STAR: Scaling Transactions through Asymmetric Replication

Yi Lu
MIT CSAIL
yilu@csail.mit.edu

Xiangyao Yu
MIT CSAIL
yxy@csail.mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

In this paper, we present STAR, a new distributed in-memory database with asymmetric replication. By employing a single-node non-partitioned architecture for some replicas and a partitioned architecture for other replicas, STAR is able to efficiently run both highly partitionable workloads and workloads that involve cross-partition transactions. The key idea is a new *phase-switching* algorithm where the execution of single-partition and cross-partition transactions is separated. In the partitioned phase, single-partition transactions are run on multiple machines in parallel to exploit more concurrency. In the single-master phase, mastership for the entire database is switched to a single designated master node, which can execute these transactions without the use of expensive coordination protocols like two-phase commit. Because the master node has a full copy of the database, this phase-switching can be done at negligible cost. Our experiments on two popular benchmarks (YCSB and TPC-C) show that high availability via replication can coexist with fast serializable transaction execution in distributed in-memory databases, with STAR outperforming systems that employ conventional concurrency control and replication algorithms by up to one order of magnitude.

PVLDB Reference Format:
Yi Lu, Xiangyao Yu and Samuel Madden. STAR: Scaling Transactions through Asymmetric Replication. *PVLDB*, 12(11): 1316-1329, 2019.
DOI: https://doi.org/10.14778/3342263.3342270

1. INTRODUCTION

Recent years have seen a number of in-memory transaction processing systems that can run hundreds of thousands to millions of transactions per second by leveraging multi-core parallelism [46, 50, 54]. These systems can be broadly classified into i) partitioning-based systems, e.g., H-Store [46] which partitions data onto different cores or machines, and ii) non-partitioned systems that try to minimize the overheads associated with concurrency control in a single-server, multi-core setting by avoiding locks and contention whenever possible [37, 38, 50, 54], while allowing any transaction to run on any core/processor.

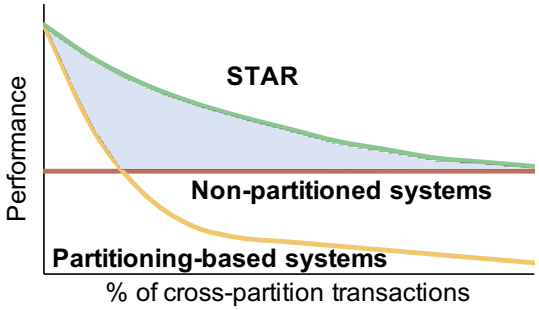


Figure 1: Partitioning-based systems vs. Non-partitioned systems

As shown in Figure 1, partitioning-based systems work well when workloads have few cross-partition transactions, because they are able to avoid the use of any synchronization and thus scale out to multiple machines. However, these systems suffer when transactions need to access more than one partition, especially in a distributed setting where expensive protocols like two-phase commit are required to coordinate these cross-partition transactions. In contrast, non-partitioned approaches provide somewhat lower performance on highly-partitionable workloads due to their inability to scale out, but are not sensitive to how partitionable the workload is.

In this paper, we propose a new transaction processing system, STAR, that is able to achieve the best of both worlds. We start with the observation that most modern transactional systems will keep several replicas of data for high availability purposes. In STAR, we ensure that at least one of these replicas is *complete*, i.e., it stores all records on a single machine, as in the recent non-partitioned approaches. We also ensure that at least one of the replicas is partitioned across several processors or machines, as in partitioned schemes. The system runs in two phases, using a novel *phase-switching* protocol: in the *partitioned* phase, transactions that can be executed completely on a single partition are mastered at one of the partial replicas storing that partition, and replicated to all other replicas to ensure fault tolerance and consistency. Cross-partition transactions are executed during the *single-master* phase, during which mastership for all records is switched to one of the complete replicas, which runs the transactions to completion and replicates their results to the other replicas. Because this node already has a copy of every record, changing the master

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342270

for a record (*re-mastering*) can be done without transferring any data, ensuring lightweight phase switching. Furthermore, because the node has a complete replica, transactions can be coordinated within a single machine without a slow commit protocol like two-phase commit. In this way, cross-partition transactions do not incur commit overheads as in typical partitioned approaches (because they all run on the single master), and can be executed using a lightweight concurrency control protocol like Silo [50]. Meanwhile, single-partition transactions can still be executed without any concurrency control at all, as in H-Store [46], and can be run on several machines in parallel to exploit more concurrency. The latency that phase switching incurs is no larger than the typical delay used in high-performance transaction processing systems for an epoch-based group commit.

Although the primary contribution of STAR is this phase-switching protocol, to build the system, we had to explore several novel aspects of in-memory concurrency control. In particular, prior systems like Silo [50] and TicToc [54] were not designed with high-availability (replication) in mind. Making replication work efficiently in these systems requires some care and is amenable to a number of optimizations. For example, STAR uses intra-phase asynchronous replication to achieve high performance. In the meantime, it ensures consistency among replicas via a replication fence when phase-switching occurs. In addition, with our phase-switching protocol, STAR can use a cheaper replication strategy than that employed by replicated systems that need to replicate entire records [55]. This optimization can significantly reduce bandwidth requirements (e.g., by up to an order of magnitude in our experiments with TPC-C.).

Our system does require a single node to hold an entire copy of the database, as in many other modern transactional systems [12, 17, 22, 26, 50, 52, 54]. Cloud service providers, such as Amazon EC2 [2] and Google Cloud [3], now provide high memory instances with 12 TB RAM, and 24 TB instances are coming in the fall of 2019. Such high memory instances are sufficient to store 10 kilobytes of online state about each customer in a database with about one billion customers, which exceeds the scale of all but the most demanding transactional workloads. In addition, on workloads with skew, a single-node in-memory database can be further extended through an anti-caching architecture [10], which provides 10x larger storage and competitive performance to in-memory databases.

In summary, STAR is a new distributed and replicated in-memory database that employs both partitioning and replication. It encompasses a number of interesting aspects:

- By exploiting multicore parallelism and fast networks, STAR is able to provide high throughput with serializability guarantees.
- It employs a *phase-switching* scheme which enables STAR to execute cross-partition transactions without two-phase commit while preserving fault tolerance guarantees.
- It uses a hybrid replication strategy to reduce the overhead of replication, while providing transactional consistency and high-availability.

In addition, we present a detailed evaluation of STAR that demonstrates its ability to provide adaptivity, high availability, and high-performance transaction execution. STAR outperforms systems that employ conventional distributed concurrency control algorithms by up to one order of magnitude on YCSB and TPC-C.

2. BACKGROUND

In this section, we describe how concurrency control allows database systems to execute transactions with serializability guarantees. We also introduce how replication is used in database systems with consistency guarantees.

2.1 Concurrency Control Protocols

Serializability — where the operations of multiple transactions are interleaved to provide concurrency while ensuring that the state of the database is equivalent to some serial ordering of the transactions — is the gold standard for transaction execution. Many serializable concurrency control protocols have been proposed, starting from early protocols that played an important role in hiding disk latency [4] to modern OLTP systems that exploit multicore parallelism [50, 54], typically employing lock-based and/or optimistic techniques.

Two-Phase locking (2PL) is the most widely used classical protocol to ensure serializability of concurrent transactions [16]. 2PL is considered pessimistic since the database acquires locks on operations even when there is no conflict. By contrast, optimistic concurrency control protocols (OCC) avoid this by only checking conflicts at the end of a transaction's execution [23]. OCC runs transactions in three phases: read, validation, and write. In the read phase, transactions perform read operations from the database and write operations to local copies of objects without acquiring locks. In the validation phase, conflict checks are done against all concurrently committing transactions. If conflicts exist, the transaction aborts. Otherwise, it enters the write phase and copies its local modifications to the database. Modern systems, like Silo [50], typically employ OCC-like techniques because they make it easier to avoid the overhead of shared locks during query execution.

In distributed database systems, cross-partition transactions involving many machines are classically coordinated using two-phase commit (2PC) [34] protocol to achieve fault tolerance, since machines can fail independently. The coordinator decides to commit or abort a transaction based on decisions collected from workers in the prepare phase. Workers must durably remember their decisions in the prepare phase until they learn the transaction outcome. Once the decision on the coordinator is made, 2PC enters the commit phase, and workers commit or abort the transaction based on its decision. Although 2PC does ensure serializability, the additional overhead of multiple log messages and network round trips for each transaction can significantly reduce throughput of distributed transactions. In STAR, we avoid two-phase commit by employing a phase-switching protocol to re-master records in distributed transactions to a single primary, which runs transactions locally and replicates them asynchronously.

2.2 Replication

Modern database systems need to be highly available. When a subset of servers in a cluster fails, the system needs to quickly reconfigure itself and replace a failed server with a standby machine, such that an end user does not experience any noticeable downtime. High availability requires

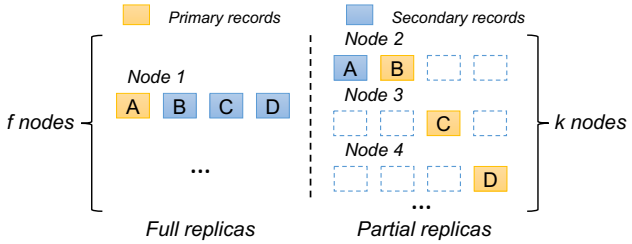


Figure 2: The architecture of STAR

the data to be replicated across multiple machines in order to allow for fast fail-over.

Primary/backup replication is one of the most widely used schemes. After the successful execution of a transaction, the primary node sends the log to all involved backup nodes. The log can either contains values [32, 33, 36, 42], which are applied by the backup nodes to the replica, or operations [43], which are re-executed by the backup nodes. For a distributed database, each piece of data has one primary node and one or multiple backup nodes. A backup node for some data can also be a primary node for some other data.

In distributed database systems, both 2PC and replication are important to ensure ACID properties. They are both expensive compared to a single node system, but in different ways. In particular, replication incurs very large data transfer but does not necessarily need expensive coordination for each transaction.

3. STAR ARCHITECTURE

STAR is a distributed and replicated in-memory database. It separates the execution of single-partition transactions and cross-partition transactions using a novel *phase-switching protocol*. The system dynamically switches between *partitioned* and *single-master* phases. In STAR, each partition is mastered by a single node. During the partitioned phase, queries that touch only a single partition are executed one at a time to completion on their partition. Queries that touch multiple partitions (even if those partitions are on a single machine) are executed in the single-master phase on a single designated master node.

To support the phase-switching protocol, STAR employs asymmetric replication. As shown in Figure 2, the system consists of two types of replicas, namely, (1) full replicas, and (2) partial replicas. Each of the f nodes (left side of figure) has a full replica, which consists of all database partitions. Each of the k nodes (right side of figure) has a partial replica, which consists of a portion of the database. In addition, STAR requires that these k partial replicas together contain at least one full copy of the database. During the partitioned phase, each node (whether a full or partial replica) acts as a master for some part of the database. During the single-master phase, one of the f nodes acts as the master for the whole database. Note that writes of committed transactions are replicated at least $f+1$ times on a cluster of $f+k$ nodes. We envision f being small (e.g., 1), while k can be much larger. Having more than one full replica (i.e., $f > 1$) does not necessarily improve system performance, but provides higher availability when failures occur (See Section 4.5).

There are several advantages of this phase switching approach. First, in the single-master phase, cross-partition transactions are run on a single master node. As in existing non-partitioned systems, two-phase commit is not needed

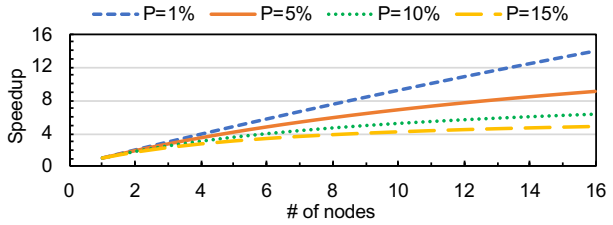


Figure 3: Performance speedup of asymmetric replication in STAR over single node execution; $\mathcal{P}$ = Percentage of cross-partition transactions

as the master node runs all cross-partition transactions and propagates writes to replicas. In contrast, transactions involving multiple nodes in existing distributed partitioning-based systems are coordinated with two-phase commit, which can significantly limit the throughput of distributed database systems [19]. Note that transactions running in the partitioned phase also do not require two-phase commit because they all run on a single partition, on a single node. Second, in the partitioned phase, single-partition transactions are run on multiple nodes in parallel, providing more concurrency, as in existing partitioning-based systems. This asymmetric replication approach is a good fit for workloads with a mix of single-partition and cross-partition transactions on a cluster of about 10 nodes. We demonstrate this both empirically and through the use of an analytical model, as shown in Sections 7 and 6.3. Figure 3 shows the speedup predicted by our model of STAR with n nodes over a single node.

STAR uses a variant of Silo's OCC protocol [50]. Each record in the database maintains a transaction ID (TID) from the transaction that last updated the record. The TID is used to determine which other transactions a committing transaction may have a conflict with. For example, once a transaction begins validation, it will abort if any one of the accessed records is locked by other transactions or has a different TID. The TID is assigned to a transaction after it is successfully validated. There are three criteria for the TID obtained from each thread: (a) it must be larger than the TID of any record in the read/write set; (b) it must be larger than the thread's last chosen TID; (c) it must be in the current global epoch. The first two criteria guarantee that the TIDs of transactions having conflicting writes are assigned in a serial-equivalent order. As in Silo, STAR uses a form of *epoch-based group commit*. The serial order of transactions running within an epoch is not explicitly known by the system except across epoch boundaries, since anti-dependencies (i.e., write-after-read conflicts) are not tracked. Different from Silo, in which the global epoch is incremented every 40 ms, a phase switch in STAR naturally forms an epoch boundary. Therefore, the system only releases the result of a transaction when the next phase switch occurs. At this time, the global epoch is also incremented.

At any point in time, each record is *mastered* on one node, with other nodes serving as secondaries. Transactions are only executed over primary records; writes of committed transactions are propagated to all replicas. One way to achieve consistency is to replicate writes *synchronously* among replicas. In other words, the primary node holds the write locks when replicating writes of a committed transaction to backup nodes. However, this design increases the latency of replication and impairs system performance. In

STAR, writes of committed transactions are buffered and *asynchronously* shipped to replicas, meaning the locks on the primary are not held during the process. To ensure correctness, we employ the Thomas write rule [47]: we tag each record with the last TID that wrote it and apply a write if the TID of the write is larger than the current TID of the record. Because TIDs of conflicting writes are guaranteed to be assigned in the serial-equivalent order of the writing transactions, this rule will guarantee correctness. In addition, STAR uses a *replication fence* when phase-switching occurs. In this way, strong consistency among replicas is ensured across the boundaries of the partitioned phase and the single-master phase.

Tables in STAR are implemented as collections of hash tables, which is typical in many in-memory databases [52, 53, 54]. Each table is built on top of a primary hash table and contains zero or more secondary hash tables as secondary indexes. To access a record, STAR probes the hash table with the primary key. Fields with secondary indexes can be accessed by mapping a value to the relevant primary key. Although STAR is built on top of hash tables, it is easily adaptable to other data structures such as Masstree [31]. As in most high performance transactional systems [46, 50, 54], clients send requests to STAR by calling pre-defined *stored procedures*. Parameters of a stored procedure must also be passed to STAR with the request. Arbitrary logic (e.g., read/write operations) is supported in stored procedures, which are implemented in C++.

STAR is *serializable* by default but can also support *read committed* and *snapshot isolation*. A transaction runs under *read committed* by skipping read validation on commit, since STAR uses OCC and uncommitted data never occurs in the database. STAR can provide snapshot isolation by retaining additional versions for records [50].

4. THE PHASE SWITCHING ALGORITHM

We now describe the phase switching algorithm we use to separate the execution of single-partition and cross-partition transactions. We first describe the two phases and how the system transitions between them. We next give a brief proof to show that STAR produces serializable results. In the end, we discuss how STAR achieves fault tolerance and recovers when failures occur.

4.1 Partitioned Phase Execution

Each node serves as primary for a subset of the records in the partitioned phase, as shown on the top of Figure 4. During this phase, we restrict the system to run transactions that only read from and write into a single partition. Cross-partition transactions are deferred for later execution in the single-master phase.

In the partitioned phase, each partition is touched only by a single worker thread. A transaction keeps a read set and a write set in its local copy, in case a transaction is aborted by the application explicitly. For example, an invalid item ID may be generated in TPC-C and a transaction with an invalid item ID is supposed to abort during execution. At commit time, it's not necessary to lock any record in the write set and do read validation, since there are no concurrent accesses to a partition in the partitioned phase. The system still generates a TID for each transaction and uses the TID to tag the updated records. In addition, writes of committed transactions are replicated to other backup nodes.

4.2 Single-Master Phase Execution

Any transaction can run in the single-master phase. Threads on the designated master node can access any record in any partition, since it has become the primary for all records. For example, node 1 is the master node on the bottom of Figure 4. We use multiple threads to run transactions using a variant of Silo's OCC protocol [50] in the single-master phase.

A transaction reads data and the associated TIDs and keeps them in its read set for later read validation. During transaction execution, a write set is computed and kept in a local copy. At commit time, each record in the write set is locked in a global order (e.g, the addresses of records) to prevent deadlocks. The transaction next generates a TID based on its read set, write set and the current epoch number. The transaction will abort during read validation if any record in the read set is modified (by comparing TIDs in the read set) or locked. Finally, records are updated, tagged with the new TID, and unlocked. After the transaction commits, the system replicates its write set to other backup nodes.

Note that fault tolerance must satisfy a transitive property [50]. The result of a transaction can only be released to clients after its writes and the writes of all transactions serialized before it are replicated. In STAR, the system treats each epoch as a commit unit through a replication fence. By doing this, it is guaranteed that the serial order of transactions is always consistent with the epoch boundaries. Therefore, the system does not release the results of transactions to clients until the next phase switch (and epoch boundary) occurs.

4.3 Phase Transitions

We now describe how STAR transitions between the two phases, which alternate after the system starts. The system starts in the partitioned phase, deferring cross-partition transactions for later execution.

For ease of presentation, we assume that all cross-partition transaction requests go to the designated master node (selected from among the f nodes with a full copy of the database). Similarly, each single-partition transaction request only goes to the participant node that has the mastership of the partition. In the single-master phase, the master node becomes the primary for all records. In the partitioned phase, the master node acts like other participant nodes to run single-partition transactions. This could be implemented via router nodes that are aware of the partitioning of the database. If some transaction accesses multiple partitions on a non-master node, the system would re-route the request to the master node for later execution.

In the partitioned phase, a thread on each partition fetches requests from clients and runs these transactions as discussed in Section 4.1. When the execution time in the partitioned phase exceeds a given threshold τ_p, STAR switches all nodes into the single-master phase, as shown in Figure 5. The phase switching algorithm is coordinated by a standalone *coordinator* outside of STAR instances. It can be deployed on any node of STAR or on a different node for better availability.

Before the phase switching occurs, the coordinator in STAR stops all worker threads. During a replication fence, all

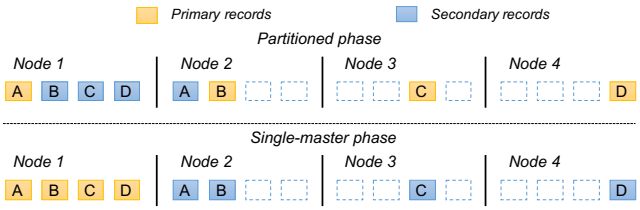


Figure 4: Illustrating the two execution phases

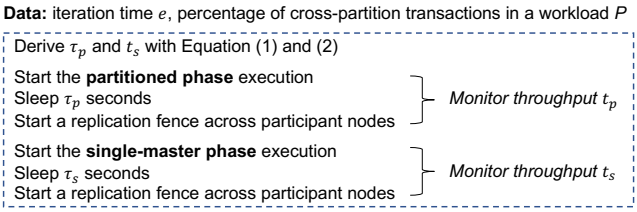


Figure 5: Phase transitions in STAR

participant nodes synchronize statistics about the number of committed transactions with one another. From these statistics each node learns how many outstanding writes it is waiting to see; nodes then wait until they have received and applied all writes from the replication stream to their local database. Finally, the coordinator switches the system to the other phase.

In the single-master phase, worker threads on the master node pull requests from clients and run transactions as discussed in Section 4.2. Meanwhile, the master node sends writes of committed transactions to replicas and all the other participant nodes stand by for replication. To further improve the utilization of servers, read-only transactions can run under read committed isolation level on non-master nodes at the client's discretion. Once the execution time in the single-master phase exceeds a given threshold τ_s, the system switches back to the partitioned phase using another replication fence.

The parameters τ_p and τ_s are set dynamically according to the system's throughput t_p in the partitioned phase, the system's throughput t_s in the single-master phase, the percentage $\mathcal{P}$ of cross-partition transactions in the workload, and the iteration time e.

$$\tau_p + \tau_s = e \quad (1)$$

$$\frac{\tau_s t_s}{\tau_p t_p + \tau_s t_s} = \mathcal{P} \quad (2)$$

Note that t_p, t_s and $\mathcal{P}$ are monitored and collected by the system in real time, and e is supplied by the user. Thus, these equations can be used to solve for τ_p and τ_s, as these are the only two unknowns. When there are no cross-partition transactions (i.e., $\mathcal{P} = 0$ and t_s is not well-defined), the system sets τ_p to e and τ_s to 0.

Intuitively, the system spends less time on synchronization with a longer iteration time e. In our experiments, we set the default iteration time to 10 ms; this provides good throughput while keeping latency at a typical level for high throughput transaction processing systems (e.g., Silo [50] uses 40 ms as a default).

Note that this deferral-based approach is symmetric so that single-partition transactions have the same expected mean latency as cross-partition transactions regardless of the iteration time (i.e., $\tau_p + \tau_s$), assuming all transactions arrive at a uniform rate. For a transaction, the latency depends on when the phase in which it is going to run ends. The mean latency is expected to be $(\tau_p + \tau_s)/2$.

4.4 Serializability

We now give a brief argument that transactions executed in STAR are serializable. A transaction only executes in a single phase, i.e., it runs in either the partitioned phase or the single-master phase. A replication fence between the partitioned phase and the single-master phase ensures that all writes from the replication stream have been applied to the database before switching to the next phase.

In the partitioned phase, there is only one thread running transactions serially on each partition. Each executed transaction only touches one partition, which makes transactions clearly serializable. In the single-master phase, STAR implements a variant of Silo's OCC protocol [50] to ensure that concurrent transactions are serializable. With the Thomas write rule, the secondary records are correctly synchronized, even though the log entries from the replication stream may be applied in an arbitrary order.

4.5 Fault Tolerance

In-memory replication provides high availability to STAR, since transactions can run on other active nodes even though some nodes failed. To prevent data loss and achieve durability, the system must log writes of committed transactions to disk. Otherwise, data will be lost when all replicas fail (e.g., due to power outage).

In this section, we first describe how STAR achieves durability via disk logging and checkpointing and then introduce how failures are detected. Finally, we discuss how STAR recovers from failures under different scenarios.

4.5.1 *Disk logging*

In STAR, each worker thread has a local recovery log. The writes of committed transactions along with some metadata are buffered in memory and periodically flushed to the log. Specifically, a log entry contains a single write to a record in the database, which has the following information: (1) key, (2) value, and (3) TID. The TID is from the transaction that last updated the record, and has an embedded epoch number as well. The worker thread periodically flushes buffered logs to disk; STAR also flushes all buffers to disk in the replication fence.

To bound the recovery time, a dedicated checkpointing thread can be used to periodically checkpoint the database to disk as well. The checkpointing thread scans the database and logs each record along with the TID to disk. A checkpoint also records the epoch number e_c when it starts. Once a checkpoint finishes, all logs earlier than epoch e_c can be safely deleted. Note that a checkpoint does not need to be a consistent snapshot of the database, as in SiloR [55], allowing the system to not freeze during checkpointing. On recovery, STAR uses the logs since the checkpoint (i.e., e_c) to correct the inconsistent snapshot with the Thomas write rule.

4.5.2 *Failure detection*

Before introducing how STAR detects failures, we first give some definitions and assumptions on failures. In this paper, we assume fail-stop failures. A *healthy node* is a node

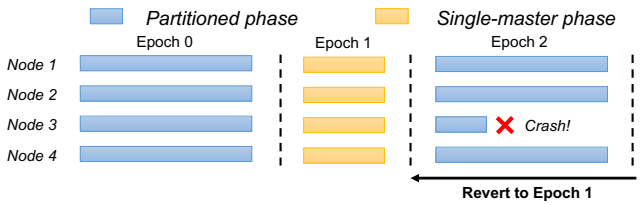


Figure 6: Failure detection in replication fence

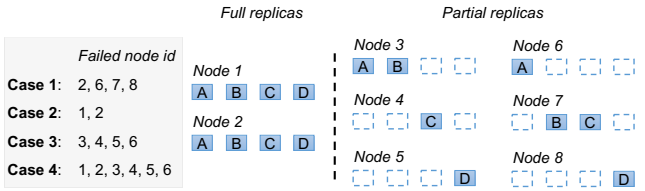


Figure 7: Illustrating different failure scenarios

that can connect to the coordinator, accept a client's requests, run transactions and replicate writes to other nodes. A *failed node* is one on which the process of a STAR instance has crashed [11] or which cannot communicate over the network.

The coordinator detects failures during the replication fence. If some node does not respond to the coordinator, it is considered to be a failed node. The list of failed nodes is broadcast to all healthy participant nodes in STAR. In this way, healthy nodes can safely ignore all replication messages from failed nodes that have lost network connectivity to the coordinator. Thus, the coordinator acts as a view service to solve the "split brain" problem, coordinating data movement across nodes on failures. To prevent the coordinator from being a single point of failure, it can be implemented as a replicated state machine with Paxos [24] or Raft [39].

Once a failure is detected by the coordinator, the system enters recovery mode and reverts the database to the last committed epoch, as shown in Figure 6. To achieve this, the database maintains two versions of each record. One is the most recent version prior to the current phase and the other one is the latest version written in the current phase. The system ignores all data versions written in the current phase, since they have not been committed by the database.

We next describe how STAR recovers from failures once the database has been reverted to a consistent snapshot.

4.5.3 Recovery

We use examples from Figure 7 to discuss how STAR recovers from failures. In these examples, there are 2 nodes with full replicas and 6 nodes with partial replicas (i.e., $f = 2$ and $k = 6$). A cluster of 8 nodes could fail in $2^8 - 1 = 255$ different ways, which fall into the following four different scenarios. Here, a "full replica" is a replica on a single node, and a "complete partial replica" is a set of partial replicas that collectively store a copy of the entire database.

(1) At least one full replica and one complete partial replica remain.
(2) No full replicas remain but at least one complete partial replica remains.
(3) No complete partial replicas remain but at least one full replica remains.
(4) No full replicas or complete partial replicas remain.

We now describe how STAR recovers under each scenario.

Case 1: As shown in Figure 7, failures occur on nodes 2, 6, 7 and 8. The system can still run transactions with the phase-switching algorithm. When a failed node recovers, it copies data from remote nodes and applies them to its database. In parallel, it processes updates from the relevant currently healthy nodes using the Thomas write rule. Once all failed nodes finish recovery, the system goes back to the normal execution mode.

Case 2: If no full replicas are available, the system falls back to a mode in which a distributed concurrency control algorithm is employed, as in distributed partitioning-based systems (e.g., Dist. OCC). The recovery process on failed nodes is the same as in Case 1.

Case 3: If no complete partial replicas are available, the system can still run transactions with the phase-switching algorithm. However, the mastership of records on lost partitions have to be reassigned to the nodes with full replicas. If all nodes with partial replicas fail, the system runs transactions only on full replicas without the phase-switching algorithm. The recovery process on failed nodes is the same as in Case 1.

Case 4: The system stops processing transactions (i.e., loss of availability) when no complete replicas remain. Each crashed node loads the most recent checkpoint from disk and restores its database state to the end of the last epoch by replaying the logs since the checkpoint with the Thomas write rule. The system goes back to the normal execution mode once all nodes finish recovery.

Note that STAR also supports recovery from nested failures. For example, an additional failure on node 3 could occur during the recovery of Case 1. The system simply reverts to the last committed epoch and begins recovery as described in Case 3.

5. REPLICATION: VALUE VS. OPERATION

In this section, we describe the details of our replication schemes, and how replication is done depending on the execution phase. As discussed earlier, STAR runs single-partition and cross-partition transactions in different phases. The system uses different replication schemes in these two phases: in the single-master phase, because a partition can be updated by multiple threads, records need to be fully-replicated to all replicas to ensure correct replication. However, in the partitioned phase, where a partition is only updated by a single thread, the system can use a better replication strategy based on replicating operations to improve performance. STAR provides APIs for users to manually program the operations, e.g., string concatenation.

To illustrate this, consider two transactions being run by two threads: `T1: R1.A = R1.B + 1; R2.C = 0` and `T2: R1.B = R1.A + 1; R2.C = 1`. Suppose `R1` and `R2` are two records from different partitions and we are running in the single-master phase. In this case, because the writes are done by different threads, the order in which the writes arrive on replicas may be different from the order in which transactions commit on the primary. To ensure correctness, we employ the Thomas write rule: apply a write if the TID of the write is larger than the current TID of the record. However, for this rule to work, each write must include the values of all fields in the record, not just the updated fields. To see this, consider the example in the left side of Figure 8

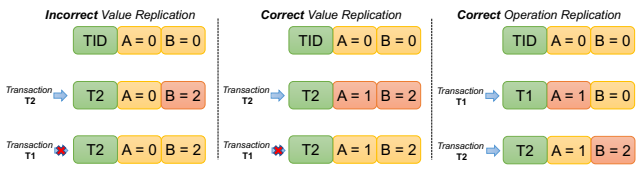


Figure 8: Illustrating different replication schemes; Red rectangle shows an updated field

(only R1 is shown); For record `R1`, if `T1` only replicates `A`, `T2` only replicates `B`, and `T2`'s updates are applied before `T1`'s, transaction `T1`'s update to field `A` is lost, since `T1` is less than `T2`. Thus, when a partition can be updated by multiple threads, all fields of a record have to be replicated as shown in the middle of Figure 8. Note that fields that are always read-only do not need to be replicated.

Now, suppose `R1` and `R2` are from the same partition, and we run the same transactions in the partitioned phase, where transactions are run by only a single thread on each partition. If `T2` commits after `T1`, `T1` is guaranteed to be ahead of `T2` in the replication stream since they are executed by the same thread. For this reason, only the updated fields need to be replicated, i.e., `T1` can just send the new value for `A`, and `T2` can just send the new value for `B` as shown in the right side of Figure 8. Furthermore, in this case, the system can also choose to replicate the *operation* made to a field instead of the value of a field in a record. This can significantly reduce the amount of data that must be sent. For example, in the `Payment` transaction in TPC-C, a string is concatenated to a field with a 500-character string in `Customer` table. With operation replication, the system only needs to replicate a short string and can re-compute the concatenated string on each replica, which is much less expensive than sending a 500-character string over network. This optimization can result in an order-of-magnitude reductions in replication cost.

In STAR, a *hybrid replication* strategy is used, i.e., the master node uses value replication strategy in the single-master phase and all nodes use the operation replication strategy in the partitioned phase. The hybrid strategy achieves the best of of both worlds: (1) value replication enables out-of-order replication, not requiring a serial order which becomes a bottleneck in the single master phase (See Section 7.5), and (2) in the partitioned phase, operation replication reduces the communication cost compared to value replication, which always replicates the values of all fields in a record.

As discussed earlier, STAR logs the writes of committed transactions to disk for durability. The writes can come from either local transactions or remote transactions through replication messages. By default, STAR logs the whole record to disk for fast and parallel recovery. However, a replication message in operation replication only has operations rather than the value of a whole record. Consider the example in the right side of Figure 8. The replication messages only have `T1: A = 1` and `T2: B = 2`. To solve this problem, when a worker thread processes a replication message that contains an operation, it first applies the operation to the database and then copies the value of the whole record to its logging buffer. In other words, the replication messages are transformed into `T1: A = 1; B = 0` and `T2: A = 1; B = 2` before logging to disk. By doing this, the logs can still be replayed in any order with the Thomas write rule during recovery.

		Non-partitioned		Partitioning-based	
	STAR	SYNC	ASYNC	SYNC	ASYNC
Write latency	Low	High	Low	High	Medium
Commit latency	High	Low	High	Low	High
Scale out	Medium	Low	Low	High	High
Performance sensitivity to cross-partition transactions	Low	Low	Low	High	High
Replication strategy	Hybrid	Operation	Value	Operation	Value

Figure 9: Overview of each approach; SYNC: synchronous replication; ASYNC: asynchronous replication + epoch-based group commit

6. DISCUSSION

We now discuss the trade-offs that non-partitioned systems and partitioning-based systems achieve and use an analytical model to show how STAR achieves the best of both worlds.

6.1 Non-partitioned Systems

A typical approach to build a fault tolerant non-partitioned system is to adopt the primary/backup model. A primary node runs transactions and replicates writes of committed transactions to one or more backup nodes. If the primary node fails, one of the backup nodes can take over immediately without loss of availability.

As we show in Figure 9, the writes of committed transactions can be replicated from the primary node to backup nodes either synchronously or asynchronously. With synchronous replication, a transaction releases its write locks and commits as soon as the writes are replicated (low commit latency), however, round trip communication is needed even for single-partition transactions (high write latency). With asynchronous replication, it's not necessary to hold write locks on the primary node during replication, and the writes may be applied in any order on backup nodes with value replication and the Thomas write rule. To address the potential inconsistency issue when a fault occurs, an epoch-based group commit (high commit latency) must be used as well. The epoch-based group commit serves as a barrier that guarantees all writes are replicated when transactions commit. Asynchronous replication reduces the amount of time that a transaction holds write locks during replication (low write latency) but incurs high commit latency for all transactions.

The performance of non-partitioned systems has low sensitivity to cross-partition transactions in a workload, but they cannot easily scale out. The CPU resources on backup nodes are often under-utilized, using more hardware to provide a lower overall throughput.

6.2 Partitioning-based Systems

In partitioning-based systems, the database is partitioned in a way such that each node owns one or more partitions. Each transaction has access to one or more partitions and commits with distributed concurrency control protocols (e.g., strict two-phase locking) and 2PC. This approach is a good fit for workloads that have a natural partitioning as the database can be treated as many disjoint sub-databases. However, cross-partition transactions are frequent in real-world scenarios. For example, in the standard mix of TPC-C,

10% of `NewOrder` and 15% of `Payment` are cross-partition transactions.

The same primary/backup model as in non-partitioned systems can be utilized to make partitioning-based systems fault tolerant. With synchronous replication, the write latency of partitioning-based systems is the same as non-partitioned systems. With asynchronous replication, the write latency depends on the number of partitions each transaction updates and on variance of communication delays.

If all transactions are single-partition transactions, partitioning-based systems are able to achieve linear scalability. However, even with a small fraction of cross-partition transactions, partitioning-based systems suffer from high round trip communication cost such as remote reads and distributed commit protocols.

6.3 Achieving the Best of Both Worlds

We now use an analytical model to show how STAR achieves the best of both worlds. Suppose we have a workload with n_s single-partition transactions and n_c cross-partition transactions. We first analyze the time to run a workload with a partitioning-based approach on a cluster of n nodes. If the average time of running a single-partition transaction and a cross-partition transaction in a partitioning-based system is t_s and t_c seconds respectively, we have

$$T_{\text{Partitioning-based}}(n) = (n_s t_s + n_c t_c)/n \quad (3)$$

In contrast, the average time of running a cross-partition transaction is almost the same as running a single-partition transaction in a non-partitioned approach (e.g., in a primary-backup database), and therefore,

$$T_{\text{Non-partitioned}}(n) = (n_s + n_c)t_s \quad (4)$$

In STAR, single-partition transactions are run on all participant nodes, and cross-partition transactions are run with a single master node. If the replication and phase transitions are not bottlenecks, we have

$$T_{\text{STAR}}(n) = (n_s/n + n_c)t_s \quad (5)$$

We let $\mathcal{K} = t_c/t_s$ and $\mathcal{P} = n_c/(n_c + n_s)$. Thus, $\mathcal{K}$ indicates how much more expensive a cross-partition transaction is than a single-partition transaction, and $\mathcal{P}$ indicates the percentage of cross-partition transactions in a workload. We now give the performance improvement that STAR achieves over the other two approaches,

$$I_{\text{Partitioning-based}}(n) = \frac{T_{\text{Partitioning-based}}(n)}{T_{\text{STAR}}(n)} = \frac{\mathcal{K}\mathcal{P} - \mathcal{P} + 1}{n\mathcal{P} - \mathcal{P} + 1}$$

$$I_{\text{Non-partitioned}}(n) = \frac{T_{\text{Non-partitioned}}(n)}{T_{\text{STAR}}(n)} = \frac{n}{n\mathcal{P} - \mathcal{P} + 1}$$

Similarly, we have the scalability of asymmetric replication by showing the speedup that STAR achieves with n nodes over a single node,

$$I(n) = \frac{T_{\text{STAR}}(1)}{T_{\text{STAR}}(n)} = \frac{n}{n\mathcal{P} - \mathcal{P} + 1}$$

For different values of $\mathcal{K}$, we plot $I_{\text{Partitioning-based}}(4)$ and $I_{\text{Non-partitioned}}(4)$ in Figure 10, when varying the percentage of cross-partition transactions on a cluster of four nodes. STAR outperforms non-partitioned systems as long as there are single-partition transactions in a workload. This is because all single-partition transactions are run on all participant nodes, which makes the system utilize more CPU resources from multiple nodes. To outperform partitioning-based systems, the average time of running a cross-partition transaction must exceed n times of the average time to run a single-partition transaction (i.e., $\mathcal{K} > n$).

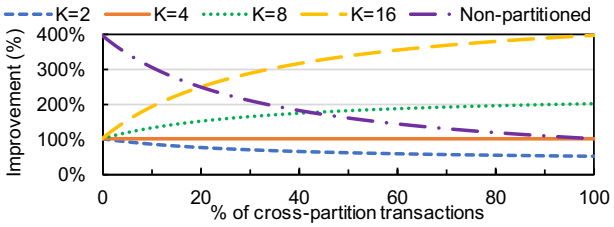


Figure 10: Illustrating effectiveness of STAR, vs. partitioning based systems for varying levels of $\mathcal{K}$, and against non-partitioned systems

7. EVALUATION

In this section, we evaluate the performance of STAR focusing on the following key questions:

- How does STAR perform compared to non-partitioned systems and partitioning-based systems?
- How does STAR perform compared to deterministic databases?
- How does the phase switching algorithm affect the throughput of STAR and what's the overhead?
- How effective is STAR's replication strategy?
- How does STAR scale?

7.1 Experimental Setup

We ran our experiments on a cluster of four `m5.4xlarge` nodes running on Amazon EC2 [2]. Each node has 16 2.50 GHz virtual CPUs and 64 GB of DRAM running 64-bit Ubuntu 18.04. `iperf` shows that the network between each node delivers about 4.8 Gbits/s throughput. We implemented STAR and other distributed concurrency control algorithms in C++. The system is compiled using GCC 7.3.0 with `-O2` option enabled.

In our experiments, we run 12 worker threads on each node, yielding a total of 48 worker threads. Each node also has 2 threads for network communication. We made the number of partitions equal to the total number of worker threads. All results are the average of three runs. We ran transactions at the serializability isolation level.

7.1.1 Workloads

To study the performance of STAR, we ran a number of experiments using two popular benchmarks:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) is a simple transactional workload designed to facilitate performance comparisons of database and key-value systems [6]. It has a single table with 10 columns. The primary key of each record is a 64-bit integer and each column consists of 10 random bytes. A transaction accesses 10 records and each access follows a uniform distribution. We set the number of records to 200K per partition, and we run a workload mix of 90/10, i.e., each transaction has 9 read operations and 1 read/write operation. By default, we run this workload with 10% cross-partition transactions that access to multiple partitions.

TPC-C: The TPC-C benchmark [1] is the gold standard for evaluating OLTP databases. It models a warehouse order

processing system, which simulates the activities found in complex OLTP applications. It has nine tables and we partition all the tables by `Warehouse ID`. We support two transactions in TPC-C, namely, (1) `NewOrder` and (2) `Payment`. 88% of the standard TPC-C mix consists of these two transactions. The other three transactions require range scans, which are currently not supported in our system. By default, we ran this workload with the standard mix, in which a `NewOrder` transaction is followed by a `Payment` transaction. By default, 10% of NewOrder and 15% of Payment transactions are cross-partition transactions that access multiple warehouses.

In YCSB, each partition adds about 25 MB to the database. In TPC-C, each partition contains one warehouse and adds about 100 MB to the database.

To measure the maximum throughput that each approach can achieve, every worker thread generates and runs a transaction to completion one after another in our experiments.

7.1.2 Concurrency control algorithms

To avoid an apples-to-oranges comparison, we implemented each of the following concurrency control algorithms in C++ in our framework.

STAR: This is our algorithm as discussed in Section 3. We set the iteration time of a phase switch to 10 ms. To have a fair comparison to other algorithms, disk logging, checkpointing, and the hybrid replication optimization are disabled unless otherwise stated.

PB. OCC: This is a variant of Silo's OCC protocol [50] adapted for a primary/backup setting. The primary node runs all transactions and replicates the writes to the backup node. Only two nodes are used in this setting.

Dist. OCC: This is a distributed optimistic concurrency control protocol. A transaction reads from the database and maintains a local write set in the execution phase. The transaction first acquires all write locks and next validates all reads. Finally, it applies the writes to the database and releases the write locks.

Dist. S2PL: This is a distributed strict two-phase locking protocol. A transaction acquires read and write locks during execution. The transaction next executes to compute the value of each write. Finally, it applies the writes to the database and releases all acquired locks.

In our experiments, PB. OCC is a non-partitioned system, and Dist. OCC and Dist. S2PL are considered as partitioning-based systems. We use `NO_WAIT` policy to avoid deadlocks in partitioning-based systems, i.e., a transaction aborts if it fails to acquire some lock. This deadlock prevention strategy was shown to be the most scalable protocol [19]. We do not report the results on PB. S2PL, since it always performs worse than PB. OCC [54]. Also note that we added an implementation of Calvin, described in Section 7.3.

7.1.3 Partitioning and replication configuration

In our experiment, we set the number of replicas of each partition to 2. Each partition is assigned to a node by a hash function. The primary partition and secondary partition are always hashed to two different nodes. In STAR, we have 1 node with full replica and 3 nodes with partial replica, i.e., $f = 1$ and $k = 3$. Each node masters a different portion of the database, as shown in Figure 2.

We consider two variations of PB. OCC, Dist. OCC, and Dist. S2PL: (1) asynchronous replication and epoch-based group commit, and (2) synchronous replication. Note that Dist. OCC and Dist. S2PL must use two-phase commit when synchronous replication is used. In addition, synchronous replication requires that all transactions hold write locks during the round trip communication for replication.

7.2 Performance Comparison

We now compare STAR with a non-partitioned system and two partitioning-based systems using both YCSB and TPC-C workloads.

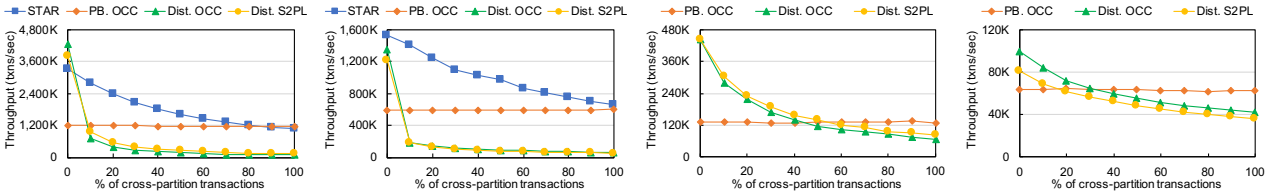
7.2.1 Results of asynchronous replication and epoch-based group commit

We ran both YCSB and TPC-C with a varying percentage of cross-partition transactions and report the results in Figure 11(a) and 11(b). When there are no cross-partition transactions, STAR has similar throughput compared with Dist. OCC and Dist. S2PL on both workloads. This is because the workload is embarrassingly parallel. Transactions do not need to hold locks for a round trip communication with asynchronous replication and epoch-based group commit. As we increase the percentage of cross-partition transactions, the throughput of PB. OCC stays almost the same, and the throughput of other approaches drops. When 10% cross-partition transactions are present, STAR starts to outperform Dist. OCC and Dist. S2PL. For example, STAR has 2.9x higher throughput than Dist. S2PL on YCSB and 7.6x higher throughput than Dist. OCC on TPC-C. As more cross-partition transactions are present, the throughput of Dist. OCC and Dist. S2PL is significantly lower than STAR (also lower than PB. OCC), and the throughput of STAR approaches the throughput of PB. OCC. This is because STAR behaves similarly to a non-partitioned system when all transactions are cross-partition transactions.

Overall, STAR running on 4 nodes achieves up to 3x higher throughput than a primary/backup system running on 2 nodes (e.g., PB. OCC) and up to 10x higher throughput than systems employing distributed concurrency control algorithms (e.g., Dist. OCC and Dist. S2PL) on 4 nodes. As a result, we believe that STAR is a good fit for workloads with both single-partition and cross-partition transactions. It can outperform both non-partitioned and partitioning-based systems, as we envisioned in Figure 1.

7.2.2 Results of synchronous replication

We next study the performance of PB. OCC, Dist. OCC and Dist. S2PL with synchronous replication. We ran the same workload as in Section 7.2.1 with a varying percentage of cross-partition transactions and report the results in Figure 11(c) and 11(d). For clarity, the y-axis uses different scales (See Figure 11(a) and 11(b) for the results of STAR). When there are no cross-partition transactions, the workload is embarrassingly parallel. However, PB. OCC, Dist. OCC, and Dist. S2PL all have much lower throughput than STAR in this scenario. This is because even single-partition transactions need to hold locks during the round trip communication due to synchronous replication. As we increase the percentage of cross-partition transactions, we observe that the throughput of PB. OCC stays almost the same, since the throughput of a non-partitioned system is not sensitive to the percentage of cross-partition transactions in a workload. Dist. OCC and Dist. S2PL have lower



(a) YCSB (b) TPC-C (c) YCSB w/ Sync. Rep. (d) TPC-C w/ Sync. Rep.

Figure 11: Performance and latency comparison of each approach on YCSB and TPC-C

% of cross-partition transactions	Synchronous replication						Asynchronous replication + Epoch-based group commit
	YCSB			TPC-C			Each approach has a similar latency due to epoch-based group commit
	10%	50%	90%	10%	50%	90%	
STAR	6.2/9.4						
PB. OCC	0.1/0.2	0.1/0.2	0.1/0.2	0.1/1.1	0.1/1.1	0.1/1.1	5.5/11.3
Dist. OCC	0.2/0.7	0.3/0.9	0.7/0.9	0.2/0.6	0.3/0.7	0.6/0.8	6.4/11.4
Dist. S2PL	0.2/4.5	0.4/6.0	0.6/6.7	0.2/4.9	0.5/6.8	0.8/8.9	6.2/11.2

Figure 12: Latency (ms) of each approach - 50th percentile/99th percentile

throughput, since more transactions need to read from remote nodes during the execution phase. They also need multiple rounds of communication to validate and commit transactions (2PC).

Overall, the throughput of PB. OCC, Dist. OCC, and Dist. S2PL is much lower than those with asynchronous replication and epoch-based group commit due to the overhead of network round trips for every transaction. STAR has much higher throughput than these approaches with synchronous replication — at least 7x higher throughput on YCSB, 15x higher throughput on TPC-C.

7.2.3 *Latency of each approach*

We now study the latency of each approach and report the latency at the 50th percentile and the 99th percentile in Figure 12, when the percentage of cross-partition transactions is 10%, 50%, and 90%. We first discuss the latency of each approach with synchronous replication. We observe that PB. OCC's latency at the 50th percentile and the 99th percentile is not sensitive to the percentage of cross-partition transactions. Dist. OCC and Dist. S2PL have higher latency at both the 50th percentile and the 99th percentile, as we increase the percentage of cross-partition transactions. This is because there are more remote reads and the commit protocols they use need multiple round trip communication. In particular, the latency of Dist. S2PL at the 99th percentile is close to 10 ms on TPC-C. In STAR, the iteration time determines the latency of transactions. Similarly, the latency of transactions in Dist. OCC and Dist. S2PL with asynchronous replication and epoch-based group commit depends on epoch size. For this reason, STAR has similar latency at the 50th percentile and the 99th percentile to other approaches with asynchronous replication. In Figure 12, we only report the results on YCSB with 10% cross-partition transactions for systems with asynchronous replication. Results on other workloads are not reported, since they are all similar to one another.

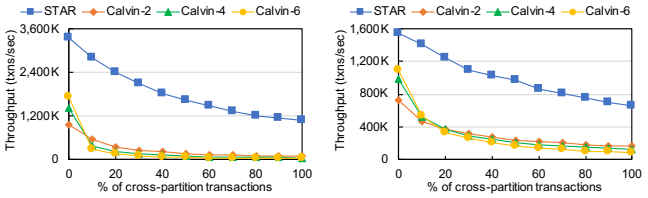
An epoch-based group commit naturally increases latency. Systems with synchronous replication have lower latency, but Figure 11(c) and 11(d) show that they have much lower throughput as well, even if no cross-partition transactions are present. In addition, the latency at the 99th percentile in systems with synchronous replication is much longer under some scenarios (e.g., Dist. S2PL on TPC-C). As prior work (e.g., Silo [50]) has argued, a few milliseconds more latency is not a problem for most transaction processing workloads, especially given throughput gains.

7.3 Comparison with Deterministic Databases

We next compare STAR with Calvin [49], which is a deterministic concurrency control and replication algorithm. In Calvin, a central sequencer determines the order for a batch of transactions before they start execution. The transactions are then sent to all replica groups of the database to execute deterministically. In Calvin, a replica group is a set of nodes containing a replica of the database. All replica groups will produce the same results for the same batch of transactions due to determinism. As a result, Calvin does not perform replication at the end of each transaction. Instead, it replicates inputs at the beginning of the batch of transactions and deterministically executes the batch across replica groups.

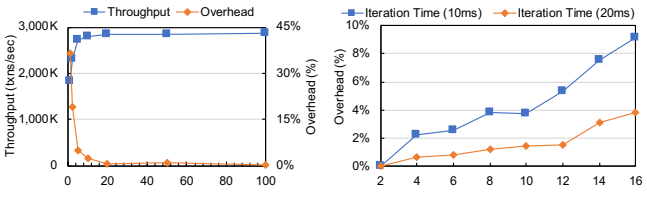
We implemented the Calvin algorithm in C++ in our framework as well to have a fair comparison. The original design of Calvin uses a single-threaded lock manager to grant locks to multiple execution threads following the deterministic order. To better utilize more CPU resources, we implemented a multi-threaded lock manager, in which each thread is responsible for granting locks in a different portion of the database. The remaining CPU cores are used as worker threads to execute transactions.

Increasing the number of threads for the lock manager does not always improve performance. The reasons are twofold: (1) fewer threads are left for transaction execution (2) more communication is needed among worker threads for cross-partition transactions. In this experiment, we consider three configurations with different number of threads used in the lock manager in Calvin, namely (1) Calvin-2, (2) Calvin-4 and (3) Calvin-6. We use Calvin-x to denote the number of threads used for the lock manager, i.e, there



(a) YCSB (b) TPC-C

Figure 13: Comparison with deterministic databases



(a) Iteration time (ms) (b) # of nodes

Figure 14: Overhead of phase transitions

are $12 - x$ threads executing transactions. In all configurations, we study the performance of Calvin in one replica group on 4 nodes. The results of Calvin-1 and Calvin-3 are not reported, since they never deliver the best performance.

We report the results on YCSB and TPC-C with a varying percentage of cross-partition transactions in Figure 13(a) and 13(b). When there are no cross-partition transactions, Calvin-6 achieves the best performance, since more parallelism is exploited (i.e., 6 worker threads on each node, yielding a total of 24 worker threads). Calvin-2 and Calvin-4 have lower throughput as the worker threads are not saturated when fewer threads are used for the lock manager. In contrast, STAR uses 12 worker threads on each node, yielding a total of 48 worker threads and has 1.4-1.9x higher throughput than Calvin-6. When all transactions are cross-partition transactions, Calvin-2 has the best performance. This is because Calvin-4 and Calvin-6 needs more synchronization and communication. Overall, STAR has 4-11x higher throughput than Calvin with various configurations.

7.4 The Overhead of Phase Transitions

We now study how the iteration time of a phase switch affects the overall throughput of STAR and the overhead due to this phase switching algorithm with a YCSB workload. Similar results were obtained on other workloads but are not reported due to space limitations. We varied the iteration time of the phase switching algorithm from 1 ms to 100 ms and report the system's throughput and overhead in Figure 14(a). The overhead is measured as the system's performance degradation compared to the one running with a 200 ms iteration time. Increasing the iteration time decreases the overhead of the phase switching algorithm as expected, since less time is spent during the synchronization. For example, when the iteration time is 1 ms, the overhead is as high as 43% and system only achieves around half of its maximum throughput (i.e., the throughput achieved with 200 ms iteration time). As we increase the iteration time, the system's throughput goes up. The throughput levels off when the iteration time is larger than 10 ms. On a cluster of 4 nodes, the overhead is about 2% with a 10 ms iteration time.

We also study the overhead of phase transitions with a varying number of nodes. We ran the same YCSB workload and report the results of 10 ms and 20 ms iteration time in Figure 14(b). Note that we also scale the number of partitions in the database correspondingly. For example, on a cluster of 16 nodes, there are $16 \times 12 = 192$ partitions in the database. In general, the overhead of phase transitions is larger with more nodes on a cluster due to variance of communication delays. In addition, a shorter iteration time makes the overhead smaller (20 ms vs. 10 ms).

Overall, the overhead of phase transitions is less than 5% with a 10 ms iteration time on a cluster of less than 10 nodes. In all experiments in this paper, we set the iteration time to 10 ms. With this setting, the system can achieve more than 95% of its maximum throughput and have good balance between throughput and latency.

7.5 Replication and Fault Tolerance

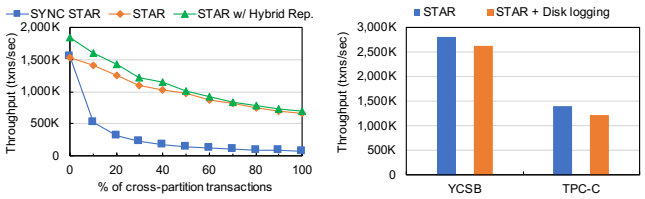
We now study the effectiveness of STAR's asynchronous replication in the single-master phase and the effectiveness of hybrid replication. We only report the results from TPC-C in this experiment, since a transaction in YCSB always updates the whole record. In Figure 15(a), `SYNC STAR` shows the performance of STAR that uses synchronous replication in the single-master phase. `STAR` indicates the one with asynchronous replication. `STAR w/ Hybrid Rep.` further enables operation replication in the partitioned phase on top of `STAR`. When there are more cross-partition transactions, `SYNC STAR` has much lower throughput than `STAR`. This is because more network round trips are needed during replication in the single-master phase. The improvement of `STAR w/ Hybrid Rep.` is also less significant, since fewer transactions are run in the partitioned phase.

We next show the performance degradation of STAR when disk logging is enabled. We ran both YCSB and TPC-C workloads and report the results in Figure 15(b). In summary, the overhead of disk logging and checkpointing is 6% in YCSB and 14% in TPC-C. Note that non-partitioned and partitioning-based systems would experience similar overheads from disk logging.

7.6 Scalability Experiment

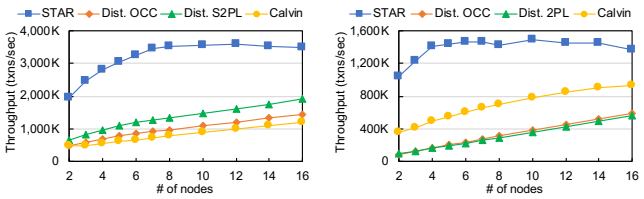
In this experiment, we study the scalability of STAR on both YCSB and TPC-C. We ran the experiment with a varying number of `m5.4xlarge` nodes and report the results in Figure 16. Note that the database is scaled correspondingly as we did in Section 7.4. On YCSB, STAR with 8 nodes achieves 1.8x higher throughput than STAR with 2 nodes. The performance of STAR stays stable beyond 8 nodes. On TPC-C, STAR with 4 nodes achieves 1.4x higher throughput than STAR with 2 nodes. The system stops scaling with more than 4 nodes. This is because the system saturates the network with 4 nodes (roughly 4.8 Gbits/sec). In contrast, Dist. OCC, Dist. S2PL and Calvin start with lower performance but all have almost linear scalability.

We believe it is possible for distributed partitioning-based systems to have competitive performance to STAR, although such systems will likely require more nodes to achieve comparable performance. Assuming linear scalability and the network not becoming a bottleneck (the ideal case for baselines), distributed partitioning-based systems maybe outperform STAR on YCSB and TPC-C with roughly 30-40 nodes.



(a) Replication strategies (b) Fault tolerance

Figure 15: Replication and fault tolerance experiment



(a) YCSB (b) TPC-C

Figure 16: Scalability experiment

8. RELATED WORK

STAR builds on a number of pieces of related work for its design, including in-memory transaction processing, replication and durability.

In-memory Transaction Processing. Modern fast in-memory databases have long been an active area of research [8, 13, 25, 38, 46, 50, 54]. In H-Store [46], transactions local to a single partition are executed by a single thread associated with the partition. This extreme form of partitioning makes single-partition transactions very fast but creates significant contention for cross-partition transactions, where whole-partition locks are held. Silo [50] divides time into a series of short epochs and each thread generates its own timestamp by embedding the global epoch to avoid shared-memory writes, avoiding contention for a global critical section. Because of its high throughput and simple design, we adopted the Silo architecture for STAR, reimplementing it and adding our phase-switching protocol and replication. Doppel [37] executes highly contentious transactions in a separate phase from other regular transactions such that special optimizations (i.e., commutativity) can be applied to improve scalability.

F1 [45] is an OCC protocol built on top of Google's Spanner [7]. MaaT [28] reduces transaction conflicts with dynamic timestamp ranges. ROCOCO [35] tracks conflicting constituent pieces of transactions and re-orders them in a serializable order before execution. To reduce the conflicts of distributed transactions, STAR runs all cross-partition transactions on a single machine in the single-master phase. Clay [44] improves data locality to reduce the number of distributed transactions in a distributed OLTP system by smartly partitioning and migrating data across the servers. Some previous work [27, 9, 5] proposed to move the master node of a tuple dynamically, in order to convert distributed transactions into local transactions. Unlike STAR, however, moving the mastership still requires network communication. FaRM [14], FaSST [20] and DrTM [53] improve the performance of a distributed OLTP database by exploiting RDMA. STAR can use RDMA to further decrease the overhead of replication and the phase switching algorithm as well.

Replicated Systems. Replication is the way in which database systems achieve high availability. Synchronous replication was popularized by systems like Postgres-R [21] and Galera Cluster [18], which showed how to make synchronous replication practical using group communication and deferred propagation of writes. Tashkent [15] is a fully replicated database in which transactions run locally on a replica. To keep replicas consistent, each replica does not communicate with each other but communicates to a certifier, which decides a global order for update transactions. Calvin [49] replicates transactions requests among replica groups and assigns a global order to each transaction for deterministic execution [48], allowing it to eliminate expensive distributed coordination. However, cross-node communication is still necessary during transaction execution because of remote reads. Mencius [30] is a state machine replication method that improves Paxos to achieve high throughput under high client load and low latency under low client load by partitioning sequence numbers, even under changing wide-area network environments. HRDB [51] tolerates Byzantine faults among replicas by scheduling transactions with a commit barrier. Ganymed [40, 41] runs update transactions on a single node and runs read-only transactions on a potentially unlimited number of replicas, allowing the system to scale read-intensive workloads. STAR is the first system that dynamically changes the mastership of records, to avoid distributed coordination. Neither a global order nor group communication is necessary, even for cross-partition transactions, since we run these cross-partition transactions in parallel on a single node.

Recoverable Systems. H-Store [29] uses transaction-level logging. It periodically checkpoints a transactionally consistent snapshot to disk and logs all the parameters of stored procedures. H-Store executes transactions following a global order and replays all the transactions in the same order during recovery. SiloR [55] uses a multi-threaded parallel value logging scheme that supports parallel replay in non-partitioned databases. In contrast, transaction-level logging requires that transactions be replayed in the same order. In STAR, different replication strategies, including both SiloR-like parallel value replication and H-Store-like operation replication are used in different phases, significantly reducing bandwidth requirements.

9. CONCLUSION

In this paper, we presented STAR, a new distributed in-memory database with asymmetric replication. STAR employs a new *phase-switching* scheme where single-partition transactions are run on multiple machines in parallel, and cross-partition transactions are run on a single machine by re-mastering records on the fly, allowing us to avoid cross-node communication and the use of distributed commit protocols like 2PC for distributed transactions. Our results on YCSB and TPC-C show that STAR is able to dramatically exceed the performance of systems that employ conventional concurrency control and replication algorithms by up to one order of magnitude.

Acknowledgments. We thank the reviewers for their valuable comments. Yi Lu is supported by the Facebook PhD Fellowship.

10. REFERENCES

[1] TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[2] Amazon EC2. https://aws.amazon.com/ec2/, 2019.

[3] Google Cloud. https://cloud.google.com/, 2019.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] P. Chairunnanda, K. Daudjee, and M. T. Özsu. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7(11):947–958, 2014.

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

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[8] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, pages 727–743, 2018.

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

[10] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[12] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[13] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[14] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[15] S. Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, pages 117–130, 2006.

[16] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[17] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[18] Galera Cluster. http://galeracluster.com/products/technology/, 2019.

[19] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *OSDI*, pages 185–201, 2016.

[21] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[22] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687, 2016.

[23] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[24] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[25] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[26] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35, 2017.

[27] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K. Tan, and Z. Wang. Towards a non-2PC transaction management in distributed database systems. In *SIGMOD Conference*, pages 1659–1674, 2016.

[28] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

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

[30] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, pages 369–384, 2008.

[31] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.

[32] D. McInnis. The basics of DB2 log shipping. https://www.ibm.com/developerworks/data/library/techarticle/0304mcinnis/0304mcinnis.html, 2003.

[33] Microsoft. About log shipping (SQL Server). https://msdn.microsoft.com/en-us/library/ms187103.aspx, 2016.

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

[35] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[36] MySQL. MySQL 8.0 reference manual,. https://dev.mysql.com/doc/refman/8.0/en/replication.html, 2019.

[37] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[38] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD Conference*, pages 677–689, 2015.

[39] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.

[40] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.

[41] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[42] PostgreSQL. PostgreSQL 9.6.13 documentation. https://www.postgresql.org/docs/9.6/static/warm-standby.html, 2019.

[43] D. Qin, A. Goel, and A. D. Brown. Scalable replay-based replication for fast databases. *PVLDB*, 10(13):2025–2036, 2017.

[44] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016.

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

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

[47] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *TODS*, 4(2):180–209, 1979.

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

[49] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.

[50] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[51] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72, 2007.

[52] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[53] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, pages 87–104, 2015.

[54] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *SIGMOD Conference*, pages 1629–1642, 2016.

[55] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.