Query Fresh: Log Shipping on Steroids

Tianzheng Wang
University of Toronto
tzwang@cs.toronto.edu

Ryan Johnson
LogicBlox
ryan.johnson@logicblox.com

Ippokratis Pandis
Amazon Web Services
ippo@amazon.com

ABSTRACT

Hot standby systems often have to trade safety (i.e., not losing committed work) and freshness (i.e., having access to recent updates) for performance. Guaranteeing safety requires synchronous log shipping that blocks the primary until the log records are durably replicated in one or multiple backups; maintaining freshness necessitates fast log replay on backups, but is often defeated by the dual-copy architecture and serial replay: a backup must generate the "real" data from the log to make recent updates accessible to read-only queries.

This paper proposes *Query Fresh*, a hot standby system that provides both safety and freshness while maintaining high performance on the primary. The crux is an append-only storage architecture used in conjunction with fast networks (e.g., InfiniBand) and byte-addressable, non-volatile memory (NVRAM). Query Fresh avoids the dual-copy design and treats the log as the database, enabling lightweight, parallel log replay that does not block the primary.

Experimental results using the TPC-C benchmark show that under Query Fresh, backup servers can replay log records faster than they are generated by the primary server, using one quarter of the available compute resources. With a 56Gbps network, Query Fresh can support up to 4–5 synchronous replicas, each of which receives and replays ∼1.4GB of log records per second, with up to 4–6% overhead on the primary compared to a standalone server that achieves 620kTPS without replication.

PVLDB Reference Format:
Tianzheng Wang, Ryan Johnson, Ippokratis Pandis. Query Fresh: Log Shipping on Steroids. *PVLDB*, 11(4): 406 - 419, 2017.
DOI: https://doi.org/10.1145/3164135.3164137

1. INTRODUCTION

Hot standby is a popular approach to high availability and is employed by many production database systems [21, 47, 48, 52, 57]. A typical hot standby system consists of a single primary server ("the primary") that processes read/write transactions, and one or more backup servers ("backups" or "secondaries"). The primary periodically ships log records to backups, which continuously replay log records. Log shipping can be configured as *synchronous* to guarantee strong safety [20]: transactions are not committed until their log records are persisted in all (or a majority of) nodes. Synchronous log shipping ensures that if the primary fails, a backup can take over as the new primary without losing committed work. In addition to replaying log records, backups can serve read-only transactions, improving resource utilization and performance.

Proceedings of the VLDB Endowment, Vol. 11, No. 4

DOI: https://doi.org/10.1145/3164135.3164137

1.1 The Freshness Gap

The ability to serve read-only transactions on backups while maintaining strong safety guarantees is a salient feature of log shipping: modern database servers are high-end machines that constitute non-trivial parts of the total cost of ownership, and offloading read-only transactions to backups can significantly increase hardware utilization and read-only throughput. Despite their popularity and usefulness, existing hot standby solutions often exhibit stale data access: queries on backups can only use a potentially much earlier snapshot of the primary database, as updates and new additions of data are not reflected on backups very quickly [68].

The reason for the freshness gap is two-fold. First, without logical log shipping and deterministic execution [39, 58, 69], the primary must continuously transfer large amounts of log records to backups, demanding high network bandwidth that traditional network often lacks. This is particularly true for modern main-memory database engines [16, 29, 32, 33, 61] that can easily produce 10–20Gb of log data per second (see details in Section 4). Such log rate is well beyond the bandwidth of ordinary 10Gbps networks. To support strong safety, log shipping must be synchronous, and the primary must wait for acknowledgement from backups on persisting log records. This puts network and storage I/O on the critical path, making the primary I/O bound.

However, the more important, fundamental reason is inefficient log replay in existing systems. These systems rely on the dual-copy architecture and permanently store data *twice*: in the log and the "real" database. Log records must be fully replayed before new updates can be queried in backups, involving non-trivial data movements and index manipulations. Moreover, in many systems log replay is single-threaded [48, 68], although it is commonplace for the primary to generate log records with multiple threads concurrently. It is unlikely for backups to easily catch up with the primary and provide fresh data access to read-only transactions.

1.2 Query Fresh

This paper proposes *Query Fresh*, a hot standby solution that provides both safety guarantees and fresh data access on backups while maintaining high performance. The key to realizing this is an append-only storage architecture that is built on modern hardware and allows fast log data transfer and lightweight, parallel log replay.

Modern hardware. Query Fresh utilizes high-speed networks (such as InfiniBand) and byte-addressable, non-volatile memory

(NVRAM) to alleviate the staleness caused by slow network and storage I/O. Recent fast networks provide high bandwidth and low latency, and allow fast remote direct memory access (RDMA). Coupled with fast NVRAM devices such as Intel 3D XPoint [14] and NVDIMMs [1, 63], the primary ships log records directly from its log buffer to *NVRAM-backed* log buffers in backups. Log records are instantly persisted once they reach NVRAM, without explicit access to the (expensive) storage stack. Moreover, Query fresh takes advantage of RDMA's asynchronous nature to hide the latency of data transfer and persistence behind forward processing, moving I/O out of the critical path with simpler implementation. Log data can be gradually de-staged in background to other bulk storage devices such as flash memory and disks when the log buffer is full.

Lightweight, parallel log replay. Fast network and NVRAM do not mitigate staleness due to inefficient log replay, caused by the traditional dual-copy architecture. Instead of maintaining two durable copies, Query Fresh employs append-only storage and keeps the log as the only durable copy of data, i.e., the log is the database [32]. This is made possible by redo-only logging: the log only contains actual data generated by committed transactions. Redo-only logging is common in modern main-memory database engines [32, 33, 61] and we design Query Fresh based on it. With the log as the database, worker threads access data through indirection arrays [12, 36, 55] where each entry maps a (logical) record identifier (RID) to the record's physical location (in memory or storage, i.e., the log). Indirection arrays are purely in-memory, but can be checkpointed for faster recovery. Indexes map keys to RIDs, instead of physical pointers. Updates can be reflected in the database by simply updating indirection arrays, without updating indexes.

The combination of append-only storage and indirection allows us to build a lightweight, parallel replay scheme without excessive data copying or index manipulations. Replay becomes as simple as scanning the log buffer, and setting up the indirection arrays. This process is inexpensive and faster than forward processing on the primary, guaranteeing backups can catch up with the primary without exhausting all compute resources. This improves freshness and frees up more resources to run read-only transactions on backups.

Evaluation results using TPC-C [60] show that with 56Gbps InfiniBand, before the network is saturated, Query Fresh can support up to 4–5 synchronous backups, each of which receives ∼1.4GB of log data from the primary, whose throughput does not drop by more than 6% over a standalone server without replication (620kTPS). Our evaluation also shows that backups are able to finish replaying 16MB of log records in ∼12ms using around one quarter of the compute resources, leaving more resources for read-only transactions.

1.3 Contributions and Paper Organization

Our first contribution is an efficient log shipping scheme that takes advantage of NVRAM and the asynchronous nature of RDMA to easily keep network I/O and replay out of the critical path. We also highlight caveats for using RDMA over NVRAM for safe persistence. Second, we propose to utilize append-only storage for log shipping and replay. Third, we devise a lightweight, parallel replay scheme for backups to keep up with the primary with fewer resources than the primary needs for forward processing.

Query Fresh is integrated with ERMIA [32], a recent open-source, main-memory database engine. Our implementation is available at: https://github.com/ermia-db/ermia.

The ideas of append-only storage and indirection have been used in other systems [3, 7, 12, 32, 35, 36, 55]. Our focus is utilizing them to speed up log replay and improve freshness. Previously, we have given the abstract idea [65]; this paper details the design, implementation, and evaluation of such a system, Query Fresh.

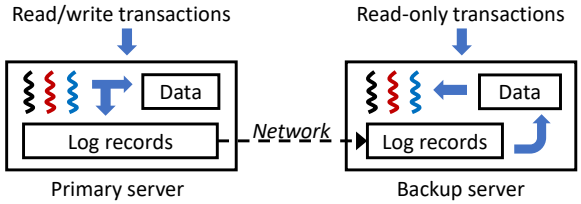

Figure 1: A typical dual-node hot standby system using log shipping. The primary persists and ships log records through the network to the backup, which replays the log and serves read-only queries.

Next, we cover background in Section 2. Section 3 describes the design of Query Fresh, followed by evaluation in Section 4. We summarize related work in Section 5 and conclude in Section 6.

2. BACKGROUND

In this section, we begin by defining the type of systems we focus on. Then we introduce the basics of fast networks and RDMA, and discuss issues related to using RDMA over NVRAM.

2.1 System Model and Assumptions

Hot standby systems. We focus on hot standby systems based on log shipping. Figure 1 shows a dual-node example. Such a system consists of a single primary server that accepts both reads and writes, and one or multiple backup servers. Backups accept log records transferred from the primary and replay them continuously. To increase hardware utilization and throughput, backups can also process read-only transactions, however, a backup never accepts writes. In dual-copy systems, log replay is necessary to transform log records into "real" database records that can be queried by read-only transactions. If the primary fails, a backup will take over and become the new primary after replaying all log records. A new node can join the cluster as a backup by requesting a recent checkpoint and the log records generated after the checkpoint from the primary.

Safety guarantees. The original concept of safety guarantees were given for dual-node pairs. Depending on whether the primary involves the backup at commit time, a dual-node system can be *1-safe*, *2-safe*, or *very safe*, as defined by Gray and Reuter [20]:

Definition 1. *1-safe: transactions commit as soon as their log records are durably stored at the primary.*

Definition 2. *2-safe: transactions are not committed until their log records have been persisted at both the primary and backup. If the backup is down, the transaction can commit after persisting log records at the primary.*

Definition 3. *Very safe: same as 2-safe, except that it does not allow transactions to commit if the backup is down.*

Under 1-safe, log records are shipped in the background without involving the backup on the commit path (asynchronous log shipping). Therefore, the primary exhibits performance similar to a single-node system's. Besides providing high primary performance, 1-safe can also be useful for reasons such as preventing human errors from propagating to all servers [52]. However, log records that are not yet shipped to the taking-over backup may appear to be lost, although it was "committed" on the then-active primary. In contrast, very safe and 2-safe involve the backup on the commit path, by not committing a transaction until its log records are persisted in the primary *and* backup. This often reduces primary performance, but provides stronger safety guarantees than 1-safe because it avoids losing committed work.

Although these definitions were originally given for dual-node pairs, we can easily extend them to systems with more nodes, by requiring log records be persisted in all backups upon commit in Definitions 2 and 3. Moreover, one could also make Definition 2 more strict (or relax Definition 3) to allow transactions to commit if a majority of nodes have persisted the log [13, 62, 48].

In this paper, we say that a system guarantees *strong safety* if it is 2-safe, very-safe or follows the above extended definitions for multi-node systems. Providing strong safety requires synchronous log shipping, i.e., transferring log records to backups and ensuring they are persisted in all or a majority of nodes upon commit. Efficiently supporting synchronous log shipping is one of the goals of this paper. Our design is applicable to both traditional dual-node pairs and quorum based systems [13, 62]. In this paper, we target cases where the primary must persist log records in all backups.

Logging. Transactions generates log records that describe changes to the database. Each log record is represented by a global, monotonically increasing log sequence number (LSN). Logging can be physical, logical, or a combination of both (physiological logging) [46]. Physical logging stores before and after images, whereas logical logging stores only the necessary information (e.g., the operation and input) needed for re-constructing the new data. Physiological logging uses physical logging for redo, and uses logical logging for undo. It is also attractive to use logical logging for redo and log shipping because logical logging generates much less log data, thus needs lower network bandwidth. However, achieving fast replay then requires non-trivial effort to implement deterministic execution on backups [39, 51, 58, 68]. Deterministic execution is tightly integrated with concurrency control; a replay scheme that works for two-phase locking might not work for snapshot isolation [39]. Many schemes only support stored procedures, and certain applications (e.g., data-dependent operations) cannot be supported. For simplicity, many systems use serial replay for logical logging [48].

Modern main-memory database systems often use redo-only physical logging [32, 33, 61] that does not store before images in the log. During forward processing, a transaction holds its log records locally and does not store them in the log buffer until the transaction successfully pre-commits. So the log contains only data generated by committed transactions. Recovery becomes a single redo pass, without analysis and undo in ARIES [46]. We build Query Fresh for main-memory systems that use redo-only physical logging.

2.2 Fast Networks and RDMA

High-speed network interconnects were mostly used by high-performance computing systems, but are now becoming cost effective and being adopted by database systems. Despite the differences in their underlying technologies, these interconnects all provide high bandwidth and low latency, and support fast RDMA, making the network no longer the slowest part of a distributed system [9].

Network interconnects. Two major kinds of fast networks are InfiniBand and Converged Ethernet. InfiniBand is a switched fabric network that can be used both within and among nodes. Today's InfiniBand already provides bandwidth that is close to that of a single memory channel [22, 25], and the aggregate bandwidth of multiple NICs can reach the memory bandwidth of a single server [70], e.g., FDR 4× gives 56Gbps bandwidth, the data rate of the upcoming HDR 4× is 200Gbps [22]. We expect the trend to continue.

InfiniBand's low latency features can be layered on top of Ethernet to implement RDMA over Converged Ethernet (RoCE) [11]. RoCE provides competitive performance, e.g., some products support 100Gbps link speed [42]. RoCE can be implemented in software or hardware. Software RoCE is compatible with any standard Ethernet, but is not as performant as the hardware-accelerated ones.

RDMA. RDMA allows nodes in a cluster to access each other's designated memory areas, without having to go through multiple layers in the OS kernel, avoiding unnecessary data copying and context switches between the user and kernel spaces which are normally unavoidable using the TCP/IP stack. RDMA is also non-blocking: posting an RDMA operation does not block the caller. The caller instead should explicitly check for completion of the posted work requests, e.g., by polling the completion queue.

RDMA exhibits a "verbs" interface [41] that is completely different from TCP/IP sockets. Peers communicate with each other by posting verbs (work requests) to queue pairs (similar to sockets in TCP/IP). A queue pair consists of a send and a receive work queue. There are two types of verbs: channel semantic (SEND and RECV) and memory semantic (READ, WRITE, and atomics). Channel semantic verbs need coordination with the remote CPU (a RECV must be posted for each remote SEND). Memory semantic verbs operate directly on remote memory without involving the remote CPU. In addition, RDMA Write with Immediate allows the sender to accompany the payload with an immediate value (e.g., to carry metadata). The only difference compared to RDMA Write is that the receiving end should post a receive request to obtain the immediate value [41]. Query Fresh uses RDMA Write and RDMA Write Immediate for log shipping.

2.3 RDMA over NVRAM

NVRAM can be attached to the memory bus and accessed remotely through RDMA. However, a completed RDMA write request does not indicate that data is written in remote NVRAM. Rather, it only guarantees that the data has arrived at the remote side, although the data might be cached by the remote NIC or CPU, instead of being stored in NVRAM due to techniques such as Intel Data Direct I/O (DDIO) [23]. As a result, data visibility and durability are decoupled: data is visible on the remote side once it is out of the I/O controller and arrives at the CPU cache; data will only get written to memory when it is evicted from CPU cache [23].

There are two stop-gap solutions, the "appliance" and "general-purpose server" methods [17]. The former requires turning off DDIO and enabling non-allocating writes in the BIOS. Each RDMA Write should be followed by an RDMA Read for the remote node to force data to NVRAM. The use of non-allocating writes and RDMA Read adds additional overheads. The remote node needs no additional operation. The required BIOS changes make it less ideal in environments such as the cloud. The general-purpose server method requires no BIOS change, but the remote side needs to explicitly issue the `CLFLUSH` or `CLWB` instruction followed by a store fence for persistence and correct ordering [24].

The general-purpose server method can incur 50% higher latency than the appliance method [17]. Both methods need at least two round trips for data durability. Simply pushing data to the remote site does not guarantee persistence [71]. Nevertheless, the general-purpose server method should give us worst-case performance; we quantify its overhead in Section 4. We expect RDMA protocol changes for NVRAM [56] to be the ultimate solution.

3. QUERY FRESH

Query Fresh consists of a synchronous log shipping mechanism that exploits RDMA and NVRAM, an append-only storage architecture with indirection designed for log shipping, and a parallel, lightweight log replay scheme that utilizes append-only storage to allow backups to keep up with the primary. Like other hot standby systems, the primary server in Query Fresh executes read/write transactions and ships log records to backups when needed. It uses a background daemon to listen to new backups requesting to join

the cluster and sets up a connection for each backup server. During connection setup, the daemon registers the log buffers in both the primary and backups to conduct RDMA. Memory registration is a one-time operation that does not affect forward processing.

Several database systems employ group or pipelined commit [26] to keep I/O out of the critical path and issue large sequential writes to storage for better performance. In these systems, worker threads submit processed transactions to a commit daemon and continue to serve the next request. A transaction's commit result is not returned to the client until its log records are persisted. Log flushes are typically issued by request, timeout, or when the log buffer's occupied space exceeds a threshold (e.g., 50% full). Query Fresh piggybacks on the commit daemon and ships log records whenever a log flush or group commit happens. This way, we take advantage of RDMA's asynchronous nature to overlap network and local I/O, as well as forward processing, hiding most of the cost of data transfer.

Query Fresh uses two key techniques to accelerate log replay on backups: (1) replay pipelining that moves log replay (mostly) out of the critical path and (2) a parallel, lightweight scheme that uses our append-only storage and indirection, to reduce the amount of work done by replay threads, making log replay itself faster.

In the rest of this section, we first describe how Query Fresh utilizes RDMA and NVRAM for log shipping. Then we cover how replay pipelining works. We then introduce the append-only storage architecture designed for log shipping and fast log replay, followed by a comparison with other popular approaches.

3.1 RDMA and NVRAM based Log Shipping

In Query Fresh, log shipping is synchronous and largely a memory-only operation that does not involve the storage stack, thanks to RDMA and NVRAM. For backups, we place the log buffer in NVRAM and register it to perform RDMA operations. The primary, however, does not have to be equipped with NVRAM, unless it needs to re-join later as a backup (e.g., after failover).

Transferring log data. Query Fresh takes advantage of RDMA's non-blocking nature to efficiently overlap the process of persisting log records locally with network I/O. Figure 2(a) illustrates how we ship log records. Upon group commit or log flush, the primary first issues an RDMA Write with Immediate request (one per backup) to store log records in each backup's NVRAM log buffer, with the size of the data being shipped as the immediate value (step 1). Since we use RDMA Write with Immediate to ship log records, the backup needs to post a receive request to retrieve the immediate value. After sending the RDMA request, the primary flushes the log locally and/or to shared or local storage if NVRAM is not in place for the primary. Otherwise it suffices to persist data in NVRAM using `CLWB` (followed by a store fence), and log records can be de-staged in the background [64]. Meanwhile, the network I/O initialized in step 1 will happen in parallel with log flush. Once the log is persisted locally, the primary starts to poll the result for the write posted for each backup (part of step 3), to ensure that the write request is sent and its completion event is consumed so that the RDMA library's internal data structure (`libibverbs` in our implementation) is kept in its correct status. We find the polling overhead is negligible.

Ensuring persistence. Returning from the polling operation in step 3 only indicates that the write request has been successfully sent; there is no guarantee on when the data will be persisted in the backup, even with NVRAM log buffer on backups (discussed in Section 2.3). If the general-purpose server method is employed, the primary should explicitly wait for acknowledgement from backups to ensure log records are received *and* persisted on backups. Our current implementation of the general-purpose server method uses RDMA Write (by the backup) and atomic read (by the primary) for

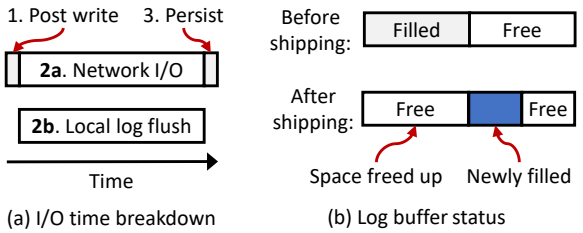

Figure 2: RDMA-based log shipping (primary server). (a) Steps to ship a batch of log records from the log buffer. Network I/O (2a) overlaps with local I/O (2b). Step 3 ensures durability on the remote side. (b) Using multi-buffering to reduce log buffer wait time.

acknowledgement. The primary exports for each backup a `status` word that can be remotely written using RDMA Write and spins on it for a "persisted" signal, which is written by the backup after it persisted the data. Upon receiving acknowledgement, the primary commits all transactions covered by the just-shipped log records.

Two approaches could be used to remove/reduce the spinning on the "persisted" signal. The first approach is to rely on large battery arrays which provide de-facto persistence across the memory bus. They are being adopted by data centers [30]. Backups could directly acknowledge the primary—or even skip it—as data is "persisted" implicitly. The other approach reduces spinning, by allowing the primary to return before backups acknowledge, and check for persistence before reusing the log buffer space, e.g., before sending the next batch. Until getting acknowledgement, the primary does not commit the transactions covered by the shipped log records. This approach trades commit latency for throughput, so it is not recommended if low commit latency is desired.

To persist the received log records, backups issue `CLWB` and a store fence while the primary is waiting in step 3 of Figure 2(a). The speed of persisting data in NVRAM is determined by memory bandwidth; a single flusher thread could make the system NVRAM-traffic bound. To reduce persistence latency, we introduce a persist-upon-replay approach that piggybacks on multiple replay threads to divide the work. As Section 3.4.2 describes, each log replay thread is responsible for redoing log records stored in one part of the log buffer. Before starting replay, each replay thread first issues `CLWB` and store fence to persist its share of log records. The log shipping daemon then acknowledges the primary once all replay threads have finished persisting their share of log data.

Guaranteeing strong safety. Our log shipping design follows the definitions of strong safety in Section 2.1. Transactions are never committed if their log records are not durably replicated across all or a majority of nodes in the cluster. Hence, Query Fresh provides strong safety. Note that log replay does not affect safety guarantees: a safe system only needs to ensure log records are properly persisted to not lose committed work; whether the log records are quickly replayed has more impact on data freshness, not safety guarantees.

Discussion. Overlapping network and storage I/O is not new and can be done with TCP/IP, but often requires extra implementation efforts. RDMA's asynchronous nature makes it possible to easily piggyback on the existing commit daemon. RDMA also provides superior performance with kernel bypassing. An inherent limitation of RDMA Write/with Immediate is that these operations are only available under unicast [41], limiting the number of backups that can be kept synchronous. For example, in theory a 56Gbps network can keep up to seven synchronous backups, if the primary's log rate is 1GB/s. The number will likely become lower with necessary overheads (e.g., acknowledgements). Section 4 shows this effect.

3.2 Replay Pipelining

Maintaining fresh data access requires the read view on backups does not fall much behind the primary's. A straightforward solution is synchronous log replay that ensures replay is finished before sending persistence acknowledgment to the primary. Although it guarantees absolute freshness, i.e., the primary and backup always exhibit exactly the same read view, synchronous replay significantly slows down the primary with replay on the critical path.

We solve this problem by replay pipelining. The key is to overlap log replay with forward processing without lagging behind. After persisting log records locally using `CLWB`, instead of waiting for replay to finish, the backup acknowledges immediately (if needed). Meanwhile, the backup notifies its own log flusher to de-stage the received log records from NVRAM to storage. Without waiting for the flush to finish, the backup starts immediately to replay log records by notifying its replay threads, each of which scans part of the log buffer. As a result, the replay of batch i is "hidden" behind forward processing that generates batch i+1. This way, log replay is kept out of the critical path as long as replay is fast enough to finish before the next batch of log records arrives.

For replay pipelining to work, the primary and backup need to coordinate with each other on when a batch of log records can be shipped: the backup must ensure the log buffer space to be used for accepting new data is available, i.e., replayed *and* de-staged to storage. We solve this problem with a `ReadyToReceive` status on the status word exported by the primary. Before sending a batch of log records, the primary waits for the backup's status to become `ReadyToReceive`, which is posted by the backup using RDMA Write after it has replayed and de-staged the previous batch.

Replay pipelining works nicely with multi-buffering, a widely used technique to reduce log buffer wait time. As Figure 2(b) shows, while the commit daemon is shipping a batch of log records, worker threads may insert to the other half of the log buffer. With multi-buffering, the log buffer space is essentially divided into multiple ranges, allowing the backup to accept new log data while replaying a previous batch. For each range, we maintain an "end LSN". If a range's end LSN is no greater than the most recently persisted and replayed batch's LSN, then the range is available for new data. The backup could issue `ReadyToReceive` and accept new data as soon as it finds that the next range is free, allowing batch i+1 to flow in while batch i is being replayed. Alternatively, the backup could delay the issuing until each current batch is replayed for better freshness. Compared to synchronous replay, replay pipelining could reduce freshness because commit does not guarantee immediate visibility on backups. As we show in Section 4, such effect is minimal and replay pipelining achieves much higher primary performance.

Replay pipelining moves log replay out of the critical path, but does not accelerate replay itself, which directly affects data freshness and determines if log replay will block the primary. Next, we show how we utilize append-only storage to accelerate log replay. We start with a review on append-only storage. Readers already familiar with it can skim and fast forward to Section 3.4.

3.3 Basic Append-Only Storage

The ideas of append-only storage and indirection are not new and have been employed in many systems [3, 7, 32, 35, 36, 55]. We first review how it works for single-node systems, and then extend the design for log shipping and fast log replay.

With append-only storage, data is never overwritten; updates to the database are always appended to the end of the log. Each record is associated with an RID that is logical and never changes. Transactions access data through another level of indirection that maps RIDs to the record's physical location. Indirection can be used by both single and multi-version systems; we base our work on ERMIA [32] (a multi-version system) and employ its append-only design.[1] As Figure 3 shows, each table is accompanied with an indirection array. Each entry in the indirection array is an 8-byte word, excluding the "RID" field shown in Figure 3, which is an index into the array and only shown for clarity. The indirection array entry could point to the in-memory version or contain an offset into the log; a later access can use approaches such as anti-caching [15] to load the version to memory and replace the entry with a pointer to the in-memory record. Our current implementation reserves a lower bit in the indirection array entry to differentiate virtual memory pointers and offsets into the durable log [32].

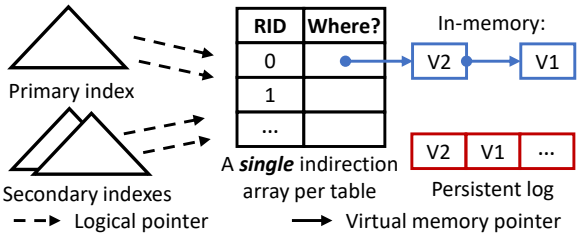

Figure 3: Append-only storage with indirection on single-node systems. Each table employs an indirection array indexed by RIDs. Indexes map keys to RIDs; updates only modify the RID entry in the indirection array, no index operations are needed.

With indirection, indexes map keys to RIDs, instead of the records' physical locations. All indexes (primary and secondary) on the same table point to the same set of RIDs. RIDs are local to their "home" indirection array, and allocated upon inserts. We assign each indirection array (i.e., logical table) a unique file ID (FID), and the combination of FID and RID uniquely identifies a database record. FIDs are maintained in exactly the same way as RIDs; a special "catalog" indirection array maps FIDs to indirection arrays. Each index stores a pointer to the underlying indirection array (or simply the FID). Worker threads can query an index first for the RID, then consult the table's indirection array to access the record.

The benefit of employing append-only storage with indirection is two-fold. First, updates that do not alter key fields require no index operations: all they need to do is to obtain the RID (e.g., by traversing an index) and modify the indirection array entry to point to the new version (e.g., by using a `compare-and-swap` instruction or latching the version chain). This is especially beneficial if a table employs multiple secondary indexes, as without indirection arrays the updater will have to make the key in every index point to the new version. Second, thanks to the adoption of redo-only physical logging, recovery becomes as simple as setting up indirection arrays while scanning the log records, without generating "real" records (unless otherwise required) like in systems without indirection.

3.4 Append-Only Storage for Log Shipping

Query Fresh employs indirection in both primary and backup servers, but follows a slightly different design to support lightweight, parallel log replay, concurrent read-only queries, and failover.

3.4.1 Making Replay Lightweight

The design in Section 3.3 has the potential of reducing the amount of work of replay down to its minimum: replay threads simply store the physical location of each log record (which is the actual data

[1] Query Fresh is also applicable to other systems, as long as they can be equipped with indirection arrays for data access and more importantly, fast log replay.

tuple) in the indirection array. However, blindly applying this technique could significantly hurt read performance: an indirection array entry usually points to a list of in-memory versions, and replacing the pointer with a log offset in fact forcibly evicts the in-memory versions, because after replay, new readers will not be able to see them and thus must reload versions from storage. Moreover, additional tracking is needed to avoid leaking memory by recycling the in-memory versions once they are not needed by existing readers.

To solve this problem, for each table we employ an additional *replay array* which only stores the latest physical, permanent address of the record. Figure 4 describes the detailed design. The data array still holds pointers to all in-memory records (or possibly pointers to the durable log after recovery). Replay threads simply parse the log in NVRAM and store each version's physical address in the replay array. Log replay threads never incarnate versions by loading data from storage or NVRAM to main memory. This way, we reduce the amount of work that must be done by log replay to its minimum while still maintaining high read performance. For inserts, the replay threads must also insert the key-RID pairs to indexes. This method works for any index. An alternative is to also use indirection arrays for index nodes, so that index node updates are logged and replayed in the same way we handle database records. Indexes such as the Bw-tree [36] will fit in this approach more easily. After the log data is persisted globally (in NVRAM or storage) and replayed, the (new or updated) records become visible to read-only transactions being run by threads that are not dedicated for log replay. Since the latest updates are all only reflected on replay arrays, readers must also examine them when trying to find suitable versions to access.

3.4.2 *Parallelizing Log Replay*

In Query Fresh we employ on the backup multiple threads to replay the log received from the primary. Besides reducing the amount of work that needs to be done for each record by employing the replay arrays, another important aspect is to reduce the amount of data that must be examined by each thread. We achieve this by assigning each replay thread part of the log buffer to scan and replay. For this to work, in addition to the log buffer data, the primary also sends metadata that describes the logical partitions of the shipped data for each replay thread to follow. Note that blindly assigning each thread an equal portion of the log to replay is not feasible: log records are of variable sizes, so we must ensure replay threads can locate the right log record in its partition to begin replaying.

The partitioning information is generated by the primary during forward processing. We logically divide the log buffer space into a predefined number (n) of partitions of equal size. Whenever a log buffer allocation crosses a partition boundary, the primary records the allocation's end log record offset. The primary sends such metadata using RDMA Write with Immediate whenever it ships log records. Upon receiving the log data and metadata, the backup will employ multiple threads to replay the log records in parallel, using the metadata. Note that we do not have to employ n threads on the backup for replay; each thread can replay multiple partitions. In addition, as we have described in Section 3.1, these replay threads can also issue `CLWB` and store fence instructions in parallel to persist log data before start replaying, amortizing the cost of persistence.

3.4.3 *Coordination of Readers*

Backups must guarantee consistent read access. Versions created by the same transaction (on the primary) must not be visible to read-only transactions (on backups) before all of them are successfully replayed and persisted globally, i.e., installed in the replay arrays and durably stored in NVRAM or storage of all or a majority of nodes. Ensuring read-only transactions only see globally persisted

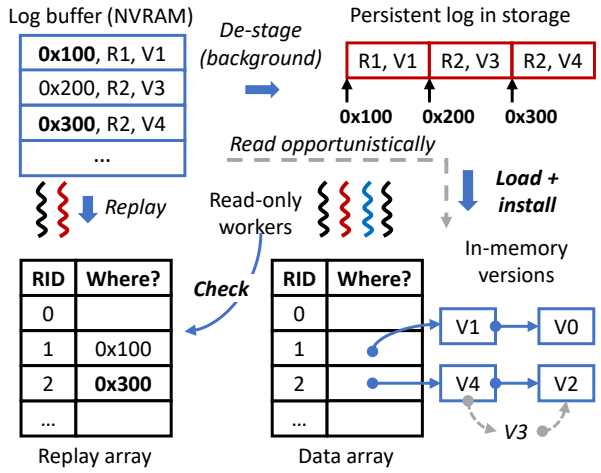

Figure 4: Append-only storage with indirection for log shipping. An additional replay array that is indexed by the same RIDs as the data array, stores the latest physical address of each record.

data guarantees correctness if the primary fails: multiple backups in the cluster could replay and persist locally in different speeds, so without proper visibility control, read-only transactions on the faster nodes could see uncommitted data if the primary crashes before receiving acknowledgement from all backup nodes.

We use LSNs to control read visibility, which is a common approach in multi-version systems. Here we describe the approach in ERMIA [32], which Query Fresh is based on; other systems might use different approaches but the general principle is similar. When entering pre-commit, the transaction acquires a globally unique commit timestamp and reserves log buffer space by atomically incrementing a shared counter by the size of its log records, using the atomic `fetch-and-add` (`FAA`) instruction.[2] The `FAA`'s return value indicates the transaction's commit order and its log records' starting offset in the durable log. For each record we use its location in the physical log, as its commit stamp. All the log records generated by the same transaction are co-located in the log and a record with a larger LSN is guaranteed to be stored in the later part of the log.

To control read visibility in backups, in each backup we maintain two additional LSNs: (1) the "replayed LSN" which is the end offset in the durable log of the most recently replayed log record, and (2) the "persistent LSN" which indicates the end durable log offset of the most recently persisted log record. A read-only transaction on the backup starts by taking the "read view LSN", which is the smaller of the visible and persistent LSNs as its begin timestamp b. It accesses versions with the maximum commit LSN that is smaller than b, and skips other versions. This way, a read-only transaction always reads a consistent snapshots of the database. During replay we install versions and bump the replayed LSN only after we have replayed all the log records for the same transaction. The persistent LSN is updated by the primary using RDMA Write after receiving acknowledgements from all backups. Thus, read-only transactions on backups will never see uncommitted data.

With a begin timestamp and target RID (obtained after querying the index or by a full-table scan), a read-only transaction on the backup starts by examining the target table's data array. If the head version is already newer than the transaction's begin timestamp, the transaction will continue to traverse the version chain to find the

[2]This instruction atomically increments the given memory word and returns the value stored in the word right before the increment [24].

proper version to read. However, if the head version is visible to the querying transaction (i.e., its commit stamp is smaller than the begin timestamp), the transaction will have to examine further the replay array in case a newer version is visible. Since a record's physical position in the log also reflects the global order of the transaction that generated it, when probing the replay array, the transaction directly compares its begin timestamp with the RID entry value. If the RID entry's value is larger than the LSN of the head version on the data array, then we continue to follow the physical address recorded by the RID entry to instantiate the latest version—no matter if it is visible to the querying thread or not—and continue to load older versions for the RID until we hit a visible version or the latest version represented by the data array (whichever is younger). Then, the querying thread will try to install this "sub-chain" of versions to the version chain on the data array and read the correct version, and retry the whole read operation if the installation failed (e.g., because another thread acted faster doing the same operation).

When loading versions, worker threads opportunistically read from the NVRAM-backed log buffer for better performance. It verifies that data read is still valid by checking the LSN range currently being represented by the log buffer. If the LSN range does not contain the read data's range, we read from storage.

3.4.4 *Failover and Catch-up*

All nodes (the primary and backups) in Query Fresh use exactly the same binary. If the primary is down, a backup (previously designated or elected [34]) will take over and become the new primary. This failover process involves both the new primary and other backups. First, the new primary needs to finish replaying the last batch of log records received from the old primary (if there is any). At the same time, it switches to the "primary" mode by notifying all other backups about the take-over. Then, the other backups will re-establish connections to the new primary and obtain the latest log data from it. Once all the remaining nodes are ready, the new primary can start processing read/write transactions. The new primary can continue to finish processing the read-only transaction requests received while it was a backup, and use the previously log replay threads for read/write transactions. When all the "old" read-only transactions are finished, the primary can employ all its threads for read/write transactions. This allows the user to run long, or even mission critical read-only queries without worrying about impact by failover. Replay arrays are the primary home for new updates on backups. Thus, when a backup takes over and starts to process read/write transactions, it must also consult the replay arrays as if it still were a backup server if the data array's head version is visible (following the steps described in Section 3.4.3). However, we do not expect this to become a major problem, as the read-only transactions that ran when the primary was a backup server should have already loaded most recent versions from storage/NVRAM to DRAM.

When the failed primary comes back online, it joins the cluster as a backup by connecting to the current primary. The process follows the same procedure as a new backup server wanting to join the cluster would follow. After connected to the primary, the new backup starts to accept and replay log records as if it already caught up with the primary using the replay techniques we proposed, but postpones the update of its replayed LSN to ensure correct read visibility. At the same time, the backup in another thread obtains from the primary a recent checkpoint plus the log records generated after the checkpoint was taken, but before the first batch of log records received from the primary since the connection is established; we call this data "catch-up records". Alternatively, the backup could read catch-up records from shared storage if data is stored there. The replay of catch-up records and "live" log records

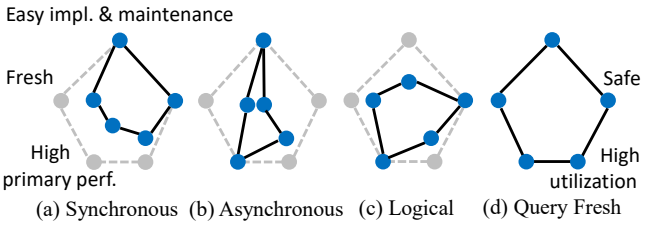

Figure 5: Relative merits of hot standby solutions. (a) Synchronous log shipping sacrifices performance and freshness in a slow network. (b) Asynchronous log shipping trades safety for performance. (c) Logical logging reduces network traffic, but is tightly coupled with concurrency control and trades utilization for freshness. (d) Query Fresh strikes a balance on all aspects.

shipped synchronously from the primary are replayed concurrently. If a recent checkpoint is available, or in the case of a re-joining primary, the new backup can start processing read-only transaction once the checkpoint is recovered or the node is recovered and synchronized with the new primary, respectively. The data might be stale depending on how old/new the checkpoint is and/or how long the node has been offline. We also parallelize the replay of catch-up records. This can be done in various ways and is orthogonal to the design choices of Query Fresh. In our current implementation, we logically partition the checkpoint and log files by RID. Each thread only replays its own partition. Recall that our replay mechanism only stores the location of the latest record in replay arrays, so the replay of catch-up records must not update the replay array entry if a newer one is already there.

Note that the size of the catch-up records is fixed, because once connected the new backup starts to accept new log records synchronously and keep up with the primary. As a result, the new backup is guaranteed to catch up with the primary eventually. Once the catch-up records are all replayed, the backup can update its replayed LSN and declare it has caught up with the primary.

3.5 Discussion

Figure 5 reasons about the relative merits of Query Fresh and three other popular approaches, by examining how well or badly each design satisfies various design goals. In the figure, we place properties of interested as dots and use the relative distance between blue and grey dots to represent the overhead. Dotted lines connect the ideal cases, while solid lines connect the real cases. Figure 5(a) summarizes our previous discussion on synchronous log shipping in a slow network: the primary is often bound by network I/O, despite its strong safety guarantees. To avoid being network bound, as Figure 5(b) shows, log records are often shipped asynchronously and a transaction is allowed to commit locally in the primary without ensuring the log records are durable also in backups. Asynchronous log shipping gives neither safety nor freshness, although it does not block the primary. Backups have to serve read-only queries with stale data that is a potentially much earlier snapshot of the primary database. Committed work whose log records are not yet shipped when the primary fails could be lost, so inconsistencies might arise after a stale secondary took over.

Logical log shipping and deterministic execution [39, 58, 69], as shown in Figure 5(c), replicate the operations to perform, instead of results (i.e., physical data) of the operations. This approach saves network bandwidth, but falls short on other aspects. First, many deterministic execution schemes only support stored procedures. Applications that use data-dependent operations may not work. Second, as Section 4 shows, the amount of compute resources needed

for replay on the backup is similar to that needed on the primary, leaving less resource for read-only transactions. Third, it requires careful handling of any randomness that might arise during transaction processing for correct log replay, such as random numbers and lock acquisition sequence, and all the work must be redone from scratch at each backup. The problem becomes even worse for multi-version concurrency control, under which commit order and serial order could be different. Replaying the log (i.e., based on commit order) might cause the primary and backup to diverge; a scheme that works for single-version 2PL might not work for snapshot isolation [39]. Solving this problem needs tight integration with the underlying concurrency control mechanism. As Figure 5(d) shows, Query Fresh strikes a balance among all the properties of interest, using modern hardware for high primary performance and strong safety guarantees, and using append-only storage with indirection for fast replay, fresh data access, and high resource utilization.

4. EVALUATION

This section empirically evaluates Query Fresh and compares it with traditional approaches, and confirms the following:

- With RDMA over fast network and NVRAM, Query Fresh maintains high performance for the primary server;
- The append-only architecture and replay strategy allow backups to keep up with the primary with little overhead;
- Query Fresh gives better freshness for read-only queries on backup servers compared to traditional approaches.

4.1 Implementation

We implemented Query Fresh in ERMIA [32], an open-source main-memory database engine designed for modern hardware. We augmented ERMIA with our lightweight replay and indirection techniques for log shipping. Each replay array uses exactly the same data structure as in original ERMIA. Therefore, with the replay arrays, the memory space needed by indirection is doubled. However, given the abundant memory space in modern main-memory database servers, we expect such addition to be acceptable. For example, since indirection array entry is 8-byte long, a million-record table would require around 16MB additional memory space (8MB for the data and replay arrays, respectively).

For indexes, we use Masstree [40] in ERMIA and all accesses are done through indexes. Our current implementation does not employ indirection for the index itself. Log replay only needs to insert new keys to the index(es) for inserts; for updates we setup indirection arrays, and do not touch indexes. In our experiments, we did not find manipulating indexes only for inserts to be a bottleneck.

4.2 Hardware

Testbed. We run experiments on the Apt cluster [53] testbed, an open platform for reproducible research. We use eight dual-socket nodes, each of which is equipped with two 8-core Intel Xeon E5-2650 v2 processors clocked at 2.6GHz (16 physical cores in total) and with 20MB cache. Each node has 64GB of main memory, a Mellanox MT27500 ConnextX-3 NIC and an Intel X520 Ethernet NIC. All nodes are connected to a 10Gbps Ethernet network and a 56Gbps FDR 4× InfiniBand network.

NVRAM emulation. The only *DIMM form-factor* NVRAM available on the market is NVDIMM [1, 63] that exhibits exactly the same performance characteristics as DRAM at runtime.[3] There is also a current trend in data centers to use battery arrays for persistence [30]. So we use DRAM in our experiments.

We show results for (1) the ideal case which assumes batteries in data centers or future RDMA protocol enhancement that guarantees NVRAM persistence, and (2) variants that employ the general-purpose server method for NVRAM persistence. In the ideal variant, backups acknowledge the primary right after receiving data without additional work. We test two variants of the general-purpose server method, denoted as "CLFLUSH" and "CLWB-EMU". The former uses the `CLFLUSH` instruction to persist log data. `CLFLUSH` will evict cachelines and impose non-trivial overhead. We only show these numbers for reference as lower-bound performance. Compared to `CLFLUSH`, the `CLWB` instruction does not evict the cacheline during write-back. `CLWB` is not available in existing processors, so we emulate its latency by busy-spinning. We calibrate the number of cycles for spinning as the number of cycles needed to write the same amount of data using non-temporal store (`MOVNTI`), which do not pollute the cache [24]. Our emulation is best-effort; we expect the actual `CLWB` instruction to perform better. For the experiments that follow, we use the ideal variant unless specified otherwise.

4.3 Experimental Setup

Workloads. On the primary server, we run the full transaction mix of the TPC-C [60] benchmark to test transactional read/write workloads. We fix the number of warehouses to the number of concurrent threads and give each thread a "home" warehouse. A major advantage of hot-standby solutions is the ability to serve read-only transactions to backup servers. Since the full TPC-C mix is a read-write workload, we run TPC-C's read-only transactions (Stock-Level and Order-Status, 1:1 breakdown) on backup servers.

System settings. To focus on evaluating the impact of network and system architecture, we keep all data memory-resident using `tmpfs`. Log data is still flushed through the storage interface. For all experiments we set the log buffer size to 16MB and ship the log at group commit boundaries. Each run lasts for 10 seconds, and is repeated for three times; we report the averages. Like most (physical) log shipping mechanisms, concurrency control and replication are not tightly coupled in Query Fresh. Our experiments use snapshot isolation; other isolation levels are transparently supported by Query Fresh, so we do not repeat experiments for them here.

Variants. We compare Query Fresh with other approaches described in Section 3.5 using end-to-end experiments. Then we conduct detailed experiments for Query Fresh to show the impact of individual design decisions; the details are described later. We compare four variants in end-to-end experiments:

- **Sync**: Traditional synchronous log shipping; log records are shipped upon group commit, and transactions are not considered committed until log records are made durable in all replicas.
- **Async**: Asynchronous log shipping that commits transactions regardless log persistence status on backup nodes.
- **Logical**: Logical log shipping that sends over the network only commands and parameters, instead of physical log records; log records are batched and shipped synchronously as in **Sync**.
- **Query Fresh**: Synchronous log shipping with modern hardware and append-only storage architecture.

All variants are implemented ERMIA for fair comparison. For Logical, we use command logging [39], an extreme case of logical logging that generates only one log record per transaction. It requires all transactions must be stored procedures. A log record only contains the unique ID of the stored procedure and other necessary parameters. We choose command logging for its small logging footprint (thus reduced network traffic). However, it does not support

[3] Truly NVRAM products (e.g., Intel 3D XPoint [14]) are available, but are only offered in PCIe interface as of this writing.

correct replay of cross-partition transactions in multi-version systems (see Section 3.5 for details) [39]. Studying how to guarantee correctness for command logging in multi-version systems is a separate issue and out of the scope of this paper. Nevertheless, we can still use it to measure the impact of log data size on performance, by slightly deviating from the TPC-C specification and restricting each worker to always access data associated with its home warehouse, i.e., no remote transactions. The benchmark may perform slightly better due to lack of conflicts, but it keeps the key benefit of command logging, which is the focus of our experiments.

Unless specified otherwise, we use four threads for log replay, and on backups threads that are not dedicated to log replay run read-only queries. Except Query Fresh, log replay is in the background and versions are fully instantiated without using replay arrays; Query Fresh uses replay pipelining. We run Sync, Async, and Logical with TCP/IP in the 10Gbps Ethernet network, and run Query Fresh in the 56Gbps InfiniBand network. This way, we show the impact of both system architecture and network on performance. To quantify the impact of individual design decisions, e.g., effect of the append-only architecture and replay policies, we run additional experiments using Query Fresh that turn on and off features (e.g., replay pipelining).

4.4 Network Bandwidth and Log Data Rate

We first calibrate our expectations by comparing the performance of a standalone server and a dual-node cluster. We focus on measuring the impact of network bandwidth and log data rate, i.e., how much bandwidth is needed for synchronous log shipping so that network is not a bottleneck. To achieve this, we turn off log replay and do not impose NVRAM delays. The backup does not run transactions; it only receives and persists log records. We then extrapolate the expected scalability of Query Fresh and other approaches, and compare our expectations with experimental results later.

Table 1 lists the throughput of a standalone server and its log rate. The standalone system scales and generates up to 1.42GB/s of log records. Such log rate well exceeds the bandwidth of a 10Gbps Ethernet. The 56Gbps InfiniBand network should in theory support up to five synchronous backups. As Section 3.1 explains, this is an inherent limitation of unicast. The log rate grows roughly at the same speed as the amount of parallelism grows. Since network bandwidth will likely be in the same ballpark with memory bandwidth [22], we estimate for larger servers, it is viable to at least have 1–2 synchronous backups. More backups can be added in a hierarchical architecture that is already widely in use today.

As the table shows, RDMA-based log shipping does not slow down the primary much, with 4–6% overhead over the standalone server. With 10Gbps Ethernet, the primary could perform up to ∼43% slower (16 threads). This indicates that 10Gbps Ethernet is unable to support any synchronous replica without significantly lowering the primary's performance. This experiment verifies the importance of network bandwidth for physical log shipping. Compared to TCP/IP, we do not observe significant performance gain from RDMA's kernel bypassing, either. This is largely because both interconnects are able to sustain high performance with bulk data transfer, which is usually the case for physical log shipping. However, the prospect of having safe RDMA over NVRAM (with protocol extensions) will likely make RDMA the preferred choice. Next, we expand our experiments to more (up to eight) nodes.

4.5 End-to-End Comparisons

Section 3.5 qualitatively compares related approaches. Now we compare them quantitatively. We focus on measuring (1) primary performance, (2) backup freshness, (3) resource utilization, and (4) implementation efforts in term of lines of code (LoC).

Table 1: Throughput and log data rate of a standalone server, vs. the primary's throughput in a dual-node cluster under 56Gbps InfiniBand RDMA and 10Gbps Ethernet TCP/IP.

Number of threads	Standalone (kTPS)	Log size (MB/s)	RDMA (kTPS)	TCP/IP (kTPS)
1	53.82	108.21	51.65	50.66
2	99.11	198.83	93.86	92.84
4	191.95	383.23	180.99	176.80
8	346.34	694.83	326.17	294.54
16	624.74	1456.62	586.80	336.20

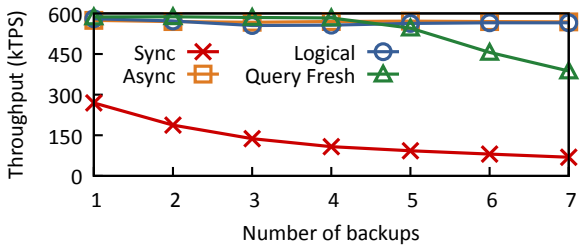

Figure 6: Primary throughput. Sync bottlenecks on the network. Query Fresh supports up to 4–5 synchronous replicas. Async and Logical keep high performance but sacrifices safety and has limited applicability/resource utilization, respectively.

Primary performance. We first measure the impact of the way (logical vs. physical) and timing (before vs. after committing locally) of log shipping. Figure 6 shows the throughput of the primary server under different approaches. As the figure shows, Query Fresh can support up to five *synchronous* backups while maintaining high performance for the primary. Once the log rate comes close to the network bandwidth (with five backups), primary throughput starts to drop. Sync bottlenecks on the network and cannot support any synchronous backups without significantly slowing down the primary. These results match our expectation in Section 4.4.

Async keeps high primary throughput regardless of the number of backups, because transactions are committed locally in the primary, before log records are sent to backups. Thus, it is possible to lose committed work and sacrifices both safety and freshness. Logical also preserves high primary performance across the x-axis in Figure 6, because command logging significantly reduces log data size, making it easy to support a large number of synchronous backups before saturating the network. However, it also exhibits various drawbacks as we have discussed in Section 3.5.

Freshness and utilization. Now we compare backup freshness and resource utilization of different approaches using two nodes; we obtained similar results with more nodes, so they are not shown here for brevity. We represent resource utilization by the percentage of CPU cores dedicated to read-only transactions. Freshness is measured by comparing the read view LSNs on the primary and backup. We synchronize the clock in each node with the same source, and take a snapshot of the read view LSN on each node every 20ms. We define the backup's *freshness score* at a given time point as $\frac{b}{p} \times 100\%$, where b and p are the read view LSN of the backup and primary, respectively. For example, if at time t the primary has a read view of 100, and the backup has finished replaying the log up to LSN 80, the freshness score will be 80%. A higher score indicates transactions can access a more recent snapshot of the database. Ideally, we can match the times between nodes and calculate freshness scores precisely. But it is difficult to obtain read

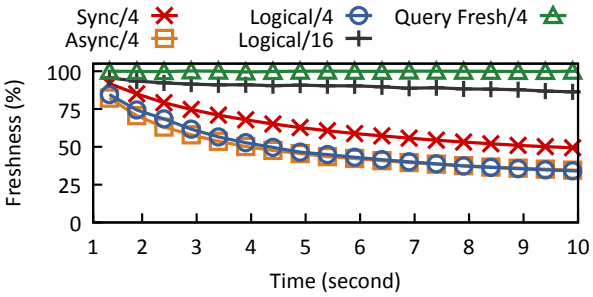

Figure 7: Backup freshness under 75% resource utilization (12 workers, 4 replay threads). Query Fresh keeps up with the primary ($\geq$ 99% freshness). Others must trade utilization for more freshness.

view LSNs on two nodes at the exact same time. So for calculation, we take the LSN at the nearest 500ms boundaries. This allows us to estimate freshness scores more easily.

Figure 7 shows the results. We also give the number of replay threads used for each variant (e.g., Sync with four replay threads is denote as "Sync/4"). Query Fresh keeps up with the primary and provides over 99% of freshness using four replay threads, i.e., it never lags behind more than 1% of the primary's read view. This shows the effectiveness of Query Fresh's append-only architecture and indirection, which allow very lightweight, parallel replay. Although Logical never bottlenecks on shipping the log, it cannot reach the same level of freshness of Query Fresh using fewer resources (four replay threads) than the primary (16 threads). To keep up with the primary, Logical must employ a similar number of threads as the primary solely for log replay (re-execution). As the figure shows, Logical/16 barely keeps up with the primary, but leaves no thread for read-only transactions with zero utilization.

Under Sync/4 and Async/4, replay cannot keep up with the log rate. The backup then falls behind the primary with a growing gap. During replay each version is fully instantiated, which is much more heavyweight than Query Fresh's replay mechanism. Sync shows better freshness than Logical and Async despite its low primary throughput (Figure 6). This is because the primary constantly blocks on network I/O waiting for log records to be delivered, which slows down forward processing. For the same reason, adding more replay threads in Sync and Async did not help much: there is not that much log data available for more replay threads to consume.

The above experiments fix the number of replay threads for all variants to four, thus fixing resource utilization for backups at 75% (with the remaining 12 threads as transaction workers). Only Query Fresh is able to sustain high freshness with 75% of utilization. Other approaches must trade utilization if more freshness is needed.

Implementation effort. Based on ERMIA, whose source code has more than 70kLoC, the core implementation of Query Fresh took around 1300LoC. The number for TCP-based physical log shipping (Sync and Async) is ~800LoC. Because the only difference between Sync and Async is whether log replay happens in background, their code differs very little, so we do not separate them here. The extra ~500LoC in Query Fresh are for implementing functionality related to log replay and data arrays. Our implementation of Logical took ~330LoC, but it is not a complete implementation that guarantees correct replay for cross-partition transactions. We expect a complete implementation for Logical will need much more code tightly integrated with concurrency control and the application, whereas Query Fresh requires no change to concurrency control or the application.

Summary. These results show that Query Fresh strikes a balance among primary performance, freshness, and resource utilization. It also keeps the easy-to-implement feature of log shipping, and does not require any change in the application. Next, we explore how each design decision impacts Query Fresh's behavior.

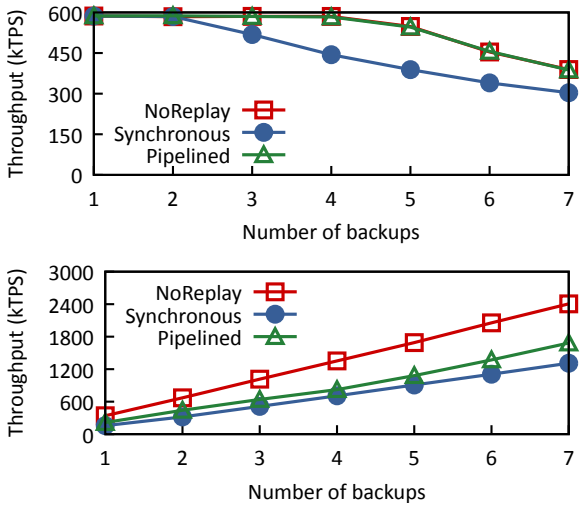

Figure 8: Throughput of the primary (top) and backups (bottom). Replay pipelining maintains high primary throughput by "hiding" log replay behind forward processing.

4.6 Effect of Replay Policy

We first explore the effectiveness of replay pipelining, by comparing it with synchronous replay described in Section 3.2. Figure 8(top) plots the primary's throughput. "NoReplay" denotes the variant where replay is disabled and reflects the amount of back pressure put on the primary by synchronous backups. Similar to what we have seen in the end-to-end experiments, after saturating the network (4–5 backups) throughput starts to drop significantly. Pipelined replay behaves similarly to NoReplay, because the overhead of log replay is hidden behind forward processing and is out of the critical path. However, synchronous replay does not scale after two backups with log replay on the critical path.

Figure 8(bottom) shows the corresponding aggregate throughput of backups. Each backup employs all of its physical threads. In NoReplay, all the 16 threads in each backup are used to run read-only transactions. Due to the lack of log replay and because we start with a warm database that has all records in memory, reading a version involves no I/O. Thus, NoReplay gives an upper bound on how fast read-only transactions can run on backups. Synchronous uses eight threads for log replay, while the number for Pipelined is four. Compared to Synchronous, Pipelined is left with more threads for read-only queries, thus achieving higher throughput.

As Figure 9(top) shows, Pipelined needs no more than four threads to keep up with the primary (after four backups network bandwidth becomes a scarcity). With 1–2 backups, Synchronous needs roughly half of the resources for log replay to keep up, as Figure 9(bottom) shows. Since Synchronous and Pipelined perform exactly the same amount of work for log replay, this experiment shows the importance of moving replay out of the critical path.

4.7 Effect of Indirection

Pipelined replay employs indirection, so log replay does not actually instantiate "real" versions. Now we quantify its effect by comparing with a variant that fully replays the log (denoted as "FullReplay"). FullReplay is exactly the same as Pipelined except

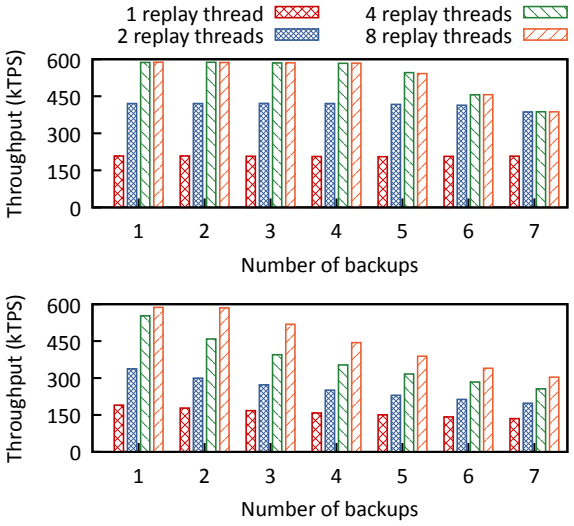

Figure 9: Throughput of the primary under pipelined (top) and synchronous replay policies.

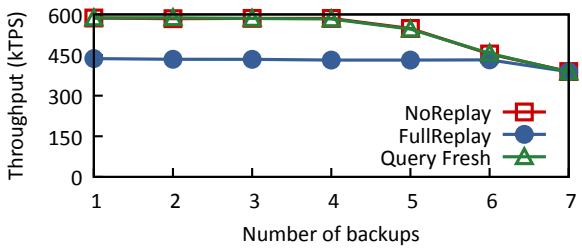

Figure 10: Throughput of the primary with and without indirection.

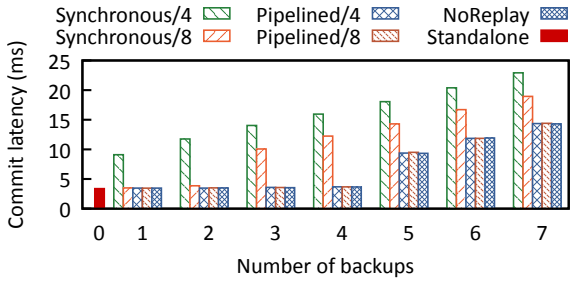

Figure 11: Commit latency on the primary. Query Fresh with replay pipelining incurs negligible overhead over the standalone case as it hides network I/O and replay behind local I/O on the primary.

that replay threads instantiate each version from the log record and install it on the data arrays, without using replay arrays. FullReplay employs multiple threads and can avoid replaying log records whose successors have already been replayed and installed on the data array. Figure 10 reflects the amount of back-pressure each variant could put on the primary. Both Query Fresh and FullReplay employ our pipelined log replay design. We use four replay threads and dedicate the remaining threads to read-only transactions. Compared to Query Fresh, FullReplay imposes significant overhead as it needs to fully instantiate versions,[4] making log replay a major bottleneck.

4.8 Bootstrapping

Now we measure the time needed to start a new backup, including the time for recovering from a recent checkpoint and replaying the log generated afterward. The primary first generates a checkpoint after started, and then starts processing transactions for five seconds. Then we start a new backup and measure the time needed to finish checkpoint recovery and replay. The new backup uses 16 threads for checkpoint recovery. Once it is done, it uses eight, four, and four threads for replaying catch-up log records, pipelined log replay, and running read-only transactions, respectively. The checkpoint size is 1.8GB, and the log data size is ∼6GB. Checkpoint recovery took 5.8 seconds, and replaying all the remaining catch-up log records took 24.21 seconds. So in total it took ∼30 seconds for the backup to replay and catch up with the primary. To replay catch-up records, the threads have to load data from storage. So it is slower than replay pipelining which scans the log buffer directly. Moreover, our current implementation stores the whole checkpoint and log data in two separate files. Although we parallelize the replay process, using multiple threads to operate on the same file concurrently incurs severe bottleneck in the file system [43]. We expect an optimized implementation will be able to achieve even shorter catch-up time.

4.9 Commit Latency

Synchronous log shipping often greatly increase commit latency on the primary in a slow network. This is avoided in Query Fresh by overlapping log shipping with local I/O and replay pipelining. We collect the primary's commit latency numbers for two Query Fresh variants using pipelined and synchronous replay, with a varying number of replay threads (denoted as "Pipelined/4" and so on). The group commit boundary is set to 4MB, i.e., log shipping is triggered whenever 4MB of new log data is generated. Other events (e.g., commit queue full or timeout) could also trigger log shipping, but we found the system is most sensitive to log data size.

Figure 11 shows the results. As a baseline, the commit latency of the standalone case (no replication) is 3.38ms. Synchronous replay exhibits ∼1.04–4.74× higher latency, even before saturating the network (four backups). With replay pipelining, Query Fresh exhibits at most 1.16× of the standalone latency before the network is saturated. Once the network is saturated ($\geq$ 5 backups), however, all variants starts to add non-negligible latency on the commit path.

4.10 Persistence Delay

Now we quantify the impact of delays caused by persisting log records on backups using pipelined replay. We compare three types of variants: a variant that adds no delay (denoted as "Ideal"), a variant that does not use NVRAM ("No-NVRAM") and four other variants that impose NVRAM delays. No-NVRAM assumes DRAM log buffer. Both Ideal and No-NVRAM use four replay threads, leaving 12 for read-only queries. For the remaining four variants, we use `CLFLUSH/CLWB-EMU` and vary the number of replay threads between four and eight (denoted as "CLFLUSH/4" and so on); persist-upon-replay is enabled to spread the work of persisting the log records to multiple threads.

Figure 12(top) shows primary's throughput. Ideal maintains high throughput with 1–5 backups. With four and more backups, network gradually becomes the bottleneck. Since CLWB-EMU does not evict cachelines, both CLWB-EMU variants achieve performance that is close to Ideal's with up to four backups. With five and more backups, CLWB-EMU performs up to 9% slower than Ideal. CLFLUSH/8 incurs ∼5–26% overhead over Ideal, due to the cache misses caused by `CLFLUSH` during log replay. With fewer replay

[4] We use a NUMA-aware allocator in ERMIA [32]. Our profiling results show that memory allocation is not a bottleneck.

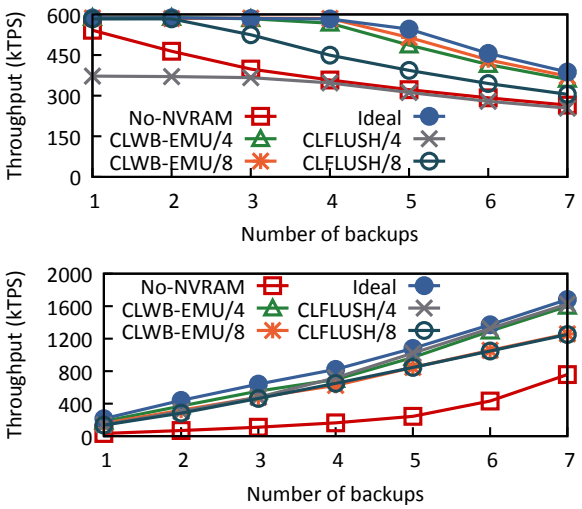

Figure 12: Throughput of the primary (top) and backups (bottom) with varying NVRAM delays.

threads, CLFLUSH/4 could add as much as 42% overhead on Ideal. These results show that data persistence is the bottleneck, rather than log replay, as we have shown that four threads are enough for pipelined replay to keep up with the primary.

Figure 12(bottom) shows the read-only transactions' aggregate throughput on backups. Despite CLFLUSH/4 shows much lower primary performance than CLWB-EMU/4, it gave higher backup throughput, because slower replay reduces the chance of a transaction seeing a newer version posted on the replay arrays. More reads can be directly served through the data arrays without having to read versions in the "gap" between the replay and data arrays from storage. However, read-only transactions will access stale data. Variants with eight replay threads (CLFLUSH/8 and CLWB-EMU/8) exhibit lower backup throughput, due to the reduced number of threads for read-only transactions and higher chance for readers to have to load versions from storage since replay is faster.

5. RELATED WORK

Primary-backup replication. Many production systems implement log shipping [21, 47, 48, 52, 57, 62]. KuaFu [68] enables parallel log replay by tracking dependencies. SHADOW systems [31] offload log shipping and other jobs to the storage layer in the cloud, reducing data duplication. Amazon Aurora [62] also offloads redo processing to the storage layer, to reduce network load. RemusDB [44] provides high availability at the virtualization layer, requiring little or no change to the database engine. Another approach is deterministic execution [58, 59] and logical log shipping [39]. A concurrent effort by Qin et al. [51] proposes a replay scheme for serializable multi-versioned systems. The scheme inherits all the benefits of logical log shipping and expands deterministic replay to multi-version systems, but is tightly coupled with concurrency control and only supports stored procedures. BatchDB [38] supports hybrid workloads using dedicated OLTP and OLAP components that operate on the same or separate nodes. It uses logical logging for OLTP and propagates updates explicitly to the OLAP node.

Utilizing fast networks. The use of fast networks are being actively studied in key-value stores [27, 45] and database engines. Rödiger et al. [54] design a distributed query engine, tackling problems associated with TCP/IP. Barthels et al. [5] use RDMA to distribute and partition data efficiently for in-memory hash join. Liu et al. [37] employ RDMA to devise efficient data shuffling operators. DrTM+R [10] combines hardware transactional memory and RDMA to handle distributed transactions. Binnig et al. [9] suggest a redesign of distributed databases using RDMA, and propose the network-attached memory (NAM) architecture that logically decouples compute and storage. NAM-DB [70] is a NAM-based distributed database engine. It uses snapshot isolation and mitigates the timestamp bottleneck using vector clocks. FaRM [18] is a more general platform that exposes memory as a shared address space. FaSST [28] uses two-sided RDMA for fast RPC, as using a large number of queue pairs under one-sided RDMA can limit scalability.

Append-only storage. Distributed shared logs also abstract storage to be append-only, but assumes a different model. CORFU [3] organizes flash chips as a global, shared log, on top of which transactions execute optimistically [6, 7, 8]. Tango [4] provides mechanisms to build in-memory data structures backed by a shared log.

Many main-memory systems use direct memory pointers [33, 61], instead of RIDs. File systems such as BPFS [12] also use indirection. Several recent studies note the usefulness of indirection in main-memory environments [66]. The Bw-Tree [36] uses indirection to aid the build of lock free B+-tree operations. Sadoghi et al. [55] use indirection and store RIDs in indexes to reduce index maintenance cost. ERMIA [32] adopts the same philosophy for easier logging, recovery and low index maintenance cost.

Logging/recovery and NVRAM. Much recent work focuses on accelerating recovery. Graefe proposes on-demand per-page redo for faster recovery [19]. PACMAN [67] uses static analysis to obtain application-specific information and speed up recovery. Adaptive logging [69] combines physical and logical logging to reduce recovery time. FPTree [49] is a hybrid index that only keeps leaf nodes in NVRAM. Several recent designs leverage NVRAM for better logging/recovery, such as alleviating the centralized logging bottleneck [64] and providing near-instant recovery [2, 50].

6. CONCLUSION

Hot standby systems often exhibit a freshness gap between the primary and backup servers, for two reasons. First, network can easily be a bottleneck, limiting the speed of log shipping and making most backups operate in asynchronous mode, especially so for modern main-memory OLTP engines. Second, the traditional dual-copy architecture stores data in two permanent places: the log and the "real" database, mandating (expensive) log replay before data can become accessible to read-only transactions on backups.

We propose Query Fresh to solve these problems with modern hardware and software architecture re-design. Query Fresh leverages RDMA over fast network and NVRAM for fast log shipping, and employs append-only storage with indirection for lightweight, parallel log replay. The result is a hot standby system that provides strong safety, freshness, and high resource utilization. Our evaluation using an 8-node cluster shows that with 56Gbps InfiniBand, Query Fresh supports up to 4–5 synchronous backups without significantly lowering the primary's performance.

Acknowledgements

We would like the thank the team behind the Apt cluster at the University of Utah and Hewlett Packard Labs for their help on provisioning and configuring machines for our experiments. We gratefully acknowledge Kangnyeon Kim, Angela Demke Brown, and colleagues at Amazon Web Services for their invaluable feedback and editorial help. We also thank the anonymous reviewers for their constructive comments and suggestions.

7. REFERENCES

[1] AgigaTech. AgigaTech Non-Volatile RAM. 2017. http://www.agigatech.com/nvram.php.

[2] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. *NSDI*, 2012.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. *SOSP*, pages 325–340, 2013.

[5] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. *SIGMOD*, pages 1463–1475, 2015.

[6] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. *SIGMOD*, pages 1295–1309, 2015.

[7] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. *CIDR*, 2011.

[8] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[9] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[10] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. *EuroSys*, pages 26:1–26:17, 2016.

[11] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun. Remote direct memory access over the converged enhanced Ethernet fabric: Evaluating the options. *Hot Inteconnects*, pages 123–130, 2009.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. *SOSP*, pages 133–146, 2009.

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

[14] R. Crooke and M. Durcan. A revolutionary breakthrough in memory technology. *Intel 3D XPoint launch keynote*, 2015.

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

[16] C. Diaconu et al. Hekaton: SQL server's memory-optimized OLTP engine. *SIGMOD*, pages 1243–1254, 2013.

[17] C. Douglas. RDMA with PMEM: Software mechanisms for enabling access to remote persistent memory. *Storage Developer Conference*, 2015. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf.

[18] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. *NSDI*, pages 401–414, 2014.

[19] G. Graefe. Instant recovery for data center savings. *SIGMOD Record*, 44(2):29–34, Aug. 2015.

[20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[21] IBM. High availability through log shipping. *IBM DB2 9.7 for Linux, UNIX, and Windows documentation*, 2015.

[22] InfiniBand Trade Association. InfiniBand roadmap. 2016. http://www.infinibandta.org/content/pages.php?pg=technology_overview.

[23] Intel Corporation. Intel data direct I/O technology (Intel DDIO): A primer. 2012.

[24] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. 2015.

[25] JEDEC. DDR3 SDRAM standard. 2012. http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[26] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1):681–692, 2010.

[27] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM*, pages 295–306, 2014.

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

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

[30] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. *ISCA*, 2017.

[31] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan. Database high availability using shadow systems. *SoCC*, pages 209–221, 2015.

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

[33] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. *SIGMOD*, pages 691–706, 2015.

[34] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.

[35] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A cache/storage subsystem for modern hardware. *PVLDB*, 6(10):877–888, 2013.

[36] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. *ICDE*, pages 302–313, 2013.

[37] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. *EuroSys*, pages 48–63, 2017.

[38] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. *SIGMOD*, pages 37–50, 2017.

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

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

[41] Mellanox Technologies. RDMA aware networks programming user manual. 2015.

[42] Mellanox Technologies. RDMA over converged ethernet (RoCE) - an efficient, low-cost, zero copy implementation. 2017. http://www.mellanox.com/page/products_dyn?product_family=79.

[43] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim.

Understanding manycore scalability of file systems. *USENIX ATC*, pages 71–85, 2016.

[44] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *PVLDB*, 4(11):738–748, 2011.

[45] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. *USENIX ATC*, pages 103–114, 2013.

[46] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery met- hod supporting fine-granularity locking and partial roll backs using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[47] Oracle. TimesTen in-memory database replication guide. *Oracle Database Online Documentation*, 2014.

[48] Oracle. Chapter 17 Replication. *MySQL 5.7 Reference Manual*, 2015.

[49] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. *SIGMOD*, pages 371–386, 2016.

[50] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. *CIDR*, 2015.

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

[52] P. S. Randal. High availability with SQL Server 2008. *Microsoft White Papers*, 2009. https://technet.microsoft.com/en-us/library/ee523927.aspx.

[53] R. Ricci, G. Wong, L. Stoller, K. Webb, J. Duerig, K. Downie, and M. Hibler. Apt: A platform for repeatable research in computer science. *SIGOPS Oper. Syst. Rev.*, 49(1):100–107, Jan. 2015. http://docs.aptlab.net/.

[54] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[55] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *PVLDB*, 6(11):997–1008, 2013.

[56] T. Talpey. RDMA extensions for remote persistent memory access. *12th Annual Open Fabrics Alliance Workshop*, 2016. https://www.openfabrics.org/images/eventpresos/2016presentations/215RDMAforRemPerMem.pdf.

[57] The PostgreSQL Global Development Group. Chapter 25. High Availability, Load Balancing, and Replication. *PostgreSQL 9.4.4 Documentation*, 2015.

[58] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1-2):70–80, 2010.

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

[60] TPC. TPC benchmark C (OLTP) standard specification, revision 5.11, 2010. http://www.tpc.org/tpcc.

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

[62] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. *SIGMOD*, pages 1041–1052, 2017.

[63] Viking Technology. DDR4 NVDIMM. 2017. http://www.vikingtechnology.com/products/nvdimm/ddr4-nvdimm/.

[64] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[65] T. Wang, R. Johnson, and I. Pandis. Fresh replicas through append-only storage. *HPTS*, 2015. http://www.hpts.ws/papers/2015/lightning/append-only-log-ship.pdf.

[66] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[67] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Fast failure recovery for main-memory DBMSs on multicores. *SIGMOD*, pages 267–281, 2017.

[68] M. Yang, D. Zhou, C. Kuo, C. Hong, L. Zhang, and L. Zhou. KuaFu: Closing the parallelism gap in database replication. *ICDE 2013*, pages 1186–1195, 2013.

[69] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. *SIGMOD*, pages 1119–1134, 2016.

[70] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transactions can scale. *PVLDB*, 10(6):685–696, 2017.

[71] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. *ASPLOS*, pages 3–18, 2015.