 that it is one of two active participants. It reads record Alice in phase 2, and sends the result to partition 2 in phase 3. In phase 4, it collects the remote read result that was already sent to it by **A**'s worker thread at partition 2 in *its* phase 3. Now, with full knowledge of **A**'s read set, it proceeds with phase 5 and executes **A**'s transaction logic. When it goes to apply the writes (recall that **A** updates records Alice and Dale), it applies only those that are local: it updates record Alice and drops its write to record Dale. The lock manager releases **A**'s lock on Alice, and **A** has now completed at partition 1.

When the **A**'s worker thread at partition 2 receives the value of Alice that was just sent from 1, it now has the full knowledge of **A**'s read set and can run phase 5. Since **A**'s transaction logic is deterministic given the same read set, **A**'s execution at partition 2 will mirror its execution at 1, but at partition 2 only the update of record Dale will be applied.

Once **A**'s lock on record Dale is released, **D** inherits it and executes to completion, after which **E** is finally granted its lock on record Frances and executes to completion as well.

3.2.1. Dependent Transactions. Transactions which must perform reads in order to determine their full read/write sets (which we term *dependent transactions*) are not natively supported in Calvin since Calvin's deterministic locking protocol requires advance knowledge of all transactions' read/write sets before transaction execution can begin. Instead, Calvin supports a scheme called Optimistic Lock Location Prediction (OLLP), which can be implemented at very low overhead cost by modifying the client transaction code itself [Thomson and Abadi 2010]. The idea is for dependent transactions to be preceded by an inexpensive, low-isolation, unreplicated, read-only *reconnaissance* *query* that performs all the necessary reads to discover the transaction's full read/write set. The *actual* transaction is then sent to be added to the global sequence and executed, using the reconnaissance query's results for its read/write set. Because it is possible for the records read by the reconnaissance query (and therefore the actual transaction's read/write set) to have changed between the execution of the reconnaissance query and the execution of the actual transaction, the read results must be rechecked, and the process may have to be (deterministically) restarted if the "reconnoitered" read/write set is no longer valid.

Particularly common within this class of transactions are those that must perform secondary index lookups in order to identify their full read/write sets. Since secondary indexes tend to be comparatively expensive to modify, they are seldom kept on fields whose values are updated extremely frequently. Secondary indexes on "inventory item *name*" or "New York Stock Exchange stock *symbol*", for example, would be common, whereas it would be unusual to maintain a secondary index on more volatile fields such as "inventory item *quantity*" or "NYSE stock *price*". One therefore expects the OLLP scheme seldom to result in repeated transaction restarts under most common real-world workloads.

The TPC-C benchmark's "Payment" transaction type is an example of this subclass of transaction. And since the TPC-C benchmark workload *never* modifies the index on which Payment transactions' read/write sets may depend, Payment transactions never have to be restarted when using OLLP.

In Section 7.4, we examine throughput and latency characteristics when using OLLP on workloads containing dependent transactions under a number of conditions, in particular when we vary the update rate of the secondary index on which the transactions' read and write sets depend.

4. DEVELOPER-FRIENDLY SERVER-SIDE TRANSACTIONS

Most commercially available database systems allow transactions to be defined and executed in the form of either ad hoc client-side transactions or server-side stored procedures.

Embedding client-side transactions in application code is extremely convenient for developers and generally leads to very readable, maintainable code. For example, the left side of Figure 3 shows a simple balance transfer transaction in a Python application. Here, when the application calls the `BalanceTransfer` function, a sequence of synchronous interactions with the database system occur, starting with a "begin transaction" request, and ending with a "commit transaction" action. We chose this simple `BalanceTransfer` transaction as an example here because it represents an extremely common transactional idiom—an "interaction" between two or more database objects that are not expected to be stored on the same partition in which records corresponding to both objects are read, modified, and written. It is easy to envision very similar transactions appearing in a finance application (e.g., as a stock trade), in an e-commerce application (e.g., as a customer paying an outstanding balance), in a massively multiplayer online game (e.g., as a player buying a magical sword from another player), in a messaging application (e.g., as one user sending a message to another), and so on.

Alternatively, implementing a transaction such as `BalanceTransfer` as a stored procedure in a DBMS would likely result in superior performance. Stored procedures are generally faster than ad hoc client-side transactions for two reasons.

(1) Stored procedures are precompiled and saved by the DBMS. Each actual invocation therefore skips steps such as SQL parsing, query planning, etc.

client-side transaction execution

```
# Define transaction.
def BalanceTransfer(from, to, amount):
  begin_transaction()

  # Check that 'from' balance is
  # sufficient for transfer. (Abort
  # transaction if not.)
  from_balance = execute_sql(
      "SELECT balance FROM user
       WHERE user_id='%s'"
          % from
  )
  if (int(from_balance) < amount):
    abort_transaction()
    return

  # Decrement 'from' balance.
  execute_sql(
      "UPDATE user
       SET balance = balance - %d
       WHERE user_id='%s'"
          % (amount, from)
  )

  # Increment 'to' balance.
  execute_sql(
      "UPDATE user
       SET balance = balance + %d
       WHERE user_id='%s'"
          % (amount, to)
  )

  commit_transaction()

# Invoke transaction.
BalanceTransfer("alice", "bob", 10)
```

code snippet executed server-side

```
ExecAsTxn("""
  # Define transaction.
  def BalanceTransfer(from, to, amount):
    begin_transaction()

    # Check that 'from' balance is
    # sufficient for transfer. (Abort
    # transaction if not.)
    from_balance = execute_sql(
        "SELECT balance FROM user
         WHERE user_id='%s'"
            % from
    )
    if (int(from_balance) < amount):
      abort_transaction()
      return

    # Decrement 'from' balance.
    execute_sql(
        "UPDATE user
         SET balance = balance - %d
         WHERE user_id='%s'"
            % (amount, from)
    )

    # Increment 'to' balance.
    execute_sql(
        "UPDATE user
         SET balance = balance + %d
         WHERE user_id='%s'"
            % (amount, to)
    )

    commit_transaction()

  # Invoke transaction.
  BalanceTransfer("alice", "bob", 10)
""")
```

Fig. 3. An example client-side transaction (left side). One simple method of executing this transaction server side would be for the developer to put the code snippet into a string and send the string to the database server, where it could be interpreted as a transaction (right side).

(2) Transactions that involve multiple SQL statements with application logic in between don't require multiple round-trip messages between the application server and the database system. This decreases overall latency and network IO costs—but more importantly, it significantly reduces transactions' total contention footprints.

However, stored procedures come with additional costs with respect to code complexity. Implementing the aforesaid transaction as a stored procedure would require the following additional steps.

(1) Translate the transaction code to whatever programming language is used for stored procedures in the DBMS being used.
(2) Register the transaction with the database system.
(3) Invoke the stored procedure using the stored procedure's name and arguments.

While this sounds reasonably simple, each of these steps can involve significant complexity. First, languages for implementing stored procedures vary drastically between database systems, making applications that rely on stored procedures significantly less portable and requiring developers to be familiar with additional DBMS-specific languages and features.

```
include/txn.h

// Abstract stored procedure class.
class Txn {
 public:
  void SetArgs(const vector<string>& args);
  void SetReadSet(const vector<Key>& readset);
  void SetWriteSet(const vector<Key>& writeset);
  void GetStatus(Status* status);
  void GetResults(string* results);

  // User-defined txn logic.
  virtual void Execute() = 0;
  ...
};
```

```
application/user-txns.h

// List of all registered
// stored procedures.
enum TxnType {
  ...
  MyTxn = 42,
  ...
};

// Definition of a particular
// stored procedure.
class MyTxn : public Txn {
 public:
  virtual void Execute() {
    ...
  }
};
```

Fig. 4. The native method of defining a transaction in Calvin is to extend the abstract `Txn` class. This results in very high performance but can be difficult for developers to reason about and can make application debugging more complex.

The second step, registering the procedure, must be carried out every time the application is modified, built, or deployed. This can introduce particular challenges in large-scale application deployments where application updates happen in a "rolling" manner rather than synchronously across the entire deployment.

Third, application testing and debugging can be made much more difficult when stored procedures are used in place of client-side transactions for several reasons.

(1) Server-side error/exception logs must be merged with application output logs to get a clear view of any errors that occurred within the transaction.
(2) Testing of application logic that appears inside a transaction requires a full database deployment. This can make unit testing of individual aspects of transaction behavior significantly harder than for client-side transactions.
(3) When running the application in a debugger (e.g., `pdb` for a Python application), code inside the stored procedure cannot be examined in a step-by-step execution.

As discussed in Section 2, Calvin's transaction execution protocol requires transaction logic to be defined in advance of executing the transaction, and annotated with the transaction's read/write sets (either by the user, through static code analysis, or as a result of the reconnaissance phase for dependent transaction). In order to achieve this and gain the performance advantages of stored procedures, we implemented Calvin such that the "native" mode for defining transactions is as an extension of a C++ class; a module containing user-defined transactions is linked with Calvin at system build/deployment time (Figure 4). While these stored procedures are extremely fast to execute due to their simplicity, implementing transactions this way clearly comes with all the drawbacks of stored procedures as implemented by commercially available DBMSs. In fact, the restrictions and problems with this model are considerably worse for several reasons.

(1) Modifying or adding transaction classes requires the entire database system to be taken down, rebuilt, and redeployed.
(2) If a user-defined `MyTxn::Execute()` function invokes any nondeterministic program logic (e.g., random number generation, accessing system clock), replica state may diverge.

(3) Buggy transaction code running completely unsandboxed in C++ can crash, stall, or corrupt the entire database system. We have observed while implementing the experiments in this article that bad transaction code can interact with database system logic to cause problems that are difficult and time consuming to debug.

Note, however, that the invariant required by Calvin's transaction scheduling protocol is not that every transaction must be a stored procedure, but rather that each transaction's logic and read/write sets must be known in advance of beginning execution of that particular transaction. To demonstrate that Calvin's transaction scheduling and replication schemes can be made to work in environments where application developers would normally prefer to define simpler client-side transactions, we devised a toolset for making server-side transaction execution less painful for developers. Our prototype for the methods we've developed targets Python as the application language, although the techniques we propose are largely language independent.

4.1. Server-Side Transactions without Stored Procedures

The basic idea is to allow applications to submit code snippets to the database system, which can then be interpreted and executed on-the-fly (for example, by an embedded Python interpreter) on the database server—with full replication and transaction guarantees. The most obvious way of implementing snippet transactions would be to provide an API call

```
ExecAsTxn(<CODE-SNIPPET>)
```

allowing the application developer to simply construct and submit a string containing the snippet of Python code—exactly as it would appear if implemented as a client-side transaction. The right side of Figure 3 shows how our example `BalanceTransfer` transaction might be implemented using this scheme. Here, very little work is required on the part of the developer to turn a client-side transaction into a server-side transaction.

This method of constructing and sending code snippets for server-side execution still comes with certain drawbacks, however, as detailed next.

(1) No local or remote IO (e.g., reading/writing files, communicating with other servers, etc.) could be permitted in the code snippet.
(2) The code snippet would not be able to reference any global variables or functions defined elsewhere in the application.
(3) The application developer would have to explicitly construct strings containing snippet code rather than defining functions directly.

Certain of these issues (remote IO limitations in particular) are inherent in Calvin's determinism guarantee. However, it is possible to address these other issues to make this scheme of executing code snippets as transactions more usable. In particular, we have devised a Python module that uses Python's reflection support to access and analyze the full application code and selectively send the appropriate information to the database system for execution. Python explicitly exposes the code for every function and method, so it is always possible at runtime to look up this code. So if a function `MyClass.MyFun` is defined

```
class MyClass:
  def MyFun(self):
    print ‘Hello, world!’
```

in `src/example.py`, then in addition to calling the function,

```
>>> MyClass().MyFun()
Hello, world!
```

one can also look up the precise file and line number where it is defined.

```
>>> MyClass.MyFun.func_code.co_filename
'src/example.py'
>>> MyClass.MyFun.func_code.co_firstlineno
2
```

We use this to allow application developers to write what looks like a client-side transaction that can still be executed server side. To resume our earlier example, using this library, one could implement the `BalanceTransfer` transaction as shown on the left in Figure 3, but would then invoke it by calling the following.

```
ExecAsTxn(BalanceTransfer, "alice", "bob", 10)
```

rather than

```
BalanceTransfer("alice", "bob", 10)
```

A global flag determines whether the `ExecAsTxn` causes the `BalanceTransfer` function to be called client side (i.e., directly by the machine running the application code) or server side (i.e., by the database server). This is designed to make it extremely painless to switch between client-side transactions (which may be easier to test and debug) and server-side transactions. If the flag is set to client-side execution, `ExecAsTxn` simply calls the specified function and returns. If the flag is set to server-side execution, `ExecAsTxn` performs several tasks.

(1) Python's support for reflection is used to identify the Python source file in which the function body is defined.[6]
(2) The function body is parsed and scanned for the following.
    (a) *Uses of Python's `time` or `random` modules*. If either of these appears, `ExecAsTxn` records the current application server system time for the database server to later use in place of its own system time (including to use as the default deterministic seed for pseudorandom number generation).
    (b) *Prohibited nondeterministic operations such as reading from a file or performing network I/O*. In this case, an error is thrown. (Note that *writing* to a file during a transaction is allowed—when execution occurs at the database server, writes can be buffered to be returned to the client and applied afterwards.)
    (c) *References to global variables and nonbuilt-in functions that are defined elsewhere in the application*.
(3) If the function body referenced any global variables, their current values are recorded to send to the database server.[7]
(4) If the function body called any nonbuilt-in functions, `ExecAsTxn` recursively performs a similar lookup-parse-scan-analyze procedure on those functions' bodies.
(5) Other additional contextual information such as Python interpreter version, active compiler flags, etc., are then recorded, and all data required to duplicate function execution is compiled into a Python script sent to the database server.

[6]This is extremely easy to do in interpreted languages such as Python, Perl, and Ruby due to runtime reflection support. For applications written in compiled languages with very different runtime environments, for example, C++ and Java, additional data structures would have to be constructed and saved at compilation and linking time in order to make this possible. While this may introduce additional implementation difficulty, the overall approach would be the same for these languages.
[7]This is only guaranteed to be possible for global variables of reasonably simple type such as integers, strings, basic structs, and other types that are explicitly serializable. This could represent a significant limitation of this scheme when implemented for application languages such as Haskell, in which lazy evaluation is commonly used to construct infinite data structures.

The database server receiving the request may then treat it as any other dependent transaction,[8] except that when Calvin's scheduler hands the transaction off to an execution thread, that thread starts up an embedded Python interpreter to execute the code sent by `ExecAsTxn` rather than invoking a stored procedure. Upon completion of the transaction, the database system notifies the client of: (a) the commit/abort status of the transaction, (b) any file output events that occurred (such as debug logging), and (c) the final states of any global variables that were modified by the transaction code.

Figure 5 illustrates this process given an extended implementation of the `BalanceTransfer` transaction described before, which accesses global variables and external functions, writes output to files, and makes use of Python's `time` and `random` modules.

Overall, we find that defining database transactions using this scheme looks and feels like using standard client-side database transactions. We envision that this could be a useful set of tools in the design of future database systems' developer APIs—regardless of whether server-side execution is critical for supporting transactions or merely desirable for improving performance.

5. CALVIN WITH DISK-BASED STORAGE

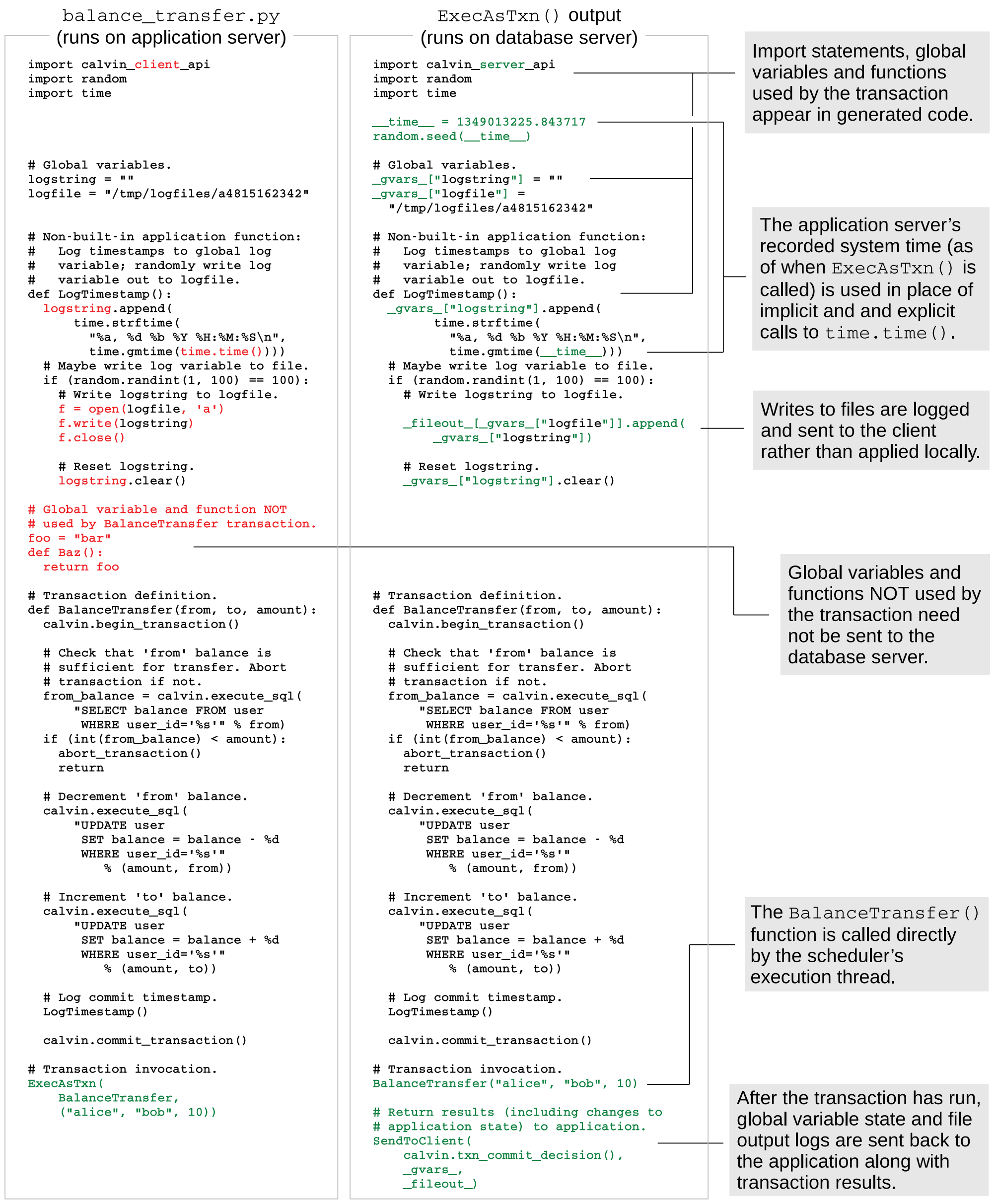
Our previous work on deterministic database systems came with the caveat that deterministic execution would only work for databases entirely resident in main memory [Thomson and Abadi 2010]. The reasoning was that a major disadvantage of deterministic database systems relative to traditional nondeterministic systems is that nondeterministic systems are able to guarantee equivalence to *any* serial order, and can therefore arbitrarily reorder transactions, whereas a system like Calvin is constrained to respect whatever order the sequencer chooses.

For example, if a transaction (let's call it A) is stalled waiting for a disk access, a traditional system would be able to run other transactions (B and C, say) that do not conflict with the locks already held by A. If B and C's write sets overlapped with A's on keys that A has not yet locked, then execution can proceed in manner equivalent to the serial order $B-C-A$ rather than $A-B-C$. In a deterministic system, however, B and C would have to block until A completed. Worse yet, other transactions that conflicted with B and C—but not with A—would also get stuck behind A. On-the-fly reordering is therefore highly effective at maximizing resource utilization in systems where disk stalls upwards of 10 ms may occur frequently during transaction execution.

Calvin avoids this disadvantage of determinism in the context of disk-based databases by following its guiding design principle: move as much as possible of the heavy lifting to earlier in the transaction processing pipeline, before locks are acquired.

Any time a sequencer component receives a request for a transaction that may incur a disk stall, it introduces an artificial delay before forwarding the transaction request to the scheduling layer and meanwhile sends requests to all relevant storage components to "warm up" the disk-resident records that the transaction will access. If the

[8]Since the read and write sets are not specified by the submitted Python code, a reconnaissance step is generally required to determine their membership in advance of beginning transaction execution. See Section 3.2.1 for additional discussion of the OLLP procedure used for handling dependent transactions. In some transactions where records are always identified by primary key, certain syntactic analysis may be sufficient to allow the reconnaissance step to be omitted. In particular, there exist tools developed in the programming languages research community (primarily for code security verification purposes) that track all variable dependencies. If, using such a tool, the primary keys of accessed records could be determined not to depend on the results of previous database accesses within the transaction, but only on a priori known quantities such as explicit string constants and function arguments, then the read and write sets could be deduced through purely syntactic analysis. The full implementation of such a tool remains as future work, however.



balance_transfer.py
(runs on application server)

```
import calvin_client_api
import random
import time



# Global variables.
logstring = ""
logfile = "/tmp/logfiles/a4815162342"


# Non-built-in application function:
#   Log timestamps to global log
#   variable; randomly write log
#   variable out to logfile.
def LogTimestamp():
  logstring.append(
      time.strftime(
        "%a, %d %b %Y %H:%M:%S\n",
        time.gmtime(time.time())))
  # Maybe write log variable to file.
  if (random.randint(1, 100) == 100):
    # Write logstring to logfile.
    f = open(logfile, 'a')
    f.write(logstring)
    f.close()

    # Reset logstring.
    logstring.clear()

# Global variable and function NOT
# used by BalanceTransfer transaction.
foo = "bar"
def Baz():
  return foo

# Transaction definition.
def BalanceTransfer(from, to, amount):
  calvin.begin_transaction()

  # Check that 'from' balance is
  # sufficient for transfer. Abort
  # transaction if not.
  from_balance = calvin.execute_sql(
      "SELECT balance FROM user
       WHERE user_id='%s'" % from)
  if (int(from_balance) < amount):
    abort_transaction()
    return

  # Decrement 'from' balance.
  calvin.execute_sql(
      "UPDATE user
       SET balance = balance - %d
       WHERE user_id='%s'"
         % (amount, from))

  # Increment 'to' balance.
  calvin.execute_sql(
      "UPDATE user
       SET balance = balance + %d
       WHERE user_id='%s'"
         % (amount, to))

  # Log commit timestamp.
  LogTimestamp()

  calvin.commit_transaction()

# Transaction invocation.
ExecAsTxn(
    BalanceTransfer,
    ("alice", "bob", 10))
```

ExecAsTxn() output
(runs on database server)

```
import calvin_server_api
import random
import time

__time__ = 1349013225.843717
random.seed(__time__)

# Global variables.
_gvars_["logstring"] = ""
_gvars_["logfile"] =
  "/tmp/logfiles/a4815162342"


# Non-built-in application function:
#   Log timestamps to global log
#   variable; randomly write log
#   variable out to logfile.
def LogTimestamp():
  _gvars_["logstring"].append(
      time.strftime(
        "%a, %d %b %Y %H:%M:%S\n",
        time.gmtime(__time__)))
  # Maybe write log variable to file.
  if (random.randint(1, 100) == 100):
    # Write logstring to logfile.

    _fileout_[_gvars_["logfile"]].append(
        _gvars_["logstring"])

    # Reset logstring.
    _gvars_["logstring"].clear()






# Transaction definition.
def BalanceTransfer(from, to, amount):
  calvin.begin_transaction()

  # Check that 'from' balance is
  # sufficient for transfer. Abort
  # transaction if not.
  from_balance = calvin.execute_sql(
      "SELECT balance FROM user
       WHERE user_id='%s'" % from)
  if (int(from_balance) < amount):
    abort_transaction()
    return

  # Decrement 'from' balance.
  calvin.execute_sql(
      "UPDATE user
       SET balance = balance - %d
       WHERE user_id='%s'"
         % (amount, from))

  # Increment 'to' balance.
  calvin.execute_sql(
      "UPDATE user
       SET balance = balance + %d
       WHERE user_id='%s'"
         % (amount, to))

  # Log commit timestamp.
  LogTimestamp()

  calvin.commit_transaction()

# Transaction invocation.
BalanceTransfer("alice", "bob", 10)

# Return results (including changes to
# application state) to application.
SendToClient(
    calvin.txn_commit_decision(),
    _gvars_,
    _fileout_)
```

Import statements, global variables and functions used by the transaction appear in generated code.

The application server's recorded system time (as of when ExecAsTxn() is called) is used in place of implicit and and explicit calls to time.time().

Writes to files are logged and sent to the client rather than applied locally.

Global variables and functions NOT used by the transaction need not be sent to the database server.

The BalanceTransfer() function is called directly by the scheduler's execution thread.

After the transaction has run, global variable state and file output logs are sent back to the application along with transaction results.

Fig. 5. An example code transformation as performed by ExecAsTxn. The left box shows a transaction (a more complex version of the BalanceTransfer transaction shown in Figure 3) defined as a client-side Python function that: (a) makes use of global variables and functions; (b) uses Python's time and random modules; and (c) writes output to a file. When the transaction is invoked using ExecAsTxn, the Python source file is examined and the code snippet in the right box is produced and sent to the database server to be executed.

artificial delay is greater than or equal to the time it takes to bring all the disk-resident records into memory, then when the transaction is actually executed, it will access only memory-resident data. Note that with this scheme the overall latency for the transaction should be no greater than it would be in a traditional system where the disk IO were performed during execution (since exactly the same set of disk operations occur in either case)—but none of the disk latency adds to the transaction's contention footprint.

To clearly demonstrate the applicability (and pitfalls) of this technique, we implemented a simple disk-based storage system for Calvin in which "cold" records are written out to the local file system and only read into Calvin's primary memory-resident key-value table when needed by a transaction. When running 10,000 microbenchmark transactions per second per machine (see Section 7 for more details on experimental setup), Calvin's total transactional throughput was unaffected by the presence of transactions that access disk-based storage, as long as no more than 0.9% of transactions (90 out of 10,000) are to disk. However, this number is very dependent on the particular hardware configuration of the servers used. We ran our experiments on low-end commodity hardware, and so we found that the number of disk-accessing transactions that could be supported was limited by the maximum *throughput* of local disk (rather than contention footprint). Since the microbenchmark workload involved random accesses to a lot of different files, 90 disk-accessing transactions per second per machine was sufficient to turn disk random access throughput into a bottleneck. With higher-end disk arrays (or with flash memory instead of magnetic disk) many more disk-based transactions could be supported without affecting total throughput in Calvin.

To better understand Calvin's potential for interfacing with other disk configurations, flash, networked block storage, etc., we also implemented a storage engine in which "cold" data was stored in memory on a separate machine that could be configured to serve data requests only after a prespecified delay (to simulate network or storage-access latency). Using this setup, we found that each machine was able to support the same load of 10,000 transactions per second, no matter how many of these transactions accessed "cold" data—even under extremely high contention (contention index = 0.01).

We found two main challenges in reconciling deterministic execution with disk-based storage. First, disk latencies must be accurately predicted so that transactions are delayed for the appropriate amount of time. Second, Calvin's sequencer layer must accurately track which keys are in memory across all storage nodes in order to determine when prefetching is necessary.

5.1. Disk I/O Latency Prediction

Accurately predicting the time required to fetch a record from disk to memory is not an easy problem. The time it takes to read a disk resident can vary significantly for many reasons:

—variable physical distance for the head and spindle to move;
—prior queued disk I/O operations;
—network latency for remote reads;
—failover from media failures;
—multiple I/O operations required due to traversing a disk-based data structure (e.g., a B+ tree).

It is therefore impossible to predict latency perfectly, and any heuristic used will sometimes result in underestimates and sometimes in overestimates. Disk IO latency estimation proved a particularly interesting and crucial parameter when tuning Calvin to perform well on disk-resident data under high contention.

We found that if the sequencer chooses a conservatively high estimate and delays forwarding transactions for longer than is likely necessary, the contention cost due to disk access is minimized (since fetching is almost always completed before the transaction requires the record to be read), but at a cost to overall transaction latency. Excessively high estimates could also result in the memory of the storage system being overloaded with "cold" records waiting for the transactions that requested them to be scheduled.

However, if the sequencer underestimates disk I/O latency and does not delay the transaction for long enough, then it will be scheduled too soon and stall during execution until all fetching completes. Since locks are held for the duration, this may come with high costs to the contention footprint and therefore overall throughput.

There is therefore a fundamental trade-off between total transactional latency and contention when estimating for disk I/O latency. In both experiments described earlier, we tuned our latency predictions so at least 99% of disk-accessing transactions were scheduled after their corresponding prefetching requests had completed. Using the simple file-system-based storage engine, this meant introducing an artificial delay of 40 ms, but this was sufficient to sustain throughput even under very high contention (contention index = 0.01). Under lower contention (contention index ≤ 0.001), we found that no delay was necessary beyond the default delay caused by collecting transaction requests into batches, which averages 5 ms. A more exhaustive exploration of this particular latency-contention trade-off would be an interesting avenue for future research, particularly as we experiment further with hooking Calvin up to various commercially available storage engines.

5.2. Globally Tracking Hot Records

In order for the sequencer to accurately determine which transactions to delay scheduling while their read sets are warmed up, each node's sequencer component must track what data is currently in memory across the entire system—not just the data managed by the storage components co-located on the sequencer's node. Although this was feasible for our experiments in this article, this is not a scalable solution. If global lists of hot keys are not tracked at every sequencer, one solution is to delay all transactions from being scheduled until adequate time for prefetching has been allowed. This protects against disk seeks extending contention footprints, but incurs latency at every transaction. Another solution (for single-partition transactions only) would be for schedulers to track their local hot data synchronously across all replicas, and then allow schedulers to deterministically decide to delay requesting locks for single-partition transactions that try to read cold data. A more comprehensive exploration of this strategy, including investigation of how to implement it for multipartition transactions, remains future work.

6. CHECKPOINTING

Deterministic database systems have two properties that are useful for ensuring fault tolerance. First, they are actively replicated, and active replication allows clients to instantaneously fail over to another replica in the event of a crash.

Second, only the transactional input is logged—there is no need to pay the overhead of physical REDO logging. Replaying history of transactional input is sufficient to recover the database system to the current state. However, it would be inefficient (and ridiculous) to replay the entire history of the database from the beginning of time upon every failure. Instead, Calvin periodically takes a checkpoint of full database state in order to provide a starting point from which to begin replay during recovery.

Calvin supports three checkpointing modes: naïve synchronous checkpointing, an asynchronous variation of Cao et al.'s Zig-Zag algorithm [Cao et al. 2011], and an asynchronous snapshot mode that is supported only when the storage layer supports full multiversioning.

The first mode uses the redundancy inherent in an actively replicated system in order to create a system checkpoint. The system can periodically freeze an entire replica and produce a full versioned snapshot of the system. Since this only happens at one replica at a time, the period during which a replica is unavailable can be hidden from the client by routing client requests to nonfrozen replicas.

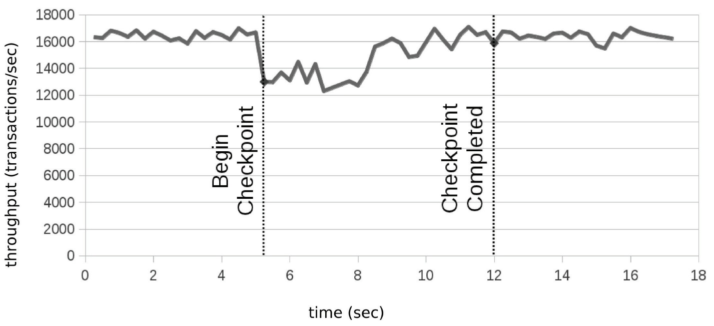


Fig. 6. Throughput over time during a typical checkpointing period using Calvin's modified Zig-Zag scheme.

One problem with this approach is that the replica taking the checkpoint may fall significantly behind other replicas, which can be problematic if it is called into action due to a hardware failure in another replica. In addition, it may take the replica significant time for it to catch back up to other replicas, especially in a heavily loaded system.

Calvin's second checkpointing mode is closely based on Cao et al.'s Zig-Zag algorithm [Cao et al. 2011]. Zig-Zag stores two copies of each record in a given datastore, $AS[K]_0$ and $AS[K]_1$, plus two additional bits per record, $MR[K]$ and $MW[K]$ (where K is the key of the record). $MR[K]$ specifies which record version should be used when reading record K from the database, and $MW[K]$ specifies which version to overwrite when updating record K. So new values of record K are always written to $AS[K]_{MW[K]}$, and $MR[K]$ is set equal to $MW[K]$ each time K is updated.

Each checkpoint period in Zig-Zag begins with setting $MW[K]$ equal to $\neg MR[K]$ for all keys K in the database during a physical point of consistency in which the database is entirely quiesced. Thus $AS[K]_{MW[K]}$ always stores the latest version of the record, and $AS[K]_{\neg MW[K]}$ always stores the last value written prior to the beginning of the most recent checkpoint period. An asynchronous checkpointing thread can therefore go through every key K, logging $AS[K]_{\neg MW[K]}$ to disk without having to worry about the record being clobbered.

Taking advantage of Calvin's global serial order, we implemented a variant of Zig-Zag that does *not* require quiescing the database to create a physical point of consistency. Instead, Calvin captures a snapshot with respect to a *virtual* point of consistency, which is simply a prespecified point in the global serial order. When a virtual point of consistency approaches, Calvin's storage layer begins keeping two versions of each record in the storage system—a "before" version, which can only be updated by transactions that precede the virtual point of consistency, and an "after" version, which is written to by transactions that appear after the virtual point of consistency. Once all transactions preceding the virtual point of consistency have completed executing, the "before" versions of each record are effectively immutable, and an asynchronous checkpointing thread can begin checkpointing them to disk. Once the checkpoint is completed, any duplicate versions are garbage-collected: all records that have both a "before" version and an "after" version discard their "before" versions, so that only one record is kept of each version until the next checkpointing period begins.

Whereas Calvin's first checkpointing mode described before involves stopping transaction execution entirely for the duration of the checkpoint, this scheme incurs only moderate overhead while the asynchronous checkpointing thread is active. Figure 6 shows Calvin's maximum throughput over time during a typical checkpoint capture period. This measurement was taken on a single-machine Calvin deployment running our microbenchmark under low contention (see Section 7 for more on our experimental setup). However, it should be noted that these results serve merely as a demonstration of the feasibility of Calvin's durability in the face of site failures; further research is forthcoming.

Although there is some reduction in total throughput due to: (a) the CPU cost of acquiring the checkpoint and (b) a small amount of latch contention when accessing records, writing stable values to storage asynchronously does not increase lock contention or transaction latency.

Calvin is also able to take advantage of storage engines that explicitly track all recent versions of each record in addition to the current version. Multiversion storage engines allow read-only queries to be executed without acquiring any locks, reducing overall contention and total concurrency control overhead at the cost of increased memory usage. When running in this mode, Calvin's checkpointing scheme takes the form of an ordinary "SELECT *" query over all records, where the query's result is logged to a file on disk rather than returned to a client.

7. PERFORMANCE AND SCALABILITY

To investigate Calvin's performance and scalability characteristics under a variety of conditions, we ran a number of experiments using two benchmarks: the TPC-C benchmark and a microbenchmark we created in order to have more control over how benchmark parameters are varied. Except where otherwise noted, all experiments were run on Amazon EC2 using High-CPU/Extra-Large instances, which promise 7GB of memory and 20 EC2 Compute Units—8 virtual cores with 2.5 EC2 Compute Units each.[9]

7.1. TPC-C Benchmark

The TPC-C benchmark consists of several classes of transactions, but the bulk of the workload—including almost all distributed transactions that require high isolation—is made up by the New Order transaction, which simulates a customer placing an order on an e-commerce application. Since the focus of our experiments is on distributed transactions, we limited our TPC-C implementation to only New Order transactions. We would expect, however, to achieve similar performance and scalability results if we were to run the complete TPC-C benchmark.

Figure 7 shows total and per-machine throughput (TPC-C New Order transactions executed per second) as a function of the number of Calvin nodes, each of which stores a database partition containing 10 TPC-C warehouses. To fully investigate Calvin's handling of distributed transactions, multi-warehouse New Order transactions (about 10% of total New Order transactions) always access a second warehouse that is *not* on the same machine as the first.

Because each partition contains 10 warehouses and New Order updates one of 10 "districts" for some warehouse, at most 100 New Order transactions can be executing concurrently at any machine (since there are no more than 100 unique districts per partition, and each New Order transaction requires an exclusive lock on a district). Therefore, it is critical that the time that locks are held is minimized, since the throughput of the system is limited by how fast these 100 concurrent transactions complete (and release locks) so that new transactions can grab exclusive locks on the districts and get started.

If Calvin were to hold locks during an agreement protocol such as two-phase commit for distributed New Order transactions, throughput would be severely limited (a detailed comparison to a traditional system implementing two-phase commit is given in Section 7.3). Without the agreement protocol, Calvin is able to achieve around 5,000

[9]Each EC2 Compute Unit provides the roughly the CPU capacity of a 1.0 to 1.2 GHz 2007 Opteron or 2007 Xeon processor.

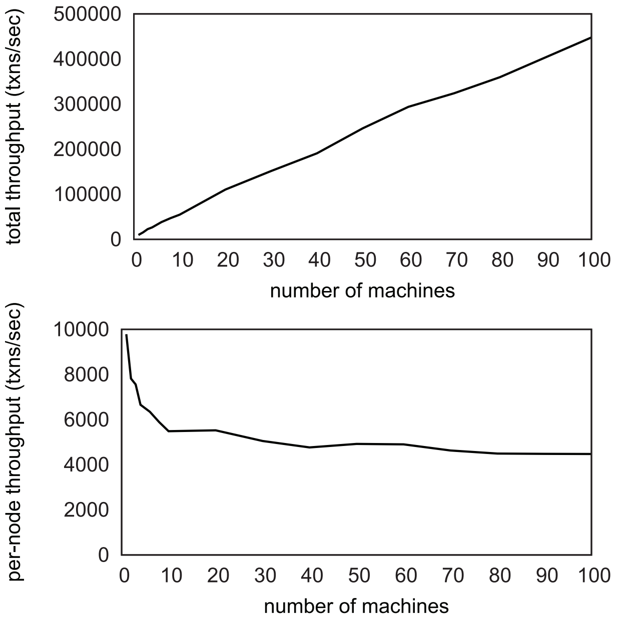


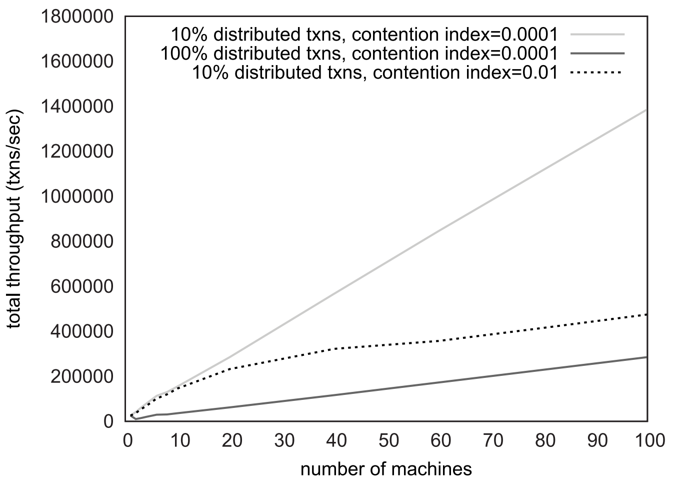
Fig. 7. Total and per-node TPC-C (100% New Order) throughput, varying deployment size.

transactions per second per node in clusters larger than 10 nodes, and scales linearly. (The reason why Calvin achieves more transactions per second per node on smaller clusters is discussed in the next section.) Our Calvin implementation is therefore able to achieve nearly half a million TPC-C transactions per second on a 100-node cluster. It is notable that the present TPC-C world-record holder (Oracle) runs 504,161 New Order transactions per second, despite running on much higher-end hardware than the machines we used for our experiments [Carlile 2010].

7.2. Microbenchmark Experiments

To more precisely examine the costs incurred when combining distributed transactions and high contention, we implemented a microbenchmark that shares some characteristics with TPC-C's New Order transaction, while reducing overall overhead and allowing finer adjustments to the workload. Each transaction in the benchmark reads 10 records, performs a constraint check on the result, and updates a counter at each record if and only if the constraint check passed. Of the 10 records accessed by the microbenchmark transaction, one is chosen from a small set of "hot" records,[10] and the rest are chosen from a very much larger set of records—except when a microbenchmark transaction spans two machines, in which case it accesses one "hot" record on *each* machine participating in the transaction. By varying the number of "hot" records, we can finely tune contention. In the subsequent discussion, we use the term *contention index* to refer to the fraction of the total "hot" records that are updated when a transaction executes at a particular machine. A contention index of 0.001 therefore means that each transaction chooses one out of one thousand "hot" records to update at each participating machine (i.e., at most 1,000 transactions could ever be executing

[10]Note that this is a different use of the term "hot" than that used in the discussion of caching in our earlier discussion of memory- versus disk-based storage engines.



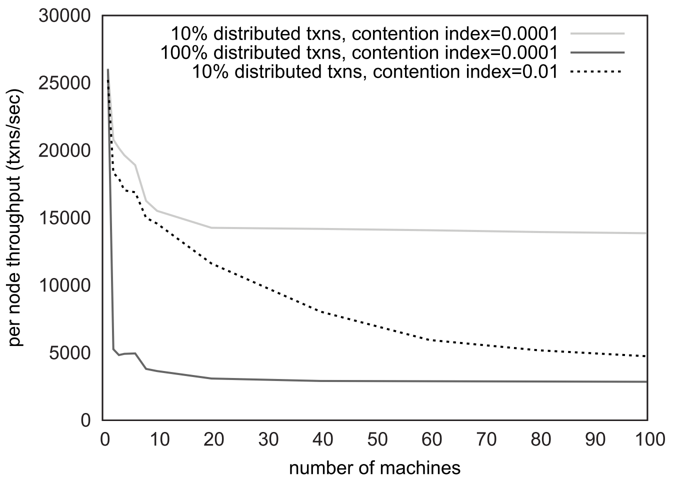


Fig. 8. Total and per-node microbenchmark throughput, varying deployment size.

concurrently), while a contention index of 1 would mean that every transaction touches *all* "hot" records (i.e., transactions must be executed completely serially).

Figure 8 shows experiments in which we scaled the microbenchmark to 100 Calvin nodes under different contention settings and with varying numbers of distributed transactions. When adding machines under very low contention (contention index = 0.0001), throughput per node drops to a stable amount by around 10 machines and then stays constant, scaling linearly to many nodes. Under higher contention (contention index = 0.01, which is similar to TPC-C's contention level), we see a longer, more gradual per-node throughput degradation as machines are added, more slowly approaching a stable amount.

Multiple factors contribute to the shape of this scalability curve in Calvin. In all cases, the sharp drop-off between one machine and two machines is a result of the CPU cost of additional work that must be performed for every multipartition transaction:

—serializing and deserializing remote read results;
—additional context switching between transactions waiting to receive remote read results;

—setting up, executing, and cleaning up after the transaction at *all* participating machines, even though it is counted *only once* in total throughput.

After this initial drop-off, the reason for further decline as more nodes are added—even when both the contention and the number of machines participating in any distributed transaction are held constant—is quite subtle. Suppose, under a high-contention workload, that machine A starts executing a distributed transaction that requires a remote read from machine B, but B hasn't gotten to that transaction yet (B may still be working on earlier transactions in the sequence, and it can not start working on the transaction until locks have been acquired for all previous transactions in the sequence). Machine A may be able to begin executing some other nonconflicting transactions, but soon it will simply have to wait for B to catch up before it can commit the pending distributed transaction and execute subsequent conflicting transactions. By this mechanism, there is a limit to how far ahead of or behind the pack any particular machine can get. The higher the contention, the tighter this limit. As machines are added, two things happen.

—*Slow machines.* Not all EC2 instances yield equivalent performance, and sometimes an EC2 user gets stuck with a slow instance. Since the experimental results shown in Figure 8 were obtained using the same EC2 instances for all three lines and all three lines show a sudden drop between 6 and 8 machines, it is clear that a slightly slow machine was added when we went from 6 nodes to 8 nodes.
—*Execution progress skew.* Every machine occasionally gets slightly ahead of or behind others due to many factors, such as OS thread scheduling, variable network latencies, and random variations in contention between sequences of transactions. The more machines there are, the more likely at any given time there will be at least one that is slightly behind for some reason.

The sensitivity of overall system throughput to execution progress skew is strongly dependent on two factors.

—*Number of machines.* The fewer machines there are in the cluster, the more each additional machine will increase skew. For example, suppose each of n machines spends some fraction k of the time contributing to execution progress skew (i.e. falling behind the pack). Then at each instant there would be a $1 - (1 - k)^n$ chance that at least one machine is slowing the system down. As n grows, this probability approaches 1, and each additional machine has less and less of a skewing effect.
—*Level of contention.* The higher the contention rate, the more likely each machine's random slowdowns will be to cause *other* machines to have to slow their execution as well. Under low contention (contention index = 0.0001), we see per-node throughput decline sharply only when adding the first few machines, then flatten out at around 10 nodes, since the diminishing increases in execution progress skew have relatively little effect on total throughput. Under higher contention (contention index = 0.01), we see an even sharper initial drop, and then it takes many more machines being added before the curve begins to flatten, since even small incremental increases in the level of execution progress skew can have a significant effect on throughput.

7.3. Handling High Contention

Most real-world workloads have low contention most of the time, but the appearance of small numbers of *extremely* hot data items is not infrequent. We therefore experimented with Calvin under the kind of workload that we believe is the primary reason that so few practical systems attempt to support distributed transactions: combining many multipartition transactions with very high contention. In this experiment we therefore

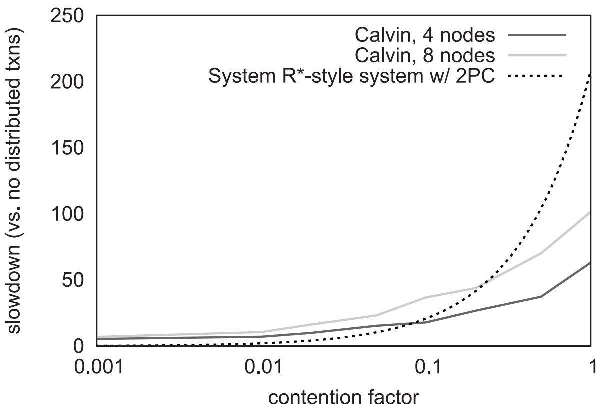


Fig. 9. Slowdown for 100% multipartition workloads, varying contention index.

do not focus on the entirety of a realistic workload, but instead we consider only the subset of a workload consisting of high-contention multipartition transactions. Other transactions can still conflict with these high-conflict transactions (on records besides those that are very hot), so the throughput of this subset of an (otherwise easily scalable) workload may be tightly coupled to overall system throughput.

Figure 9 shows the factor by which 4-node and 8-node Calvin systems are slowed down (compared to running a perfectly partitionable, low-contention version of the same workload) while running 100% multipartition transactions, depending on contention index. Recall that contention index is the fraction of the total set of hot records locked by each transaction, so a contention index of 0.01 means that up to 100 transactions can execute concurrently, while a contention index of 1 forces transactions to run completely serially.

Because modern implementations of distributed systems do not implement System R*-style distributed transactions with two-phase commit, and comparisons with any earlier-generation systems would not be an apples-to-apples comparison, we include for comparison a simple model of the contention-based slowdown that would be incurred by this type of system. We assume that in the nonmultipartition, low-contention case this system would get similar throughput to Calvin (about 27,000 microbenchmark transactions per-second per-machine). To compute the slowdown caused by multipartition transactions, we consider the extended contention footprint caused by two-phase commit. Since given a contention index C at most $1/C$ transactions can execute concurrently, a system running 2PC at commit time can never execute more than

$$\frac{1}{C * D_{2PC}}$$

total transactions per second where D_{2PC} is the duration of the two-phase commit protocol.

Typical round-trip ping latency between nodes in the same EC2 datacenter is around 1 ms, but including delays of message multiplexing, serialization/deserialization, and thread scheduling, one-way latencies in our system between transaction execution threads are almost never less than 2 ms, and usually longer. In our model of a system similar in overhead to Calvin, we therefore expect for locks to be held for approximately 8 ms on each distributed transaction. Note that this model is somewhat naïve since the contention footprint of a transaction is assumed to include nothing but the latency of two-phase commit. Other factors that contribute to Calvin's actual slowdown are completely ignored in this model, including:

—CPU costs of multipartition transactions;
—latency of reaching a local commit/abort decision before starting 2PC (which may require additional remote reads in a real system);
—execution progress skew (all nodes are assumed to begin execution of each transaction and the ensuing 2PC in perfect lockstep).

Therefore, the model does not establish a specific comparison point for our system, but a strong lower bound on the slowdown for such a system. In an actual System R*-style system, one might expect to see considerably more slowdown than predicted by this model in the context of high-contention distributed transactions.

There are two very notable features in the results shown in Figure 9. First, under low contention, Calvin gets the same approximately 5x to 7x slowdown—from 27,000 to about 5,000 (4 nodes) or 4,000 (8 nodes) transactions per second—as seen in the previous experiment going from 1 machine to 4 or 8. For all contention levels examined in this experiment, the difference in throughput between the 4-node and 8-node cases is a result of increased skew in workload execution progress between the different nodes; as one would predict, the detrimental effect of this skew to throughput is significantly worse at higher contention levels.

Second, as expected, at very high contentions, even though we ignore a number of the expected costs, the model of the system running two-phase commit incurs significantly more slowdown than Calvin. This is evidence that: (a) the distributed commit protocol is a major factor behind the decision for most modern distributed systems not to support ACID transactions and (b) Calvin alleviates this issue.

7.4. Dependent Transactions

To demonstrate the effectiveness of OLLP (described in Section 3.2.1) in the presence of dependent transactions in Calvin, we modified our microbenchmark so that sometimes (*D*% of the time, say) a secondary index lookup is required to determine the full read-write set. Recall that in OLLP each transaction begins with a reconnaissance step (in which the transactions read and write sets are predicted via low-isolation index lookups) *before* the transaction request is routed to the sequencer. The latency between the reconnaissance step and the actual transaction execution therefore includes the latency of the Paxos replication protocol. This latency also affects the likelihood of having to abort and restart the transaction. After all, as more time passes between the optimistic prediction and the final check, the index will be more likely to have been updated. To examine this effect, we ran our modified microbenchmark workload[11] varying three parameters.

—*The percentage D of transactions that are dependent transactions*. In particular, we looked at workloads consisting of 0%, 20%, 50%, and 100% dependent transactions.
—*Expected replication latency*. To model the three replication modes that were discussed in Section 3.1—asynchronous master-slave replication, within-datacenter Paxos, and WAN Paxos—we introduced artificial delays of 0, 100, and 500 milliseconds respectively during the transaction sequencing phase.[12]

[11]The experiments in this section were not run on EC2 machines, but on a local cluster at Yale with similar overall hardware specifications to those promised by EC2 High-CPU/Extra-Large instances: 2.6 GHz quad-core Intel Xeon X5550 processors providing 8 hardware threads; 12GB of memory.

[12]We chose to use artificial delays rather than running the actual replication protocols in order to limit the amount of random variation, making for a clearer analysis. Furthermore, for simplicity we limited the experiments in this section to single-partition transactions only. However, since replication latency generally

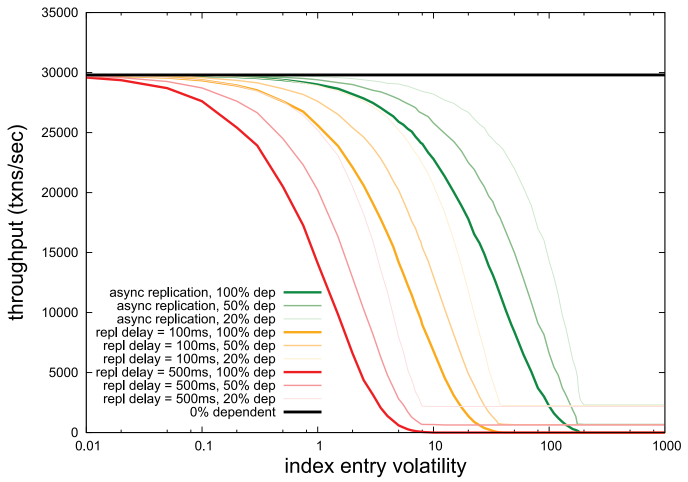


Fig. 10. Per-machine throughput for microbenchmark workloads with various dependent transaction compositions and replication latencies, as a function of index entry volatility.

—*Index entry volatility*. This is the average update rate of *each* entry in the index. This means that an index with one million entries and an index entry volatility of 0.1 is updated 100,000 times per second.

First, we examine throughput performance that can be achieved in the presence of dependent transactions. Figure 10 shows total throughput for various dependent microbenchmark workloads. We see here that, with higher replication latencies, performance is indeed more sensitive to high index update rates, whereas as replication latency is lower, it takes much higher index entry volatility for a significant adverse effect on throughput to be observed. For example, for workloads consisting fully of dependent transactions, it takes an index entry volatility of 1 to reduce throughput by 50% replication delay 500 ms, whereas volatility must reach 5 and 35 to cause 50% throughput decreases with replication delays of 100 ms and 0 ms respectively.

To better understand these results, we examined the number of times that dependent transactions are forced to restart due to failed optimistic checks. For example, with a volatility of 1 and a replication delay of 500 ms, one would expect about half of optimistic checks to fail. Therefore half of all dependent transactions would restart at least once, then half of these restarted transactions (so a quarter of total transactions) would be forced to restart once again, and so on. We would thus expect to see a total expected restart rate of

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^{i} = 1.$$

Similarly, we would expect roughly one-tenth of optimistic checks to fail with an index volatility of 1 and a replication delay of 100 ms, resulting in a restart rate of

$$\frac{1}{10} + \frac{1}{100} + \frac{1}{1000} + \cdots = \sum_{i=1}^{\infty} \left(\frac{1}{10}\right)^{i} = \frac{1}{9}.$$

dominates the delay between reconnaissance and execution steps, we expect the overall throughput and latency effects to be very similar in the presence of distributed transactions.

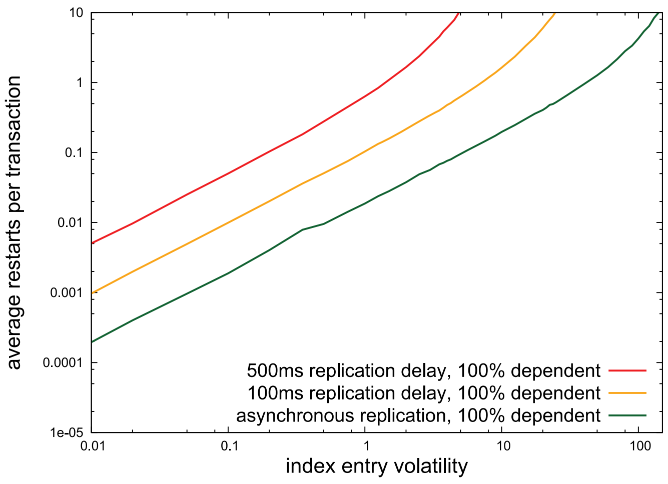


Fig. 11. Average number of times that a dependent transaction has to restart as a function of index entry volatility.

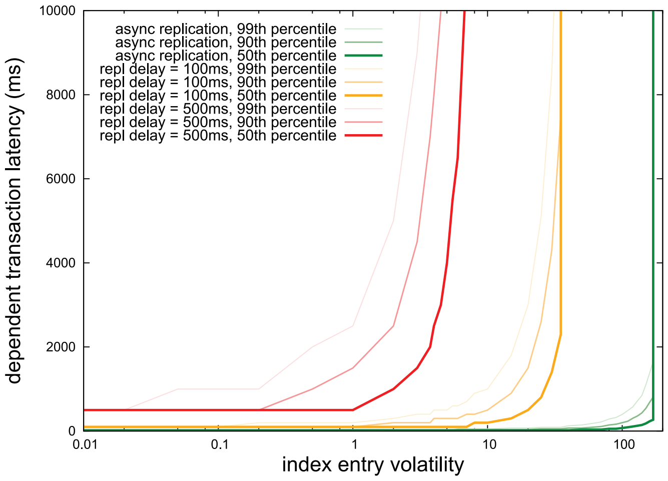


Fig. 12. Here we show 50th, 90th, and 99th percentile latency of dependent microbenchmark transactions as a function of index entry volatility.

With asynchronous replication, the timing between the reconnaissance phase and actual execution is usually between 10 and 20 milliseconds, which should result in an expected restart rate below 1/50 when volatility is 1.

Figure 11 shows the number of times on average that a dependent transaction restarts due to an index change between the reconnaissance phase and the actual transaction execution, depending on replication mode and index entry volatility. As predicted, high replication delays result in increased restart rates. In fact, at volatility 1 we see just the restart rates predicted previously: approximately 1 when replication delay is 500 ms, approximately 0.1 when replication is 100 ms, and slightly over 0.01 for asynchronous replication.

Figure 12 extends this analysis with actual transaction latencies. In particular, we plot the 50th, 90th, and 99th percentile latencies for dependent transactions as index volatility grows.

Once again, we see the same effect as given earlier, but "magnified" in each case by the imposed replication latency—we see that total latency is generally equal to the number of restarts *times* the replication delay associated with each restart. Once again high replication delays make performance more vulnerable to high index entry volatility. Yet once again, volatility must be upwards of 0.2 to see more than 1-second delays—even for the 99th percentile latency when performing Paxos over a WAN.

As discussed in Section 3.2.1, real-world databases typically maintain indexes only on tables with very low volatility. Recall that index entry volatility refers here to the total updates per second for *each* index entry. Therefore these analyses only examine the unusual case of extremely high index entry update rates. Secondary indexes are commonly maintained on fields such as user names, addresses, stock symbols, and other "static" attributes by which applications are likely to group and reference records. For such common real-world scenarios, we expect the restart costs of running OLLP on dependent transactions to be completely negligible.

8. RELATED WORK

Many of the contributions of this article—including the introduction of the general Calvin architecture and many of the performance measurements discussed here—have appeared in our previous work [Thomson et al. 2012]. This article's major contributions that did not appear in earlier publications include our scheme for implementing developer-friendly server-side transactions, introduced in Section 4, and our analysis of Calvin's performance in the presence of dependent transactions, which appears in Section 7.4.

One key contribution of the Calvin architecture is that it features active replication, where the same transactional input is sent to multiple replicas, each of which processes transactional input in a deterministic manner so as to avoid diverging. There have been several related attempts to actively replicate database systems in this way. Pacitti et al. [2003], Whitney et al. [1997], Stonebraker et al. [2007], and Jones et al. [2010] all propose performing transactional processing in a distributed database without concurrency control by executing transactions serially—and therefore equivalently to a *known* serial order—in a single thread on each node (where a node in some cases can be a single CPU core in a multicore server [Stonebraker et al. 2007]). By executing transactions serially, nondeterminism due to thread scheduling of concurrent transactions is eliminated, and active replication is easier to achieve. However, serializing transactions can limit transactional throughput, since if a transaction stalls (e.g., for a network read), other transactions are unable to take over. Calvin enables concurrent transactions while still ensuring logical equivalence to a given serial order. Furthermore, although these systems choose a serial order in advance of execution, adherence to that order is not as strictly enforced as in Calvin (e.g., transactions can be aborted due to hardware failures), so two-phase commit is still required for distributed transactions.

Each of the aforesaid works implements a system component analogous to Calvin's sequencing layer that chooses the serial order. Calvin's sequencer design most closely resembles the H-store design [Stonebraker et al. 2007], in which clients can submit transactions to any node in the cluster. Synchronization of inputs between replicas differs, however, in that Calvin can use either asynchronous (log-shipping) replication or Paxos-based, strongly consistent synchronous replication, while H-store replicates inputs by stalling transactions by the expected network latency of sending a transaction to a replica, and then using a deterministic scheme for transaction ordering assuming all transactions arrive from all replicas within this time window.

Bernstein et al.'s Hyder [Bernstein et al. 2011] bears conceptual similarities to Calvin despite extremely different architectural designs and approaches to achieving high scalability. In Hyder, transactions submit their “intentions”—buffered writes—after executing based on a view of the database obtained from a recent snapshot. The intentions are composed into a global order and processed by a deterministic “meld” function, which determines what transactions to commit and what transactions must be aborted (e.g., due to a data update that invalidated the transaction’s view of the database after the transaction executed, but before the meld function validated the transaction). Hyder’s globally ordered log of things-to-attempt-deterministically is comprised of the after-effects of transactions, whereas the analogous log in Calvin contains unexecuted transaction requests. However, Hyder’s optimistic scheme is conceptually very similar to the Optimistic Lock Location Prediction scheme (OLLP) discussed in Section 3.2.1. OLLP’s “reconnaissance” queries determine the transactional inputs, which are deterministically validated at “actual” transaction execution time in the same optimistic manner that Hyder’s meld function deterministically validates transactional results.

Lomet et al. propose “unbundling” transaction processing system components in a cloud setting in a manner similar to Calvin’s separation of different stages of the pipeline into different subsystems [Lomet et al. 2009]. Although Lomet et al.’s concurrency control and replication mechanisms do not resemble Calvin’s, both systems separate the Transactional Component (scheduling layer) from the Data Component (storage layer) to allow arbitrary storage backends to serve the transaction processing system depending on the needs of the application. Calvin also takes the unbundling one step further, separating out the sequencing layer, which handles data replication.

Google’s Megastore [Baker et al. 2011] and IBM’s Spinnaker [Rao et al. 2011] recently pioneered the use of the Paxos algorithm [Lamport 1998, 2001] for strongly consistent data replication in modern, high-volume transactional databases (although Paxos and its variants are widely used to reach synchronous agreement in countless other applications). Like Calvin, Spinnaker uses ZooKeeper [Hunt et al. 2010] for its Paxos implementation. Since they are not deterministic systems, both Megastore and Spinnaker must use Paxos to replicate transactional effects, whereas Calvin only has to use Paxos to replicate transactional inputs.

9. FUTURE WORK

In its current implementation, Calvin handles hardware failures by recovering the crashed machine from its most recent complete snapshot and then replaying all more recent transactions. Although other nodes within the same replica may depend on remote reads from the afflicted machine, throughput in the rest of the replica is apt to slow or halt until recovery is complete.

In the future we intend to develop a more seamless failover system. For example, failures could be made completely invisible with the following simple technique. The set of all replicas can be divided into replication subgroups—pairs or trios of replicas located near one another, generally on the same local area network. Outgoing messages related to multipartition transaction execution at a database node A in one replica are sent not only to the intended node B within the same replica, but also to every replica of node B within the replication subgroup—just in case one of the subgroup’s node A replicas has failed. This redundancy technique comes with various trade-offs and would not be implemented if inter-partition network communication threatened to be a bottleneck (especially since active replication in deterministic systems already provides high availability), but it illustrates a way of achieving a highly “hiccup free” system in the face of failures.

A good compromise between these two approaches might be to integrate a component that monitors each node’s status, which could then detect failures and carefully orchestrate quicker failover for a replica with a failed node by directing other replicas of the afflicted machine to forward their remote read messages appropriately. Such a component would also be well situated to oversee load balancing of read-only queries, dynamic data migration and repartitioning, and load monitoring.

Another area of future work involves investigation into deterministic lock management. The current implementation of Calvin uses a traditional lock management data structure, with a hash table that maps tuple keys to queues of transactions that have made lock requests on that tuple. However, in a deterministic system that is deadlock free and guarantees equivalence to a given serial order, the oldest active transaction is always guaranteed to be able to acquire all of its locks. Therefore, it is not necessary to track which transactions are waiting for which data items—all that is required is a simple count be tracked for each data item of how many transactions have made requests for that lock, and this count be incremented/decremented whenever transactions make requests/finish working on a particular data item. Whenever a transaction gets the status of being the oldest transaction in the system, it can immediately assume that any locks it failed to acquire originally (because the lock count was nonzero when it tried to acquire the lock) it now has automatically acquired, and can proceed without concern for other transactions that have also made requests on the same lock. This approach significantly reduces lock management overhead at the cost of potentially reducing concurrency due to the inability for one transaction to directly hand over the lock on a data item to the next transaction that made a request for it. We plan to investigate when exactly this approach is a good idea.

A third area of future work involves application of deterministic transaction execution to distributed file systems. Distributed file systems that store all name-space and file metadata on a single server suffer from both a potential single point of failure and a scalability bottleneck. However, if file metadata is partitioned across the memory of a shared-nothing cluster of commodity machines, then file updates turn into distributed transactions as both name-space and file metadata information must be updated in a single atomic transaction. Calvin could be used to scale these distributed transactions (and thus also scale the distributed file system) while also replicating the metadata for high availability.

10. CONCLUSIONS

This article presents Calvin, a transaction processing and replication layer designed to transform a generic, nontransactional, unreplicated data store into a fully ACID, consistently replicated distributed database system. Calvin supports horizontal scalability of the database and unconstrained ACID-compliant distributed transactions while supporting both asynchronous and Paxos-based synchronous replication, both within a single datacenter and across geographically separated datacenters. By using a deterministic framework, Calvin is able to eliminate distributed commit protocols, the largest scalability impediment of modern distributed systems. Consequently, Calvin scales near linearly and has achieved near-world-record transactional throughput on a simplified TPC-C benchmark.

Furthermore, we introduce a set of tools that could make defining server-side transactions easier for developers. This is essential for making deterministic systems like Calvin usable, since all transaction execution must occur server side, but this contribution could also be used to improve usability of other database systems in which server-side execution is not required but can still boost transactional throughput.

REFERENCES

D. J. Abadi. 2012. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *IEEE Comput.* 45, 2.

J. C. Anderson, J. Lehnardt, and N. Slater. 2010. Fast distributed transactions and strongly consistent replication for oltp database systems. In *CouchDB: The Definitive Guide* 1st Ed., O'Reilly Media, 1337:35.

J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data System Research (CIDR'11).* 223–234.

P. A. Bernstein, C. W. Reid, and S. Das. 2011. Hyder—A transactional record manager for shared flash. In *Proceedings of the Conference on Innovative Data System Research (CIDR'11).* 9–20.

D. Campbell, G. Kakivaya, and N. Ellis. 2010. Extreme scale with full sql language support in microsoft sql azure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 1021–1024.

T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11).* 265–276.

B. Carlile. 2010. Tpc benchmark c full disclosure report: Oracle sparc supercluster with t3-4 servers using oracle database 11g release 2 with oracle real application clusters and partitioning. http://c970058.r58.cf2.rackcdn.com/fdr/tpcc/Oracle_SPARC_SuperCluster_with_T3-4s_TPC-C_FDR_120210.pdf.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06).* 205–218.

B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. 2008. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2, 1277–1288.

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. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12).* 251–264.

G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Oper. Syst. Rev.* 41, 6, 205–220.

S. Gilbert and N. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2, 51–59.

P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. 2010. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference.*

E. P. C. Jones, D. J. Abadi, and S. R. Madden. 2010. Concurrency control for partitioned databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 603–614.

A. Lakshman and P. Malik. 2009. Cassandra: Structured storage system on a p2p network. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (PODC'09).* 47.

L. Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2, 133–169.

L. Lamport. 2001. Paxos made simple. *ACM SIGACT News* 34, 4, 18–25.

D. Lomet and M. F. Mokbel. 2009. Locking key ranges with unbundled transaction services. *Proc. VLDB Endow.* 2, 1, 265–276.

D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. 2009. Unbundling transaction services in the cloud. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research* (*CIDR'09).*

C. Mohan, B. G. Lindsay, and R. Obermarck. 1986. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.* 11, 4, 378–396.

E. Pacitti, M. T. Ozsu, and C. Coulon. 2003. Preventive multi-master replication in a cluster of autonomous databases. In *Proceedings of the 9th Euro-Par Conference on Parellel Processing.* 318–327.

E. Plugge, T. Hawkins, and P. Membrey. 2010. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA.

J. Rao, E. J. Shekita, and S. Tata. 2011. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.* 4, 4, 243–254.

M. Seltzer. 2011. Oracle nosql database. http://www.oracle.com/webapps/dialogue/ns/dlgwelcome.jsp?p_ext= Y&p_dlg_id= 14620894&src= 7912319&Act= 63&sckw= WWMK13067492MPP001.

M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. 2007. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07).* 1150–1160.

A. Thomson and D. J. Abadi. 2010. The case for determinism in database systems. *Proc. VLDB. Endow.* 3, 1–2, 70–80.

A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 1–12.

A. Whitney, D. Shasha, and S. Apter. 1997. High volume transaction processing without concurrency control, two phase commit, sql or c++. In *Proceedings of the International Workshop on High Performance Transaction Systems (HPTS'97)*.

Received October 2012; revised October 2013; accepted December 2013