Scalable Replay-Based Replication For Fast Databases

Dai Qin
University of Toronto
mike@eecg.toronto.edu

Angela Demke Brown
University of Toronto
demke@cs.toronto.edu

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

ABSTRACT

Primary-backup replication is commonly used for providing fault tolerance in databases. It is performed by replaying the database recovery log on a backup server. Such a scheme raises several challenges for modern, high-throughput multi-core databases. It is hard to replay the recovery log concurrently, and so the backup can become the bottleneck. Moreover, with the high transaction rates on the primary, the log transfer can cause network bottlenecks. Both these bottlenecks can significantly slow the primary database.

In this paper, we propose using record-replay for replicating fast databases. Our design enables replay to be performed scalably and concurrently, so that the backup performance scales with the primary performance. At the same time, our approach requires only 15-20% of the network bandwidth required by traditional logging, reducing network infrastructure costs significantly.

1. INTRODUCTION

Databases are often a critical part of modern computing infrastructures and hence many real-world database deployments use backup and failover mechanisms to guard against catastrophic failures. For example, many traditional databases use log shipping to improve database availability [19, 20, 24]. In this scheme, transactions run on a primary server and after they commit on the primary, the database recovery log is transferred asynchronously and replayed on the backup. If the primary fails, incoming requests can be redirected to the backup.

This replication scheme raises several challenges for in-memory, multi-core databases. These databases support high transaction rates, in the millions of transactions per second, for online transaction processing workloads [14, 16, 30]. These fast databases can generate 50 GB of data per minute on modern hardware [36]. Logging at this rate requires expensive, high-bandwidth storage [30] and leads to significant CPU overheads [17]. For replication, the log transfer requires a 10 Gb/s link for a single database. Failover and disaster recovery in enterprise environments, where the backup is located across buildings (possibly separated geographically), is thus an expensive proposition. These network links are expensive to operate or lease, and upgrades require major investment. A second challenge is that the backup ensures consistency with the primary by replaying the database log in serial order. This replay is hard to perform concurrently, and so the backup performance may not scale with the primary performance [13].

These challenges can lead to the network or the backup becoming a bottleneck for the primary, which is otherwise scalable. Two trends worsen these issues: 1) the availability of increasing numbers of cores, and 2) novel databases [21, 15], both of which further improve database performance.

While there is much recent work on optimizing logging and scalable recovery [17, 36, 35, 33], replication for fast databases requires a different set of tradeoffs, for two reasons. First, when logging for recovery, if the storage throughput becomes a bottleneck, storage performance can be easily upgraded. For instance, a 1 TB Intel SSD 750 Series PCIe card costing less than $1000 can provide 1 GB/s sequential write performance, which can sustain the logging requirements described above. Much cheaper SSDs can also provide similar performance in RAID configurations. In comparison, when logging for replication, if a network link becomes a bottleneck, especially high-speed leased lines, an upgrade typically has prohibitive costs and may not even be available.

Second, unlike recovery, which is performed offline after a crash, a backup needs to be able to catch up with the primary, or it directly impacts primary performance [1]. Databases perform frequent checkpointing, so the amount of data to recover is bounded. If the recovery mechanism doesn't scale with the primary, the consequence is a little more time for recovery. However, for replication, if the backup cannot sustain the primary throughput, then it will fall increasingly far behind and may not be able to catch up later.

Our goal is to perform database replication with minimal performance impact on the primary database. We aim to 1) reduce the logging traffic, and 2) perform replay on the backup efficiently so that the backup scales with the primary. To reduce network traffic, we propose using deterministic record-replay designed for replicating databases. The primary sends transaction inputs to the backup, which then replays the transactions deterministically. This approach reduces network traffic significantly because, as we show later, transaction inputs for OLTP workloads are much smaller than their output values.

Proceedings of the VLDB Endowment, Vol. 10, No. 13

For deterministic replay, we record and send the transaction write-set so that the backup can determine the records that were written by the transaction. On the backup, we use multi-versioning, so that writers can execute concurrently, and safely create new versions while readers are accessing the old versions. Our replay uses epoch-based processing, which allows both readers and writers to efficiently determine the correct versions to access, while allowing readers to execute concurrently with the writers. Together, these techniques allow highly concurrent and deterministic replay.

Our main contribution is a generic and scalable replay-based database replication mechanism. By decoupling our replication scheme from the primary database, we enable supporting different database designs. Our approach allows the primary to use any concurrency control scheme that supports total ordering of transactions, and imposes no restrictions on the programming model. In addition, the backup is designed so that it makes no assumptions about the workloads, data partitioning, or the load balancing mechanism on the primary. For example, to support different kinds of applications fast databases often make various design tradeoffs, such as data partitioning [14, 7] and a special programming model [8]. Our approach is designed to scale without relying on any specific primary database optimizations, and without requiring any developer effort for tuning the backup for these optimizations.

We have implemented replication for ERMIA [15], an in-memory database designed to support heterogeneous workloads. Our backup database is specifically designed and optimized for replaying transactions concurrently. Our experiments with TPC-C workloads show that our approach requires 15-20% of the network bandwidth required by traditional logging. The backup scales well, replaying transactions as fast as the primary, and the primary performance is comparable to its performance with traditional logging. An added reliability benefit of our generic replication mechanism is that it can be used to validate the concurrency control scheme on the primary. We found and helped fix several serious bugs in the ERMIA implementation that could lead to non-serializable schedules.

The rest of the paper describes our approach in detail. Section 2 provides motivation for our approach, and Section 3 describes related work in the area. Section 4 describes our multi-version replay strategy and the design of our system. Section 5 provides details about our implementation, and Section 6 describes experimental results that help evaluate our approach. Finally, Section 7 describes bugs found using our replay approach and Section 8 gives conclusions.

2. MOTIVATION

A recent incident at gitlab.com, a popular source code hosting site, illustrates the importance of fast and scalable database replication. On Jan 31, 2017, a spam robot created large numbers of read-write transactions on the production database [10]. Due to limited network bandwidth, the backup database lagged far behind the primary and eventually stopped working. The maintenance crew decided to wipe the backup database and set up a new backup instance from scratch. However, while setting up the backup instance, a team member logged into the wrong server and accidentally deleted the production database. The gitlab.com site had to shut down for more than 24 hours to recover from an offline backup and lost 6 hours of data.

While log shipping is commonly used for backup and failover, it raises several challenges in a production environment. A recent post by Uber describes these issues in detail [31]. First, log shipping incurs significant network traffic because it sends physical data, and this is expensive because backup machines are usually situated across buildings or data centers. Second, databases usually do not maintain storage compatibility between major releases. Because log shipping ships physical data, the backup and the primary database have to run exactly the same version of the database software, making database upgrades a challenge. Traditional databases usually provide an offline storage format converter [25], and both the primary and the backup have to shut down to perform a major upgrade. This significantly increases the downtime and maintenance complexity. Finally, if the primary database has bugs that can corrupt data, log shipping will propagate the corruption to the backup database as well.

Uber's current solution to these problems is to use MySQL's statement level replication [22]. This approach simplifies the upgrade process and saves some network traffic because the logging granularity is row level. However, this approach can still generate close to 10Gb/s for a fast database, and the re-execution needs to be performed serially. It also doesn't help with data corruption. Our replay-based approach is designed to address these issues.

3. RELATED WORK

In this section, we describe related work in the areas of database replication and deterministic replay schemes. Table 1 provides a summary comparing our scheme with various logging and replay schemes.

Primary-backup replication based on traditional log shipping is easy to implement and commonly available in many traditional databases such as SQL Server [20], DB2 [19], MySQL [22], and PostgreSQL [24], but it has significant logging requirements as discussed earlier. While it can be executed relatively efficiently, Hong et al. suggest that serially replaying the recovery log on the backup database can become a bottleneck with increasing cores on the primary database [13]. To enable concurrent log replay, their KuaFu system constructs a dependency graph on the backup, based on tracking write-write dependencies in the log. Then it uses topological ordering to concurrently apply the logs of non-conflicting transactions. KuaFu is implemented for a traditional MySQL database with a throughput on the order of 1000 transactions per second. Its dependency tracking and transaction ordering need to be performed serially, and so it is unclear whether the scheme will scale for high-throughput databases.

The rest of the schemes in Table 1 are designed for fast, in-memory databases. Calvin [28] uses deterministic locking and scheduling to guarantee deterministic database execution [27]. Incoming transactions are assigned a serial order and then replicas execute transactions concurrently using a deterministic concurrency control scheme that guarantees equivalence to the predetermined serial order. The deterministic scheme requires knowing the read and write sets of transactions. Compared to our approach, Calvin requires significantly more network bandwidth for logging the read set, especially when OLTP transactions perform scans. Our evaluation shows that our backup scales better than Calvin's deterministic locking scheme and single

Table 1: A comparison of logging, recovery and replay schemes

	Read Set Needed?	Write Set Needed?	Row Data Needed?	Multi-Version	Network BW Needs	Developer Effort	Scalability Bottlenecks on Backup
Traditional Log Shipping	No	Yes	Yes	No	High	N/A	Serial replay
Kuafu [13]	No	Yes	Yes	No	High	N/A	Dependency analysis; Frequent write-write dependencies
Calvin [28]	Yes	Yes	No	No	Depends on read-set	N/A	Deterministic locking
Bohm [7]	For performance	Yes	No	Link list	Low	Specify data partitioning	Either data or thread partitions imbalanced
Command Logging [17]	No	No	No	No	Low	Specify data partitioning	Data partitions imbalanced; Cross-partition transactions
Adaptive Logging [35]	No	Adaptive	Adaptive	No	High	Specify data partitioning	Same as above; Dependency analysis on primary
Pacman [33]	No	No	No	No	Low	Limited by static analysis	Data accesses are imbalanced
Our Approach	No	Yes	No	Array	Low	N/A	Wait on read dependency

version store. Furthermore, Calvin does not easily support serializable schemes that allow transactions to commit back in time [9, 32], as described later in Section 4.4.2.

Our work is closest to Bohm's multi-versioning scheme [7]. Bohm's placeholder approach for tracking tuple versions is similar to our approach, but it uses a linked list versus our array implementation which has a significant impact on performance, as we show in Section 6.4.2. The key difference between the two schemes is that Bohm partitions data while we partition transactions on the backup. To support its design, Bohm requires its log to contain transaction data in serial order. All partitions (cores) process the entire log while creating tuple versions. Bohm uses a set of concurrency control threads and execution threads that operate on batches of transactions.

While data partitioning works well for a primary, it is problematic for a backup that is designed to serve different primary databases. Consider a non-partitioned, high-performance primary such as ERMIA [15] or Silo [30] being served by a Bohm backup. First, a developer would need to decide how the keyspace should be partitioned on the backup. Improper data partitioning would lead to load imbalance because the concurrency control threads operate on a batch of transactions in lock-step. Second, unlike our scheme, the primary's log would need to be serialized, and processed by all cores. Third, Bohm requires the read set for optimizing its multi-versioning performance because of its linked-list based placeholder approach. Fourth, Bohm's performance depends on carefully tuning the number of threads for the concurrency control and the execution layers, with the optimal choice being non-trivial because it is workload dependent. Finally, the execution layer runs transactions on cores based on data dependencies. When a thread performs a read, if the version is not available, the transaction that produces the value is run by the same thread. However, it is unclear whether this execution model provides good locality when dependencies cross partitions. For example, 15% of the payment transactions in TPC-C access two warehouses.

Our approach sidesteps issues of determining partitioning on the backup by simply relying on the primary's transaction partitioning scheme to execute transactions on the same core on the backup as on the primary. This allows the backup to scale as well as the primary, for both non-partitioned or partitioned databases.

Malviya et al. show that logging in high-throughput databases imposes significant processing overheads [17]. They propose recovering databases using command logging, and show that their approach improves database performance by 50%. Command logging logs the transaction inputs and replays the transactions deterministically for recovery. In their VoltDB distributed database, command logs of different nodes are merged at the master node during recovery. To guarantee correctness during recovery, transactions are processed in serial order based on their timestamps, and significant synchronization is needed for cross-partition transactions. As a result, a pure command logging approach on the backup will not scale with primary performance, as shown by the adaptive logging work [35].

Adaptive logging improves recovery for command logging by generating a graph based on data dependencies between transactions. Their focus is on single node failure in a distributed database. When a node fails, all the transactions on the failed node and any transactions on which it depends on other nodes are replayed concurrently, in topological order, similar to Kuafu [13]. Adaptive logging reduces recovery time further by logging rows for some transactions, thus avoiding serial replay of long chains of transactions. Adaptive logging is designed for recovery and thus generates its dependency graph offline. Unfortunately, the paper does not evaluate how long it takes to generate this graph, whether it can be performed scalably, and the size of the graph. For replication, this graph would need to be generated online, impacting primary performance, and sent to the backup, requiring significant network bandwidth.

These command logging approaches are designed for partitioned databases, making them unsuitable for a backup, as described earlier. Our design uses command logging style deterministic replay, and although it requires sending the write set, it enables the replay to be performed concurrently without requiring dependency analysis.

Pacman [33] proposes a novel method for parallelizing recovery for in-memory databases that use command logging. They use a combination of static and dynamic analysis to parallelize replaying of transactions during recovery and do

not depend on partitioned-data parallelism. The static analysis performs intra- and inter-procedure analysis to create a global dependency graph based on coarse-grained table-level dependencies. The dynamic analysis performs fine-grained dependency analysis for nodes in the global graph, allowing concurrent execution of pieces of code with no data dependencies. This approach can be applied for a backup and we believe its dynamic analysis can scale well for replication. However, its static analysis assumes that the transaction logic is simple enough to determine the read and write sets easily, limiting the programming model. It also assumes that all the stored procedures are known in advance, complicating the deployment of new stored procedures.

Silo [30] uses a variant of optimistic concurrency control (OCC), with a novel, decentralized transaction ID (TID) assignment protocol that avoids scaling bottlenecks during commit, and enables a concurrent recovery scheme in SiloR [36]. However, SiloR uses value logging, which generates a GB per second of log data, requiring high-bandwidth storage. Silo's TID assignment makes it hard to use command logging without logging the read-set as well. To support Silo and hardware transactional memory databases using a similar TID assignment scheme [34], we would need to use a global counter to determine the serial order of transactions. However, our experiments with up to 32 cores in Section 6.3 show that such a counter is not a scaling bottleneck for OLTP workloads.

Write-behind logging [1] is an efficient logging and recovery scheme for in-memory multi-version databases that use NVM for durable storage. This logging scheme logs the timestamp ranges of uncommitted transactions, requiring minimal logging and it provides very fast recovery because it reuses the multi-versioned data in the database. However, since the log only contains timestamps and not the data, replication still requires traditional row-level logging.

Many recent in-memory databases use multi-versioning, with an emphasis on providing serializability as efficiently as single-versioned designs [21, 15]. Our backup scheme knows the serialization order and thus requires much simpler synchronization. However, it requires an initialization phase for each batch of transactions that needs to be performed efficiently, so that the backup can keep up with a scalable primary. Furthermore, in traditional MVCC systems, readers tend to access the latest versions, while in our system, readers frequently access older versions in order to guarantee that they can read identical data, motivating our binary search based multi-versioning.

4. MULTI-VERSION REPLAY

We now describe our approach for replicating fast databases. Our aim is to replay transactions on the backup concurrently, while requiring low network bandwidth. We begin by estimating the network bandwidth that can be saved by our record-replay method. Next we describe our transaction model, followed by an overview of our approach. Finally we provide more details about the design of the primary and the backup databases.

4.1 Record-Replay

Our design is motivated by the observation that for many OLTP workloads, the transaction input size is smaller than the output size. As a result, replicating transaction inputs to replay the transactions will reduce network traffic compared to replicating the transaction outputs [17].

Table 2: Input Size versus Output Size in TPC-C

	Input Size	Output Size (Row Level)	Output Size (Cell Level)
NewOrder[a]	200 bytes	996 bytes	903 bytes
Delivery	12 bytes	741 bytes[b]	356 bytes
Payment	32 bytes	346 bytes	118 bytes

[a]We assume that the customer orders 15 items, the maximum allowed for a NewOrder transaction.
[b]This output includes the `c_data` column in the Customer Table, which is a `VARCHAR(500)` type. We estimate its length to be around 30.

To estimate the likely benefits of our approach, we consider the TPC-C benchmark, designed to be representative of common OLTP applications. TPC-C uses several stored procedures to simulate an online-shopping workload. It contains three read-write transactions. Table 2 shows the input and output size of these transactions. For example, in the TPC-C NewOrder transaction, the input parameters are the customer and the products the customer would like to purchase, and the outputs are the rows updated by the NewOrder transaction.

The input size is the total size of the parameters of the stored procedures. We estimate the output size based on commonly used row-level logging (Column 3), as well as cell-level logging (Column 4).

We observe that the input parameter size for OLTP transactions is significantly smaller than the output size. This suggests that our replay-based approach can save network traffic by replicating the transaction inputs and re-executing transactions on the backup. As a side effect of reducing network traffic, our approach can help reduce the recent modifications that may be lost after a failure.

4.2 Transaction Model

We assume that all transaction inputs are available at the start of a transaction and requests do not require any further client input. This avoids long latencies due to client communication during the transaction, which can raise abort rates. This model is commonly used in fast OLTP databases [30].

For replaying transactions deterministically, we need to handle two sources of non-determinism in transactions. The first is non-deterministic database functions such as `NOW()` and `RAND()`. We capture the return values of these function calls on the primary and pass them as input parameters to the transaction during replay. While our current system does not perform this record-replay automatically, the functionality can be implemented at the SQL layer or the C library layer [11], or the system call layer [26], without requiring any changes to the transaction processing code. SQL queries may also return non-deterministic results such as performing a read or write based on the results from "Select count(*) FROM Table T1". Unlike OLAP workloads, OLTP workloads generally do not use such queries to avoid returning non-deterministic results. However, to support them, the developer could add order clauses to ensure determinism, or we can revert to row-level logging for such queries. All command logging based schemes must handle these issues.

The second source of non-determinism arises due to concurrently reading and writing data to the database, which presents challenges for concurrent recording and replay, similar to the challenges with replaying other concurrent programs [11] and systems [6].

We assume that data integrity is maintained by executing transactions serializably, so that the outcome of the concurrent execution is the same as the transactions executing in some total order, or serial order. Serializability is commonly supported by fast, in-memory databases [5, 30, 15, 34].

4.3 Overview of Approach

Serializability regulates non-determinism, making it simple to replay transactions correctly. For example, command logging can be used to replay transactions deterministically in a serial order. This serial replay scheme requires minimal network traffic because the primary only needs to send the input parameters and the serial order of the transactions. However, replaying transactions in serial order will not scale, making the backup a bottleneck.

An alternative replay scheme is to send the read-set for each transaction (the data for all records read by the transaction). With this data, transactions can be replayed concurrently because read values are available and reads do not need to be synchronized with writes. A similar approach is used to replay concurrent programs in multi-core systems [6]. However, OLTP transactions have a high read-write ratio and issue large scans, and so read-set based replay will generate significant network traffic.

Our multi-version replay approach lies in between, generating logs that are similar in size to command logging, while allowing concurrent and scalable replay similar to read-set based replay. Besides the data needed for serial replay, we also send the write-set of each transaction to the backup. The write-set contains the keys of all the records written by the transaction, but not the row data. The write-set, together with the serial order, allows transactions to infer the correct data to read during concurrent replay. This approach has lower logging requirements than value logging because keys are generally smaller than row data.

On the backup, we use two techniques to replay transactions concurrently. First, we use multi-versioning, so that transactions can read and write different versions of rows concurrently. In particular, writers can create new row versions safely while readers are accessing old versions.

Multi-versioning, however, is not sufficient for performing replay concurrently. During replay, when a transaction reads a row, it doesn't know the specific version to read because we don't send read-sets. Instead, the transaction needs to read the version that was written by a previous transaction that last updated this row. However, with concurrent replay, this transaction may still be running, and so the required version may not exist yet. Even if the version exists, we don't know whether it is the correct version until all previous transactions have been replayed in serial order. Thus replay must still be performed serially.

We need a mechanism to identify the version to read while allowing concurrent replay. If we know about all the possible versions of a record before starting replay, we can infer the correct version to read based on the serial order of transactions. Based on this insight, we replay transactions in epochs, so that all versions of a row that will be created in an epoch are known in advance. This allows the reads of a transaction to select the correct version, without waiting for all previous transactions to complete in an epoch. The result is that replay can proceed concurrently, while requiring synchronization for true read-after-write dependencies only.

T_1		Write A (A1)	Commit	
T_2	Read A (A0)		Write C	Commit

Figure 1: An Example of Commit Back In Time

4.4 Recording on the Primary

On the primary, we record the inputs and the write-set of transactions, which is relatively simple. In addition, we need to determine the serial order of transactions and batch transactions in epochs. Next, we describe these two operations in more detail.

4.4.1 Determining the Serial Order

Our approach allows the *primary* database to use any concurrency control mechanism that ensures serializability. The primary may be a single or multi-versioned store, and it may or may not partition data. We assign a *serial id* to each transaction, representing its serial order, and send this serial id with the rest of the transaction information to the backup. The serial id is closely tied to the concurrency control scheme implemented by the database. For example, the traditional methods for implementing serializability are two-phase locking and optimistic concurrency control [16, 30]. For both, when the transaction commits, we use a global timestamp (often maintained by the database implementation) to assign the serial id of the transaction.

Multi-version databases often implement more advanced concurrency control schemes such as Serializable Snapshot Isolation (SSI) [9] and Serial Safety Net (SSN) [32]. Obtaining the serial id is more involved in these schemes because they allow transactions to commit "back in time".

Consider the example shown in Figure 1: T_1 writes a new version of A, while T_2 reads A and writes C concurrently. T_2 reads A before T_1 commits and so it reads A_0, the old version of A. Then T_1 writes to A and commits, creating A_1, a new version of A. Finally, T_2 writes to C and commits.

Under two-phase locking, T_2 would hold the lock on A and so T_1 would not have been able to proceed with writing A until T_2 had committed. This schedule would not be valid under optimistic concurrency control either, because T_2's read set has been overwritten, and so T_2 would be aborted.

However in multi-version protocols like SSI and SSN, T_1 and T_2 can commit successfully. In fact, this concurrent interleaving is serializable, and the serial order is T_2, T_1. In this case, we say that T_2 commits back in time, before T_1. Later, in Section 5, we describe how we obtain the serial id for the SSI and the SSN protocols in the ERMIA database.

4.4.2 Batching Transactions in Epochs

As mentioned in Section 4.3, we need to batch transactions into epochs so that the backup can replay the transactions within an epoch concurrently. An epoch must satisfy the constraint that any data read in the epoch must have been written before the end of the epoch. In other words, transactions should never read data from future epochs, or else epoch-based replay could not be performed correctly.

The simplest method for implementing epochs is to batch transactions in actual commit order. Since transactions are serializable, this order satisfies our epoch constraint. Epochs

based on commit order work correctly even when transactions can commit back in time in our multi-versioned backup database. A transaction in a later epoch may need to read an older version of a data item, but a transaction never needs to read from a future epoch. Consider the example in Figure 1 again. Suppose T_1 is batched in an epoch and T_2 is batched in the next epoch. T_1 will create version A_1 during replay, but when T_2 is replayed in the next epoch, it will still read the previous version, A_0, because its serial order is before T_1. The benefit of using the commit order for defining epoch boundaries is that we do not need to wait for transactions to commit back in time. Note that a single-versioned backup design, e.g., based on Calvin [28], cannot easily support such back in time commits across epochs.

It may appear that epochs are expensive to implement but they can be defined more flexibly. In practice, databases often use epochs for various purposes such as recovery or resource management. We can reuse the same epochs for batching transactions. For example, ERMIA supports snapshot isolation and uses epochs based on the start times of transactions, which allows more efficient garbage collection of dead versions [15]. Our implementation for ERMIA uses the same epochs because snapshot isolation guarantees that reads see the last committed values at the time the transaction started, hence satisfying our epoch constraint.

4.5 Replaying on the Backup

We designed our backup database from scratch because there are several differences between replay and normal database execution. First, non-deterministic functions need to be handled by replaying the return values of these functions recorded at the primary. Second, we require an initialization phase at each epoch so that transactions in the epoch can be replayed correctly. Third, we replay committed transactions only, and thus require a simple synchronization scheme and no machinery for handling aborts or deadlocks.

To simplify the design of a failover scheme, we implemented the backup database with a structure similar to a fast, in-memory database, such as Silo [30] and ERMIA [15]. We use a B-Tree index to represent a table in the database, with the keys of the B-Tree representing the primary keys of the table. Like many multi-version databases, the values in the B-Tree are pointers to indirect objects that maintain the different versions of the rows. In our case, this object is an array, with each entry containing a version number and a row. The version number is the serial id of the transaction that created this row version. It not only indicates the transaction that created this version but also the serial order in which the version was created. The array is kept sorted by the version number.

4.5.1 Replay Initialization

We need to first perform an initialization step before replaying an epoch. For each transaction in the epoch, we process each key in its write-set by creating a new row version. The version number of the row is the transaction's serial id, and its value is set to an empty value, indicating the value has not yet been produced. This empty value will be overwritten by actual row data when this transaction is replayed later. New row versions are added to the version array using insertion sort to maintain sorted order.

We perform the initialization step concurrently on the backup. This step scales for three reasons. First, updates

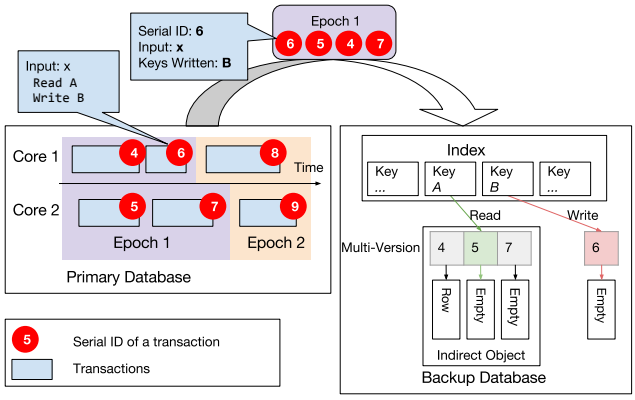

Figure 2: An Example of Replaying a Transaction

are more frequent than inserts in OLTP transactions. Since the values in the B-Tree leaf nodes point to indirect objects, while inserting new keys can modify the B-Tree index, updating existing keys does not modify the B-Tree. As a result, concurrent initialization will generate a read-mostly workload on the B-Tree index, and existing B-Tree implementations scale well under a read-mostly workload [18]. Second, write-write conflict rates are low in OLTP transactions, and so acquiring locks on the indirect objects should not cause significant contention. Third, we send transaction data from each primary core to a separate backup core to minimize contention on the sending or receiving path. This data is mostly sorted by serial id. As a result, when new versions are inserted in the version array, they are mostly appended, and insertion sort is efficient in this case.

4.5.2 Replay

After the initialization step, we replay the transactions in the epoch concurrently. Any read during replay will see all the possible row versions created during the epoch. When a transaction with a serial id t reads a key, we simply read from the greatest version that is smaller than t, i.e., the previous version. We use binary search to find this correct version in the version array. If this version contains an empty row, we need to wait until the actual row data is written (by the transaction that creates this version).

Figure 2 shows an example of replaying a transaction (Transaction 6) at the backup. The figure shows that transactions with serial id 4-7 are batched in Epoch 1. Transaction 6 has an input parameter x, and it reads key A and writes key B. Transactions 4, 5 and 7 write to key A.

On the backup, the Index represents a table or an index to the table. We initialize replay for Epoch 1 by creating versions for keys A and B. Each key points to an indirect object containing a sorted array of row versions. For epoch 1, we initialize versions 4, 5 and 7 for key A, and version 6 for key B. When Transaction 6 is replayed, it reads version 5 of key A (the largest version smaller than 6). At this point, this version is still an empty row, and so Transaction 6 will wait until Transaction 5 updates this version. During replay, Transaction 6 will write to version 6 of key B.

One corner case is that a transaction can read its own writes. On the first read, the replay will correctly read the previous version. However, after a write, we can no longer read the previous version. We resolve this issue with a per-transaction write buffer. On a read, the transaction first searches the buffer for the updates it has made before search-

ing the database. When the transaction commits, it releases the writes from the buffer into the database and frees the entire buffer. Since OLTP transactions are read-dominated, the overhead of using the write buffer is small.

We do not need to grab any lock or perform any synchronization, other than waiting for empty row versions, during replay. This avoids the significant overheads associated with performing concurrency control [12]. Since we only replay committed transactions, there are no aborts, and the replay is deadlock free.

4.5.3 Garbage Collection

We perform garbage collection of old versions of rows at epoch boundaries, when we know that replay is not running and the old versions are not being accessed concurrently. At replay initialization, we only perform garbage collection for keys that are overwritten in the epoch. For each such key, our garbage collector requires that the previous versions of these keys will be read from only the last epoch in which they were written. Thus it keeps all the versions of that last epoch, and reclaims all previous versions from preceding epochs. The garbage collector's requirement is met by ERMIA because it only creates new epochs when all cores have entered the current epoch. For example, it will only create Epoch N when all transactions in Epoch $N-2$ have committed. As a result, transactions in epoch N will not read a version created in epoch $N-2$, if there is an update in epoch $N-1$. When we replay epoch N, we need to keep all the versions of epoch $N-1$ but can reclaim previous versions.

5. IMPLEMENTATION

Our backup database is designed to work with any primary database that provides serializability. To stress our replay server, we chose to use ERMIA [15], a fast in-memory database, as the primary database. ERMIA performs well under heterogeneous workloads such TPC-E and TPC-C+ in which OLTP transactions run together with some long running transactions. ERMIA also achieves high throughput on traditional OLTP workloads like TPC-C. Next, we describe our replay implementation for ERMIA.

5.1 Recording on the Primary

The TPC-C and TPC-C+ benchmarks implemented in ERMIA are not fully deterministic. For benchmarking purposes, these transactions generate random data for the database contents. In practice, these random contents would be specified by the end user, as inputs to transactions. We modified the TPC-C and TPC-C+ benchmark in ERMIA to pass the randomly generated content as input to the transactions, making the benchmark logic deterministic.

5.1.1 Obtaining Serial Order

ERMIA implements two serializability-enforcing concurrency control protocols, SSI [9] and SSN [32]. Both use timestamps to disallow certain transaction interleavings that may cause serializability violations, and they allow "back in time" commits for additional concurrency. Our prototype supports obtaining the serial order for both SSI and SSN. Next, we describe how the SSI protocol works, and then how we derive the transaction serial order from the protocol.

In SSI, similar to OCC, the transaction checks if its read set has been overwritten before commit. If not, it acquires a commit timestamp and assigns it to all the tuples in its write set and then commits. If its read set has been overwritten, then a write skew is possible, making it dangerous to commit the transaction [2]. SSI determines whether the transaction can be committed without violating serializability by looking for dangerous structures, in which two adjacent rw-dependency edges occur between concurrent transactions [9]. ERMIA implements SSI by first determining the minimum commit timestamp among all the overwriters of this transaction.

We call this timestamp the `skew_timestamp`. Next, the transaction checks the previous versions of the tuples in its write set. For these tuples, we calculate the maximum of their commit timestamps and their readers' commit timestamps[1], and we refer to this timestamp as the `predecessor_stamp`. If the `predecessor_stamp` is larger than or equal to the `skew_timestamp`, the transaction aborts, otherwise it can commit at the `skew_timestamp`, i.e., commit back in time.

While the SSI concurrency control protocol is complicated, calculating the serial id is relatively simple. When a transaction commits back in time, we assign its serial id to be 2 × `skew_timestamp` - 1. Otherwise, the serial id is assigned as 2 × `commit_id` of the transaction. This assignment ensures that the serial id of a transaction that commits back in time is below the skew timestamp.

Unlike SSI, SSN allows nested commit back in time. For example, with three transactions T1, T2 and T3, T2 can commit back before T1, and T3 can commit back before T2. The SSN serial order needs to handle these nested commits, but is otherwise obtained in a similar way to SSI, and is not described here.

5.1.2 Batching Transactions in Epochs

ERMIA uses an epoch manager for tracking various time lines, including for garbage collection of dead versions, and RCU-style memory management. We reuse these epochs to batch transactions. In each epoch, transactions are committed on different cores. We create a TCP connection for each core, and send the transactions committed on that core to a specific core on the backup database, where those transactions are replayed. This allows the backup to perform the replay initialization on different cores concurrently with minimal contention.

We use one worker thread per core for processing an epoch. Each epoch consists of three stages: fetch, initialization and execution. Each worker performs all these stages sequentially and the workers execute concurrently. During fetch, the worker fetches the data from the network socket and parses the packet. During initialization, empty values are inserted for transactions using their serial id. Last, transactions execute deterministically.

We allow the fetch and the initialization phase of the current epoch, and the execution phase of the current epoch and the fetch phase of the next epoch to run concurrently. However, the initialization stage and the execution stage do not overlap in time (within or across epochs). As a result, during execution, the B-Tree index and the version arrays

[1]Since keeping track of all commit timestamps of readers is expensive, ERMIA only tracks which core has read the data and the latest commit timestamp on that core, which may cause false positives when aborting a transaction.

are read-only, and thus do not need any synchronization. We use a single control thread to synchronize the stages.

5.2 Replaying on the Backup

Our prototype system uses Masstree [18] as the B-Tree index because it scales well for read-mostly workloads. We need to acquire a lock on the indirect object associated with a key when updating the key during replay initialization. Since this lock is unlikely to be contended, as explained in Section 4.5.1, we use a spinlock for its implementation.

During replay, the B-Tree indices and the version arrays are read but their structures are not updated. The only synchronization is while waiting for empty rows. To make the wait efficient, we replay transactions using co-routines. When a transaction needs to wait, it performs a user-space context switch to the next transaction in the run queue, so threads make progress even when a transaction needs to wait.

5.3 Failover

When the primary database fails, database requests need to be redirected to the backup database, so that it can take over from the primary. Since our backup database is replay-based, and its internal multi-version storage structures are different from the primary database, we need to migrate the database to a new primary.

Our prototype can export the backup database into an ERMIA checkpoint image and start ERMIA from the image. We scan the tables and indices and export them in ERMIA's checkpoint format. Our current implementation is single-threaded due to limitations in ERMIA's checkpoint format, thus it takes roughly two minutes to export a 32 warehouse TPC-C workload. This time can be significantly reduced by using concurrent checkpointing methods [36].

A more efficient failover scheme can convert data structures in memory. In our current prototype, all the row data storage format and indices are compatible with ERMIA, with the multi-version storage structures needing conversion. To implement this feature efficiently, our replay engine could be a plugin for ERMIA, making it aware of our storage structure, allowing it to convert them lazily.

Our failover functionality can also be used to ease the process of upgrading primary database software. Upgrading of databases is often a cumbersome process that needs to be performed offline [25], with little support for streaming replication [4]. Our replay-based replication can export the backup to the new storage format asynchronously, and then minimize downtime by performing a controlled shutdown of the primary and replay-based failover to the backup.

5.4 Read-Only Transactions

Since our backup database is multi-versioned, it can provide consistent snapshots to read-only transactions. The simplest option is to serve snapshots at epoch boundaries by executing these transactions in the same way as we replay transactions. To reduce staleness, we can also track the serial id before which all transactions have been replayed, and assign this serial id to the read-only transaction. This serial id can be maintained scalably by using per-core counters.

Currently, we do not support read-only transactions because our garbage collector is unaware of them. To support read-only transactions, we would need to implement a second epoch timeline that tracks read-only transactions and thus can be used to reclaim versions safely.

6. EVALUATION

In this section, we evaluate the performance of our replication method. We use ERMIA as the primary database. We compare our approach against a baseline ERMIA (no log shipping), ERMIA with its log shipping implementation, and our Calvin implementation on the backup [28].

ERMIA uses row-level logging, which consumes less network traffic than traditional ARIES style page-level logging. All our experiments are performed with ERMIA using the SSI concurrency control protocol. We also performed experiments using the SSN protocol; the numbers were similar and are not presented here.

The Calvin primary ships both the read-set and the write-set keys. Our Calvin backup implementation closely mirrors the original design. However, unlike the original implementation that uses a centralized lock manager for deterministic locking during initialization, we reuse the version arrays in our backup to implement a per-tuple lock queue that allows concurrent initialization. As described in Section 4.4.2, Calvin is unable to handle back in time commits across epochs, and our implementation ignores these errors.

Our evaluation uses two metrics. First, we compare the network traffic of the different methods. Second, we measure the throughput and scalability on the primary and the backup databases. We use two network configurations in our log shipping and replay experiments. For most experiments, we use a 10 Gb/s Ethernet network so that the network is not a bottleneck. We also show the impact on the primary when using a slower 1 Gb/s network.

We execute the ERMIA primary database and our backup database on machines with the same hardware configuration: 4-socket Intel Xeon CPU E5-2650 (32 cores total) and 512 GB DRAM. Data sets for all workloads fit into DRAM. Both databases run on Linux 3.10 with glibc 2.17. ERMIA is compiled with gcc 4.8.5 and our backup database is compiled with clang 3.8, both with the `-Ofast` optimization.

6.1 Workloads

We use 4 kinds of workloads in our experiments: TPC-C, TPC-C+, TPC-C Spread and TPC-C+ Spread. TPC-C is a typical OLTP workload that simulates an E-Commerce workload: customers query and purchase items online, and items are delivered from warehouses to customers. There are three types of read-write transactions in TPC-C: `NewOrder`, `Delivery`, and `Payment`. These read-write transactions constitute 92% of the total transactions; the remaining 8% are read-only transactions.

TPC-C+ [3] is designed to evaluate heterogeneous (non-pure OLTP) workloads. It is similar to TPC-C, but it adds `CreditCheck`, a type of analytic transaction. This transaction scans the customer's order history and account balance to determine the customer's credit level, and it takes significantly longer to run than other TPC-C transactions.

By default, ERMIA partitions the database by warehouse id and associates a worker thread with each warehouse. Since most TPC-C/TPC-C+ transactions tend to operate on a single warehouse, ERMIA runs a transaction on the worker thread serving its warehouse. To evaluate how our approach will perform when data cannot be easily partitioned, we disable this thread pinning in ERMIA by letting transactions

Table 3: Network Traffic

	Record	Log Shipping	Calvin	Calvin Optimized
TPC-C	6.988 GB	35.739 GB (5.11x)	8.445 GB (1.21x)	7.257 GB (1.04x)
TPC-C+	5.081 GB	27.398 GB (5.39x)	12.789 GB (2.52x)	11.969 GB (2.36x)
TPC-C Spread	3.600 GB	24.481 GB (6.80x)	4.730 GB (1.31x)	3.814 GB (1.06x)
TPC-C+ Spread	2.832 GB	19.120 GB (6.75x)	8.157 GB (2.88x)	7.486 GB (2.64x)

run on arbitrary worker threads. We call these the "spread" workloads: TPC-C Spread and TPC-C+ Spread.

During replay, we avoid second-guessing the data partitioning policy. We replay the transaction on the same backup core as the transaction was run on the primary, which helps preserve the partitioning policy of the primary.

6.2 Network Traffic

In this section, we measure the network traffic using the 10 Gb/s network so that it is not a bottleneck. We use 16 cores on the primary database, because with 32 cores, log shipping can generate traffic close to 10 Gb/s.

Table 3 shows the network traffic generated by each workload in 60 seconds. Compared to our approach, log shipping requires 5x more network bandwidth for the TPC-C and TPC-C+ workloads, and 7x more bandwidth for the spread workloads. Our bandwidth savings are higher for the spread workloads because they commit fewer `NewOrder` transactions (due to higher abort rates). We save more network traffic on the other transactions types, which make up a larger share of the committed transactions.

For the Calvin backup, the primary sends read-sets as well as write-sets to the backup. This requires 1.2-2.9x network traffic compared to our approach because some transactions issue long table scans. Most of these scans are issued on read-only tables and thus read locks are not needed for these rows during replay. To measure the benefits of optimizing for read-only tables, we modify the Calvin primary so that it specifically does not send the read-set keys for these tables.

With this optimization, the network traffic for Calvin is close to our approach for the TPC-C and TPC-C Spread workloads, but still incurs much higher network traffic on the TPC-C+ workload. For TPC-C, most of the records being read are updated in the transaction, and so the read-set is mostly covered by the write-set. However for TPC-C+, the `CreditCheck` transaction issues scans on many read-write tables, and sending the read-set for these scans consumes significant network traffic. By default, the `CreditCheck` transactions are 4% of all the issued transactions. Increasing this ratio will increase the network traffic as well.

We also evaluate statement level logging in MySQL [22] using the TPC-C implementation from Percona Lab [23]. We configure the benchmark with just 1 warehouse and run the workload for 60 seconds. MySQL is able to commit 7078 transactions and generates a 22MB replication log. With ERMIA generating 700K transactions/s, MySQL would require 20x network traffic compared to our approach.

6.3 Primary Performance

In this section, we measure the performance impact on the primary database for recording transaction inputs, the write-set keys and sending them to the backup. For the baseline, we use default ERMIA (no log shipping). We also show the performance of ERMIA with log shipping. To avoid any performance bottlenecks on the backup, we discard any packets that it receives during this experiment. We warm up

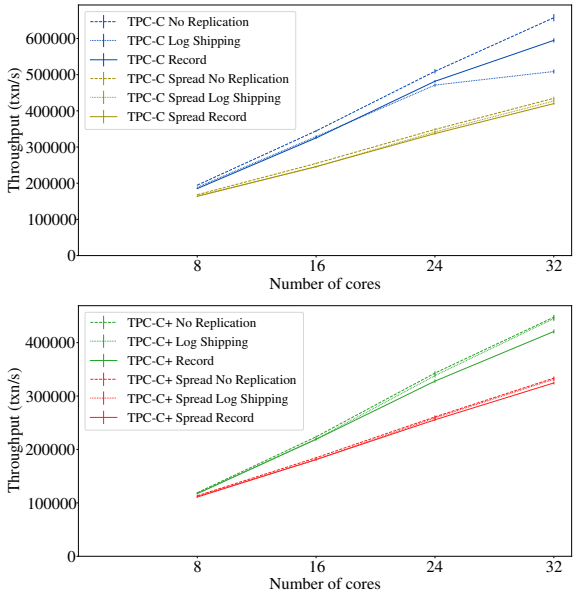

Figure 3: Primary Performance

ERMIA by running each workload for 30 seconds and then we measure the average throughput for the next 30 seconds.

Fast databases are designed to scale with the number of cores. Thus, we show the throughput of the workloads with increasing numbers of cores (from 8 to 32). Figure 3 shows that the throughput of our recording approach is close to the performance of the original database and scales with the number of cores. While the throughput slows down slightly per core, it has no visible scalability impact. Although our approach requires global serial ordering, it has minimal effect under transactional workloads.

Log shipping also performs similarly to the original database, but the 10 Gb/s becomes a bottleneck for the TPC-C workload at 32 cores. Both our approach and log shipping send data in a background thread. Although log shipping sends large amounts of data, it has only about 2% overhead when the network is not the bottleneck. Our prototype has slightly more overhead (3-4%), because it requires copying the transaction input parameters and the write-set keys in the commit path.

6.3.1 *Performance Over Slow Network*

Table 3 shows that replicating fast databases consumes significant network traffic. If the network is slow, then replication will have a significant impact on primary performance. In this section, we compare log shipping and our approach using a 1Gb/s network. Though expensive, a 1Gb/s link is practical in the wide area. Depending on the region and the ISP, the estimated cost of such a wide area link is roughly between $100K-500K per year. Both ERMIA's log shipping and our approach use a 512MB send

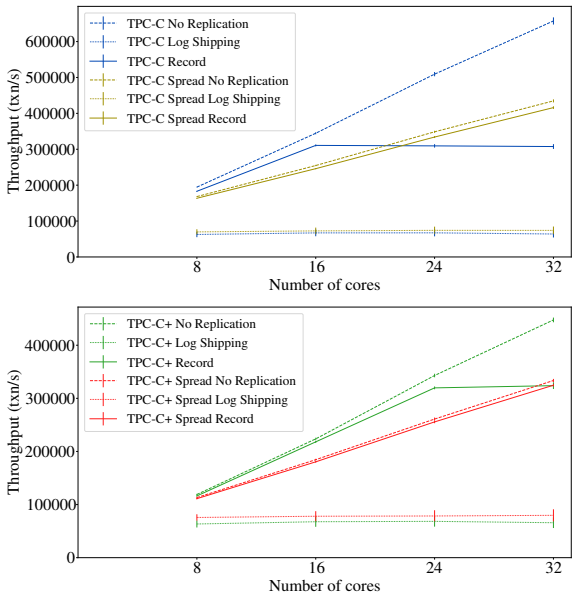

Figure 4: Primary Performance, 1 Gb/s Network

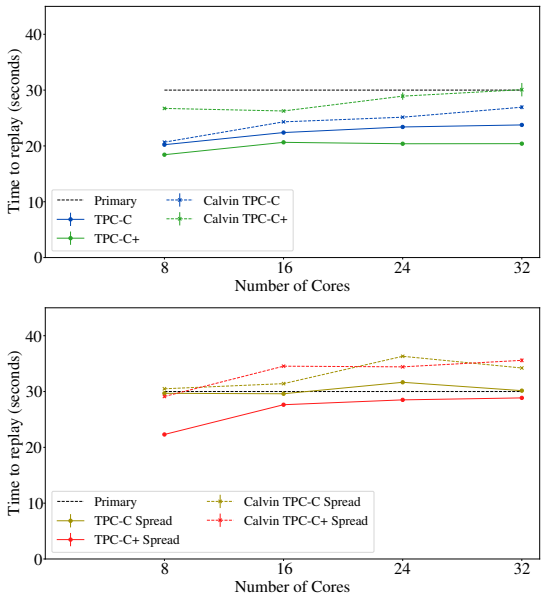

Figure 5: Concurrent Replay Performance

buffer. If the network is slow, the send buffer will fill up and stall the primary.

Figure 4 shows the primary throughput with increasing number of cores. Our approach can sustain the primary throughput up to 16 cores for TPC-C and 24 cores for TPC-C+. In contrast, log shipping performs much worse than our approach. For the Spread workloads, our approach scales until 32 cores, while log shipping still performs poorly.

6.4 Replay Performance

In this section, we measure the performance of our concurrent replay method on the backup server. To avoid performance artifacts caused by the primary, we collect the packet traces generated by the primary, and send these traces to the backup on the 10 Gb/s network.

We evaluate backup performance by measuring the replay time for a 30 second trace.[2] If the backup can finish replaying within 30 seconds, then it will not be a bottleneck on primary performance. We also compare with Calvin's deterministic execution scheme to show the benefits of using multiple versions. Figure 5 shows the replay time on the backup with increasing numbers of cores. For each data point, the primary and the backup use the same number of cores. Our numbers are marked using solid lines and Calvin numbers are marked using dashed lines.

Our approach is able to replay the trace within 30 seconds under all 4 workloads, except TPC-C Spread at 24 cores. For TPC-C and TPC-C+, our approach takes roughly 18-23s. The spread workloads represent a worst-case scenario with no data locality and much higher contention. Even in this extreme setup, our approach can replay the trace in 29-32s.

Calvin shows slightly worse performance (5-10% overhead) than our approach with the TPC-C and TPC-C spread workloads. As mentioned in Section 6.2, TPC-C transactions update most of the records they read. This type of access pattern does not benefit from multiversioning during replay, since most versions will be read only once before being updated. For TPC-C+ workloads however, the `CreditCheck` transaction performs many reads without updating these values. As a result, a single-versioned system like Calvin suffers from write-after-read dependencies, leading to higher overheads for TPC-C+ (up to 50%) compared to our approach. For the challenging spread workloads, Calvin consistently fails to complete the replay within 30s.

6.4.1 *Epoch Length*

Our approach replicates transactions at epoch granularity. Thus, a failure on the primary risks losing transactions in the current epoch. The epoch length represents a trade-off between performance and data loss, since shorter epochs lose less data but may have a performance impact on the primary and the backup. We use ERMIA's epoch manager to implement our epochs; our default epoch length is 1 second.

We measure throughput on the primary and the backup while varying the epoch length. We use the TPC-C and TPC-C Spread workloads with 32 cores because this setup is performance sensitive and should show the largest impact. We find that ERMIA's performance is relatively stable when the epoch length is larger than 100 ms. At 50ms epoch length, the primary throughput decreases by roughly 15% due to the cost of epoch processing.

We also measured the replay time with different epoch lengths, and found that it is similar to the numbers shown in Figure 5. This suggests that the cost of epoch processing has a more significant impact on the primary than on the backup.

6.4.2 *Version Array Vs. Linked List*

Existing multi-version databases use a linked list for tracking versions because it allows scalable, lock-free access (e.g., removal during aborts). However, this imposes a non-trivial cost for traversing pointers when the linked list is long [7].

As described in Section 4.5, we track row versions using a sorted array, which allows using binary search for finding a given version. In our case, the array is created during

[2]Similar to previous experiments, this trace is captured after the primary has been warmed up for 30 seconds.

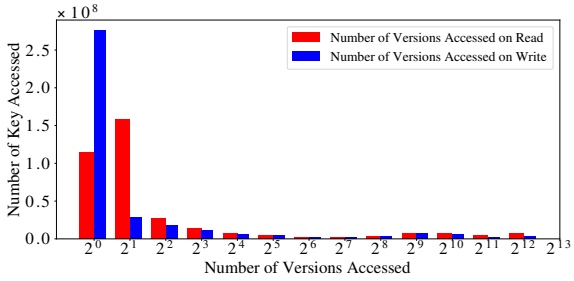

Figure 6: Link List Versions

replay initialization, and can be accessed lock free during replay. We also do not need to handle aborts.

Here, we evaluate the cost of using a linked list implementation by measuring the replay time for the 30-second TPC-C and TPC-Spread traces with different epoch lengths. With a 50 ms epoch size, the TPC-C replay time is 30.3 seconds (backup about to keep up), and the TPC-C Spread replay time is 29.4 seconds (backup can keep up). When the epoch size is 100 ms or more, the replay cannot keep up with the primary. The slowdown (vs. using an array) for TPC-C ranges from 24% (100 ms epoch) to 2.9x (750 ms epoch). The slowdown for TPC-C Spread ranges from 71.5% (100 ms epoch) to 5.3x (750 ms epoch). However, since epoch lengths below 100 ms impact primary performance, and epoch lengths above 100 ms impact backup performance, we conclude that version arrays are essential for ensuring that the backup can keep up with peak primary throughput.

The main reason for the slowdown is update hotspots [29] in the TPC-C family. Figure 6 shows a histogram of the number of key versions that are looked up in the linked list when a key is accessed (note that X axis has log scale). While most accesses are cheap (1 or 2 versions), a significant number of accesses skip over 4000 versions, because some keys are updated frequently in TPC-C resulting in long linked lists within an epoch. These accesses lead to cascading slowdown for dependent transactions.

6.4.3 Memory Overhead

In this section, we measure the memory overhead imposed by multi-versioning. When the backup database finishes replay and starts generating the ERMIA checkpoint file, we collect the distribution of the number of versions for all keys. The total number of keys in the database is roughly 110M and the total number of versions is 164M. Thus the memory overhead of multiversioning is roughly 48%. We find that most keys (99.7%) only have 1 or 2 versions, however the update hotspots in TPC-C lead to some keys having high numbers of versions (e.g., 352 keys had 2048-65536 versions).

7. BUGS FOUND IN ERMIA

Our replay scheme is mainly designed for replication, but it can also be used to catch corruption bugs in the primary database. The concurrency scheme on the backup is different from the primary and thus uncorrelated corruption bugs can be detected with a simple checksum scheme. On the primary database, we add a checksum to each committed transaction. The checksum is calculated over the transaction's write set, containing keys and row data. On the backup, after replaying a transaction, we recalculate the checksum and compare it with the primary's checksum. A checksum mismatch indicates a bug. This scheme imposes 12% overhead on the primary performance, although a faster checksum will help improve performance.

Using the checksum scheme, we have found and fixed 3 bugs in ERMIA's concurrency control implementation. Two of them are related to timestamp tracking in ERMIA's SSI and SSN implementation. We also find ERMIA's phantom protection protocol a significant flaw. ERMIA reuses Silo's phantom protection protocol, but Silo's protocol only works under a single-versioned database. All of these three bugs are non-crashing concurrency bugs and can lead to a non-serializable schedule. Without our system, it would have been hard to identify them.

8. CONCLUSIONS

We have designed a primary-backup replication scheme for providing fault tolerance for high-throughput, in-memory, databases. Our approach uses deterministic record-replay for replication. A key motivation for this work is to minimize the network traffic requirements due to logging. We have shown that recording the keys in the write-set requires 15-20% of the network bandwidth needed by traditional logging for OLTP workloads. The write-set keys can be used to perform deterministic replay concurrently, using epoch-based processing and a multi-version database. We have shown that this approach allows the backup to scale as well as ERMIA, a modern in-memory database that supports heterogeneous workloads.

9. REFERENCES

[1] Arulraj, J., Perron, M., and Pavlo, A. Write-behind logging. *Proc. VLDB Endow. 10*, 4 (Nov. 2016), 337–348.

[2] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A critique of ANSI SQL isolation levels. In *Proc. of the 1995 ACM SIGMOD International Conference on Management of Data* (May 1995), SIGMOD '95, pp. 1–10.

[3] Cahill, M. J., Röhm, U., and Fekete, A. D. Serializable isolation for snapshot databases. *ACM Trans. Database Syst. 34*, 4 (Dec. 2009), 20:1–20:42.

[4] Davis, J. pg_upgrade + streaming replication. https://www.postgresql.org/message-id/1332194822.1453.4.camel%40sussancws0025, Mar. 2012.

[5] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1243–1254.

[6] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. a., and Chen, P. M. Execution replay of multiprocessor virtual machines. *Proc. of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)* (Mar. 2008), 121–130.

[7] Faleiro, J. M., and Abadi, D. J. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow. 8*, 11 (July 2015), 1190–1201.

[8] Faleiro, J. M., Abadi, D. J., and Hellerstein, J. M. High performance transactions via early write visibility. *Proc. VLDB Endow. 10*, 5 (2017).

[9] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. Making snapshot isolation serializable. *ACM Trans. Database Syst. 30*, 2 (June 2005), 492–528.

[10] GitLab Team, 2017. *https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/*.

[11] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An Application-Level Kernel for Record and Replay. *Operating Systems Design and Implementation* (2008), 193–208.

[12] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. OLTP through the looking glass, and what we found there. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), SIGMOD '08, pp. 981–992.

[13] Hong, C., Zhou, D., Yang, M., Kuo, C., Zhang, L., and Zhou, L. KuaFu: Closing the parallelism gap in database replication. *Proceedings - International Conference on Data Engineering* (2013), 1186–1195.

[14] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1496–1499.

[15] Kim, K., Wang, T., Johnson, R., and Pandis, I. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, pp. 1675–1687.

[16] Larson, P.-A., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow. 5*, 4 (Dec. 2011), 298–309.

[17] Malviya, N., Weisberg, A., Madden, S., and Stonebraker, M. Rethinking main memory OLTP recovery. In *Proc. of the IEEE 30th Intl. Conference on Data Engineering* (Mar. 2014), pp. 604–615.

[18] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 183–196.

[19] McInnis, D. The Basics of DB2 Log Shipping. *IBM developerWorks* (April 2003). *http://www.ibm.com/developerworks/data/library/techarticle/0304mcinnis/0304mcinnis.html*.

[20] Microsoft. About Log Shipping (SQL Server), May 2016. *https://msdn.microsoft.com/en-us/library/ms187103.aspx*.

[21] Neumann, T., Mühlbauer, T., and Kemper, A. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 677–689.

[22] Oracle Corporation. *MySQL 5.7 Reference Manual*, 2016. Chapter 18, *https://dev.mysql.com/doc/refman/5.7/en/replication.html*.

[23] Percona Lab. TPC-C MySQL, 2016. *https://github.com/Percona-Lab/tpcc-mysql*.

[24] The PostgreSQL Global Development Group. *PostgreSQL 9.5.4 Documentation*, 2016. Chapter 25, *https://www.postgresql.org/docs/current/static/warm-standby.html*.

[25] The PostgreSQL Global Development Group. *PostgreSQL 9.5.6 Documentation*, 2017. `pg_upgrade`, *https://www.postgresql.org/docs/9.5/static/pgupgrade.html*.

[26] Saito, Y. Jockey: a user-space library for record-replay debugging. *Proceedings of the sixth international symposium on Automated analysis driven debugging* (2005), 69–76.

[27] Thomson, A., and Abadi, D. J. The case for determinism in database systems. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 70–80.

[28] Thomson, A., Diamond, T., and Weng, S. Calvin: fast distributed transactions for partitioned database systems. In *Sigmod '12* (2012), pp. 1–12.

[29] Tözün, P., Pandis, I., Kaynak, C., Jevdjic, D., and Ailamaki, A. From A to E: Analyzing TPC's OLTP Benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proc. of the 16th Intl. Conference on Extending Database Technology* (2013), pp. 17–28.

[30] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proc. of the 24th ACM Symposium on Operating Systems Principles* (2013), pp. 18–32.

[31] Uber Technologies Inc. Why uber engineering switched from postgres to mysql, July 2016.

[32] Wang, T., Johnson, R., Fekete, A., and Pandis, I. The Serial Safety Net: Efficient Concurrency Control on Modern Hardware. *Proceedings of the 11th International Workshop on Data Management on New Hardware* (2015), 8:1—-8:8.

[33] Wu, Y., Guo, W., Chan, C.-Y., and Tan, K.-L. Fast failure recovery for main-memory dbmss on multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 267–281.

[34] Wu, Y., and Tan, K.-L. Scalable in-memory transaction processing with HTM. In *Proc. of the 2016 USENIX Annual Technical Conference* (2016), ATC '16, pp. 365–377.

[35] Yao, C., Agrawal, D., Chen, G., Ooi, B. C., and Wu, S. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proc. of the 2016 Intl. Conference on Management of Data* (2016), SIGMOD '16, pp. 1119–1134.

[36] Zheng, W., Tu, S., Kohler, E., and Liskov, B. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 465–477.