Lotus: Scalable Multi-Partition Transactions on Single-Threaded Partitioned Databases

Xinjing Zhou
MIT CSAIL
xinjing@mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Goetz Graefe
Google
goetzg@google.com

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

This paper revisits the H-Store/VoltDB concurrency control scheme for partitioned main-memory databases, which we term run-to-completion-single-thread (RCST), with an eye toward improving its poor performance on multi-partition (MP) workloads. The original scheme focused on maximizing single partition (SP) performance, producing results in millions of transactions per second on modest clusters, but at the expense of dismal MP performance. In this paper, we show that original RCST algorithms be made to dramatically improve MP performance with very limited impact on SP performance. That makes RCST superior to popular optimistic and pessimistic schemes without optimizations for batch execution, including OCC and 2PL, on a wide range of multi-node workloads with up to 60% throughput improvement.

Our second contribution is to propose a multiplexed-execution-single-thread (MEST) algorithm based on RCST to amortize the network stalls from MP transactions over a *batch* of MP transactions. This scheme delivers up to 21× higher throughput for SP transactions and comparable MP throughput compared to state-of-the-art distributed deterministic concurrency control algorithms that are optimized for batch execution. Finally, our MEST scheme offers dramatically superior performance when *straggler* transactions are present in the workload. Our conclusion is that the H-Store/VoltDB concurrency control scheme can be dramatically improved and dominates state-of-the-art algorithms over a variety of MP workloads.

PVLDB Reference Format:
Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Lotus: Scalable Multi-Partition Transactions on Single-Threaded Partitioned Databases. PVLDB, 15(11): 2939 - 2952, 2022.
doi:10.14778/3551793.3551843

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

1 INTRODUCTION

In 2008, some of us wrote a paper [27] about OLTP performance. In a conventional disk-based system of the time [31] with all data resident in a main memory buffer pool, we found the following approximate distribution of CPU activity: 1) Useful Work: 10%. 2) Concurrency Control: 18%. 3) Buffer Management: 30%. 4) Logging and Recovery: 18%. 5) Latching: 18%. 6) Unknown: 6%. Hence, 90% of the CPU cycles went to services, limiting the possible performance of conventional DBMSs on OLTP applications. That paper motivated the design of a main memory DBMS, H-Store [33], with the following characteristics: ❶ Partition the database over multiple nodes for scalability. ❷ Main memory only. ❸ Run transactions to completion with a single thread to eliminate contention from multi-threading. ❹ Command-based replication to remove log processing overhead. ❺ Active-active replication — SQL commands sent to replicas; deterministic execution order for consistency.

H-Store and its commercial successor VoltDB [61] were unbeatable for workloads where every transaction was local to a single partition and to a single interaction with the database server. In this case, the CPU assigned to a partition could be kept completely loaded with minimal overhead. However, performance on multi-partition transactions was terrible as they were done one by one with no concurrency. As a result, the available market for H-Store-like systems was limited to SP transaction workloads.

Although SP transactions are prevalent in many workloads [16, 51], some applications require MP transactions [8]. For example, DBOS [12, 57] which runs on top of VoltDB is overwhelmingly SP transactions. However, as noted in [40], better support for MP transactions would make DBOS life a lot easier. Also, there are many applications, for example, electronic funds transfer, where there are exactly two partitions involved in most transactions. In this case, it should be straightforward to do much better.

The purpose of this paper is to revisit the H-Store/VoltDB concurrency control scheme, which we call Run-to-Completion-Single-Thread (RCST), with the goal of dramatically improving MP performance, at the expense of slightly degraded SP performance. We make three main contributions in this paper.

First, we identify partition-level locking as the main factor limiting concurrency for MP transactions. We introduce lightweight *granules*, which are lock units smaller than a partition but much larger than a record. This enhances concurrency and makes RCST superior to non-deterministic concurrency control schemes such as OCC and 2PL with 2PC and synchronous primary-backup replication over most of the MP landscape.

Second, even with granule locking, significant network stalls still exist in MP workloads. We recognize that most of the stalls come from distributed commit processing and synchronous replication. Therefore we propose a new scheme called Multiplexed-Execution-Single-Thread (MEST) for a batch of transactions. In this scheme, multiple MP transactions are executed in an interleaved fashion, exploiting logical concurrency. They are then committed in a single batch to amortize the cost of commit processing.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551843

In addition, we detach the replication of transactions from commit processing. Commands are first partially ordered and then committed by the *sequencer* (on *primary* nodes). Replicas are updated asynchronously with a deterministic execution order. As a result, MEST outperforms batch-oriented state-of-the-art distributed deterministic concurrency control protocols [41, 63] on a wide variety of SP and MP workloads.

Finally, batch deterministic systems have poor performance on imbalanced workloads containing long-running transactions (i.e., stragglers) due to required cluster-wide synchronization [41, 42]. Our approach does not need cluster-wide synchronization for batch execution and is more resilient to workloads with stragglers.

We summarize the contributions of our system, Lotus, as:

- Through extensive experiments, we show that Lotus outperforms conventional non-deterministic schemes (2PL, OCC) with 2PC on YCSB [13] and TPC-C [62] over all SP/MP workloads.
- Compared with state-of-the-art deterministic schemes [41, 63], Lotus consistently achieves higher (up to 21× for SP) or comparable throughput on the above workloads.
- Other batch-oriented deterministic schemes degrade badly when long-running transactions are present. Lotus is up to 3.3× faster on workloads with stragglers.

As a result, the concurrency control community may want to rethink the dismissal of RCST-based algorithms, as they appear to be dominant on a wide range of transaction workloads.

2 BACKGROUND

This section gives an overview of approaches to building serializable transaction processing systems with replication for high availability. These approaches generally fall into two categories: non-deterministic schemes and deterministic algorithms. We also recap the H-Store/VoltDB approach for readers unfamiliar with these systems.

2.1 Non-Deterministic Concurrency Control

In this paper we only consider serializability guarantees as this is the gold standard and is implemented by H-Store/VoltDB. Hence, the DBMS ensures that the final state of the database will be equivalent to one produced by running transactions in some serial order. Popular approaches to achieving serializability include Two-Phase Locking [9, 19] and OCC [36, 64].

For high-availability some form of replication is required. Typical replication schemes include primary-backup replication [10, 11] and state-machine replication [37, 47, 48]. In primary-backup systems, a primary database processes all transactions first and then replicates modifications to backup replicas. Primary-backup replication schemes can be further classified into two groups: synchronous and asynchronous. In synchronous primary-backup replication, a transaction is committed only after the backups have acknowledged the application of changes. This provides strong consistency among replicas but incurs an extra network round-trip delay during transaction commit. In asynchronous primary-backup replication, the primary finishes the commit without waiting for an acknowledgment from the backups. This has minimal impact on transaction latency; however, replicas can be inconsistent when failures occur. In this case, a cleanup phase is required during failover processing. As a result synchronous replication is often favored. In state-machine replication, transaction writes can go to all replicas, depending on consensus protocols [37, 47, 48] to ensure that data modifications are processed in an agreed-upon order.

Table 1: Comparison against Existing Deterministic Approaches

	Distributed	Read-Write-Set	Program Analysis	Batching
LADS [67]	✗	✓	✓	✓
BOHM [20]	✗	✓	✗	✓
PWV [21]	✗	✓	✓	✓
QueCC [53]	✗	✓	✓	✓
QueStore [52]	✓	✓	✓	✓
Calvin [63]	✓	✓	✗	✓
SLOG [54]	✓	✓	✗	✓
Aria [41]	✓	✗	✗	✓
Lotus	✓	✗	✗	✓

If the database is partitioned, a distributed transaction needs some form of atomic commit protocols such as two-phase commit [45, 46] or its variants [7, 24, 56, 58, 59] to handle failures (e.g., node failures, concurrency control conflicts) to ensure ACID properties [25]. These protocols typically require multiple network round-trips and durable writes. Therefore, 2PC is typically considered the major bottleneck in distributed transactions.

2.2 Deterministic Concurrency Control

Given the shortcomings of non-deterministic systems, a class of deterministic databases [20, 21, 41, 52–54, 63, 67] provides another way of guaranteeing serializable transactions. Instead of guaranteeing equivalence to some serial order only after execution, these systems predetermine a single serial order (input order) prior to execution, typically through a sequencing layer that also logs the transaction inputs. For throughput, the sequence layer can be partitioned over multiple nodes [63]. For high availability, nodes in the sequence layer agree on an order through consensus protocols [37, 48] . These systems run transactions in large batches to amortize the sequencing overhead. Once the serial order of a batch of transactions is determined by the sequencer, the batch is sent to multiple replicas to be executed concurrently and independently while reaching the same final state as their execution in serial order. This is much like shipping logs to replicas and each invoking recovery logic for each transaction. This reduces replication overhead as replicas do not communicate with each other to reach the same final state. Perhaps more importantly, the overhead of a distributed transaction is also reduced by simplifying the commit protocol. In contrast to non-deterministic concurrency control, non-deterministic failures, such as node failure, will not affect the commit decision [4]. Instead, during recovery a transaction will be executed again to reproduce the correct state. Therefore, costly distributed commit protocol such as 2PC can be avoided.

We summarized the properties of deterministic approaches in Table 1. One critique about existing systems [4] is that they typically rely on the strong assumption that the read-write-set of a transaction is known beforehand to allow concurrent execution (except

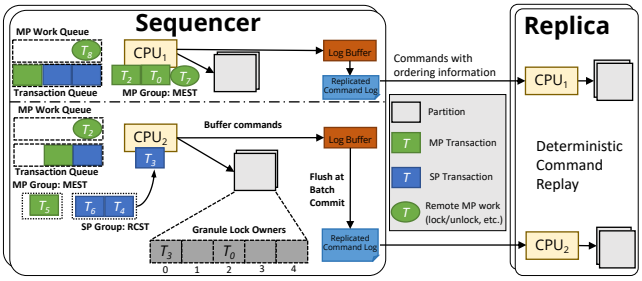


Figure 1: Architecture of Lotus

Aria [41]). Some [21, 52, 53, 67] take a step further to require manual program analysis for more concurrency. Lotus does not rely on these assumptions for achieving high performance.

2.3 H-Store/VoltDB Background

Although H-Store/VoltDB also exploits determinism, they leverage it differently from the systems mentioned in the previous section.

H-Store [2, 50, 60] partitions the database among the nodes in a cluster and binds each partition exclusively to a single worker thread. Transactions are ordered globally through physical timestamps. Each partition has a replica set that contains one or more copies (K-safety). One replica of a partition is designated as the primary that synchronously replicates all the transaction requests to the other replicas. Since transactions are ordered by timestamp, all replicas executing the same sequence of transactions end up in the same state. For multi-partition transactions, partition locking is used during execution for exclusive access to affected partitions. 2PC is required during the commit process for such transactions. Logging is avoided in 2PC. Instead of writing 2PC votes to disk, persistence is implemented through synchronous replication of prepare/commit requests to the backups of a partition.

Its commercial successor, VoltDB [3, 61], inherits the architecture of H-Store but concluded that timestamp-based serialization was not practical due to its reliance on NTP services. Poorly configured NTP services allowed unacceptably large clock skew. As a result, VoltDB opted for a centralized controller (MP Initiator) to serialize MP transactions and execute them one at a time. SP transactions bypass MP Initiator as their serialization is achieved using the FIFO queue of the server hosting their partition data. Because of minimal sequencing overhead, SP transactions are extremely fast. However, a single MP Initiator with no concurrency among MP transactions creates scaling challenges for MP transactions.

In summary, H-Store/VoltDB achieves unparalleled performance on SP transactions at the cost of high overhead for MP transactions because of limited concurrency due to partition locking, 2PC with synchronous replication, and global serialization. In contrast, conventional deterministic systems reduce the overhead of MP transactions at the cost of strong assumptions and mediocre SP transaction performance due to scheduling and sequencing overhead. We now turn to how Lotus achieves low replication/commit overhead without the limitations and assumptions discussed above.

3 Lotus OVERVIEW

Lotus addresses the inefficiencies of VoltDB/H-Store on MP transactions in the following ways: ❶ Support more concurrency when waiting for the network by refining partition-level locking into granule-level locking. ❷ Amortize network stalls coming from execution and commit processing through batch execution and commit. ❸ Decouple replication from the commit protocol by asynchronously and deterministically executing partially-ordered transactions on backups.

Lotus partitions data over a set of physical nodes. We define such a collection of physical nodes as *replica*. Different replicas do not share physical nodes to isolate machine failures. We refer to the replica accepting user transactions as the primary/sequencer replica (primary for short). We refer to replicas replaying replicated transactions from primary as backup replicas (backup for short). Lotus inherits its transaction model from H-Store/VoltDB. In contrast to VoltDB/H-Store, there is no global controller for serializing MP transactions or reliance on physical timestamps for ordering transactions. As can be seen in Figure 1, Lotus contains a sequencer for every partition. User programs submit SP transactions to the correct sequencer for the desired partition. MP transactions can be submitted to any sequencer. Every sequencer assembles a batch of queued transactions. Then the batch is decomposed into an SP collection and an MP collection. The SP transactions are done using RCST, as in VoltDB/H-store. Subsequently, the sequencer acts as the coordinator for each MP transaction in the collection. As such, it sends sub-transactions to remote workers, waiting for a reply while interleaving with execution of MP transactions in the same batch. Every partition is sub-divided into *granules*, which are lock units. Mapping between a partition and its granules can be done via hash- or range-partitioning. Lotus uses strict two-phase locking with a *NO_WAIT* policy for these granules. Hence, on a lock request failure, the MP transaction is aborted and rescheduled for the next batch. A successful transaction queues a log record containing the various commands in the log buffer.

The successful execution of MP transactions on the various sequencers yields a partial ordering among the transactions that have lock conflicts. This ordering is also recorded in the command log. When a batch finishes, the composite log records are persisted to the replicated command log at which point the whole batch is considered committed. Replicas then asynchronously read from the replicated command logs produced by the sequencers and deterministically execute transactions. Results of transactions are returned to clients once they are committed on the sequencers. Lotus assumes a fail-stop failure model and runs in a trusted environment. It does not tolerate Byzantine faults [38]. Lotus assumes a networking technology that provides ordered reliable delivery of messages, such as TCP, on an asynchronous network.

4 Lotus DESIGN

We now describe the detailed design of Lotus. Section 4.1 presents the design of granule-based locking. Section 4.2 introduces the design of batch execution and the commit protocol. Section 4.3 covers the Lotus asynchronous replication scheme. Finally, Section 4.4 discusses the limitations of Lotus.

4.1 Granule-based Locking

In Lotus, transactions lock granules instead of partitions. Our scheme strikes a balance between concurrency and lock management overhead, i.e., between partition-level locking and tuple-level

```
1  Function: BatchExecution (wid, LB, BatchNo) # LB: log buffer
2      TG = Queued transactions ∪ aborted transactions
3      SPTG, MPTG = break TG into SP and MP transactions
4      for T in SPTG:                    RCST Batch Execution
5          ExecuteSP(T,LB)               not processing remote requests
6      Persist LB to log
7      Reply to clients
8      for T in MPTG:                    MEST Batch Execution
9          multiplex ExecuteMP(T) with ProcessMPWork(LB)
10     Persist [LB, BatchNo] to log
11     for T in MPTG where T.abort == false:
12         InstallCommands(T, LB)
13     Install writes and release locks of transactions in MPTG
14     Send BCOMMIT[wid, BatchNo] to all other workers
15     Reply to clients
```

Figure 2: Algorithm for Batch Execution Control Loop

locking. A partition is decomposed into a fixed set of logical sub-partitions (granules) through hash- or range-partitioning the keys of tuples in the partition. Granules are virtual. LOTUS does not physically decompose a partition into sub-partitions. Instead, granules are derived from partitioning on the keys of tuples. It supports reasonable concurrency for MP transactions at low cost as shown in the experiments. Also, every partition is assigned to one processor (core). Therefore, there is no latching overhead or multi-core contention. Another important distinction is that the granules also serve as the unit of replication. LOTUS supports both shared and exclusive lock modes.

One might be tempted to shard the database into more partitions to get more concurrency. While it seems intuitive and simple, this has two major problems. On the one hand, the database might need to be physically fragmented into very small pieces in order to get high level of concurrency. This imposes non-trivial partition management overhead and scalability challenges. On the other hand, it is not flexible. For example, when the workload changes and the existing partitioning is no longer suitable, such an approach requires expensive re-partitioning of database to adjust to the new workloads. The root cause of these problems is that the sharding approach tightly couples physical data distribution with concurrency control. Our approach decouples the two. Since granules are logical partitions used only for concurrency control, it is actually easier to change the number of granules at run-time. One could simply use a system-level distributed transaction that exclusively locks all partitions and changes the number of granules per partition to adapt to a new workload. This is lightweight as it involves no data movement. Therefore, our approach is more scalable and serves as a good basis for adapting to changing workloads.

4.2 Batch Execution and Commit in LOTUS

MP transactions submitted to a worker thread in H-Store/VoltDB are processed one by one. This includes executing the transaction processing logic, waiting for results from a remote partition, and performing commit. This results in CPU stalls while waiting for the network. Moreover, MP transaction commit overhead is still high because the commit protocol requires multiple network round trips between partitions and between replicas. We now show how to reduce the overhead of the commit protocol for MP transactions

```
1  Function: InstallCommands(T, LB)
2    for each local granule (p, g) in T: # p stands for partition
3      LB.AddParticipantRecord({T.tid, p.id, g.id, g.last-writer,   g.lock-
4  type})
5    p,g = choose any local (partition, granule) T accessed
6    LB.AddCoordinatorRecord(T.tid, p.id, g.id, T)
7
8  Function: ExecuteSP (T, LB)
9    Execute T's logic to completion:
10     for each granule (p, g) T accesses during execution:
11       if (p, g) is X-locked or
12           ((p, g) is S-locked and T intends to modify (p, g)) :
13           T.abort = true # concurrency control abort
14   if T.abort == false:
15         update any granule g's last-writer field if written by T
16         InstallCommands(T, LB)
17         Install writes
18
19 Function: ExecuteMP(T)
20   Execute T's logic:
21     for each granule (p, g) T accesses during execution:
22       if (p, g)  is remote:
23           send lock/read requests to remote workers
24       else:
25           process lock/read requests locally
26   if T.abort == true: unlock granules locally and remotely
27
28 Function: ProcessMPWork(LB)
29   for r in MPWork:
30     if r is COMMIT_UNLOCK on granule g:
31         Update g.last-writer if g is X-locked
32         LB.AddParticipantRecord(r.tid, r.pid, r.gid, g.last-writer, g.lock-
33 type)
34     process lock/read/write/unlock/commit_unlock … requests
```

Figure 3: Algorithms for Transaction Execution and Commit

through batch execution and commit. The basic ideas are to overlap MP transaction execution with network stalls from commit protocol/execution as well as to buffering transactions' log writes until the end of a batch commit. Network stalls and commit protocol overhead are therefore amortized over a batch of MP transactions. In addition, LOTUS makes replication of transactions asynchronous by having sequencer nodes persist the transactions to replicated command logs and by having backups re-execute the transactions from the command logs asynchronously and independently.

Each sequencer runs the algorithm listed in Figure 2 for processing a batch of transactions. At the start of batch execution, the sequencer reorders transactions into an SP group and an MP group. Each group goes through two phases: execution and batch commit. As noted in Section 3, SP transactions are run RCST, as in VoltDB/H-Store. For the MP group, sequencer acts as the coordinator for the entire group of transactions. For either type of transaction, any user-initiated aborts and integrity violations are detected during the execution phase and the transaction will be aborted. To handle such aborts, LOTUS keeps all the writes in a local write set and these writes are only applied to the database when the transaction is ready to commit.

4.2.1 ***RCST for SP Group*** Since there is no network involved for SP transactions, they are processed to completion in RCST fashion, maximizing CPU utilization. If execution is successful, writes are immediately applied to the local database after a transaction finishes execution (Line 16 of Figure 3). SP transaction will also record

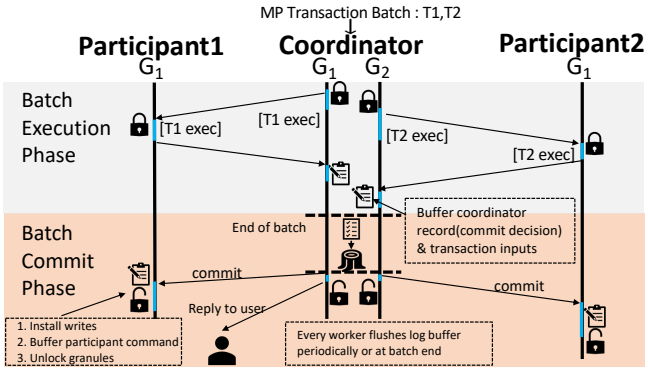


Figure 4: Workflow of Lotus Batch Execution and Commit: T1, T2 are transactions targeting granules in the coordinator worker and two granules from two other workers (participants). Logging of T1 and T2 are buffered in memory. The transaction results are not released to user until the buffer has been flushed to the replicated log.

granules accessed in the log buffer (Line 3-6 of Figure 3). Transaction inputs are also recorded as a coordinator record in the log buffer. These records are essential for correct replication. Among SP transactions in a same batch, there is no possible lock conflicts since they are executed one by one. Therefore, Lotus does not lock granules for SP transactions. However, a SP transaction must check whether a granule is locked by a remote MP transaction before the RCST phase starts and the intent of the SP transaction is incompatible with the lock (Line 11-12 of Figure 3). For example, the SP transaction intends to modify a granule that is exclusively locked. In such cases, the SP transaction is aborted. Also, a sequencer needs to disable processing requests from remote MP transactions during the RCST period. This is to ensure not to expose modifications from uncommitted SP transactions to remote MP transactions, avoiding cascading aborts. Note that Lotus applies the writes before persisting the commands to the log. The persistence could fail due to network failures. In such a case, we need to roll back the database to the state before the RCST phase. To achieve this, similar to H-Store/VoltDB [50, 61], Lotus maintains a transient in-memory undo buffer that stores undo information for each SP transaction. The buffer is discarded when the batch is committed.

Differences from H-Store/VoltDB: Our SP execution differs from that of H-Store/VoltDB in that replication is not involved in the execution phase, reducing potential network stalls. Data replication occurs after the log buffer is persisted in the replicated command log, at which point backups get the commands and asynchronously execute transactions. Details will be discussed in Section 4.3.

4.2.2 ***MEST for MP Group*** We execute MP transactions asynchronously (Line 9-10). Whenever an MP transaction has to wait for results from remote granules, the coordinator suspends the transaction and switches to a different MP transaction in the batch. In the meantime, the coordinator also processes requests from remote MP transactions. If an MP transaction finishes execution without an abort, the coordinator installs participant and coordinator records locally. In contrast to SP transactions, locks are not released after an MP transaction finishes execution. Instead, they are released after the log buffer is flushed at which point the commit decision is propagated to participants. This ensures the state of uncommitted MP transactions is not exposed to remote MP transactions. An illustration of such a process is shown in Figure 4. Note that no undo buffer is maintained as Lotus only applies writes after persisting the commands for MP transaction.

Distinction from H-Store/VoltDB: Our MP execution differs from H-Store/VoltDB in that multiple MP transactions can multiplex the CPU. Another difference is that replication is not involved in the execution phase, similar to SP Group execution.

4.2.3 ***Supporting Shared Locks*** It is trivial to support shared locks on granules within the primary replica. The challenge lies in allowing shared locks during replay on backups that produce the same state as the primary. Between any two exclusive lock events on a granule, Lotus allows shared locks among multiple transactions running concurrently on the primary. On backups, Lotus only needs to ensure these shared lock events are ordered after the first exclusive lock event and before the second one. Among these shared lock events, there is no need to maintain any order as reads do not change the state of the database. To capture this ordering information, Lotus maintains for each granule a last writer field that indicates the globally-unique transaction id of the last committed transaction that exclusively locked the granule. We highlighted the pseudo-code related to shared locks in grey in Figure 3. When a transaction commits, the last writer field is updated during lock release for an exclusive lock on a granule (Line 15 and 31 of Figure 3). The lock type as well as the last writer on a granule that a lock operation observed at the time of locking are included in the command log records (Line 3 and 32 of Figure 3). Therefore, shared lock operations in between two exclusive lock events on a granule will observe the same last writer which will enable shared locks during replay on backups as we will see in Section 4.3.

4.2.4 ***Batch Commit Phase*** Lotus makes sure that all possible aborts are processed during the execution phase. For concurrency control aborts, transactions will be retried during the next batch. For integrity constraint violations or user-initiated aborts found during execution, transactions are discarded, releasing any locks held. Therefore, the remaining transactions are ready to commit. Committing a batch of transactions occurs by flushing the log buffer to the replicated command log. Since the log buffer flush contains multiple coordinator records, the commit overhead is amortized. Transaction results are released to clients once the log buffer is persisted in the command log and a quorum of replicas responded.

Once the log buffer is successfully flushed, Lotus sends *COMMIT_UNLOCK* requests to all the workers to unlock granules (Line 13 of Figure 2). This step must happen after persisting the log buffer. Before unlocking participant granules, participant records are recorded in their log buffer (Line 30-33 of Figure 3). A participant record contains information about which granules are locked by which transactions. This is critical as the participant record captures the lock ordering of the participants and will be used in our recovery and replication schemes. Such participant records do not need to be forced to log, thereby reducing logging overhead.

Protocol Properties: Our commit protocol does not require forced (expensive) log writes of votes and redo records on participants as typically seen in the standard 2PC protocol. This resembles one-phase commit (1PC) protocol. The durability and atomicity properties of a transaction are guaranteed by the persistence of

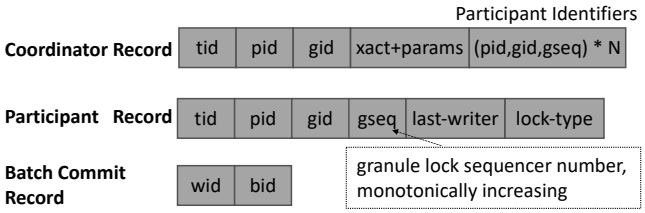


Figure 5: Command Logging Record Format

coordinator records. Such records contain the transaction inputs as well as granules locked, as shown in Figure 5. The isolation property is guaranteed collectively by the S2PL (Strict Two-Phase Locking) lock orderings recorded in the participant's records for each participant. Our protocol also adheres to the principle of the presumed-abort optimization [45, 46]. That is, if no coordinator record is found in the log, the transaction is considered aborted. The commit decision is implicitly represented as the presence of inputs of a transaction contained in the coordinator record. The abort decision is implicitly represented by the absence of the coordinator record. It should be noted that 1PC protocol using logical logging has been proposed previously [5]. However, our protocol focuses on the extreme form of logical logging (i.e., command logging). Also notice our protocol does not wait for the acknowledgments from the participants, which is required in [5], before replying to users.

Recovery: We now discuss how Lotus recovers from failures. Since we buffer commands for a batch of transactions, node failures could result in buffer contents being lost. Our recovery protocol needs to repair the lost contents. As shown on the left side of Figure 6, granules $G_i/G_j/G_k$ on three different nodes have persisted records up to (exclusive) sequence numbers 1/2/1, respectively (Sequence numbers are for illustration only). Upon a crash, the contents after the persisted portion of the sequence numbers are lost. Since we only buffer participant records on remote workers after the coordinator has successfully forced its log buffer, there is only one failure case Log Repair needs to handle: missing participant records. As shown in the figure, transactions T2 and T5 are committed since their coordinator records are persisted for granule G_i and G_k. However, their participant records are actually lost since they were in the log buffer of the worker hosting granule G_j before the crash. On recovery, the worker hosting granule G_j is instructed by coordinators of transactions T2 and T5 to check the existence of corresponding participant records using the information (*pid* for partition id, *gid* for granule id in Figure 5) in coordinator records. Because the coordinator record is the ground truth of the fate of the transaction, the command log of the worker hosting granule G_j will be patched with participant records generated from the coordinator records of T2 and T5 (on the right side of Figure 6).

It should also be noted that repair must reproduce the same lock ordering information. As shown in Figure 6, T2 is ordered before T5 by the lock conflict on granule G_j before the crash. Fortunately, the coordinator record contains the granule lock sequence number (gseq in Figure 5) that reflects this ordering. The lock sequence number is a per-granule monotonically increasing number that reflects the order of lock operations on a same granule. The sequence number is increased every time a granule is locked. During log repair, coordinators cooperatively patch the participant records

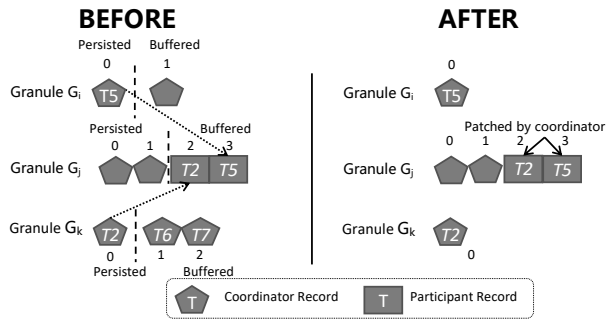


Figure 6: Granule-level Log Repair: $G_i/G_j/G_k$ are granules on three partitions each on a different node; T2 should be patched before T5.

by comparing granule lock sequence numbers recorded in their coordinator records.

Accelerating Log Repair: The naive log repair algorithm mentioned previously requires checking every coordinator record in the log for patching any missing participant records. We now show how to reduce the overhead by persisting a batch commit record. After a coordinator W_x flushed its log buffer, a batch commit record containing worker identifier(*wid* in Figure 5) and batch number(*bid* in Figure 5) is sent to all the other workers in the cluster to be recorded in their own log buffer, denoting the fact that MP transactions of this batch initiated by coordinator W_x are committed (Line 14 of Figure 2). These commit records will be flushed by batch execution of other workers in a piggybacked way. Therefore, if a worker W_x finds a batch commit record with batch number i in another worker W_y's command log, any MP transactions prior to batch $(i+1)$ initiated by worker W_x must have its participants records persisted on W_y's command log. (Participant records from an MP transaction in batch i from worker W_x will appear before batch commit record for batch i). Therefore, we can skip checking for missing participant records for these MP transactions. After all the participant records are properly patched, we can patch any missing batch commit records, hence concluding the log repair. As shown in Figure 7, a worker goes through the command log backwards to find the last batch commit records for all workers (Line 20). Then the algorithm reads the command log again backwards to patch any missing participant records (Line 23-28). It stops at the batch B_i such that all other workers in the system have a batch commit record for B_i durable in their log. Therefore, we can bound the repair overhead to just processing a few batches of transactions at the tail of command logs.

Once all the logs are properly repaired, workers can start cooperatively recovering the states of the transactions by re-executing the transactions recorded in the command logs. We defer the discussion of scheduling of re-execution of these transactions to Section 4.3. Notice, that our commit protocol does not maintain site autonomy property [5]. That is, the recovery of an MP transaction cannot be done independently. It needs the coordinator to initiate the re-execution of the transaction using the inputs stored in the coordinator record to recover the effects of the committed transaction.

Distinction from H-Store/VoltDB: Our commit protocol for MP transactions differs from H-Store/VoltDB [44]. First, Lotus logs and commits a batch of MP transactions from the same sequencer.

```
Function: ReplayCommandLog()
  Event Loop:
  on receiving commands from log:
    de-multiplex commands into partition-granule-level queues
  on trigger events for granule g of partition p:
    r = p.queue[g].head
    if r.type == PARTICIPANT and r.lock_type == S-MODE:
        S-lock g if not X-locked and g.last_writer == r.last_writer,
increase g.reader_count by one upon success
    else if r.type == PARTICIPANT and r.lock_type == X-MODE:
        X-lock g if g not X-locked and g.reader_count == 0, set
g.last_writer = r.tid upon success
    else: # coordinator record
      if r.xact is SP and all granules locked: execute r. xact
      else if r. xact is MP: async execute r. xact
    pop r from queue if locking succeeded or r. xact committed
  on remote requests: processMPWork()

Function: RepairLog(CommandLog)
  LastBCommits = go through CommandLog backwards to find the last
BCOMMIT record of each worker
  WORKERS = ALL_WORKERS_IN_CLUSTER excluding self
  for batch in reversed(CommandLog):
    check and patch any missing participant records on
      WORKERS for coordinator records in batch
    remove any worker from WORKERS that has a BCOMMIT
      record with batch number == batch.no
    if WORKERS is empty: break
  sync with all workers
```

Figure 7: Algorithms for Recovery and Deterministic Replay

Second, H-Store only records transaction inputs on the coordinator node. On recovery, command logs from different nodes need to be merged and replayed serially based on timestamps. Significant coordination is required for scheduling MP transactions during recovery. In Lotus, in addition to logging the transaction inputs on each coordinator worker, Lotus also logs (buffers) the lock orderings on participants after commit which allows transactions to be replayed in parallel during recovery.

4.3 Asynchronous Active Replication

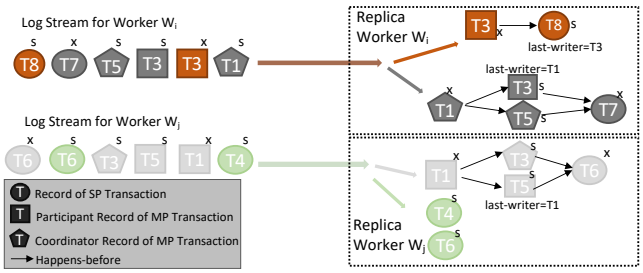


Figure 8: Example of Deterministic Replay: Colored shapes denote records for different granules; Dashed ovals shows the granules a transaction accesses; Records are also marked with the lock type(s for shared mode; x for exclusive mode); Records with the same last-writer field indicates their transactions were ran concurrently in the primary replica; Partial orderings produced by the sequencers can be rebuilt from these commands: T1 →T3 →T6, T1 →T5 →T6, T1 →T3 →T7, T1 →T5 →T7.

Lotus is designed to replicate command logs through consensus algorithms [37, 48] or consensus-based log store [6, 65] for high-availability and strong consistency. However, Lotus architecture does not preclude other replication schemes such as synchronously shipping commands to all replicas before commit or asynchronously shipping commands to replicas. These schemes trade consistency and availability for performance. Notice all of the schemes only concern the replication of commands. They do not concern when replicas replay the commands.

As pointed out previously, different from H-Store/VoltDB, execution of the commands on backups is not in lockstep with the primary. One problem with such an asynchronous replication scheme is potential replication lag. The replication lag problem occurs when the rate of backup applying updates cannot keep up with the rate of updates by the primary database. A backup that is lagging too far behind cannot be promoted as the new primary in a timely fashion, reducing availability and causing real-world incidents [1, 35]. Common causes for replication lag include insufficient network bandwidth and non-scalable replay schemes on the backup. Fortunately, the Lotus replication scheme is designed around replicating commands with small network footprint and granule-level parallelism, greatly mitigating this problem. We now describe how Lotus replicates transactions and ensures that all replicas will reach the same state as the sequencers in a scalable and efficient manner.

In Lotus, backups mirror the primary in terms of data partitioning and worker setups. Backup worker W_i subscribes to the command log that is being actively written by W_i in the primary. As illustrated in Figure 8, two workers in the backup running on different nodes are de-multiplexing the log stream into granule-level commands in log order. From these commands, Lotus can precisely rebuild the partial ordering of transactions captured by the locking protocol on the primary. A backup worker executes commands concurrently at the granule level. The algorithm for the replay is shown in Figure 7. The replay process is similar to a long-running topological sort. Specifically, our backup worker runs in a event loop in function *ReplayCommandLog*. Events, e.g., the arrival of a new command for a granule or the granule being unlocked, trigger the execution of the commands against a granule. For participant records, the worker attempts to lock the granule. For exclusive locking, it succeeds if the granule is not currently being locked in exclusive mode and the reader count is zero. Upon success, the last writer field is updated. For shared locks, it succeeds when the granule is not currently locked in exclusive mode and the last writer field of the granule matches what is stored in the participant record. Reader count is incremented upon success. Therefore there could be multiple readers on the same granule, allowing more concurrency. If a locking operation succeeds, the granule waits for instructions (read/write/unlock, etc.) from the coordinator. If locking fails, it means the granule is still locked by previous transaction and lock acquisition will be retried on the next lock release event. For coordinator records, the transaction will be initiated. For SP transaction, when it obtains all the necessary granule locks, the transaction is executed synchronously since there is no need to access remote partition. For MP transactions, we execute the transaction asynchronously (Line 11). When an MP transaction requests a shared lock on a remote granule, the last writer information for the granule observed at the primary is sent to remote nodes. The worker continues to process commands instead of waiting for remote results of an MP transaction. It is possible that MP transactions will fail due to granule lock conflicts. This is

because the remote participant granule has not caught up yet on the participant record that needs to be executed. Therefore, in such a case, we simply restart the execution until it eventually succeeds.

Notice, that we do not need to run an atomic commit protocol for MP transactions. This is because LOTUS only replay committed transactions. It is also easy to see that each backup runs transactions deterministically. The execution of commands strictly honors the partial orderings produced by the locking protocol on granules. In addition, concurrent transactions have no lock conflicts(T4 and T1, T8 and T7 in Figure 8) can be run in any order without breaking determinism. Therefore, they all end up in the same state. Two transactions having conflicts on a granule with the same *last-writer* (T3 and T5 in Figure 8) can also be run concurrently with shared locks. Moreover, the performance of SP transactions is maximized as they are run to completion without interruption. The parallelism is the same as that in the primary. Therefore, LOTUS replication can deliver almost the same throughput and scalability as the primary as we will see in the experiment section.

4.4 Limitations

We now list the limitations of LOTUS. First, although LOTUS greatly improves the performance of MP transactions, the number of granules is still significantly smaller than the number of tuples in a partition. Therefore for workloads with contention among concurrent MP transactions, LOTUS will perform worse than tuple-level schemes. We should also point out that if the contention is among a batch of SP transactions, conflicts will be rare because SP transactions are run one by one. Second, our replication scheme is asynchronous. Although LOTUS guarantees all replicas will execute the same sequences of committed commands, it carries an inevitable replication time lag. On failover, a new replica needs a time window to catch up before servicing new transactions. Therefore, LOTUS trades instantaneous fail-over as in H-Store/VoltDB for lower per-transaction overhead. Third, the granule-based locking scheme only enables logical concurrency for MP transactions. LOTUS does not provide physical concurrency within a partition.

5 PRACTICAL CONSIDERATIONS

We discuss some of the practical considerations during implementation, including fail-over and checkpoint.

5.1 Failover

We describe the failover procedure for LOTUS as follows. There is a fault-tolerant configuration service, like Zookeeper [30], that monitors the live-ness of nodes and membership of the replicas in the system. We assume that only the primary is able to write to replicated command logs which can be achieved through leases or locks [22, 30] provided by the configuration service. When any sequencer in the primary replica fails, our system must failover to one of the backups which will become the new primary. On fail-over, one candidate is chosen via leader election mechanisms provided by the configuration service. The new primary will have each worker run the log repair algorithm (*RepairLog* in Figure 7) to patch any missing participant records by looking at the tail of their command log. After that, workers start replaying from where they were left off when they were backups until they drain their own command log. Then the new replica switches from replay to accepting new transactions from clients.

5.2 Checkpoint

To bound the log volume and recovery time, some form of checkpointing is required. LOTUS inherits its checkpoint strategy from H-Store [50]. During the checkpointing process, a node in a cluster is elected as the coordinator to issue a special system MP transaction that locks all the partitions. This ensures that a global transactionally consistent point is established. The transaction instructs all the partitions to go into copy-on-write mode. A background thread for each worker will start to write the snapshots of the partitions it manages as of the consistent point to storage. New transactions after the checkpoint transaction will copy the tuples it wants to modify to a new memory location before applying the updates. The checkpoint transaction then flushes out the log buffers of all the workers, ensuring the system transaction is durable and committed. After all the snapshots are written to storage, the locations of these partition snapshots along with the transaction id of the system transaction are recorded in a snapshot catalog. Then the coordinator instructs all the workers to go back to normal operating mode. To recover from a checkpoint, a complete snapshot is chosen from the catalog. Partition data are first recovered from the snapshots. Then the same recovery protocol discussed can be used to replay all commands after the system transaction.

6 IMPLEMENTATION

We implemented LOTUS in the Star test-bed [43] in 4.6K lines of C++ code. LOTUS provides a relational model of data stored in tables with schema of typed attributes. Transactions are submitted to the sequencers in the form of predefined stored procedures in C++. Each table has a primary key. Indexes are implemented using hash tables. For secondary indexes, the primary key is stored in the record of a hash table. Therefore, two look-ups are required to find the tuple stored in the primary index. LOTUS requires transaction ids to be unique globally. Transaction id in LOTUS consists of a unique worker id and a monotonically increasing integer.

On every node in the system, each partition is bound to a single-threaded sequencer that handles all the accesses. Sequencers in a replica communicate purely through message-passing. Each sequencer is associated with two queues: *Transaction Queue* and *MP Work Queue*. A *Transaction Queue* buffers all the transactions received from clients targeting partitions managed by the sequencer. A *MP Work Queue* buffers the work (e.g., lock/unlock/read/write messages) of MP transactions initiated from remote coordinators. Typically, partitions to be accessed can be inferred from the stored procedure parameters (e.g., most transactions in TPC-C). For transactions accessing only a single partition (the majority of TPC-C) are submitted directly to the sequencer hosting the partition. Transactions accessing multiple partitions, clients can submit them to any sequencer. Sequencer receiving a transaction serves as its coordinator. For transactions without hints on the partitions they will access, they are sent to any sequencer and will be bound to a partition and granule on that sequencer.

For granule locking, hash- or range-partitioning can be used to map the primary key of tuples to a fixed set of lock buckets. We

experimentally find the number of granules that works well for a given workload. A more automatic approach is left for future work.

7 EXPERIMENTAL EVALUATION

In this section, we conduct experiments aimed to answer the following questions:

- How does Lotus perform compared to conventional non-deterministic databases with primary-backup replication?
- How does Lotus perform compared to the state-of-the-art deterministic systems?
- How much overhead does granule-level locking impose?
- How robust are the existing deterministic systems when there are stragglers in the workload?
- How does Lotus scale on a large number of nodes ?
- What is the performance impact of the checkpoint process?

7.1 Evaluation Setup

We ran our experiments on a cluster of six *n1-highcpu-16* nodes on Google Cloud, each equipped with 16 2.30 GHZ virtual CPUs and 14.8GB RAM. Each node runs Debian 10 with Linux Kernel 4.19.208. All code is compiled using GCC 8.3 with -O3 option. On each node, we run 6 worker threads and 2 threads for network I/O. In addition, one thread is dedicated to logging and one thread is dedicated to coordinating workers in the node. Any two nodes are connected through a network with a measured (*iperf3*) bandwidth of 1.8 GB/s.

Benchmarks. To evaluate Lotus, we perform extensive experiments on two popular benchmarks:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source performance benchmark suite used to evaluate databases and key-value systems. The benchmark includes a table with a 64-bit primary key attribute and ten data attributes. Each data attribute has ten random bytes. Each transaction performs ten read or write operations on these attributes. All operations are keyed using a uniform distribution. In our evaluation, we use a workload with a 50/50 mixture of read and write operations. Each multi-partition transaction accesses two partitions by default.

TPC-C: The TPC-C benchmark is the industry-standard benchmark suite for OLTP databases. In our experiments, we implemented NewOrder and Payment transactions. The NewOrder benchmark mimics customers submitting orders to their local district of a warehouse. The Payment transaction mimics the step of making payments on the orders customers submitted. The two transactions comprise 90% of the workload. The remaining three transactions are omitted because they require range scan which is not supported in the current implementation. We partition the database by warehouse. By default, NewOrder and Payment transactions contain 15% and 10% multi-partition transactions that access a remote warehouse.

Concurrency Control Algorithms. We compare Lotus to the following distributed protocols in our framework using the same data structures for table storage, indexing, and networking:

H-Store: Our simulation of H-Store without global timestamp ordering using the Lotus framework. We simulate partition-level exclusive locking in H-Store with *NO_WAIT* policy by configuring one granule per partition. Transactions are sent to backups once they have obtained all the locks on primary (active-active replication). Locks are held till the end of the replication. MP transactions are committed using Lotus commit protocol without batching.

DS2PL: An implementation of a tuple-level distributed S2PL algorithm. NO_WAIT locking policy is employed to prevent deadlocks which has been shown to be the most scalable algorithm [26]. Two-phase commit and synchronous primary-backup replication are employed for distributed transactions and replication.

Sundial: An implementation of a recent distributed OCC protocol [69]. The protocol uses logical leases and pessimistic locks for writes to permit more serializable schedules and reduce aborts. We did not implement the caching feature of Sundial. Tuples are write locked with the NO_WAIT policy during the execution phase. To commit, a transaction needs to validate all its reads. Similarly, two-phase commit and synchronous primary-backup replication are employed for distributed transactions and replication.

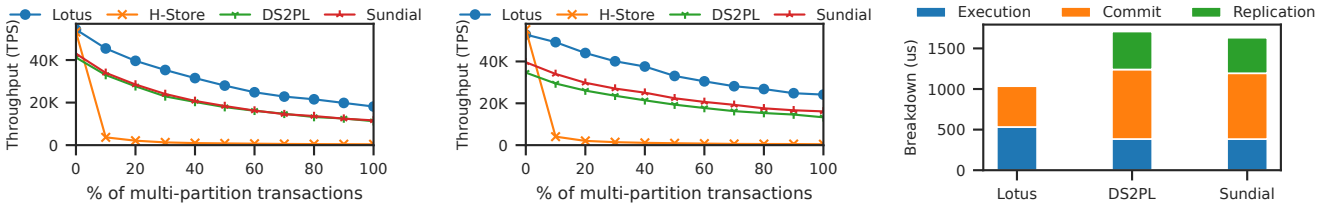
Calvin: A distributed deterministic database system. Ordered locks are employed to ensure determinism. The read/write set of a transaction must be declared before execution.

Aria: Another distributed deterministic database system. Aria proposes a determinstic reservation technique to ensure determinism and does not need pre-declared read/write set for a transaction.

Replication and Workload Generation. For Lotus, we run a cluster of three primary nodes and three backup nodes. For DS2PL and DOCC, we designate half of the nodes as primaries and the other half as backups. For Calvin and Aria, we use half the of six nodes to form a replica. Since replica does not communicate with each other in Calvin and Aria, we only study the performance of one replica. For Aria and Calvin we use one thread as the sequencer in every node of a replica. Transactions are generated by the local worker and then handed to the sequencer thread for determining order and logging. After the transactions are persisted, sequencer threads schedule workers for execution. For DS2PL and DOCC, transactions are generated at primaries directly to avoid network stalls. Similarly, transactions are generated at the sequencers of Lotus. Lotus, Calvin, and Aria deliver the strongest consistency when their logs are replicated using consensus. However, To focus on concurrency control aspect, our experiments did not implement logging with consensus algorithms. Instead, logging in all the systems is implemented by writing to the local disk using a dedicated thread. For Aria and Calvin, logging is performed by the sequencers. In Lotus, the sequencer ships commands in the log buffer to backups after it is persisted. Therefore, Lotus, Aria and Calvin pay the same logging overhead per batch of transactions.

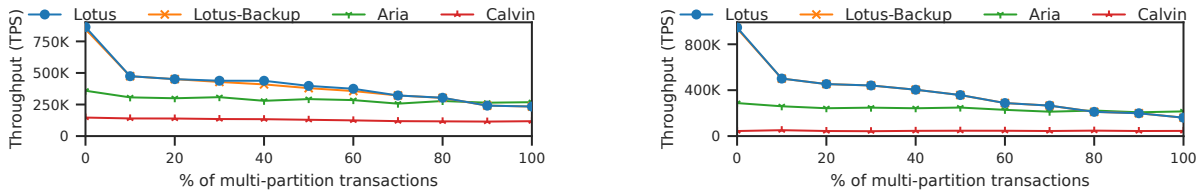
7.2 Non-Deterministic Systems

We configure the number of partitions to be equal to the total number of worker threads. We ran YCSB and TPC-C varying the percentage of multi-partition transactions. Each MP transaction accessed at most five partitions for YCSB. In addition, for all the protocols, we implemented the standard group-commit optimization that combines concurrent durable writes to the log from multiple workers on a node into one durable write. To ensure fairness for non-deterministic protocols, we turned off the batch execution for Lotus as other protocols do not do batching. Each transaction is immediately committed after execution. Hence, the difference



(a) YCSB **(b)** TPC-C **(c)** Average Latency Breakdown of YCSB MP transaction

Figure 9: Comparison with Non-Deterministic Systems



(a) YCSB - batch size: 1200 **(b)** TPC-C - batch size: 600, # warehouses: 414

Figure 10: Comparison with Deterministic Systems

between Lotus and H-Store in the experiment is the addition of granule-based locking.

The results are shown in Figure 9a and Figure 9b. We see that Lotus and H-Store perform similarly when there are no MP transactions. Both outperform DS2PL and Sundial by 20%. This is largely due to active-active replication. The primary streams SP transactions to its backup and their executions overlap. This is in contrast to DS2PL and Sundial which employ synchronous primary-backup-based log shipping. Writes are applied synchronously to backups at the end of execution. Therefore, there is little overlap.

As we raise the percentage of MP transactions, the throughput of H-Store plummets. Transactions almost always abort due to partition lock conflicts. For Lotus, which adds granule-level locking on top of H-Store, throughput degrades gracefully as granule-level locking allows more concurrency. Lotus keeps its dominance over DS2PL and Sundial by 20-60% across all percentages of MP transactions. To drill down on performance gains, we show the breakdown of MP transaction average latency in Figure 9c. We see that Lotus's overall latency is smaller than DS2PL/Sundial by 57%/63%. This is due to synchronous active replication that overlaps the transaction execution on two replicas. Therefore, there is no replication afterward. Our commit latency is also smaller than DS2PL and Sundial. The reason is that Lotus commit protocol eliminates the expensive forced log write on participants. Sundial has a similar breakdown as D2SPL because its number of phases and synchronous log writes during commit are the same.

Overall, with granule-level locking, the H-Store architecture can be made competitive against fine-grained distributed concurrency control protocols.

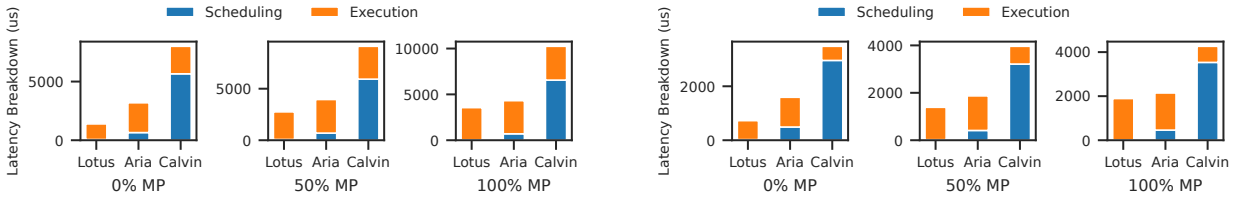
7.3 Deterministic Systems

YCSB Results. We start with the YCSB benchmark for deterministic systems. All accesses are uniform. We set the number of partitions to be 180. We configure 1,000 granules per partition, which we find works well for Lotus. We configure the batch size to be 1,200 transactions which is much larger than the number of partitions to fully saturate all the cores. The result is shown in Figure 10a. At 100% SP transactions, Lotus outperforms Aria and Calvin by 2.4×/5.9× respectively. As we increase the MP percentage, Lotus throughput starts to drop. This is expected as each MP transaction goes through the commit protocol which incurs network communication. However, even at 100% MP, Lotus is comparable to Aria and outperforms Calvin.

To drill down on the performance difference, we show a time breakdown during a batch execution within a worker in Figure 11a. We classify time into two activities: Scheduling and Execution. Execution is time spent doing useful work such as execution and commit. Scheduling refers to performing transaction scheduling such as sequencing and logging in Aria and Calvin. Scheduling also includes scheduling locks in Calvin. Since Lotus does not log before transaction execution and transactions start execution once they are submitted to the sequencer replica, there is nearly zero scheduling time. At 100% SP, Lotus spends 2.5x/5.7x less time finishing a batch of transactions. This is because transactions are local and run to completion without interruption or coordination.

Aria and Calvin both suffer from sequencing overhead and execution overhead for SP transactions. For Aria, although our benchmark ensures that SP transactions are generated on the nodes holding the data, each transaction still needs to check tuple-level conflicts among workers in the same node at the end of the batch. Besides, multiple cluster-wide barriers are required at batch boundaries. For Calvin, it is bottle-necked by the single-threaded lock manager for scheduling locks. With more MP transactions in the workload, Lotus execution time increases due to network stalls in execution and commit protocol. However, its execution time is still on par with Aria and beats Calvin thanks to the batch execution and commit scheme that overlaps network stalls with execution. Aria and Calvin are not affected because they do not run a commit protocol.

TPC-C Results: TPC-C results are shown in Figure 10a. The database is partitioned on warehouse id and runs with a total batch size of 600 transactions with 414 partitions(warehouses). Each worker gets 33 transactions per batch. The overall trend is similar



(a) Batch Latency Breakdown on YCSB

(b) Batch Latency Breakdown on TPC-C

Figure 11: Batch Latency Breakdown

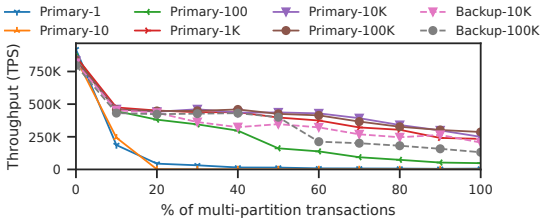


Figure 12: YCSB Performance Varying Number of Granules

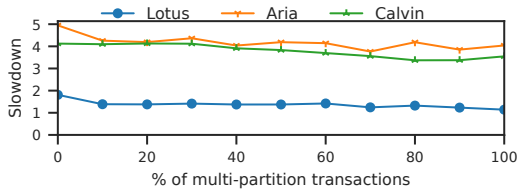


Figure 13: Slowdown by an 20ms Straggler YCSB Transaction

to YCSB. Lotus dominates Aria and Calvin with low MP percentage due to the same reasons described above. Lotus throughput degrades more quickly compare to YCSB as we increase the percentage of MP transactions. At 100% MP, Lotus throughput is 30% below that of Aria. This is because TPC-C has more conflicts than YCSB. Each warehouse tuple and district tuples within a warehouse are frequently updated. Lotus incurs an abort rate of 56% compared to 32% in Aria at 100% MP. Therefore, even though Lotus takes less time in batch execution, as shown in Figure 11b, more work is wasted. This is one of the limitations of Lotus.

We also included the throughput of Lotus backup replica (Lotus-Backup in Figure 10a) on YCSB and TPC-C. Our results show that the Lotus backup can closely follow the throughput of the primary which ensures fast failover. Typically, the backup is only a few batches behind the primary. Overall, on the low-to-medium contention workloads described here, Lotus can significantly outperform deterministic systems such Calvin and Aria on SP workloads. Lotus also stays competitive with MP workloads through granule-level locking and batching. The Lotus replica is also able to keep up with the throughput of its primary replica.

7.4 Granule Locking Overhead

We now explore the overhead of granule-level locking. First, we studied the locking overhead on the primary. We perform the same YCSB experiments as in Section 7.3 varying the number of granules from 1 to 100K (tuple-level locking). With one granule, it is effectively partition-level locking. Once the partition is locked at the transaction start, no locking is needed during the execution. The results are shown in Figure 12. At 0% MP, partition-level locking outperforms the configuration with 1K granules by only 6%. The overhead of granule-level locking is because of the hash function computation for identifying the granule and more lock acquisitions and releases. However, with more MP transactions, the throughput dramatically increases with more granules.

We also studied the throughput of backups for each configuration. For brevity, we only show configurations with a significant replication lag that occurs with 10K/100K granules. In this setting, the throughput of the backup (Backup-10K/Backup-100K in Figure 12) is up to 2.2× lower than their primaries. This is because tuple-level dependency tracking imposes a significant overhead. The memory usage of a backup is up to 8× higher than the primary for tracking dependencies. Therefore, coarser-grained granules are a better trade-off between concurrency and replay overhead.

7.5 Impact of Stragglers

Next, we experimentally show the impact of straggler transactions in existing batch-based transaction processing systems. Straggler transactions are harmful to such systems because they typically need to wait for all the transactions in a batch to finish before moving on to the next batch. This means performance is determined by the slowest transaction. We use the YCSB benchmark in Figure 10a to explore this topic. For each batch of YCSB transactions, we randomly insert one straggler transaction into each batch that sleeps for 20 milliseconds in addition to performing its transaction logic. We measure the total throughput slowdown caused by this transaction. The result is shown in Figure 13. We observe that the straggler impact to Lotus throughput is the smallest: 1.8× slowdown observed at 100% SP. For Aria and Calvin, the impact could be 4.9× and 4.1×. In Aria and Calvin, a single straggler transaction in a batch will cause all workers to wait for it to finish before starting the next batch. The impact of stragglers is lessened in Lotus as batching happens at the individual worker level.

7.6 Scalability Experiment

To show the scalability of Lotus, we ran the same YCSB experiments varying the number of nodes in a replica. Each node stores 180 partitions and each worker uses 200 transactions per batch. We studied various percentages of MP transactions. We also studied the scalability of the backup replaying transactions and whether the backup can keep up with the primary when scaling out to additional nodes. Therefore, the throughput of the replica is also studied. The results are shown in Figure 14. Lotus shows near-linear scalability. When running with 12 nodes in a primary, the throughput is 6.0×/5.6×/5.3×/4.8× of the throughput with 2 nodes respectively for workloads with 0%/5%/10%/25% MP transactions. The backup

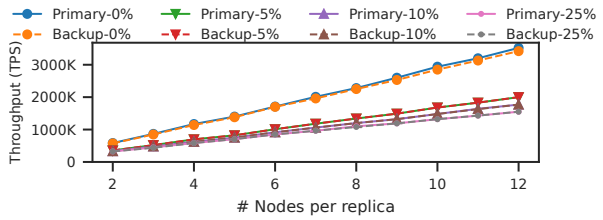


Figure 14: Scalability of Lotus

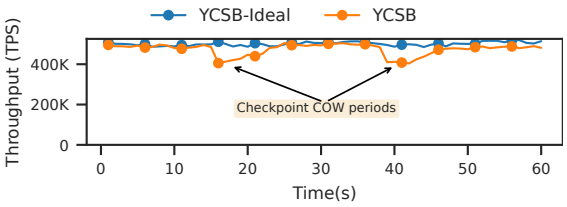


Figure 15: Asynchronous Checkpoint Overhead

also scales linearly. The throughput of backup in these workloads closely follows the primary, implying a minimum replication lag.

7.7 Asynchronous Checkpoint Overhead

We next performed an experiment using YCSB to show the overhead of the checkpoint process described in Section 5.2. The database is hashed into 180 partitions among 3 nodes each with 6 workers. Each partition contains 400K tuples and 2000 granules. The transactions in the workload consist of 50% reads and 50% updates. Checkpoint is triggered every 15 seconds. We set the batch size to be 1200 transactions. 50% of the transactions are multi-partition transactions. We ran the experiment for 60 seconds. The system writes log and checkpoint files to two separate storage devices to eliminate interference. The results are shown in Figure 15. For comparison, we also run the experiment with checkpoints disabled (labeled as YCSB-Ideal in Figure 15). We observe that during periods without checkpoint-ing, throughput is similar to that of YCSB-Ideal. When there are checkpoint activities, around time ranges 15s–25s and 35s–45s, we observe a maximum throughput drop of 26%. These overheads mainly come from copying tuples to the shadow table. We do not show the performance impact of backups for brevity as they were similar. Overall, the performance impact is moderate. Note that the checkpointing overhead needs to be paid for any DBMS that adopts command logging and is not unique to Lotus.

8 RELATED WORK

Lotus is inspired by previous works on in-memory transaction processing, distributed commit, and highly available DBMSes.

8.1 Main-Memory Transaction Processing

There has been a vibrant research community around in-memory transaction processing. This includes multi-core in-memory transaction processing [18, 34, 49, 64, 66, 68, 70] as well as distributed transaction processing [14, 26, 43, 50]. Epoch-based [64, 70] transaction processing exploits the idea of time-window-based transaction batching to trade latency for throughput. The idea is applicable in distributed setting [15, 41, 43] as well. Jones et al[32] proposed to speculatively execute transactions during stalls of 2PC in H-Store. However, a significant amount of CPU cycles are still wasted in stalls from MP execution and cascading aborts. Lotus dramatically reduces network stalls in an H-Store style architecture by executing a batch of MP transactions concurrently without losing the competitive edge of SP transaction performance.

8.2 Distributed Commit

There are many works on optimizing atomic commit protocols [5, 7, 23, 24, 42, 58, 59]. Most related to Lotus is the line of work focuses on one-phase commit protocol by having only coordinator performing either physical [7, 39, 58, 59] or logical [5] logging. The durability and atomicity properties are guaranteed by the coordinator. Lotus is in the similar spirit of these protocols. However, Lotus is more focused on the extreme form of logical logging, i.e., command logging. COCO [42], proposes amortizing the overhead of 2PC by performing commit for transactions in an epoch. However, it adds latency to individual transactions and suffers from imbalanced workloads with stragglers. Lotus also amortize the overhead of distributed commit. However, batching in Lotus is local to a sequencer without relying on fixed time window. Therefore, Lotus does not add much latency and is more robust against straggler.

8.3 Highly-Available Databases

DBMS high-availability is typically achieved through replication. Existing replication schemes can be classified as active-passive [10, 55] or active-active [2, 41, 50, 52, 63]. Active-passive scheme replicates outputs after the primary has finished execution through log shipping. Whereas active-active scheme replicates inputs which is used extensively in deterministic systems. For H-Store and VoltDB, active-active is implemented by synchronously replicating commands to all replicas. Systems like Calvin and Aria rely on consensus algorithms [37, 48] or consensus-based log store [6] to ensure data consistency during the replication of input transactions. The replication scheme of Lotus is similar to that of Calvin and Aria. The difference is that Lotus only enforces partial orderings of transactions. Some recent works [17, 28, 29] explore multi-shard transaction processing in resilient systems (e.g., blockchains). Lotus does not tolerate the Byzantine faults assumed in these systems. Therefore, the commit protocol in Lotus is made more lightweight.

9 CONCLUSION

In this paper, we presented the design of Lotus, a distributed and scalable in-memory database based on the concurrency control scheme from H-Store. We identified bottlenecks that cause poor MP performance in the H-Store architecture. Lotus improves the MP transaction throughput by introducing granule locks for more concurrency and batch execution/commit for overlapping computation and communication. We explored decoupling transaction replay on replica from the commit protocol in the original H-Store/VoltDB design. We showed that on YCSB and TPC-C, Lotus can outperform 2PL and OCC on workloads with MP transactions. Compared to state-of-the-art deterministic protocols, Lotus is both more performant and robust.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Qian Li, and Peter Kraft for their insightful feedback on the early draft of this work.

REFERENCES

[1] [n.d.]. Gitlab.com Database Incident. https://about.gitlab.com/blog/2017/02/01/gitlab-dot-com-database-incident/

[2] [n.d.]. H-Store. http://hstore.cs.brown.edu.

[3] [n.d.]. VoltDB. http://voltdb.com.

[4] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.

[5] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. 1998. One-phase commit: does it make sense?. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*. IEEE, 182–192.

[6] Jung-Sang Ahn, Woon-Hak Kang, Kun Ren, Guogen Zhang, and Sami Ben-Romdhane. 2019. Designing an efficient replicated log store with consensus protocol. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[7] Y Al-Houmaily and P Chrysanthis. 1995. Two-phase commit in gigabit-networked distributed databases. In *Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*. Citeseer.

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable atomic visibility with RAMP transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 1–45.

[9] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[10] Thomas C Bressoud and Fred B Schneider. 1996. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 80–107.

[11] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. 1993. The primary-backup approach. *Distributed systems* 2 (1993), 199–216.

[12] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. 2020. Dbos: A proposal for a data-centric operating system. *arXiv preprint arXiv:2007.11112* (2020).

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.

[14] James Cowling and Barbara Liskov. 2012. Granola:{Low-Overhead} Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 223–235.

[15] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 727–743.

[16] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[17] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[18] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[19] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (1976), 624–633.

[20] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201.

[21] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[22] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

[23] Zhihan Guo, Xinyu Zeng, Ziwei Ren, and Xiangyao Yu. 2021. Cornus: One-Phase Commit for Cloud Databases with Storage Disaggregation. *arXiv preprint arXiv:2102.10185* (2021).

[24] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol.. In *EDBT*. 157–168.

[25] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)* 15, 4 (1983), 287–317.

[26] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment* 10, 5 (2017), 553–564.

[27] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD*. 981–992.

[28] Jelle Hellings, Daniel P Hughes, Joshua Primero, and Mohammad Sadoghi. 2020. Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing. *arXiv preprint arXiv:2008.04450* (2020).

[29] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.

[30] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[31] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*. 24–35.

[32] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. 603–614.

[33] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*. 1496–1499.

[34] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*. 1675–1687.

[35] Evan Klitzke. 2020. Why uber engineering switched from postgres to mysql. https://eng.uber.com/postgres-to-mysql-migration/

[36] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[37] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[38] Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*. 203–226.

[39] Inseon Lee and Heon Young Yeom. 2002. A single phase distributed commit protocol for main memory database systems. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 8–pp.

[40] Qian Li, Peter Kraft, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, et al. [n.d.]. A Progress Report on DBOS: A Database-oriented Operating System. ([n. d.]).

[41] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.

[42] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. *Proceedings of the VLDB Endowment* 14, 5 (2021), 743–756.

[43] Yi Lu, Xiangyao Yu, and Samuel Madden. 2018. Star: Scaling transactions through asymmetric replication. *arXiv preprint arXiv:1811.02059* (2018).

[44] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 604–615.

[45] C Mohan and Bruce Lindsay. 1985. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review* 19, 2 (1985), 40–52.

[46] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[47] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2012. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*.

[48] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[49] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. *Proceedings of the VLDB Endowment* 3, ARTICLE (2010), 928–939.

[50] Andrew Pavlo. 2014. *On scalable transaction execution in partitioned main memory database management systems*. Ph.D. Dissertation. PhD thesis, Brown University.

[51] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 61–72.

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

[53] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[54] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1747–1761.

[55] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. 2010. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review* 44, 4 (2010), 30–39.

[56] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 133–142.

[57] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe,

et al. 2021. DBOS: a DBMS-oriented Operating System. *Proceedings of the VLDB Endowment* 15, 1 (2021), 21–30.

[58] James W Stamos and Flaviu Cristian. 1990. A low-cost atomic commit protocol. In *Proceedings Ninth Symposium on Reliable Distributed Systems*. IEEE, 66–75.

[59] James W Stamos and Flaviu Cristian. 1993. Coordinator log transaction execution protocol. *Distributed and Parallel Databases* 1, 4 (1993), 383–408.

[60] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*. 1150–1160.

[61] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[62] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/.

[63] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. 1–12.

[64] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 18–32.

[65] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1654–1655.

[66] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[67] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.

[68] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*. 1629–1642.

[69] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[70] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*.