Write-Behind Logging

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Matthew Perron
Carnegie Mellon University
mperron@cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

The design of the logging and recovery components of database management systems (DBMSs) has always been influenced by the difference in the performance characteristics of volatile (DRAM) and non-volatile storage devices (HDD/SSDs). The key assumption has been that non-volatile storage is much slower than DRAM and only supports block-oriented read/writes. But the arrival of new non-volatile memory (NVM) storage that is almost as fast as DRAM with fine-grained read/writes invalidates these previous design choices.

This paper explores the changes that are required in a DBMS to leverage the unique properties of NVM in systems that still include volatile DRAM. We make the case for a new logging and recovery protocol, called write-behind logging, that enables a DBMS to recover nearly instantaneously from system failures. The key idea is that the DBMS logs what parts of the database have changed rather than how it was changed. Using this method, the DBMS flushes the changes to the database before recording them in the log. Our evaluation shows that this protocol improves a DBMS's transactional throughput by 1.3×, reduces the recovery time by more than two orders of magnitude, and shrinks the storage footprint of the DBMS on NVM by 1.5×. We also demonstrate that our logging protocol is compatible with standard replication schemes.

1. INTRODUCTION

A DBMS ensures that the database state is not corrupted due to an application, operating system, or device failure [19]. It ensures the durability of all updates made by a transaction by writing changes out to durable storage, such as an HDD, before returning an acknowledgement back to the application. Such storage devices, however, are much slower than DRAM (especially for random writes) and only support bulk data transfers as blocks. In contrast, a DBMS can quickly read and write a single byte from a volatile DRAM device, but all data is lost after the system restarts or there is a power failure.

These differences between the two types of storage are a major factor in the design of DBMS architectures. For example, disk-oriented DBMSs employ different data layouts optimized for non-volatile and volatile storage. This is because of the performance gap between sequential and random accesses in HDD/SSDs. Further, DBMSs try to minimize random writes to the disk due to its high random write latency. During transaction processing, if the DBMS were to overwrite the contents of the database before committing the transaction, then it must perform random writes to the database at multiple locations on disk. It works around this constraint by flushing the transaction's changes to a separate log on disk with only sequential writes on the critical path of the transaction. This method is referred to as *write-ahead logging* (WAL).

But emerging *non-volatile memory* (NVM) technologies are poised to upend these assumptions. NVM storage devices support low latency reads and writes similar to DRAM, but with persistent writes and large storage capacity like a SSD [9]. The CPU can also access NVM at cache line-granularity. This means that the canonical approaches for DBMS logging and recovery that assume slower storage are incompatible with this new hardware landscape [14].

In this paper, we present a new protocol, called **write-behind logging** (WBL), that is designed for a hybrid storage hierarchy with NVM and DRAM. We demonstrate that tailoring these algorithms for NVM not only improves the runtime performance of the DBMS, but it also enables it to recovery nearly instantaneously from failures. The way that WBL achieves this is by tracking *what* parts of the database have changed rather than *how* it was changed. Using this logging method, the DBMS can flush the changes to the database before recording them in the log. By ordering writes to NVM correctly, the DBMS can guarantee that all transactions are durable and atomic. This allows the DBMS to write less data per transaction, thereby improving an NVM device's lifetime [6].

To evaluate our approach, we implemented it in the Peloton [2] in-memory DBMS and compared it against WAL using three storage technologies: NVM, SSD, and HDD. These experiments show that WBL with NVM improves the DBMS's throughput by 1.3× while also reducing the database recovery time and the overall system's storage footprint. Our results also show that WBL only achieves this when the DBMS uses NVM; the DBMS actually performs worse than WAL when WBL is deployed on the slower, block-oriented storage devices (i.e., SSD, HDD). This is expected since our protocol is explicitly designed for fast, byte-addressable NVM. We also deployed Peloton in a multi-node configuration and demonstrate how to adapt WBL to work with standard replication methods.

The remainder of this paper is organized as follows. We begin in Section 2 with an overview of the recovery principles of a DBMS and how NVM affects them. We then discuss logging and recovery implementations in modern DBMSs. We start with the ubiquitous WAL protocol in Section 3, followed by our new WBL method in Section 4. In Section 5, we discuss how this logging protocol can be used in replicated environments. We then provide an overview of the NVM hardware emulator that we use in our experiments in Section 6. We present our experimental evaluation in Section 7. We conclude with a discussion of related work in Section 8.

Proceedings of the VLDB Endowment, Vol. 10, No. 4
Copyright 2016 VLDB Endowment 2150-8097/16/12.

2. BACKGROUND

We begin with an overview of DBMS recovery principles and discuss how emerging NVM technologies affect them. We then make the case for adapting the logging protocol for NVM.

2.1 Recovery Principles

A DBMS guarantees the integrity of a database by ensuring (1) that all of the changes made by committed transactions are durable and (2) that none of the changes made by aborted transactions or transactions that were running at the point of a failure are visible after recovering from the failure. These two constraints are referred to as *durability of updates* and *failure atomicity*, respectively [5, 19].

There are three types of failures that a DBMS must protect against: (1) transaction failure, (2) system failure, and (3) media failure. The first happens when a transaction is aborted either by the DBMS due to a conflict with another transaction or because the application chose to do so. System failures occur due to bugs in the DBMS/OS or hardware failures. Finally, in the case of a data loss or corruption on the non-volatile storage, the DBMS must recover the database from an archival version. It must also remove the updates of incomplete transactions to satisfy the failure atomicity constraint.

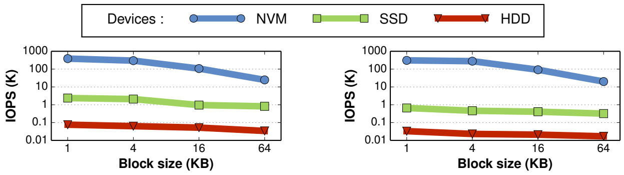
Almost every DBMS adopts the *steal* and *no-force* policies for managing the data stored in the volatile buffer pool and the database on durable storage [19]. The former policy allows a DBMS to flush the changes of uncommitted transactions at any time. With the latter, the DBMS is not required to ensure that the changes made by a transaction are propagated to the database when it commits. Instead, the DBMS records a transaction's changes to a *log* on durable storage before sending an acknowledgement to the application. Further, it flushes the modifications made by uncommitted transactions to the log before propagating them to the database.

During recovery, the DBMS uses the log to ensure the atomicity and durability properties. The recovery algorithm reverses the updates made by failed transactions using their *undo* information recorded in the log. In case of a system failure, the DBMS first ensures the durability of updates made by committed transactions by reapplying their *redo* information in the log on the database. Afterwards, the DBMS uses the log's undo information to remove the effects of transactions that were aborted or active at the time of the failure. DBMSs can handle media failures by storing the database, the log, and the archival versions of the database (i.e., checkpoints) on multiple durable storage devices.

2.2 Non-Volatile Memory Database Systems

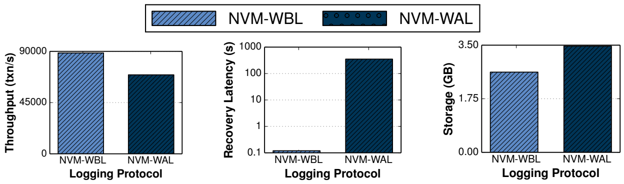
Modern HDDs are based on the same high-level design principles from the 1960s: a magnetic platter spins and an arm reads data off of it with a block-granularity (typically 4 KB). Since moving the platter and the arm is a mechanical process, these drives are the slowest of all the durable storage devices. Sequential reads and writes to the device are faster than random accesses as they do not require the arm to be re-positioned. HDDs have a high data density and thus offer a lower storage price per capacity.

SSDs are faster than HDDs because they use non-volatile NAND storage cells; their read and write latencies are up to three orders of magnitude lower than an average HDD. But there are three problems that make SSDs less than ideal for DBMSs. Foremost is that they only support block-oriented access to data. Each storage cell in a SSD can only be written to a fixed number of times before it can no longer reliably store the data. Lastly, SSDs are currently 3–10× more expensive per GB than an HDD.



(a) Sequential Writes **(b)** Random Writes

Figure 1: I/O Performance – Synchronous file write throughput obtained on different storage devices including emulated NVM, SSD, and HDD.



(a) Throughput **(b)** Recovery Time **(c)** Storage Footprint

Figure 2: WBL vs. WAL – The throughput, recovery time, and storage footprint of the DBMS for the YCSB benchmark with the write-ahead logging and write-behind logging protocols.

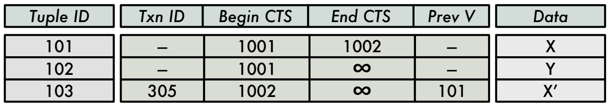
The performance of DBMSs that use HDDs/SSDs for durable storage is constrained by the speed with which they persist changes to the log stored on these devices. This is because there is a large gap in the read/write latencies of DRAM and HDDs/SSDs, as well as a mismatch in their data access granularities (i.e., coarse-grained block writes vs. fine-grained byte-level writes).

NVM technologies, such as phase change memory, STT-MRAM, and memristors, provide low-latency, byte-addressable loads and stores [6]. In contrast to the other durable storage devices that use the PCIe or SATA interfaces, NVM can be plugged into DIMM slots to deliver higher bandwidths and lower latencies to CPUs. Consequently, it can help reduce the performance overhead associated with persisting the changes on durable storage.

To better understand the performance characteristics of these devices, we benchmark them using `fio` [8] with different access patterns. In this experiment, we measure the write throughput of a single thread performing synchronous writes to a large file (64 GB) stored on a HDD, SSD, and emulated NVM [17]. The results in Figure 1 show that NVM delivers more than two orders of magnitude higher write throughput compared to the SSD and HDD. More importantly, the gap between sequential and random write throughput of NVM is much smaller.

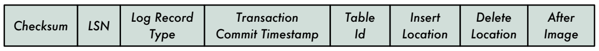
Although the performance advantages of NVM are obvious, it is still not clear how to make full use of it in a DBMS running on a *hybrid storage hierarchy* with both DRAM and NVM. Previous work has shown that optimizing the storage methods for NVM improves both the DBMS performance and the lifetime of the storage device [6]. These techniques, however, cannot be employed in a hybrid storage hierarchy, as they target an NVM-only system. Another line of research focuses on using NVM only for storing the log and managing the database still on disk [22]. This is a more cost-effective solution, as the cost of NVM devices are expected to be higher than that of disk. But this approach only leverages the low-latency sequential writes of NVM, and does not exploit its ability to efficiently support random writes and fine-grained data access. Given this, we contend that it is better to employ logging and recovery algorithms that are designed for NVM.

We designed such an approach that we call write-behind logging (WBL). Before we present WBL in Section 4, we first show its benefits for a DBMS running with NVM. We compared it against the canonical *write-ahead logging* (WAL) protocol that is used in most DBMSs today [19]. For this microbenchmark, we executed a write-heavy variation of the YCSB workload on Peloton. We defer the description of our experiment environment until Section 7. The results in Figure 2 show that WBL improves the DBMS's throughput



Tuple ID	Txn ID	Begin CTS	End CTS	Prev V	Data
101	–	1001	1002	–	X
102	–	1001	∞	–	Y
103	305	1002	∞	101	X′

Figure 3: Tuple Version Meta-data – The additional data that the DBMS stores to track tuple versions in an MVCC protocol.



Checksum	LSN	Log Record Type	Transaction Commit Timestamp	Table Id	Insert Location	Delete Location	After Image

Figure 4: Structure of WAL Record – Structure of the log record constructed by the DBMS while using the WAL protocol.

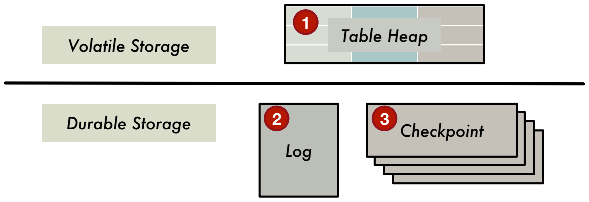


Figure 5: WAL Commit Protocol – The ordering of writes from the DBMS to durable storage while employing the WAL protocol.

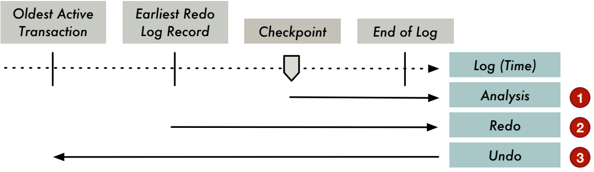


Figure 6: WAL Recovery Protocol – The phases of the recovery protocol.

by 1.3× over WAL, while also reducing the recovery time by over 100× and storage footprint on NVM by 1.5×.

To appreciate why WBL is better than WAL when using NVM, we now discuss how WAL is implemented in both disk-oriented and in-memory DBMSs.

3. WRITE-AHEAD LOGGING

The most well-known recovery method based on WAL is the ARIES protocol developed by IBM in the 1990s [28]. ARIES is a *physiological logging* protocol where the DBMS combines a physical redo process with a logical undo process [19]. During normal operations, the DBMS records transactions' modifications in a durable log that it uses to restore the database after a crash.

In this section, we provide an overview of ARIES-style WAL. We begin with discussing the original protocol for a disk-oriented DBMS and then describe optimizations for in-memory DBMSs. Our discussion is focused on DBMSs that use the multi-version concurrency control (MVCC) protocol for scheduling transactions [7, 29]. MVCC is the most widely used concurrency control scheme in DBMSs developed in the last decade, including Hekaton [16], MemSQL, and HyPer. The DBMS records the *versioning* meta-data alongside the tuple data, and uses it determine whether a tuple version is visible to a transaction. When a transaction starts, the DBMS assigns it a unique *transaction identifier* from a monotonically increasing global counter. When a transaction commits, the DBMS assigns it a unique *commit timestamp* by incrementing the timestamp of the last committed transaction. Each tuple contains the following meta-data:

- **TxnId:** A placeholder for the identifier of the transaction that currently holds a latch on the tuple.
- **BeginCTS & EndCTS:** The commit timestamps from which the tuple becomes visible and after which the tuple ceases to be visible, respectively.
- **PreV:** Reference to the previous version (if any) of the tuple.

Figure 3 shows an example of this versioning meta-data. A tuple is visible to a transaction if and only if its last visible commit timestamp falls within the `BeginCTS` and `EndCTS` fields of the tuple. The DBMS uses the previous version field to traverse the version chain and access the earlier versions, if any, of that tuple. In Figure 3, the first two tuples are inserted by the transaction with commit timestamp `1001`. The transaction with commit timestamp `1002` updates the tuple with `ID 101` and marks it as deleted. The newer version is stored with `ID 103`. Note that the `PreV` field of the third tuple refers to the older version of tuple. At this point in time, the transaction with identifier `305` holds a latch on the tuple with `ID 103`. See [7, 26] for a more detailed description of in-memory MVCC.

We now begin with an overview of the runtime operation of the DBMS during transaction processing and its commit protocol. Table 1 lists the steps in a WAL-based DBMS to execute database operations, process transaction commits, and take checkpoints. Later, in Section 4, we present our WBL protocol for NVM systems.

3.1 Runtime Operation

For each modification that a transaction makes to the database, the DBMS creates a log record that corresponds to that change. As shown in Figure 4, a log record contains a unique log sequence number (LSN), the operation associated with the log record (i.e., `INSERT`, `UPDATE`, or `DELETE`), the transaction identifier, and the table modified. For `INSERT` and `UPDATE` operations, the log record contains the location of the inserted tuple or the newer version. Each record also contains the *after-images* (i.e., new values) of the tuples modified, as shown in Table 1. In case of `UPDATE` and `DELETE` operations, it contains the location of the older version or the deleted tuple, respectively. This is known as the *before-images* (i.e., old values) of the modified tuples and is used to ensure failure atomicity.

A disk-oriented DBMS maintains two meta-data tables at runtime that it uses for recovery. The first is the *dirty page table* (DPT) that contains the modified pages that are in DRAM but have not been propagated to durable storage. Each of these pages has an entry in the DPT that marks the log record's LSN of the oldest transaction that modified it. This allows the DBMS to identify the log records to replay during recovery to restore the page. The second table is the *active transaction table* (ATT) that tracks the status of the running transactions. This table records the LSN of the latest log record of all active transactions. The DBMS uses this information to undo their changes during recovery.

To bound the amount of work to recover a database after a restart, the DBMS periodically takes checkpoints at runtime. ARIES uses *fuzzy checkpointing* where the checkpoint can contain the effects of both committed and uncommitted transactions [28]. Consequently, the DBMS must write out the DPT and ATT as a part of the checkpoint so that it can restore committed transactions and undo uncommitted transactions during recovery. After all the log records associated with a transaction are safely persisted in a checkpoint, the DBMS can remove those records from the log.

With an in-memory DBMS, transactions access tuples through pointers without indirection through a buffer pool [7]. The ARIES protocol can, therefore, be simplified and optimized for this architecture. Foremost is that a MVCC DBMS does not need to perform fuzzy checkpointing [34]. Instead, it constructs transactionally-consistent checkpoints that only contain the changes of committed transactions by skipping the modifications made by transactions that began after the checkpoint operation started. Hence, a MVCC DBMS neither stores the before-images of tuples in the log nor tracks dirty data (i.e., DPT) at runtime. Its recovery component, however, maintains an ATT that tracks the LSN of the latest log record written by each active transaction.

3.2 Commit Protocol

We now describe how a WAL-based DBMS processes and commits transactions. When a transaction begins, the DBMS creates an entry in the ATT and sets it status as *active*. For each modification that the transaction makes to the database, the DBMS constructs the corresponding log record and appends it to the log buffer. It then updates the LSN associated with the transaction in the ATT.

The DBMS flushes all the log records associated with a transaction to durable storage (using the `fsync` command) before committing the transaction. This is known as *synchronous logging*. Finally, the DBMS marks the status of the transaction in the ATT as *committed*. The ordering of writes from the DBMS to durable storage while employing WAL is presented in Figure 5. The changes are first applied to the table heap and the indexes residing in volatile storage. At the time of commit, WAL requires that the DBMS flush all the modifications to the durable log. Then, at some later point the DBMS writes the changes to the database in its next checkpoint.

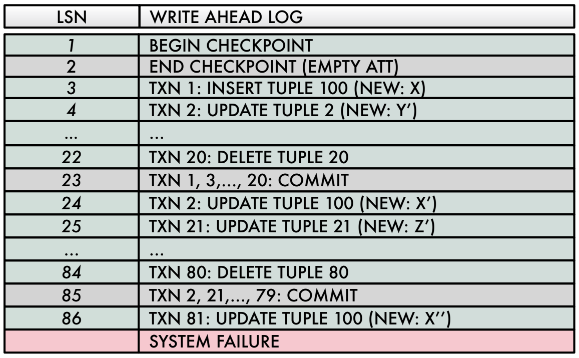
As transactions tend to generate multiple log records that are each small in size, most DBMSs use *group commit* to minimize the I/O overhead [15]. It batches the log records for a group of transactions in a buffer and then flushes them together with a single write to durable storage. This improves the transactional throughput and amortizes the synchronization overhead across multiple transactions.

3.3 Recovery Protocol

The traditional WAL recovery algorithm (see Figure 6) comprises of three phases: (1) analysis, (2) redo, and (3) undo. In the *analysis phase*, the DBMS processes the log starting from the latest checkpoint to identify the transactions that were active at the time of failure and the modifications associated with those transactions. In the subsequent *redo phase*, the DBMS processes the log forward from the earliest log record that needs to be redone. Some of these log records could be from transactions that were active at the time of failure as identified by the analysis phase. During the final *undo phase*, the DBMS rolls back uncommitted transactions (i.e., transactions that were active at the time of failure) using the information recorded in the log. This recovery algorithm is simplified for the MVCC DBMS. During the redo phase, the DBMS skips replaying the log records associated with uncommitted transactions. This obviates the need for an undo phase.

Figure 7 shows the contents of the log after a system failure. The records contain the after-images of the tuples modified by the transactions. At the time of system failure, only transactions `80` and `81` are uncommitted. During recovery, the DBMS first loads the latest checkpoint that contains an empty ATT. It then analyzes the log to identify which transactions must be redone and which are uncommitted. During the redo phase, it reapplies the changes made by transactions committed since the latest checkpoint. It skips the records associated with the uncommitted transactions `80` and `81`. After recovery, the DBMS can start executing new transactions.

Correctness: For active transactions, the DBMS maintains the before-images of the tuples they modified. This is sufficient to reverse the changes of any transaction that aborts. The DBMS ensures that the log records associated with a transaction are forced to durable storage before it is committed. To handle system failures during recovery, the DBMS allows for repeated undo operations. This is feasible because it maintains the undo information as before-images and not in the form of compensation log records [5, 19].



LSN	WRITE AHEAD LOG
1	BEGIN CHECKPOINT
2	END CHECKPOINT (EMPTY ATT)
3	TXN 1: INSERT TUPLE 100 (NEW: X)
4	TXN 2: UPDATE TUPLE 2 (NEW: Y′)
...	...
22	TXN 20: DELETE TUPLE 20
23	TXN 1, 3,..., 20: COMMIT
24	TXN 2: UPDATE TUPLE 100 (NEW: X′)
25	TXN 21: UPDATE TUPLE 21 (NEW: Z′)
...	...
84	TXN 80: DELETE TUPLE 80
85	TXN 2, 21,..., 79: COMMIT
86	TXN 81: UPDATE TUPLE 100 (NEW: X′′)
	SYSTEM FAILURE

Figure 7: WAL Example – Contents of the WAL during recovery.

Although WAL supports efficient transaction processing when memory is volatile and durable storage cannot support fast random writes, it is inefficient for NVM storage [6]. Consider a transaction that inserts a tuple into a table. The DBMS first records the tuple's contents in the log, and it later propagates the change to the database. With NVM, the logging algorithm can avoid this unnecessary data duplication. We now describe the design of such an algorithm geared towards a DBMS running on a hybrid storage hierarchy comprising of DRAM and NVM.

4. WRITE-BEHIND LOGGING

Write-behind logging (WBL) leverages fast, byte-addressable NVM to reduce the amount of data that the DBMS records in the log when a transaction modifies the database. The reason why NVM enables a better logging protocol than WAL is three-fold. Foremost, the write throughput of NVM is more than an order of magnitude higher than that of an SSD or HDD. Second, the gap between sequential and random write throughput of NVM is smaller than that of older storage technologies. Finally, individual bytes in NVM can be accessed by the processor, and hence there is no need to organize tuples into pages or go through the I/O subsystem.

WBL reduces data duplication by flushing changes to the database in NVM during regular transaction processing. For example, when a transaction inserts a tuple into a table, the DBMS records the tuple's contents in the database *before* it writes any associated meta-data in the log. Thus, the log is always (slightly) behind the contents of the database, but the DBMS can still restore it to the correct and consistent state after a restart.

We begin this section with an overview of the runtime operations performed by a WBL-based DBMS. We then present its commit protocol and recovery algorithm. Table 1 provides a summary of the steps during runtime, recovery, and checkpointing. Although our description of WBL is for MVCC DBMSs, we also discuss how to adapt the protocol for a single-version system.

4.1 Runtime Operation

WBL differs from WAL in many ways. Foremost is that the DBMS does not construct log records that contain tuple modifications at runtime. This is because the changes made by transactions are guaranteed to be already present on durable storage before they commit. As transactions update the database, the DBMS inserts entries into a *dirty tuple table* (DTT) to track their changes. Each entry in the DTT contains the transaction's identifier, the table modified, and additional meta-data based on the operation associated with the change. For `INSERT` and `DELETE`, the entry only contains the location of the inserted or deleted tuple, respectively. Since `UPDATE`s are executed as a `DELETE` followed by an `INSERT` in MVCC, the entry contains the location of the new and old version of the tuple. DTT

Table 1: An overview of the steps performed by the DBMS during its runtime operation, commit processing, and checkpointing.

	Runtime Operation	Commit Processing	Checkpointing
WAL	• Execute the operation. • Write changes to table heap on DRAM. • Construct a log record based on operation (contains after-image of tuple). • Append log record to log entry buffer.	• Collect log entries from log entry buffers. • Sync the collected entries on durable storage. • Mark all the transactions as committed. • Inform workers about group commit.	• Construct checkpoint containing after-images of visible tuples. • Write out transactionally consistent checkpoint to durable storage. • Truncate unnecessary log records.
WBL	• Execute the operation. • Write changes to table heap on DRAM. • Add an entry to the DTT for that modification (does not contain after-image of tuple).	• Determine dirty tuples using the DTT. • Compute c_p and c_d for this group commit. • Sync dirty blocks to durable storage. • Sync a log entry containing c_p and c_d. • Inform workers about group commit.	• Construct a checkpoint containing only the active commit identifier gaps (no after-images). • Write out transactionally consistent checkpoint to durable storage. • Truncate unnecessary log records.

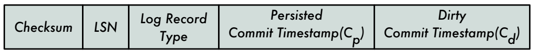

Checksum
LSN
Log Record Type
Persisted Commit Timestamp(C_p)
Dirty Commit Timestamp(C_d)

Figure 8: Structure of WBL Record – Structure of the log record constructed by the DBMS while using the WBL protocol.

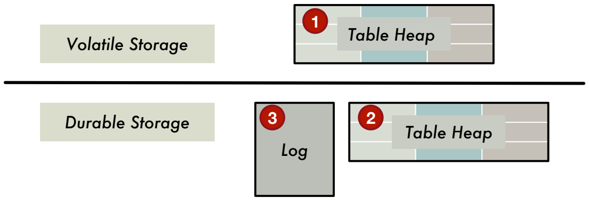


Figure 9: WBL Commit Protocol – The ordering of writes from the DBMS to durable storage while employing the WBL protocol.

entries never contain the after-images of tuples and are removed when their corresponding transaction commits. As in the case of WAL, the DBMS uses this information to ensure failure atomicity. But unlike in disk-oriented WAL, the DTT is never written to NVM. The DBMS only maintains the DTT in memory while using WBL.

4.2 Commit Protocol

Relaxing the ordering of writes to durable storage complicates WBL's commit and recovery protocols. When the DBMS restarts after a failure, it needs to locate the modifications made by transactions that were active at the time of failure so that it can undo them. But these changes can reach durable storage even before the DBMS records the associated meta-data in the log. This is because the DBMS is unable to prevent the CPU from evicting data from its volatile caches to NVM. Consequently, the recovery algorithm must scan the entire database to identify the dirty modifications, which is prohibitively expensive and increases the recovery time.

The DBMS avoids this problem by recording meta-data about the clean and dirty modifications that have been made to the database by tracking two commit timestamps in the log. First, it records the timestamp of the latest committed transaction all of whose changes and updates of prior transactions are safely persisted on durable storage (c_p). Second, it records the commit timestamp (c_d, where $c_p < c_d$) that the DBMS *promises* to not assign to any transaction before the subsequent group commit finishes. This ensures that any dirty modifications that were flushed to durable storage will have only been made by transactions whose commit timestamp is earlier than c_d. When the DBMS restarts after a failure, it considers all the transactions with commit timestamps earlier than c_p as committed, and ignores the changes of the transactions whose commit timestamp is later than c_p and earlier than c_d. In other words, if a tuple's begin timestamp falls within the (c_p, c_d) pair, then the DBMS's transaction manager ensures that it is not visible to any transaction that is executed after recovery.

When committing a group of transactions, as shown in Table 1, the DBMS examines the DTT entries to determine the dirty modifications. For each change recorded in the DTT, the DBMS persists the change to the table heap using the device's sync primitive (see Section 6.2). It then constructs a log entry containing c_p and c_d to record that any transaction with commit timestamps earlier than c_p has committed, and to indicate that it will not issue a commit timestamp later than c_d for any of the subsequent transactions before the next group commit. It appends this commit record (see Figure 8) to the log. The DBMS flushes the modifications of all the transactions with commit timestamps less than c_p before recording c_p in the log. Otherwise, it cannot guarantee that those transactions have been committed upon restart.

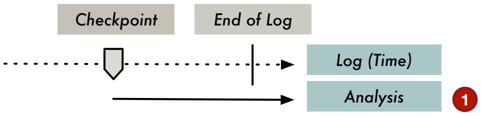


Figure 10: WBL Recovery Protocol – The phases of the recovery protocol.

For long running transactions that span a group commit window, the DBMS also records their commit timestamps in the log. Without this information, the DBMS cannot increment c_p before the transaction commits. During recovery, it uses this information to identify the changes made by uncommitted transactions. With WBL, the DBMS writes out the changes to locations spread across the durable storage device. For example, if a transaction updates tuples stored in two tables, then the DBMS must flush the updates to two locations in the durable storage device. This design works well for NVM as it supports fast random writes. But it is not a good choice for slower devices that incur expensive seeks to handle random writes.

To abort a transaction, the DBMS uses the information recorded in the DTT to determine the changes made by the transaction. It then discards those changes and reclaims their table heap storage.

The diagram in Figure 9 shows WBL's ordering of writes from the DBMS to durable storage. The DBMS first applies the changes on the table heap residing in volatile storage. But unlike WAL, when a transaction commits, the DBMS flushes all of its modifications to the durable table heap and indexes. Subsequently, the DBMS appends a record containing c_p and c_d to the log.

4.3 Recovery Protocol

Before describing WBL's recovery algorithm, we first introduce the notion of a *commit timestamp gap*. A commit timestamp gap refers to the range of timestamps defined by the pair (c_p, c_d). The DBMS must ignore the effects of transactions that fall within such a gap while determining the tuple visibility. This is equivalent to undoing the effects of any transaction that was active at the time of failure. The set of commit timestamp gaps that the DBMS needs to track increases on every system failure. To limit the amount of work performed while determining the visibility of tuples, the DBMS's garbage collector thread periodically scans the database to undo the dirty modifications associated with the currently present gaps. Once all the modifications in a gap have been removed by the garbage

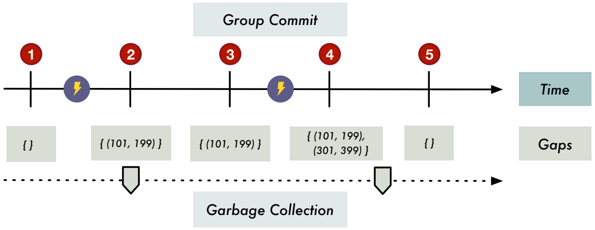


Figure 11: WBL Commit Timestamp Gaps – An illustration of successive system failures resulting in multiple commit timestamp gaps. The effects of transactions in those gaps are eventually undone by the garbage collector.

collector, the DBMS stops checking for the gap in tuple visibility checks and no longer records it in the log.

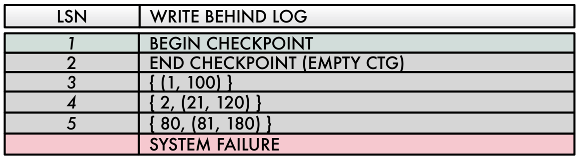
The example in Figure 11 depicts a scenario where successive failures result in multiple commit timestamp gaps. At the end of the first group commit operation, there are no such gaps and the current commit timestamp is `101`. The DBMS promises to not issue a commit timestamp higher than `199` in the time interval before the second commit. When the DBMS restarts after a system failure, it adds `(101, 199)` to its set of gaps. The garbage collector then starts cleaning up the effects of transactions that fall within this gap. Before it completes the scan, there is another system failure. The system then also adds `(301, 399)` to its gap set. Finally, when the garbage collector finishes cleaning up the effects of transactions that fall within these two gaps, it empties the set of gaps that the DBMS must check while determining the visibility of tuples.

With WBL, the DBMS does not need to periodically construct WAL-style physical checkpoints to speed up recovery. This is because each WBL log record contains all the information needed for recovery: the list of commit timestamp gaps and the commit timestamps of long running transactions that span across a group commit operation. The DBMS only needs to retrieve this information during the analysis phase of the recovery process. It can safely remove all the log records located before the most recent log record. This ensures that the log's size is always bounded.

As shown in Figure 10, the WBL recovery protocol only contains an analysis phase. During this phase, the DBMS scans the log backward until the most recent log record to determine the currently present commit timestamp gaps and timestamps of long running transactions. There is no need for a redo phase because all the modifications of committed transactions are already present in the database. WBL also does not require an WAL-style undo phase. Instead, the DBMS uses the information in the log to ignore the effects of uncommitted transactions.

Figure 12 shows the contents of the log after a system failure. This example is based on the same the workload used in Figure 7. We note that transactions `2` and `80` span across a group commit operation. At the time of system failure, only transactions `80` and `81` are uncommitted. During recovery, the DBMS loads the latest log record to determine the currently present commit timestamp gaps and timestamps of long running transactions. After this brief analysis phase, it can immediately start handling transactions again.

Correctness: When a transaction modifies the database, the DBMS only writes those changes to DRAM. Then when that transaction commits, the DBMS persists its changes to the table heap on durable storage. This prevents the system from losing the effects of any committed transaction, thereby ensuring the durability property. It ensures atomicity by tracking the uncommitted transactions using commit timestamp gaps. WBL allows repeated undo operations as it maintains logical undo information about uncommitted transactions.



LSN	WRITE BEHIND LOG
1	BEGIN CHECKPOINT
2	END CHECKPOINT (EMPTY CTG)
3	{ (1, 100) }
4	{ 2, (21, 120) }
5	{ 80, (81, 180) }
	SYSTEM FAILURE

Figure 12: WBL Example – Contents of the WBL during recovery.

Single-Versioned System: In a single-versioned DBMS with WBL, the system makes a copy of a tuple's before-image prior to updating it and propagating the new version to the database. This is necessary to support transaction rollbacks and to avoid *torn writes*. The DBMS stores the before-images in the table heap on durable storage. The DBMS's recovery process then only consists of an analysis phase; a redo phase is not needed because the modifications for all committed transactions are already present in the database. The DBMS, however, must roll back the changes made by uncommitted transactions using the before-images. As this undo process is done on demand, the DBMS starts handling transactions immediately after the analysis phase. Similar to the multi-versioned case, the DBMS uses the commit timestamps to determine the visibility of tuples and identify the effects of uncommitted transactions.

5. REPLICATION

With both the WAL and WBL protocols described above, the DBMS can recover from system and transaction failures. These protocols, however, are not able to handle media failures or corrupted data. This is because they rely on the integrity of durable data structures (e.g., the log) during recovery. These failures are instead overcome through replication, wherein the DBMS propagates changes made by transactions to multiple servers. When the *primary* server incurs a media failure, replication ensures that there is no data loss since the *secondary* servers can be configured to maintain a transactionally consistent copy of the database.

But replicating a database using WBL DBMS is different than in a WAL DBMS. With WAL, the DBMS sends the same log records that it stores on its durable storage device over the network. The secondary server then applies them to their local copy of the database. But since WBL's log records only contain timestamps and not the actual data (e.g., after-images), the DBMS has to perform extra steps to make WBL compatible with a replication protocol.

We now describe the different replication schemes for a primary-secondary configuration. We later present how a DBMS transforms WBL log records to work with these schemes.

5.1 Replication Schemes

There are two schemes for replicating a database in a primary-secondary configuration that each provide different consistency guarantees: (1) synchronous, and (2) asynchronous. Figure 13 presents the steps executed by the DBMS during replication. With *synchronous* replication, the primary server sends log records and waits for acknowledgments from the secondary servers that they have flushed the changes to durable storage (steps ❶,❷,❸,❹ are on the transaction's critical path). In *asynchronous* replication, the primary server does not wait for acknowledgments from any of the secondary servers (steps ❶,❷).

5.2 Record Format

The primary server in a WBL-based system cannot simply send its log records to the secondary servers because they do not contain the after-images of the modified tuples. Thus, to support replication, the DBMS must construct additional WAL records that contain the physical modifications to the database and send them to the

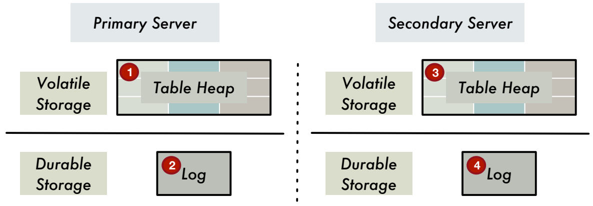


Figure 13: Replication – The steps taken by the DBMS during replication.

secondary servers. As we show in Section 7.3, this additional step adds minimal computational overhead since replication is bound by network communication costs.

We now describe the failover procedure in the secondary server when the primary server goes down. By design, the DBMS only transfers the changes associated with the committed transactions to the secondary servers. Consequently, there is no need for an undo process on the secondary servers on a failover. After a failover, the secondary server can immediately start handling transactions on a transactionally consistent database. But if the DBMS uses asynchronous replication, then the effects of recently committed transactions might not be present in the secondary server.

6. NVM EVALUATION PLATFORM

We next describe the hardware emulator that we use in our experiments. We then present the design of our NVM-aware allocator.

6.1 NVM Emulator

Existing NVM devices cannot store large databases due to their limited capacities and prohibitive costs. We instead use Intel Labs' persistent memory evaluation platform (PMEP) [17, 37]. PMEP models the latency and bandwidth characteristics of Intel's upcoming NVM technologies. It also emulates the newly proposed persistence primitives [4]. PMEP emulates NVM's higher read/write latencies for the NVM partition by using custom CPU microcode. This microcode estimates the number of cycles that the CPU would have to wait if DRAM is replaced by slower NVM and then stalls the CPU for that amount of time. The emulator provides two interfaces to access NVM storage:

Allocator Interface: The emulator contains a NVM-aware memory allocator (see Section 6.2) that exports the POSIX `malloc` interface. Internally, this allocator uses *libnuma* to allocate memory only from the emulated NVM partition [1].

Filesystem Interface: The emulator also supports a special *persistent memory filesystem* (PMFS) interface to read/write data to files stored on a NVM-backed volume [17, 6].

6.2 NVM-aware Allocator

A NVM-aware allocator needs to provide a *naming* mechanism so that pointers to locations in memory remain valid even when the system restarts. That is, it ensures that any pointer to a virtual memory address assigned to a memory-mapped region is always the same after the DBMS restarts. We refer to these pointers as *non-volatile pointers* [32, 6]. These allow us to construct non-volatile data structures that are guaranteed to always be consistent.

The allocator also provides a *durability* mechanism that the DBMS uses to ensure that database modifications are persisted on NVM. This is required because stores to NVM share the same volatile micro-architectural buffers in the processor and can therefore be lost on a power failure. The CPU must provide instructions that the allocator uses to expose a special NVM *sync* primitive.

Internally, the allocator implements the sync primitive by writing back the modifications to NVM using the cache-line write back (`CLWB`) instruction [4]. This instruction writes back the modified data in the cache-lines to NVM. Unlike the cache-line flush (`CLFLUSH`) instruction that is generally used for flushing operations, `CLWB` does not invalidate the line from the cache and instead only transitions it to a non-modified state. This reduces the possibility of a compulsory cache miss when the same data is accessed momentarily after the line has been flushed. As we later present in Section 7.7, an efficient cache flushing primitive is critical for a high-performance DBMS. We developed the memory allocator using the open-source NVM programming library [3]. After a system failure, the allocator reclaims memory that has not been persisted and restores its internal meta-data to a consistent state.

6.3 Three-Tier Storage Hierarchy

In our analysis, we focus on a two-tier storage hierarchy comprising of volatile DRAM and a durable storage device that is either NVM, SSD, or HDD. WBL can also be used in a three-tier storage hierarchy with DRAM, NVM, and SSD. In this case, the DBMS uses SSD to store the less frequently accessed tuples in the database. It stores the log and the more frequently accessed tuples on NVM. As bulk of the data is stored on SSD, the DBMS only requires a less expensive NVM device with smaller storage capacity.

The latency of a transaction that accesses a *cold* tuple will be higher in a three-tier storage hierarchy. This is because NVM supports faster reads than SSD. During update operations, however, the DBMS quickly writes to the log and database on NVM. Eventually, it moves the cold data to SSD. We plan to explore the impact of the data placement on a three-tier storage hierarchy in future work.

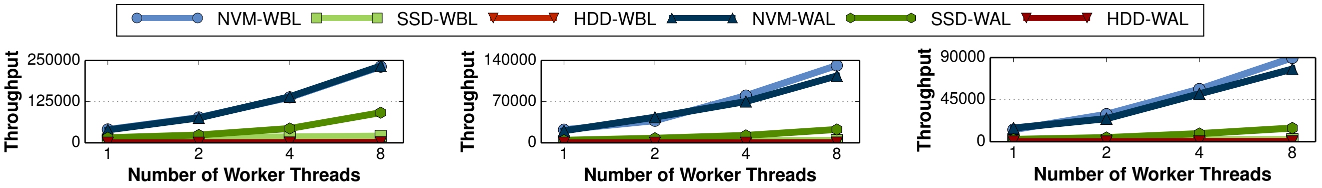
7. EXPERIMENTAL EVALUATION

We now present our analysis of the logging protocols. We implemented both WAL and WBL in Peloton, an in-memory HTAP DBMS that supports NVM [2]. We compare the DBMS's runtime performance, recovery times, and storage footprint for two OLTP workloads. We then analyze the effect of using WBL in a replicated system. Next, we compare WBL against an instant recovery protocol based on WAL [20, 21]. Finally, we examine the impact of storage latency, group commit latency, and new CPU instructions for NVM on the system's performance.

We performed these experiments using Intel Lab's PMEP hardware emulator that we described in Section 6. It contains two Intel Xeon E5-4620 CPUs (2.6 GHz), each with eight cores and a 20 MB L3 cache. The PMEP contains 256 GB of DRAM. It dedicates 128 GB of DRAM for the emulated NVM. We configured the NVM latency to be 4× that of DRAM and validated these settings using Intel's memory latency checker [6]. The PMEP also includes two additional storage devices:

- **HDD:** Seagate Barracuda (3 TB, 7200 RPM, SATA 3.0)
- **SSD:** Intel DC S3700 (400 GB, SATA 2.6)

We modified Peloton to use the PMEP's allocator and filesystem interfaces to store its logs, checkpoints, and table heap on NVM (see Section 6.1). When employing WAL, the DBMS maintains the log and the checkpoints on the filesystem, and uses `fsync` to ensure durability. When it adopts WBL, the DBMS uses the allocator for managing the durable table heap and indexes. Internally, it stores indexes in persistent B+trees [11, 10]. It relies on the allocator's *sync* primitive to ensure database durability. All the transactions execute with the same snapshot isolation level and durability guarantees. To evaluate replication, we use a second PMEP with the same hardware that is connected via 1 Gb Ethernet with 150 μs latency.

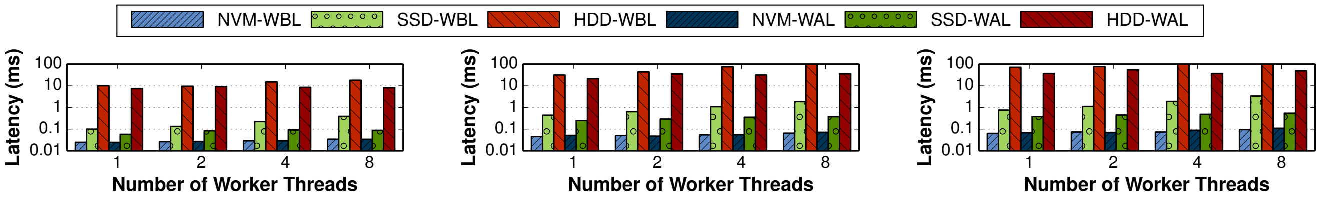


(a) Read-Heavy Workload

(b) Balanced Workload

(c) Write-Heavy Workload

Figure 14: YCSB Throughput – The throughput of the DBMS for the YCSB benchmark with different logging protocols and durable storage devices.



(a) Read-Heavy Workload

(b) Balanced Workload

(c) Write-Heavy Workload

Figure 15: YCSB Latency – The latency of the DBMS for the YCSB benchmark with different logging protocols and durable storage devices.

7.1 Benchmarks

We next describe the benchmarks that we use in our evaluation.

YCSB: This is a widely-used key-value store workload from Yahoo! [13]. It is representative of the transactions handled by web-based companies. The workload consists of two transaction types: (1) a *read* transaction that retrieves a single tuple using its primary key, and (2) an *update* transaction that modifies a single tuple based on its primary key. The distribution of the transactions' access patterns is based on a Zipfian skew. We use three workload mixtures to vary the amount of I/O operations that the DBMS executes:

- **Read-Heavy:** 90% *reads*, 10% *updates*
- **Balanced:** 50% *reads*, 50% *updates*
- **Write-Heavy:** 10% *reads*, 90% *updates*

The YCSB database contains a single table comprised of tuples with a primary key and 10 columns of random string data, each 100 bytes in size. Each tuple's size is approximately 1 KB. We use a database with 2 million tuples (∼2 GB).

TPC-C: This benchmark simulates an order-entry application of a wholesale supplier [35]. The TPC-C workload consists of five transaction types, of which 88% of them modify the database. We configured the benchmark to use eight warehouses and 100,000 items. The initial storage footprint of the database is ∼1 GB.

7.2 Runtime Performance

We begin with an analysis of the recovery protocols' impact on the DBMS's runtime performance. To obtain insights that are applicable for different storage technologies, we run the YCSB and TPC-C benchmarks in Peloton while using either the WAL or WBL. For each configuration, we scale up the number of worker threads that the DBMS uses to process transactions. The clients issue requests in a closed loop. We execute all the workloads three times under each setting and report the average throughput and latency. To provide a fair comparison, we disable checkpointing in the WAL-based configurations, since it is up to the administrator to configure the checkpointing frequency. We note that throughput drops by 12–16% in WAL when the system takes a checkpoint.

YCSB: We first consider the read-heavy workload results shown in Figure 14a. These results provide an approximate upper bound on the DBMS's performance because the 90% of the transactions do not modify the database and therefore the system does not have to construct many log records. The most notable observation from this experiment is that while the DBMS's throughput with the SSD-WAL configuration is 4.5× higher than that with the SSD-WBL configuration, its performance with the NVM-WBL configuration is comparable to that obtained with the NVM-WAL configuration. This is because NVM supports fast random writes unlike HDD.

The NVM-based configurations deliver 1.8–2.3× higher throughput over the SSD-based configurations. This is because of the ability of NVM to support faster reads than SSD. The gap between the performance of the NVM-WBL and the NVM-WAL configurations is not prominent on this workload as most transactions only perform reads. The throughput of all the configurations increases with the number of worker threads as the increased concurrency helps amortize the logging overhead. While the WAL-based DBMS runs well for all the storage devices on a read-intensive workload, the WBL-based DBMS delivers lower performance while running on the HDD and SSD due to their slower random writes.

The benefits of WBL are more prominent for the balanced and write-heavy workloads presented in Figures 14b and 14c. We observe that the NVM-WBL configuration delivers 1.2–1.3× higher throughput than the NVM-WAL configuration because of its lower logging overhead. That is, under WBL the DBMS does not construct as many log records as it does with WAL and therefore it writes less data to durable storage. The performance gap between the NVM-based and SSD-based configurations also increases on write-intensive workloads. With the read-heavy workload, the NVM-WBL configuration delivers only 4.7× higher throughput than the SSD-WBL configuration. But on the balanced and write-heavy workloads, NVM-WBL provides 10.4–12.1× higher throughput.

The transactions' average response time is presented in Figure 15. As expected, the HDD-based configurations incur the highest latency across all workloads, especially for WBL. For example, on the write-heavy workload, the average latency of the HDD-WBL configuration is 3.9× higher than the HDD-WAL configuration. This is because the random seek time of HDDs constrains the DBMS performance. The SSD-based configurations have up to two orders of magnitude lower transaction latency compared to HDD configurations because of their better write performance. On the write-heavy workload shown in Figure 14c, the transaction latency of the NVM-WBL configuration is 21% and 65% lower than that observed with NVM-WAL and SSD-WAL respectively. We attribute this to WAL's higher overhead and higher write latency of SSD.

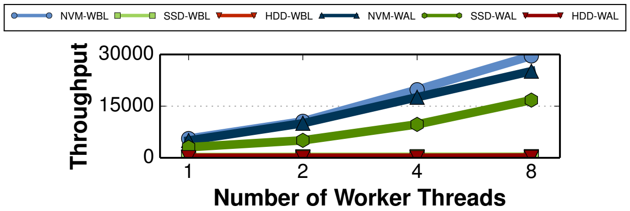


Figure 16: TPC-C Throughput – The measured throughput for the TPC-C benchmark with different logging protocols and durable storage devices.

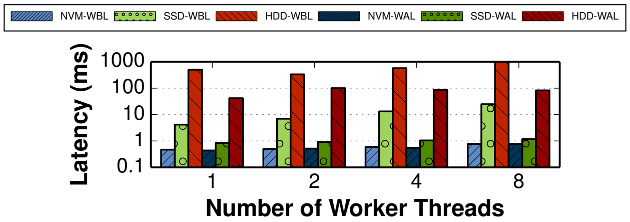
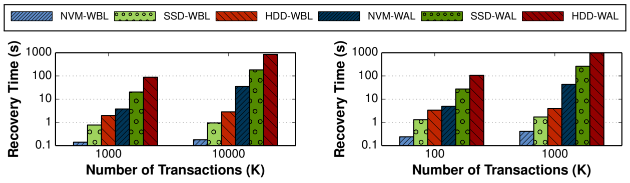


Figure 17: TPC-C Latency – The latency of the DBMS for the TPC-C benchmark with different logging protocols and durable storage devices.



(a) YCSB

(b) TPC-C

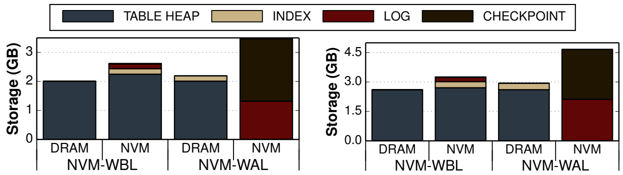
Figure 18: Recovery Time – The time taken by the DBMS to restore the database to a consistent state after a restart with different logging protocols.

TPC-C: Figures 16 and 17 show the throughput and latency of the DBMS while executing TPC-C with varying number of worker threads. Like with YCSB, the DBMS achieves the highest throughput and the lowest latency using the NVM-WBL configuration. The NVM-WAL and SSD-WAL configurations provide 1.3× and 1.8× lower throughput compared to NVM-WBL. We attribute this to the large number of writes performed per transaction in TPC-C. We observe that the performance obtained across all configurations on the TPC-C benchmark is lower than that on the YCSB benchmark. This is because the transactions in TPC-C contain more complex program logic and execute more queries per transaction.

7.3 Recovery Time

We evaluate the recovery time of the DBMS using the different logging protocols and storage devices. For each benchmark, we first execute a fixed number of transactions and then force a hard shutdown of the DBMS (`SIGKILL`). We then measure the amount of time for the system to restore the database to a consistent state. That is, a state where the effects of all committed transactions are durable and the effects of uncommitted transactions are removed. Recall from Section 3 that the number of transactions that the DBMS processes after restart in WAL depends on the frequency of checkpointing. With WBL, the DBMS performs garbage collection to clean up the dirty effects of uncommitted transactions at the time of failure. This garbage collection step is done asynchronously and does not have a significant impact on the throughput of the DBMS.

YCSB: The results in Figure 18a present the recovery measurements for the YCSB benchmark. The recovery times of the WAL-based configurations grow linearly in proportion to the number of transactions that the DBMS recovers. This is because the DBMS needs to replay the log to restore the effects of committed transactions. In contrast, with WBL, we observe that the recovery time is independent of the number of transactions executed. The system only reverses the effects of transactions that were active at the time of failure as the changes made by all the transactions committed after the last checkpoint are already persisted. The WBL-based configurations, therefore, have a short recovery.



(a) YCSB Storage Footprint

(b) TPC-C Storage Footprint

Figure 20: Storage Footprint – The storage space occupied by the internal components of the DBMS while using different recovery protocols.

TPC-C: The results for the TPC-C benchmark in Figure 18b show that the recovery time of the WAL-based configurations is higher than that in the YCSB benchmark. This is because the TPC-C transactions perform more operations, and consequently require a longer redo phase. The recovery time of the WBL-based configurations, however, is still independent of the number of transactions executed unlike their WAL counterparts because they ensure that the effects of committed transactions are persisted immediately on durable storage.

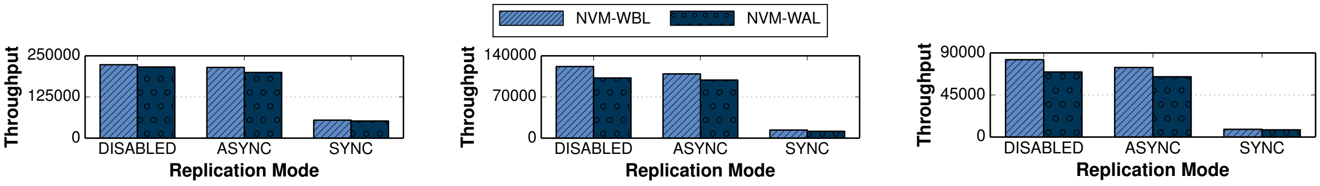
7.4 Storage Footprint

We compare the storage utilization of the DBMS using either the WAL and WBL protocols while running on NVM. This metric is important because we expect that the first NVM products will initially be more expensive than current technologies [24], and thus using less storage means a lower procurement cost.

We measure Peloton's storage footprint as the amount of space that it uses in either DRAM or NVM to store tables, logs, indexes, and checkpoints. We periodically collect statistics from the DBMS's storage manager and the filesystem meta-data during the workload execution. We perform these measurements after loading the initial database and report the peak storage footprint of the DBMS for each trial. For all of the configurations, we allow the DBMS's background processes (e.g., group commit, checkpointing, garbage collection) to execute while we collect these measurements.

YCSB: We use the balanced workload mixture for this experiment with an initial database size of 2 GB. The results in Figure 20a show that the WAL-based configuration has a larger storage footprint than WBL. This is because WAL constructs log records that contain the physical changes associated with the modified tuples. In contrast, as described in Section 4.1, WBL's log records do not contain this information. Another important difference is that while the WAL-based DBMS periodically constructs transactionally-consistent checkpoints of the database, WBL only requires the DBMS to write log records that contain the list of currently present commit identifier gaps. As such, its logical checkpoints have a smaller storage footprint than WAL's physical checkpoints. Unlike WAL, WBL persists the indexes on durable storage to avoid rebuilding it during recovery. The WBL-based DBMS consume 26% less storage space on NVM than its WAL counterpart.

TPC-C: The graph in Figure 20b shows the storage footprint of the engines while executing TPC-C. For this benchmark, the initial size of the database is 1 GB and it grows to 2.4 GB. Transactions inserting new orders increase the size of the table heap, log,



(a) Read-Heavy Workload **(b)** Balanced Workload **(c)** Write-Heavy Workload

Figure 19: Replication – The throughput of the DBMS for the YCSB benchmark with different replication schemes and logging protocols.

and checkpoints in the WAL-based configuration. By reducing unnecessary data duplication using NVM's persistence property, the NVM-WBL configuration has a 31% smaller storage footprint on NVM. The space savings are more significant in this benchmark because the workload is write-intensive with longer running transactions. Thus, the log in the WAL-based configuration grows more quickly compared to the smaller undo log in WBL.

7.5 Replication

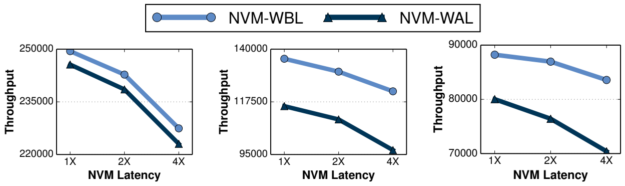
We now examine the impact of replication on the runtime performance of the DBMS while running the YCSB benchmark and using the NVM-based configurations. The results shown in Figure 19 indicate that the synchronous replication scheme reduces the throughput. On the read-heavy workload, the throughput drops by 3.6× with both NVM-WAL and NVM-WBL configurations. This shows that the overhead of constructing WAL-style log records when using WBL is lower than the overhead of sending the log records over the network. Under the asynchronous replication scheme, the DBMS's throughput drops by less than 1.1× across all the workloads. The DBMS should, therefore, be configured to use this replication scheme when the user can afford to lose the effects of some recently committed transactions on a media failure.

The impact of replication is more prominent in the write-heavy workload shown in Figure 19c. We observe that throughput of the DBMS drops by 10.1× when it performs synchronous replication. This is because the round trip latency between the primary and secondary server (150 μs) is higher than the durable write latency (0.6 μs) of NVM. The networking cost is, thus, the major performance bottleneck in replication. We conclude that a faster replication standard, such as the NVMe over Fabrics [30], is required for efficient transaction processing in a replicated environment containing NVM [37]. With this technology, the additional latency between a local and remote NVM device is expected to be less than a few microseconds. As every write to NVM must be replicated in most datacenter usage models, we expect WBL to outperform WAL in this replicated environment because it executes fewer NVM writes. We plan to investigate this in future work.

7.6 Impact of NVM Latency

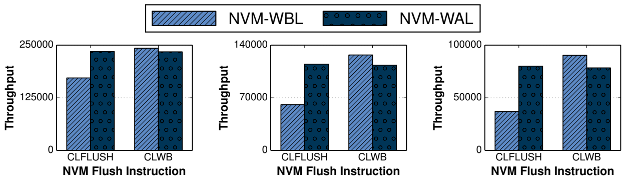
In this experiment, we analyze how the latency of the NVM affects the runtime performance of the WBL and WAL protocols in the DBMS. We ran YCSB under three latency configurations for the emulator: (1) default DRAM latency (160 ns), (2) a *low* latency that is 2× slower than DRAM (320 ns), and (3) a *high* latency that is 4× slower than DRAM (640 ns). Prior work has shown that the sustained bandwidth of NVM is likely to be lower than that of DRAM [17]. We therefore use the PMEP's throttling mechanism to reduce the NVM bandwidth to be 8× lower (9.5 GB/s) than DRAM.

The key observation from the results in Figure 21 is that the NVM-WAL configuration is more sensitive to NVM latency compared to NVM-WBL. On the write-heavy workload shown in Figure 21c, with a 4× increase in NVM latency, the throughput of NVM-WAL drops by 1.3×, whereas NVM-WBL only drops by 1.1×. This is because the DBMS performs fewer stores to NVM with WBL. We observe that NVM latency has a higher impact on the performance for write-intensive workloads. On the read-heavy workload shown in Figure 21a, the throughput of the DBMS only drops by 1.1–1.3× with a 4× increase in latency. We attribute this to the effects of caching and memory-level parallelism.



(a) Read-Heavy Workload **(b)** Balanced Workload **(c)** Write-Heavy Workload

Figure 21: Impact of NVM Latency – The throughput for the YCSB benchmark with different logging protocols and NVM latency settings.



(a) Read-Heavy Workload **(b)** Balanced Workload **(c)** Write-Heavy Workload

Figure 22: NVM Instruction Set Extensions (`CLFLUSH` vs. `CLWB`) – The throughput of the DBMS for the YCSB benchmark under the NVM-based configurations with different flush instructions.

7.7 NVM Instruction Set Extensions

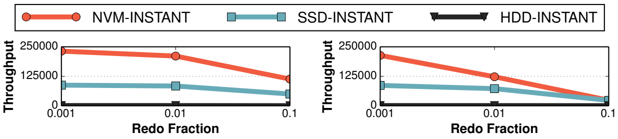
We next measure the impact of proposed NVM-related instruction set extensions on the DBMS's performance with the NVM-WBL configuration[1] [4]. We examine the impact of using the `CLWB` instruction for flushing the cache-lines instead of the `CLFLUSH` instruction. Recall from Section 6.1 that the `CLWB` instruction reduces the possibility of compulsory cache misses during subsequent data accesses.

Figure 22 presents the throughput of the DBMS with the NVM-WBL configuration while using either the `CLWB` or `CLFLUSH` instructions in its *sync* primitive. The throughput obtained with the NVM-WAL configuration, that does not use the *sync* primitive, is provided for comparison. We observe that the throughput under the NVM-WBL configuration exceeds that obtained with NVM-WAL when the DBMS uses the `CLWB` instruction. We attribute this to the effects of caching. The impact of the `CLWB` instruction is more prominent on the write-intensive workloads, where the WBL-based DBMS delivers 1.7× higher throughput when using the `CLWB` instruction instead of the `CLFLUSH` instruction. Thus, an efficient cache flushing primitive is critical for a high-performance NVM-aware DBMS.

7.8 Instant Recovery Protocol

We now compare WBL against an instant recovery protocol based on WAL [20, 21]. This protocol uses *on-demand* single-tuple redo

[1] Intel added these extensions in 2014 and they are expected to be commercially available in processors shipping in 2017.



(a) Chain length ∈ (0, 10) **(b)** Chain length ∈ (0, 100)

Figure 23: Instant Recovery Protocol – The throughput of the DBMS for YCSB with the instant logging protocol on different storage devices.

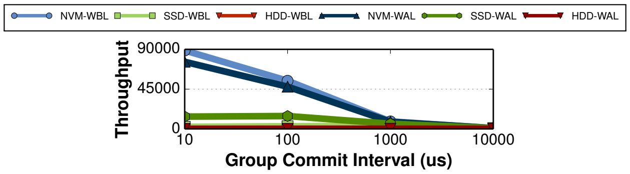


Figure 24: Group Commit Latency – Impact of the group commit latency setting on the throughput of the DBMS for the write-heavy YCSB workload with different logging protocols and durable storage devices.

and single-transaction undo mechanisms to support almost instantaneous recovery from system failures. While processing transactions, the DBMS reconstructs the desired version of the tuple on demand using the information in the write-ahead log. The DBMS can, therefore, start handling new transactions almost immediately after a system failure. The downside is that the DBMS performance is lower than that observed after the traditional ARIES-style recovery while the recovery is not yet complete.

Unlike the WAL-based instant recovery protocol, WBL relies on NVM's ability to support fast random writes. It does not contain a redo process. To better understand the impact of the instant-recovery protocol on the performance of the DBMS, we implemented it in our DBMS. We run the read-heavy YCSB workload on the DBMS, while varying the fraction of the tuples in the table that must be reconstructed from 0.001 to 0.1. With more frequent checkpointing, a smaller fraction of tuples would need be to reconstructed. We configure the length of a tuple's log record chain to follow an uniform distribution over the following ranges: (0, 100) and (0, 10).

The results shown in Figures 23a and 23b indicate that the performance drops with longer log record chains, especially when a larger fraction of tuples need to be reconstructed. When the maximum length of a long record chain is limited to 10 records, the throughput drops by 1.5× when the DBMS needs to reconstruct 10% of the tuples in comparison to the throughput observed after recovery. In contrast, when the length is limited to 100 records, the throughput drops by 8×. After the recovery process is complete, the performance of the DBMS converges to that observed after the traditional recovery. We conclude that the instant recovery protocol works well when the DBMS runs on a slower durable storage device. However, on a fast NVM device, WBL allows the DBMS to deliver high performance immediately after recovery. Unlike WAL, as we showed in Section 7.4, it improves device utilization by reducing data duplication.

7.9 Impact of Group Commit Latency

In this experiment, we analyze how the group commit latency affects the runtime performance of the WBL and WAL protocols in the DBMS. As the DBMS sends the results back to the client only after completing the group commit operation, this parameter affects the latency of the transaction. We run the write-heavy YCSB workload under different group commit latency settings ranging from 10 through 10000 μs.

The most notable observation from the results in Figure 24 is that different group commit latency settings work well for different durable storage devices. Setting the group commit latency to 10, 100, and 1000 μs works well for the NVM, SSD, and HDD respectively. We observe that there is a two orders of magnitude gap between the optimal group commit latency settings for NVM and HDD. The impact of this parameter is more pronounced in the case of NVM compared to the slower durable storage devices. When the group commit latency of the DBMS running on NVM is increased from 10 to 1000 μs, the throughput drops by 62×.

8. RELATED WORK

We now discuss the previous research on using NVM, especially in the context of DBMSs and file-systems.

NVM-Aware Logging: A previous study demonstrated that in-memory DBMSs perform only marginally better than disk-oriented DBMSs when using NVM because both systems still assume that memory is volatile [14]. As such, there has been recent work on developing new DBMS logging protocols specifically for NVM. Pelley et al. introduced a group commit mechanism to persist transactions' updates in batches to reduce the number of write barriers required for ensuring correct ordering on NVM [33]. Their work is based on Shore-MT [23], which means that the DBMS records page-level before-images in the log before performing in-place updates. This results in high data duplication.

Wang et al. present a passive group commit method for a distributed logging protocol extension to Shore-MT [36]. Instead of issuing a barrier for every processor at commit time, the DBMS tracks when all of the records required to ensure the durability of a transaction are flushed to NVM. This is similar to another approach that writes log records to NVM, and addresses the problems of detecting partial writes and recoverability [18]. Both of these projects rely on software-based NVM simulation.

SOFORT [32] is a hybrid storage engine designed for both OLTP and OLAP workloads. The engine is designed to not perform any logging and uses MVCC. Similar to SOFORT, we also make use of non-volatile pointers [3], but we use these pointers in a different way. SOFORT's non-volatile pointers are a combination of page ID and offset. We eschew the page abstraction in our engines since NVM is byte-addressable, and thus we use *raw* pointers that map to data's location in NVM.

REWIND is an userspace library for efficiently managing persistent data structures on NVM using WAL to ensure recoverability [10]. FOEDUS is a scalable OLTP engine designed for a hybrid storage hierarchy [25]. It is based on the *dual page* primitive that points to a pair of logically equivalent pages, a mutable volatile page in DRAM containing the latest changes, and an immutable snapshot page on NVM. SiloR is an efficient parallelized logging, checkpointing, and recovery subsystem for in-memory DBMSs [38]. Oh et al. present a *per-page logging* approach for replacing a set of successive page writes to the same logical page with fewer log writes [31]. Unlike WBL, all these systems require that the changes made to persistent data must be preceded by logging.

NVM-Aware File-systems: Beyond DBMSs, others have explored using NVM in file-systems. Rio is a persistent file cache that relies on uninterruptible power supply to provide a safe, in-memory buffer for filesystem data [27]. BPFS uses a variant of shadow paging on NVM to support atomic fine-grained updates by relying on a special hardware instruction that ensures ordering between writes in different epochs [12]. PMFS is another filesystem from Intel Labs that is designed for byte-addressable NVM [17]. It relies on a WAL for meta-data and uses shadow paging for data.

Instant Recovery Protocol: This protocol comprises of on-demand single-tuple redo and single-transaction undo mechanisms to support almost instantaneous recovery from system failures [20, 21]. While processing transactions, the DBMS reconstructs the desired version of the tuple on demand using the information in the write-ahead log. The DBMS can, therefore, start handling new transactions almost immediately after a system failure. The downside is that the DBMS performance is lower than that observed after the traditional ARIES-style recovery while the recovery is not yet complete. This protocol works well when the DBMS runs on a slower durable storage device. But with NVM, WBL enables the DBMS to deliver high performance than instant recovery immediately after recovery as it does not require an on-demand redo process.

9. ACKNOWLEDGEMENTS

This work was supported (in part) by the Intel Science and Technology Center for Big Data, the U.S. National Science Foundation (CCF-1438955), and the Samsung Fellowship Program. We are grateful to Jinwoong Kim, Jiajun Wang, Haibin Lin, and Abhishek Joshi for their assistance in implementing WBL in Peloton. We would like to thank Garth Gibson and Mike Stonebraker for their feedback on this work.

10. CONCLUSION

This paper presented the write-behind logging protocol for emerging non-volatile storage technologies. We examined the impact of this redesign on the transactional throughput, latency, availability, and storage footprint of the DBMS. Our evaluation of recovery algorithm in Peloton showed that across different OLTP workloads it reduces the system's recovery time by 100× and shrinks the storage footprint by 1.5×.

11. REFERENCES

[1] NUMA policy library. http://linux.die.net/man/3/numa.

[2] Peloton Database Management System. http://pelotondb.org.

[3] Persistent memory programming library. http://pmem.io/.

[4] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf, 2016.

[5] R. Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. IWDM'89, pages 269–285.

[6] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD'15*.

[7] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD'16*.

[8] J. Axboe. Flexible io tester. http://freecode.com/projects/fio.

[9] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. Res. Dev.*, 52(4):449–464, July 2008.

[10] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 2015.

[11] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 2015.

[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. In *SOSP*, pages 133–146, 2009.

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

[14] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS@VLDB*, 2014.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[16] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, 2013.

[17] S. R. Dulloor, S. K. Kumar, A. Keshavamurthy, P. Lantz, D. Subbareddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[18] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. ICDE, pages 1221–1231, 2011.

[19] M. Franklin. Concurrency Control and Recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997.

[20] G. Graefe, W. Guy, and C. Sauer. Instant recovery with write-ahead logging: Page repair, system restart, and media restore. *Synthesis Lectures on Data Management*, 2015.

[21] T. Härder, C. Sauer, G. Graefe, and W. Guy. Instant recovery with write-ahead logging. *Datenbank-Spektrum*, pages 235–239, 2015.

[22] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, pages 389–400, Dec. 2014.

[23] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[24] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *FAST*, 2014.

[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

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

[27] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *SOSP*, 1997.

[28] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[29] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, 2015.

[30] NVM Express Inc. NVM Express over Fabrics specification. http://www.nvmexpress.org/specifications, 2016.

[31] G. Oh, S. Kim, S.-W. Lee, and B. Moon. Sqlite optimization with phase change memory for mobile applications. *Proc. VLDB Endow.*, 8(12):1454–1465, Aug. 2015.

[32] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. DaMoN, 2014.

[33] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[34] S. Pilarski and T. Kameda. Checkpointing for distributed databases: Starting from the basics. *IEEE Trans. Parallel Distrib. Syst.*, 1992.

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

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

[37] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.

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