The Design and Implementation of a Non-Volatile Memory Database Management System

Joy Arulraj
jarulraj@cs.cmu.edu

THESIS PROPOSAL

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

THESIS COMMITTEE:
Andrew Pavlo (Chair)
Garth Gibson
Todd Mowry
Samuel Madden, MIT
Donald Kossmann, Microsoft Research

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Non-Volatile Memory, Write-Behind Logging, Peloton

Abstract

This dissertation explores the implications of emergent non-volatile memory (NVM) technologies for database management systems (DBMSs). The advent of NVM will fundamentally change the dichotomy between volatile memory and durable storage in DBMSs. These new NVM devices are almost as fast as DRAM, but all writes to it are potentially persistent even after power loss. Existing DBMSs are unable to take full advantage of this technology because their internal architectures are predicated on the assumption that memory is volatile. With NVM, many of the components of legacy DBMSs are unnecessary and will degrade the performance of the data intensive applications.

We present the design and implementation of a new DBMS tailored specifically for NVM. The dissertation focuses on three aspects of a DBMS: (1) logging and recovery, (2) storage management, and (3) indexing. Our primary contribution in this dissertation is the design of a new logging and recovery protocol, called write-behind logging, that improves the availability of the system by more than two orders of magnitude compared to the ubiquitous write-ahead logging protocol. Besides improving availability, we demonstrate that write-behind logging extends the lifetime and increases the the space utilization of the NVM device. Second, we propose a new storage engine architecture that leverages the durability and byte-addressability properties of NVM to avoid unnecessary data duplication. Third, the dissertation presents the design of a range index tailored for NVM that supports near-instantaneous recovery without requiring special-purpose recovery code.

Contents

Chapter 1

Introduction

This dissertation explores the changes needed in database management systems (DBMSs) to leverage emergent non-volatile memory (NVM) technologies that blur the gap between volatile memory and durable storage.

The design of DBMSs has always been influenced by the difference in the performance characteristics of volatile memory and non-volatile storage [20]. The key assumption has been that non-volatile storage is much slower than DRAM and only supports block-oriented read/writes. But new NVM [1] technologies that are almost as fast as DRAM with fine-grained read/writes invalidates these previous design choices.

Existing DBMSs can be classified into two types based on the primary storage location of the database: (1) disk-oriented, and (2) memory-oriented DBMSs. Disk-oriented DBMSs are based on the same hardware assumptions that were made in the first relational DBMSs from the 1970s, such as IBM's System R [16]. The design of these systems target a two-level storage hierarchy comprising of: a fast but volatile byte-addressable memory for caching (i.e., DRAM), and a slow, non-volatile block-addressable device for permanent storage (i.e., HDD). These systems take a pessimistic assumption that a transaction could access data that is not in memory, and thus will incur a long delay to retrieve the needed data from disk. They employ legacy techniques such as heavyweight concurrency-control schemes to overcome these limitations [75].

Recent advances in manufacturing technologies have greatly increased the capacity of DRAM available on a single computer. But, disk-oriented systems were not designed for the case where most, if not all, of the data resides entirely in memory. The result is that many of their legacy components have been shown to impede their scalability for OLTP workloads [51]. In contrast, the architecture of memory-oriented DBMSs assumes that all data fits in main memory, and it therefore does away with the slower, disk-oriented components from the system. As such, these memory-oriented DBMSs have been shown to outperform disk-oriented DBMSs [40, 53, 55, 84]. But, they still have to employ components that can recover the database after a restart because DRAM is volatile.

The need to ensure that all changes are durable has dominated the design of both disk-oriented and memory-oriented DBMSs. This has involved optimizing the layout of data for each storage layer depending on how fast it can perform random accesses. Further, the DBMS needs to propagate

[1] NVM is also referred to as *storage-class memory* or *persistent memory*.

	DRAM	PCM	RRAM	MRAM	SSD	HDD
Read latency	60 ns	50 ns	100 ns	20 ns	25 μs	10 ms
Write latency	60 ns	150 ns	100 ns	20 ns	300 μs	10 ms
Sequential bandwidth	60 GB/s	10 GB/s	10 GB/s	5 GB/s	1 GB/s	0.1 GB/s
$/GB	8.6	~4	~4	~4	0.25	0.02
Addressability	Byte	Byte	Byte	Byte	Block	Block
Persistent	No	Yes	Yes	Yes	Yes	Yes
Endurance	$>10^{16}$	10^{10}	10^{8}	10^{15}	10^{5}	$>10^{16}$

Table 1.1: Comparison of emerging NVM technologies with other storage technologies [26, 43, 67, 78]: phase-change memory (PCM) [7, 9, 81], memristors (RRAM) [8, 85], and STT-MRAM (MRAM) [42].

updates that the transactions make on tuples stored in memory to an on-disk representation to ensure durability.

The advent of NVM technologies, such as phase-change memory [7, 9, 81] and memristors [8, 85], offers an intriguing blend of the two storage mediums. These emergent storage technologies remove the data transformation and propagation costs by supporting byte-addressable loads and stores with low latency. This means that they can be used for efficient architectures that are used in memory-oriented DBMSs. But unlike DRAM, all writes to the NVM are potentially durable, and therefore a DBMS can access the tuples directly in the NVM after a restart or crash without needing to reload the database first. As shown in Table 1.1, NVM differs from other storage technologies in the following ways:

- **Byte-Addressability:** NVM supports byte-addressable loads and stores unlike other non-volatile devices that only support slow, bulk data transfers as blocks.
- **High Write Throughput:** NVM delivers two orders of magnitude higher write throughput compared to SSD and HDD. More importantly, the gap between sequential and random write throughput of NVM is much smaller than other durable storage technologies.
- **Read-Write Asymmetry:** In certain NVM technologies, writes take longer to complete compared to reads. Further, excessive writes to a single memory cell can destroy it.

Although the advantages of NVM are obvious, making full use of them in an DBMS is non-trivial. Our evaluation of state-of-the-art disk-oriented and memory-oriented DBMSs on NVM shows that the two architectures achieve almost the same performance when using NVM [37]. This is because current DBMSs assume that memory is volatile, and thus their architectures are predicated on making redundant copies of changes on durable storage. This illustrates the need for a complete rewrite of the database system in order to leverage the unique properties of NVM.

This dissertation presents the design and implementation of Peloton, a new DBMS tailored specifically for NVM. The resulting NVM-aware DBMS architecture has several key advantages over current systems:

1. It adopts a new logging and recovery protocol, called write-behind logging, that improves the availability of the system by 100× compared to the ubiquitous write-ahead logging protocol.

2. Its storage engine architecture leverages the durability and byte-addressability properties of NVM to avoid unnecessary data duplication. This improves the space utilization of the NVM device and extends its lifetime by reducing the number of device writes.
3. It employs a latch-free range index tailored for NVM that supports near-instantaneous recovery without requiring special-purpose recovery code. This reduces the implementation and maintenance complexity of critical components of the DBMS.
4. It supports both single and multi-tier storage hierarchies depending on the performance and cost constraints imposed by the application.

We implemented this NVM-aware DBMS architecture in Peloton. Our evaluation using different online transaction processing (OLTP) and analytical processing (OLAP) benchmarks show that this architecture improves the runtime performance, availability, and operational cost of database systems [12–15, 38]. From a longer-term perspective, we believe that Peloton will serve as a research platform for exploring the changes required in DBMSs to better support emergent non-volatile memory technologies.

Thesis Statement: *Rethinking the key algorithms and data structures employed in a database management system to leverage the characteristics of non-volatile memory improves availability, operational cost, development cost, and performance.*

In the remainder of this chapter, we first illustrate the impact of NVM on the design of DBMSs using a new logging algorithm that allows instantaneous recovery from system failures. We then summarize the primary contributions of this work, and conclude with an outline of this dissertation.

1.1 Write-Behind Logging

A DBMS protects the database state from corruption due to application, operating system, and device failures [45]. 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 volatile DRAM, but all data on DRAM is lost after a power failure.

These differences between the two types of storage are a major factor in the design of DBMS architectures [39, 46]. 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* [20, 66].

But emerging NVM technologies are poised to upend these assumptions. This means that the canonical approaches for DBMS logging and recovery that assume slower storage are incompatible with this new hardware landscape [37].

We have designed a new protocol, called *write-behind logging*, that is better suited for a hybrid storage hierarchy with NVM and DRAM. We found that tailoring the logging and recovery algorithms for NVM not only improves the runtime performance of the DBMS, but it also enables it to recover instantaneously from failures. The way that write-behind logging 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 the utilization of the NVM device and shrinking the cost of ownership [13].

Write-behind logging illustrates that the work performed by state-of-the-art DBMSs to ensure recoverability can be avoided by leveraging NVM. More broadly, it demonstrates that the best way to fully take advantage of NVM is by designing a new DBMS tailored specifically for NVM, with principles of both disk-oriented and memory-oriented systems [12].

1.2 Contributions

This dissertation illustrates the importance of rethinking the key algorithms and data structures employed in DBMSs for emergent NVM technologies. In particular, this dissertation answers the following research questions with the specific contributions listed:

1. **[COMPLETED]** How do state-of-the-art memory-oriented and disk-oriented DBMSs perform on non-volatile memory? (Chapter 2)

   - A study of the impact of NVM on two OLTP DBMSs.
   - We explore two possible architectures using non-volatile memory (i.e., NVM-only and NVM+DRAM architectures).

   This work is **completed** and was published in SIGMOD 2017 [12] and ADMS 2014 [37].

2. **[COMPLETED]** How should the storage engine architecture evolve to leverage NVM? (Chapter 3)

   - We implement three storage engine architectures in a single DBMS: (1) in-place updates with logging, (2) copy-on-write updates without logging, and (3) log-structured updates.
   - We then develop NVM-optimized variants for these architectures that improve the computational overhead, storage footprint, and wear-out of NVM devices.

   This work is **completed** and was published in SIGMOD 2015 [13].

3. **[COMPLETED]** What changes are required in the logging and recovery algorithms to support fast recovery from failures? (Chapter 4)

   - We present a new logging and recovery protocol that is designed for a hybrid storage hierarchy with NVM and DRAM.
   - We examine the impact of this redesign on the transactional throughput, latency, availability, and storage footprint of the DBMS.

   This work is **completed** and was published in VLDB 2017 [15] and SIGMOD 2016 [14].

4. **[COMPLETED]** How should we adapt the design of a latch-free range index for NVM?
   - We propose the design of such an index that supports near-instantaneous recovery without requiring special-purpose recovery code.
   - An evaluation of the impact of this redesign on the software development and maintenance complexity, performance, and availability.

   This work is **completed** and is under review.
5. **[PROPOSED]** How should the DBMS migrate data in a multi-tier storage hierarchy comprising of DRAM, NVM, and SSD?
   - We intend to examine the impact of introducing NVM into the storage hierarchy on the performance of the DBMS while executing different OLTP and OLAP benchmarks.
   - We plan to evaluate the total cost of ownership and operational throughput under different NVM device capacity and latency settings.

   This will be **new work**, involving some implementation effort and subsequent evaluation.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents our initial foray into the use of NVM in OLTP DBMSs. We test several DBMS architectures on an experimental, hardware-based NVM emulator and explore their trade-offs using OLTP benchmarks. To the best of our knowledge, our investigation is the first to use emulated NVM for OLTP DBMSs. Chapter 3 covers the design of NVM-aware variants of three different storage engine architectures that leverage the persistence and byte-addressability properties of NVM in their storage methods. Chapter 4 makes the case for a new logging and recovery protocol, called write-behind logging, that enables a DBMS to recover nearly instantaneously from system failures. We discuss our proposed work on data migration in Chapter 5. Chapter 6 presents a discussion of the related work. We conclude with a proposed timeline in Chapter 7.

Chapter 2

Motivation

This chapter presents our investigation of the impact of NVM on existing DBMSs. We explore two possible use cases of NVM for OLTP DBMSs. The first is where NVM completely replaces DRAM and the other is where NVM and DRAM coexist in the system. For each configuration, we compare the performance of a disk-oriented DBMS with a memory-oriented DBMS using OLTP benchmarks.

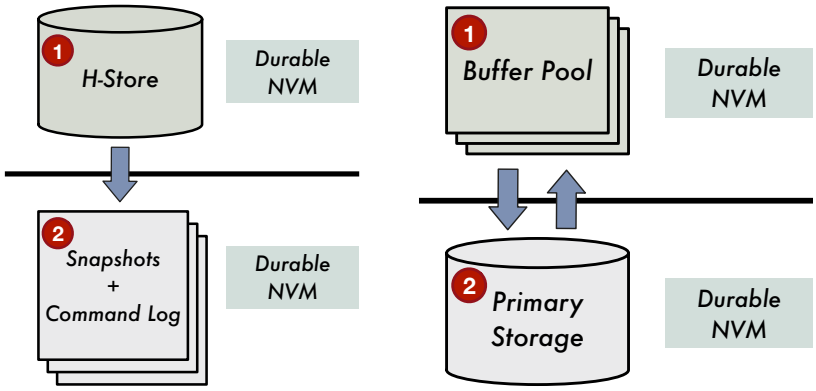
2.1 NVM Hardware Emulator

For this study, we use a NVM hardware emulator developed by Intel Lab. The emulator is implemented on a dual-socket Intel® Xeon® processor-based platform. Each processor has eight cores that run at 2.6 GHz and supports four DDR3 channels with two DIMMs per channel. The emulator's custom BIOS partitions the available DRAM memory into emulated NVM and regular (volatile) memory. Half of the memory channels on each processor are reserved for emulated NVM while the rest are used for regular memory. The emulated NVM is visible to the OS as a single NUMA node that interleaves memory (i.e., cache lines) across the two sockets. We are also able to configure the latency and bandwidth for the NVM partition by writing to CPU registers through the OS kernel. We use special CPU microcode and an emulation model for latency emulation where the amount of bandwidth throttling in the memory controller is programmable. We refer interested readers to [43] for more technical details on the emulator.

We divide the NVM partition into two sub-partitions. The first sub-partition is available to software as a NUMA node. The second sub-partition is managed by *PMFS*, a file system optimized for persistent memory [43]. Applications allocate and access memory in the first sub-partition using `libnuma` library or tools such as `numactl` [2]. We refer to this interface provided by the emulator as the *NUMA interface*. Applications can also use regular POSIX file system interface to allocate and access memory in the second sub-partition through the *PMFS interface*.

2.2 NVM-Only Architecture

In the NVM-only architecture, the DBMS uses NVM exclusively for its storage(see Figure 2.1). We compare a memory-oriented DBMS with a disk-oriented DBMS when both are running entirely on NVM storage using the emulator's NUMA interface. For the former, we use the H-Store



(a) Memory-oriented system (b) Disk-oriented system

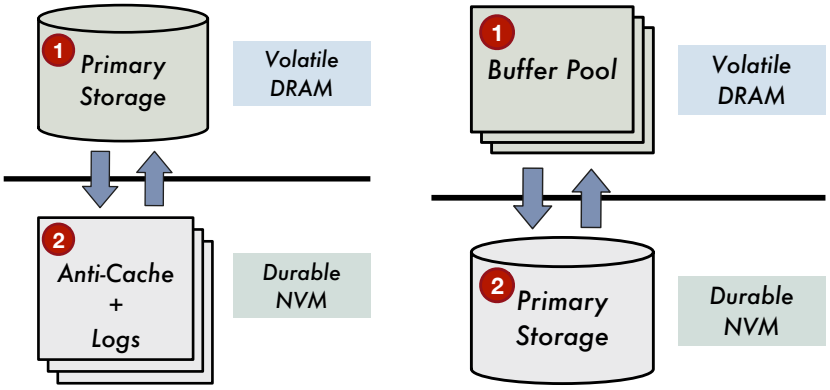
Figure 2.1: NVM-Only Architecture

DBMS [1], while for the latter we use MySQL (v5.5) with the InnoDB storage engine. Both systems are tuned according to their "best practice" guidelines for OLTP workloads.

The NVM-only architecture has implications for the DBMS's recovery scheme. In all DBMSs, some form of logging is used to guarantee recoverability in the event of a failure [45]. Disk-oriented DBMSs provide durability through the use of a write-ahead log, which is a type of *physical logging* wherein updated versions of data are logged to disk with each write operation. Such an approach has a significant performance overhead for main memory-oriented DBMSs [51, 64]. Thus, others have argued for the use of *logical logging* for main memory DBMSs where the log contains a record of the high-level operations that each transaction executed. The overhead of writing out logical log records and the size of the log itself is much smaller for logical logging. The downside, however, is that the recovery process takes longer because the DBMS must re-execute each transaction to restore the database state. In contrast, during recovery in a physical logging system, the log is replayed forward to redo the effects of committed transactions and then replayed backwards to undo the effects of uncommitted transactions [45, 66]. But since all writes to memory are persistent under the NVM-only configuration, heavyweight logging protocols such as these are excessive and inefficient.

2.3 NVM+DRAM Architecture

In this configuration, the DBMS relies on both DRAM and NVM for satisfying its storage requirements. If we assume that the entire dataset cannot fit in DRAM, the question arises of how to split data between the two storage layers. Because of the relative latency advantage of DRAM over NVM, one strategy is to attempt to keep the hot data in DRAM and the cold data in NVM. One way is to use a buffer pool to cache hot data, as in traditional disk-oriented DBMSs. With this architecture, there are two copies of cached data, one persistent copy on disk and another copy cached in the DRAM-based buffer pool. The DBMS copies pages into the buffer pool as they are needed, and then writes out dirty pages to the NVM for durability. Another approach is to use the *anti-caching* system design proposed in [38] where all data initial resides in memory and then cold data is evicted out to disk over time. One key difference in this design is that exactly one copy of the data exists at any point in time. Thus, a tuple is either in memory or in the anti-cache. An overview of these two architectures is shown in Figure 2.2.



(a) Anti-Caching system (b) Disk-oriented system

Figure 2.2: NVM+DRAM Architecture

2.4 Evaluation

To evaluate these different memory configuration and DBMS designs, we performed a series of experiments on the NVM emulator. We deployed four different system configurations: two executing entirely on NVM and two executing on a hybrid NVM+DRAM hierarchy. For the NVM-only analysis, we configured MySQL to execute entirely out of NVM and have compared it with H-Store configured to execute entirely in NVM. For the NVM+DRAM hierarchy analysis, we configured MySQL to use a DRAM-based buffer pool and store all persistent data in PMFS. As a comparison, we implemented the NVM adaptations to the anti-caching system described above by modifying the original H-Store based anti-caching implementation. We used two benchmarks in our evaluation and a range of different configuration parameters.

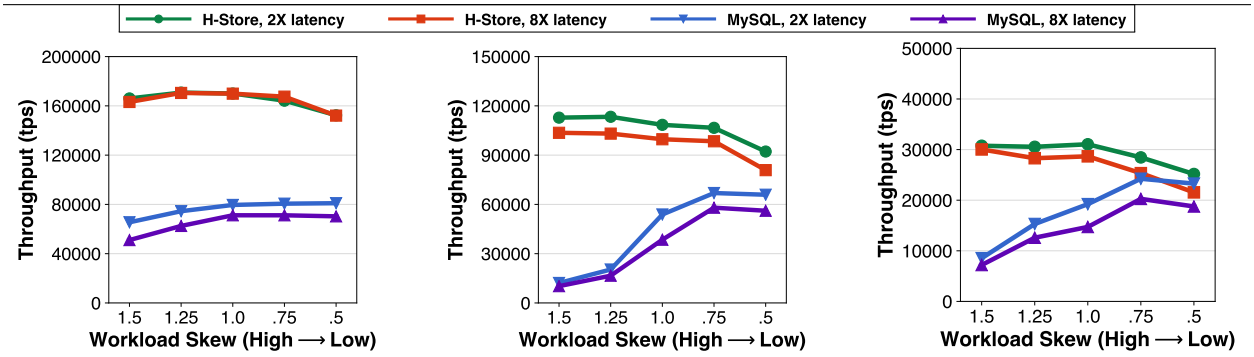
2.4.1 System Configuration

All experiments were conducted on the NVM emulator described in Section 2.1. For each system, we evaluate the benchmarks on two different NVM latencies: 2× DRAM and 8× DRAM, where the base DRAM latency is approximately 90 ns. We consider these latencies to represent the best case and worst case NVM latencies respectively [43]. We chose this range of latencies to make our results as independent from the underlying NVM technology as possible.

2.4.2 Benchmarks

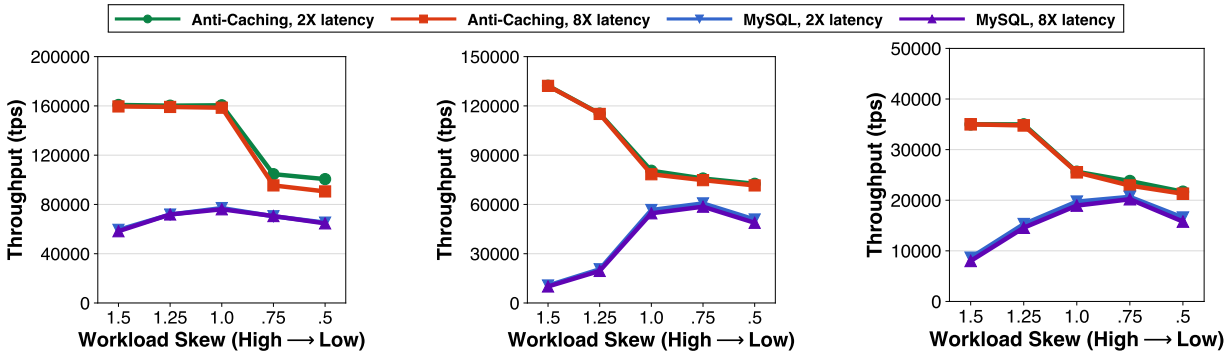
We now describe the two benchmarks that we use for our evaluation. We use H-Store's internal benchmarking framework for both the H-Store on NVM and the anti-caching analysis. For the MySQL benchmarking, we use the OLTP-Bench [41] framework.

YCSB: The Yahoo! Cloud Services Benchmark (YCSB) is a workload that is representative of large-scale services provided by web-scale companies. It is a key-value store workload. We configure each tuple to consist of a unique key and 10 columns of random string data, each 100 bytes in size. Thus, the total size of a tuple is approximately 1KB. The workload used for this analysis consists of two transaction types, a `read` and an `update` transaction. The read randomly selects a key and reads a single tuple. The update randomly selects a key and updates all 10 non-key values for the tuple



(a) Read-Only (b) Read-Heavy (c) Write-Heavy

Figure 2.3: NVM-only Architecture – YCSB.



(a) Read-Only $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$ (b) Read-Heavy $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$ (c) Write-Heavy $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$

Figure 2.4: NVM+DRAM Architecture – YCSB.

selected. The mix of read and update transactions in a workload is an important parameter in our analysis, as writes are much more costly, especially if data in the buffer pool must be kept consistent. We use three different workload mixtures:

- **Read-Heavy:** 90% reads, 10% updates
- **Write-Heavy:** 50% reads, 50% updates
- **Read-Only:** 100% reads

In addition to the read-write mix, we also control the amount of skew that determines how often a tuple is accessed by transactions. We use YCSB's Zipfian distribution to model temporal skew in the workloads, meaning that newer items are accessed much more frequently than older items. The amount of skew is controlled by the Zipfian constant $s > 0$, where higher values of s generate higher skewed workloads. We pick values of s in the range of 0.5 to 1.5, which is representative of a range of skewed workloads.

TPC-C: This benchmark is an industry standard for evaluating the performance of OLTP systems [87]. The benchmark simulates an order-processing application, and consists of nine tables and five different transaction types. Only two of the transaction types modify tuples, but they make up 88% of a TPC-C workload. We use 100 warehouses and 100,000 items, resulting in a total data size of 10GB. For simplicity, we have configured transactions to only access data from a single warehouse.

Thus, all transactions are single-sited (i.e., there are no distributed transactions) because warehouses are mapped to partitions. For the anti-cache trials, we evict data from the `HISTORY`, `ORDERS`, and `ORDER_LINE` tables, as these are the only tables where transactions insert new data.

We will now discuss the results of executing the two benchmarks, YCSB and TPC-C, on each of the NVM-only and NVM-DRAM architectures described in Sections 2.2 and 2.3.

2.4.3 NVM-Only Architecture

YCSB: We evaluate YCSB on each system across the range of skew parameters and workload mixtures described above. We first consider the impact of NVM latency on the throughput of memory-oriented and disk-oriented systems. The results for the read-heavy workload shown in Figure 2.3b indicate that increasing NVM latency decreases throughput of H-Store and MySQL by 12.3% and 14.8% respectively. There is no significant impact on H-Store's performance in the read-only workload shown in Figure 2.3a, which indicates that latency mainly impacts the performance of logging.

The throughput of these systems vary with the amount of skew in the workload. The impact of skew on H-Store's performance is more pronounced in the read-heavy workload shown in Figure 2.3b. Throughput drops by 18.2% in the read-heavy workload as the skew level is reduced. The drop in throughput is due to the application's larger working set size, which increases the number of cache misses and subsequent accesses to NVM. In contrast, MySQL performs poorly on high-skew workloads but its throughput improves by 5× as skew decreases. This is because a disk-oriented system uses locks to allow transactions to execute concurrently. Thus, if a large portion of the transactions access the same tuples, then lock contention becomes a bottleneck.

We can summarize the above observations as follows: (1) increasing NVM latency mainly impacts the performance of the logging mechanism, and (2) the throughput of memory-oriented and disk-oriented systems vary differently as skew decreases. We contend that the ideal system for a NVM-only architecture will possess features of both memory-oriented and disk-oriented systems.

TPC-C: For the TPC-C benchmark, most transactions insert or access new records (i.e., `NewOrder`), and older records are almost never accessed. As such, there is strong temporal skew built into the semantics of the benchmark. Only a subset of the tables are actually increasing in size, and the rest are static. In Figure 2.5a, we see that throughput of both systems only varies slightly with an increase in NVM latency, and that for both latencies the throughput of H-Store is 10× higher than that of the disk-oriented system.

2.4.4 NVM+DRAM Architecture

YCSB: We use the same YCSB skew and workload mixes, but configure the amount of DRAM available to the DBMSs to be $\frac{1}{8}$ of the total database size. There are several conclusions to draw from the results shown in Figure 2.4. The first is that the throughput of the two systems trend differently as skew changes. For the read-heavy workload in Figure 2.4b, anti-caching achieves 13× higher throughput over MySQL when skew is high, but only a 1.3× improvement when skew is low. Other workload mixes have similar trends. This is because the anti-caching system performs best when there is high skew since it needs to fetch fewer blocks and restart fewer transactions. In contrast,

the disk-oriented system performs worse on the high skew workloads due to high lock contention. We note that at the lowest skew level, MySQL's throughput decreases due to lower hit rates for data in the CPU's caches.

Another notable finding is that both systems do not exhibit a major change in performance with longer NVM latencies. This is significant, as it implies that neither architecture is bottlenecked by the I/O on the NVM. Instead, the decrease in performance is due to the overhead of fetching and evicting data from NVM. For the disk-oriented system, this overhead comes from managing the buffer pool, while in the anti-caching system it is from restarting transactions and asynchronously fetching previously evicted data.

We can summarize the above observations as follows: (1) the throughput of the anti-caching system decreases as skew decreases, (2) the throughput of the disk-oriented system increases as skew decreases, and (3) neither architecture is bottlenecked by I/O when the latency of NVM is between 2-8× the latency of DRAM. Given these results, we believe that the ideal system architecture for a NVM+DRAM memory hierarchy would need to posses features of both anti-caching and disk-oriented systems to enable it to achieve high throughput regardless of skew.

TPC-C: We next ran the TPC-C benchmark on the anti-caching and disk-oriented DBMSs using different NVM latencies. The results in Figure 2.5b show that the throughput of both DBMSs do not change significantly as NVM latency increases. This is expected, since all of the transactions' write operations are initially stored on DRAM. These results corroborate previous studies that have shown the 10× performance advantage of an anti-caching system over the disk-oriented DBMS [38]. For the anti-caching system, this workload essentially measures how efficiently it is able to evict data to PMFS (since no transaction reads old data).

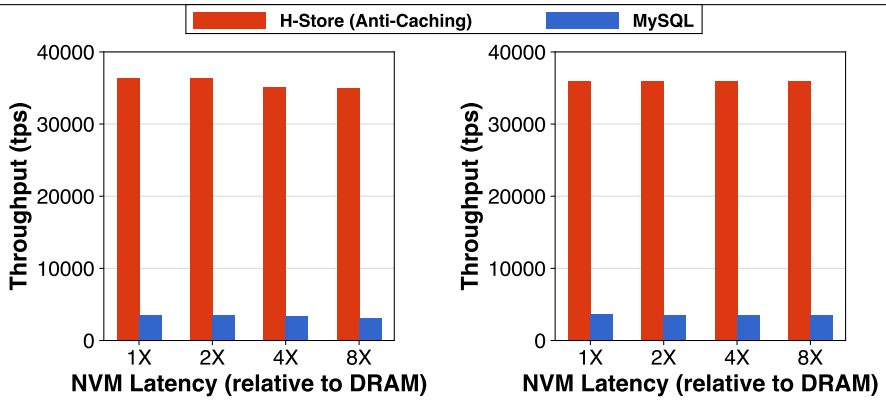
2.4.5 Recovery

Lastly, we evaluate recovery schemes in H-Store using the emulator's NUMA interface. We implemented logical logging (i.e., command logging) and physical logging (i.e., ARIES) recovery schemes within H-Store. For each scheme, we first measure the DBMS's runtime performance when executing a fixed number of TPC-C transactions (50,000). We then simulate a system failure and then measure how long it takes the DBMS to recover the database state from each scheme's corresponding log stored on PMFS.

For the runtime measurements, the results in Figure 2.6a show that H-Store achieves 2× higher throughput when using logical logging compared to physical logging. This is because logical logging only records the executed commands and thus is more lightweight. The amount of logging data for the workload using scheme is only 5MB. In contrast, physical logging keeps track of all modifications made at tuple-level granularity and its corresponding log 220MB. This reduced footprint makes logical logging more attractive for the first NVM devices that are expected to have limited capacities.

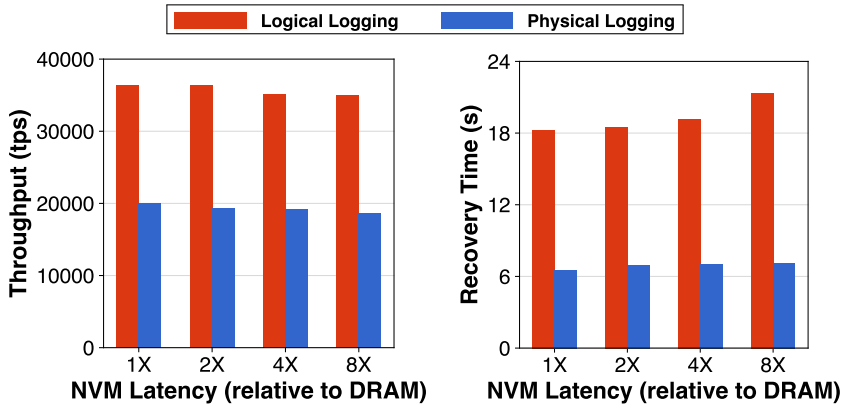
Next, in the results for the recovery times, Figure 2.6b shows that logical logging 3× is slower than physical logging. One could reduce this time in logical logging by having the DBMS checkpoint more frequently, but this will impact steady-state performance [64].

We note that both schemes are essentially doing unnecessary work, since all writes to memory when using the NUMA interface are potentially durable. A better approach is to use a recovery



(a) NVM-only Architecture (b) NVM+DRAM Architecture

Figure 2.5: NVM Latency Evaluation – Performance comparison for the TPC-C benchmark using different NVM latencies.



(a) Throughput (b) Recovery Time

Figure 2.6: Recovery Evaluation – Comparison of recovery schemes in H-Store using the TPC-C benchmark.

scheme that is designed for NVM. This would allow a DBMS to combine the faster runtime performance of logical logging with the faster recovery of physical logging.

2.5 Summary

In this chapter, we presented the results of our investigation on the impact of NVM on existing DBMSs. We explored two possible architectures using non-volatile memory (i.e., NVM-only and NVM+DRAM architectures). For each architecture, we evaluated memory-oriented and disk-oriented OLTP DBMSs. Our analysis shows that memory-oriented systems are better-suited to take advantage of NVM and outperform their disk-oriented counterparts. However, in both the NVM-only and NVM+DRAM architectures, the throughput of the memory-oriented systems decreases as workload skew is decreased while the throughput of the disk-oriented architectures increases as workload skew is decreased. Because of this, we conclude that neither system is ideally suited for NVM. Instead, a new system is needed with principles of both disk-oriented and memory-oriented systems and a lightweight recovery scheme designed to utilize the non-volatile property of NVM.

In the remainder of this dissertation, we will present the design and implementation of such a DBMS that leverages the properties of NVM in its storage and recovery methods.

Chapter 3

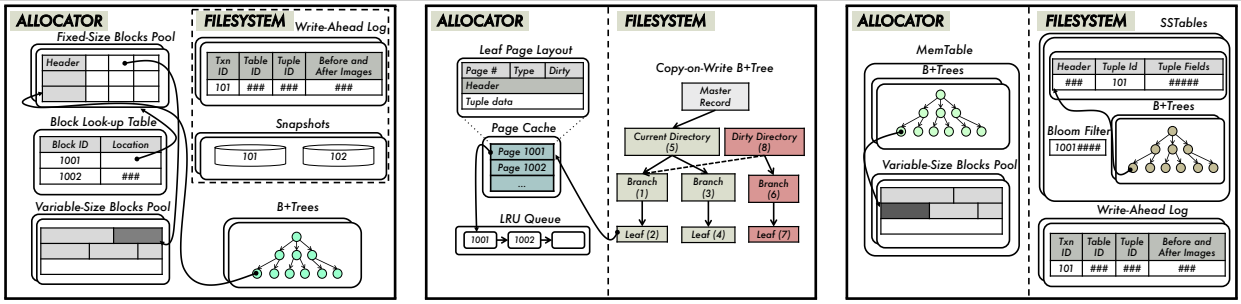
Storage Management

In this chapter, we explore the fundamentals of storage and recovery methods in OLTP DBMSs running on an NVM-only storage hierarchy. Changes in computer trends have given rise to new OLTP applications that support a large number of concurrent users and systems. What makes these modern applications unlike their predecessors is the scale in which they ingest information [58]. DBMSs are the critical component of these applications because they are responsible for ensuring transactions' operations execute in the correct order and that their changes are not lost after a crash. Optimizing the DBMS's performance is important because it determines how quickly an application can take in new information and how quickly it can use it to make new decisions. This performance is affected by how fast the system can read and write data from storage.

DBMSs have always dealt with the trade-off between volatile and non-volatile storage devices. In order to retain data after a loss of power, the DBMS must write that data to a non-volatile device, such as a SSD or HDD. Such devices only support slow, bulk data transfers as blocks. Contrast this with volatile DRAM, where a DBMS can quickly read and write a single byte from these devices, but all data is lost once power is lost. In addition, there are inherent physical limitations that prevent DRAM from scaling to capacities beyond today's levels [65]. *Non-volatile memory* (NVM) offers an intriguing blend of the two storage mediums. NVM is a broad class of technologies, including phase-change memory [81], memristors [85], and STT-MRAM [42] that provide low latency reads and writes on the same order of magnitude as DRAM, but with persistent writes and large storage capacity like a SSD [22].

It is unclear at this point, however, how to best leverage these new technologies in a DBMS. There are several aspects of NVM that make existing DBMS architectures inappropriate for them [24, 37]. For example, disk-oriented DBMSs (e.g., Oracle RDBMS, IBM DB2, MySQL) are predicated on using block-oriented devices for durable storage that are slow at random access. As such, they maintain an in-memory cache for blocks of tuples and try to maximize the amount of sequential reads and writes to storage. In the case of memory-oriented DBMSs (e.g., VoltDB, MemSQL), they contain certain components to overcome the volatility of DRAM. Such components may be unnecessary in a system with byte-addressable NVM with fast random access.

In this chapter, we evaluate different storage and recovery methods for OLTP DBMSs from the ground-up, starting with an NVM-only storage hierarchy. We implemented three storage engine architectures in a single DBMS: (1) in-place updates with logging, (2) copy-on-write updates



(a) In-place Updates (InP) (b) Copy-on-Write Updates (CoW) (c) Log-structured Updates (Log)

Figure 3.1: Architectural layout of the three traditional storage engines supported in the DBMS testbed. The engine components accessed using the allocator interface and those accessed using the filesystem interface are bifurcated by the dashed line.

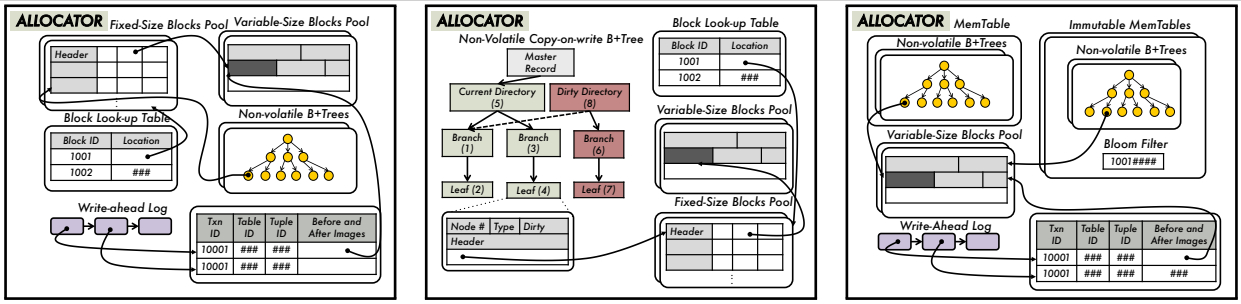
without logging, and (3) log-structured updates. We then developed optimized variants for these approaches that reduce the computational overhead, storage footprint, and wear-out of NVM devices. For our evaluation, we use a hardware-based emulator where the system only has NVM and volatile CPU-level caches (i.e., no DRAM). Our analysis shows that the NVM-optimized storage engines improve the DBMS's throughput by a factor of 5.5× while reducing the number of writes to NVM in half. These results also suggest that NVM-optimized in-place updates is the ideal method as it has lowest overhead, causes minimal wear on the device, and allows the DBMS to restart almost instantaneously.

The remainder of this chapter is organized as follows. We begin in Section 3.1 with a description of our DBMS testbed and its storage engines that we developed for this study. We then present in Section 3.2 our optimizations for these engines that leverage NVM's unique properties. We then present our experimental evaluation in Section 3.3.

3.1 DBMS Testbed

We developed a lightweight DBMS to evaluate different storage architecture designs for OLTP workloads. We did not use an existing DBMS as that would require significant changes to incorporate the storage engines into a single system. Although some DBMSs support a pluggable storage engine back-end (e.g., MySQL, MongoDB), modifying them to support NVM would still require significant changes. We also did not want to taint our measurements with features that are not relevant to our evaluation.

The DBMS's internal coordinator receives incoming transaction requests from the application and then invokes the target stored procedure. As a transaction executes in the system, it invokes queries to read and write tuples from the database. These requests are passed through a query executor that invokes the necessary operations on the DBMS's active storage engine. Using a DBMS that supports a pluggable back-end allows us to compare the performance characteristics of different storage and recovery methods on a single platform. We implemented three storage engines that use different approaches for supporting durable updates to a database: (1) *in-place* updates engine, (2) *copy-on-write* updates engine, and (3) *log-structured* updates engine. Each engine also supports both primary and secondary indexes.



(a) In-place Updates (NVM-InP) (b) Copy-on-Write Updates (NVM-CoW) (c) Log-structured Updates (NVM-Log)

Figure 3.2: NVM-Aware Engines – Architectural layout of the NVM-optimized storage engines.

We now describe these engines in detail. For each engine, we first discuss how they apply changes made by transactions to the database and then how they ensure durability after a crash. All of these engines are based on the architectures found in state-of-the-art DBMSs. That is, they use memory obtained using the allocator interface as volatile memory and do not exploit NVM's persistence. Later in Section 3.2, we present our improved variants of these engines that are optimized for NVM.

3.1.1 In-Place Updates Engine (InP)

The first engine uses the most common storage engine strategy in DBMSs. With *in-place* updates, there is only a single version of each tuple at all times. When a transaction updates a field for an existing tuple, the system writes the new value directly on top of the original one. This is the most efficient method of applying changes, since the engine does not make a copy of the tuple first before updating it and only the updated fields are modified. The design of this engine is based on VoltDB [5], which is a memory-oriented DBMS that does not contain legacy disk-oriented DBMS components like a buffer pool. The InP engine uses the STX B+tree library for all of its indexes [21].

3.1.2 Copy-on-Write Updates Engine (CoW)

The second storage engine performs *copy-on-write* updates where instead of modifying the original tuple, it creates a copy of the tuple and then modifies that copy. As the CoW engine never overwrites committed data, it does not need to record changes in a WAL for recovery. The CoW engine instead uses different look-up *directories* for accessing the versions of tuples in the database. With this approach, known as *shadow paging* in IBM's System R [49], the DBMS maintains two look-up directories at all times: (1) the *current* directory, and (2) the *dirty* directory. The current directory points to the most recent versions of the tuples and only contains the effects of committed transactions. The dirty directory points to the versions of tuples being modified by active transactions. To ensure that the transactions are isolated from the effects of uncommitted transactions, the engine maintains a *master record* that always points to the current directory. Figure 3.1b presents the architecture of the CoW engine. After applying the changes on the copy of the tuple, the engine updates the dirty directory to point to the new version of the tuple. When the transaction commits, the engine updates the master record atomically to point to the dirty directory. The engine maintains an internal page cache to keep the hot pages in memory.

3.1.3 Log-structured Updates Engine (Log)

Lastly, the third storage engine uses a *log-structured* update policy. This approach originated from log-structured filesystems [83], and then it was adapted to DBMSs as *log-structured merge* (LSM) trees [71] for write-intensive workloads. The LSM tree consists of a collection of *runs* of data. Each run contains an ordered set of entries that record the changes performed on tuples. Runs reside either in volatile memory (i.e., *MemTable*) or on durable storage (i.e., *SSTables*) with their storage layout optimized for the underlying storage device. The LSM tree reduces write amplification by batching the updates in MemTable and periodically cascading the changes to durable storage [71]. The design for our Log engine is based on Google's LevelDB [35], which implements the log-structured update policy using LSM trees.

3.2 NVM-Aware Engines

All of the engines described above are derived from existing DBMS architectures that are predicated on a two-tier storage hierarchy comprised of volatile DRAM and a non-volatile HDD/SSD. These storage devices have distinct hardware constraints and performance properties [78]. First, the read and write latency of non-volatile storage is several orders of magnitude higher than DRAM. Second, the DBMS accesses data on non-volatile storage at block-granularity, while with DRAM it accesses data at byte-granularity. Third, the performance gap between sequential and random accesses is greater for non-volatile storage compared to DRAM.

The traditional engines were designed to account for and reduce the impact of these differences. For example, they maintain two layouts of tuples depending on the storage device. Tuples stored in memory can contain non-inlined fields because DRAM is byte-addressable and handles random accesses efficiently. In contrast, fields in tuples stored on durable storage are inlined to avoid random accesses because they are more expensive. To amortize the overhead for accessing durable storage, these engines batch writes and flush them in a deferred manner.

Many of these techniques, however, are unnecessary in a system with a NVM-only storage hierarchy [37, 43, 67]. As shown in Table 1.1, the access latencies of NVM are orders of magnitude shorter than that of HDDs and SSDs. Further, the performance gap between sequential and random accesses on NVM is comparable to that of DRAM. We therefore adapt the storage and recovery mechanisms of these traditional engines to exploit NVM's characteristics. We refer to these optimized storage engines as the *NVM-aware* engines. As we show in our evaluation in Section 3.3, these engines deliver higher throughput than their traditional counterparts while still ensuring durability. They reduce write amplification using NVM's persistence thereby expanding the lifetime of the NVM device.

3.2.1 In-Place Updates Engine (NVM-InP)

One of the main problems with the InP engine described in Section 3.1.1 is that it has high rate of data duplication. When a transaction inserts a tuple, the engine records the tuple's contents in the WAL and then again in the table storage area. The InP engine's logging infrastructure also assumes that the system's durable storage device has orders of magnitude higher write latency compared to DRAM. It therefore batches multiple log records and flushes them periodically to the WAL using

sequential writes. This approach, however, increases the mean response latency as transactions need to wait for the group commit operation.

Given this, we designed the NVM-InP engine to avoid these issues. Now when a transaction inserts a tuple, rather than copying the tuple to the WAL, the NVM-InP engine only records a non-volatile pointer to the tuple in the WAL. This is sufficient because both the pointer and the tuple referred to by the pointer are stored on NVM. Thus, the engine can use the pointer to access the tuple after the system restarts without needing to re-apply changes in the WAL. It also stores indexes as non-volatile B+trees that can be accessed immediately when the system restarts without rebuilding.

3.2.2 Copy-on-Write Updates Engine (NVM-CoW)

The original CoW engine stores tuples in self-containing blocks without pointers in the copy-on-write B+tree on the filesystem. The problem with this engine is that the overhead of propagating modifications to the dirty directory is high; even if a transaction only modifies one tuple, the engine needs to copy the entire block to the filesystem. When a transaction commits, the CoW engine uses the filesystem interface to flush the dirty blocks and updates the master record (stored at a fixed location in the file) to point to the root of the dirty directory [28]. These writes are expensive as they need to switch the privilege level and go through the kernel's VFS path.

The NVM-CoW engine employs three optimizations to reduce these overheads. First, it uses a non-volatile copy-on-write B+tree that it maintains using the allocator interface. Second, the NVM-CoW engine directly persists the tuple copies and only records non-volatile tuple pointers in the dirty directory. Lastly, it uses the lightweight durability mechanism of the allocator interface to persist changes in the copy-on-write B+tree.

3.2.3 Log-structured Updates Engine (NVM-Log)

The Log engine batches all writes in the MemTable to reduce random accesses to durable storage [61, 71]. The benefits of this approach, however, are not as evident for a NVM-only storage hierarchy because the performance gap between sequential and random accesses is smaller. The original log-structured engine that we described in Section 3.1.3 incurs significant overhead from periodically flushing MemTable to the filesystem and compacting SSTables to bound read amplification. Similar to the NVM-InP engine, the NVM-Log engine records all the changes performed by transactions on a WAL stored on NVM.

Our NVM-Log engine avoids data duplication in the MemTable and the WAL as it only records non-volatile pointers to tuple modifications in the WAL. Instead of flushing MemTable out to the filesystem as a SSTable, it only marks the MemTable as immutable and starts a new MemTable. This immutable MemTable is physically stored in the same way on NVM as the mutable MemTable. The only difference is that the engine does not propagate writes to the immutable MemTables. We also modified the compaction process to merge a set of these MemTables to generate a new larger MemTable. The NVM-Log engine uses a NVM-aware recovery protocol that has lower recovery latency than its traditional counterpart.

3.3 Evaluation

In this section, we present our analysis of the six different storage engine implementations. Our DBMS testbed allows us to evaluate the throughput, the number of reads/writes to the NVM device, the storage footprint, and the time that it takes to recover the database after restarting.

The experiments were all performed on Intel Lab's NVM hardware emulator [43]. It contains a dual-socket Intel Xeon E5-4620 processor. Each socket has eight cores running at 2.6 GHz. It dedicates 128 GB of DRAM for the emulated NVM and its L3 cache size is 20 MB. We use the Intel memory latency checker [88] to validate the emulator's latency and bandwidth settings. The engines access NVM storage using the allocator and filesystem interfaces of the emulator as described in Chapter 2. We set up the DBMS to use eight partitions in all of the experiments. We configure the node size of the STX B+tree and the CoW B+tree implementations to be 512 B and 4 KB respectively. All transactions execute with the same serializable isolation level and durability guarantees.

3.3.1 Benchmarks

We first describe the benchmark we use in our evaluation. The tables in each database are partitioned in such way that there are only single-partition transactions [76].

TPC-C: This benchmark is the current industry standard for evaluating the performance of OLTP systems [87]. It simulates an order-entry environment of a wholesale supplier. The workload consists of five transaction types, which keep track of customer orders, payments, and other aspects of a warehouse. Transactions involving database modifications comprise around 88% of the workload. We configure the workload to contain eight warehouses and 100,000 items. We map each warehouse to a single partition. The initial storage footprint of the database is approximately 1 GB.

3.3.2 Runtime Performance

We begin with an analysis of the impact of NVM's latency on the performance of the storage engines. To obtain insights that are applicable for various NVM technologies, we run the TPC-C benchmark under three latency configurations on the emulator: (1) default DRAM latency configuration (160 ns), (2) a *low* NVM latency configuration that is 2× higher than DRAM latency (320 ns), and (3) a *high* NVM latency configuration that is 8× higher than DRAM latency (1280 ns). We execute all workloads three times on each engine and report the average throughput.

TPC-C: Figure 3.3 shows the engines' throughput while executing TPC-C under different latency configurations. Among all the engines, the NVM-InP engine performs the best. The NVM-aware engines are 1.8–2.1× faster than the traditional engines. The NVM-CoW engine exhibits the highest speedup of 2.3× over the CoW engine. We attribute this to the write-intensive nature of the TPC-C benchmark. Under the high NVM latency configuration, the NVM-aware engines deliver 1.7–1.9× higher throughput than their traditional counterparts.

3.3.3 Recovery

In this experiment, we evaluate the recovery latency of the storage engines. For each benchmark, we first execute a fixed number of transactions and then force a hard shutdown of the DBMS

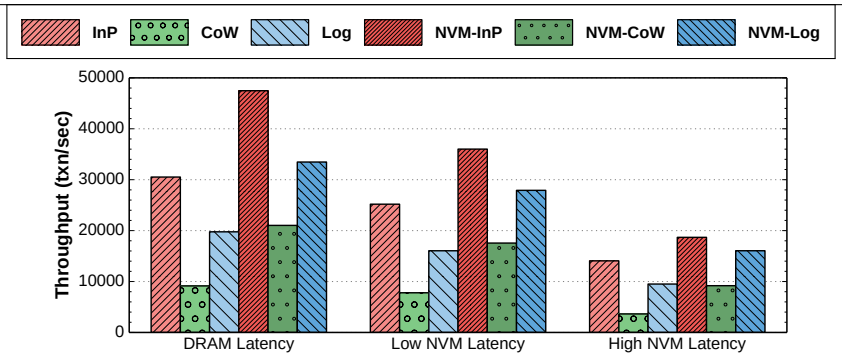


Figure 3.3: TPC-C Throughput – The performance of the engines for TPC-C benchmark for all three NVM latency settings.

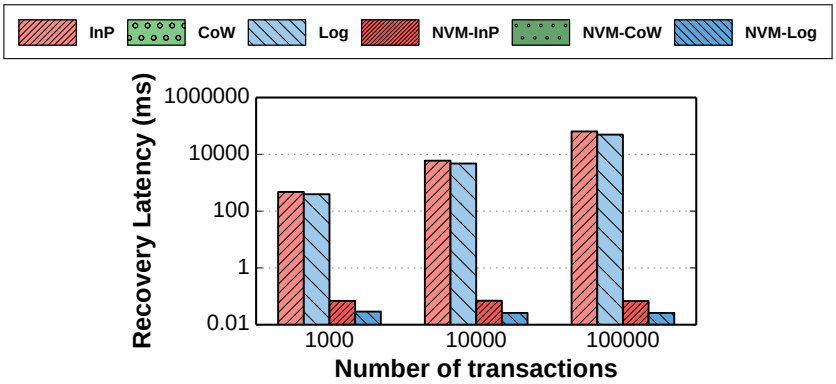


Figure 3.4: Recovery Latency – The amount of time that the engines take to restore the database to a consistent state after a restart.

(`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. The number of transactions that need to be recovered by the DBMS depends on the frequency of checkpointing for the InP engine and on the frequency of flushing the MemTable for the Log engine. The CoW and NVM-CoW engines do not perform any recovery mechanism after the OS or DBMS restarts because they never overwrite committed data. They have to perform garbage collection to clean up the previous dirty directory. This is done asynchronously and does not have a significant impact on the throughput of the DBMS.

TPC-C: The results for the TPC-C benchmark are shown in Section 3.3.3. The recovery latency of the NVM-InP and NVM-Log engines is slightly higher than that in the YCSB benchmark because the TPC-C transactions perform more operations. However, the latency is still independent of the number of transactions executed unlike the traditional engines because the NVM-aware engines ensure that the effects of committed transactions are persisted immediately.

3.4 Summary

We explored the fundamentals of storage and recovery methods in OLTP DBMSs running on an NVM-only storage hierarchy in this chapter. We implemented three storage engines in a modular DBMS testbed with different architectures: (1) in-place updates, (2) copy-on-write updates, and (3) log-structured updates. We then developed optimized variants of each of these engines that better make use of NVM's characteristics. Our experimental analysis with two different OLTP workloads showed that our NVM-aware engines outperform the traditional engines by up to 5.5× while reducing the number of writes to the storage device by more than half on write-intensive workloads. We found that the NVM access latency has the most impact on the runtime performance of the engines, more so than the workload skew or the number of modifications to the database in the workload. Our evaluation showed that the NVM-aware in-place updates engine achieved the best throughput among all the engines with the least amount of wear on the NVM device.

This chapter focused on an NVM-only storage hierarchy. In case of high NVM latency technologies and analytical workloads, a hybrid DRAM and NVM storage hierarchy is another viable alternative. In the next chapter, we will describe how our NVM-aware engines can be expanded to run on such a hybrid storage hierarchy.

Chapter 4

Logging and Recovery

In this chapter, 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 [13].

A DBMS protects the database state from corruption due to application, operating system, and device failures [45]. 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 volatile DRAM, but all data on DRAM is lost after a power failure.

These differences between the two types of storage are a major factor in the design of DBMS architectures [39, 46]. 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* [20, 66].

But 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 [22]. 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 [37].

We present a new logging and recovery protocol that is designed for a hybrid storage hierarchy with NVM and DRAM in this chapter. To evaluate our approach, we implemented it in the Pelo-

ton [3] 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 chapter is organized as follows. We begin in Section 4.1 with a discussion of the ubiquitous WAL protocol, followed by our new WBL method in Section 4.2. We present our experimental evaluation in Section 4.3. 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.

4.1 Write-Ahead Logging

The most well-known recovery method based on WAL is the ARIES protocol developed by IBM in the 1990s [66]. ARIES is a *physiological logging* protocol where the DBMS combines a physical redo process with a logical undo process [45]. 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 [14, 69]. MVCC is the most widely used concurrency control scheme in DBMSs developed in the last decade, including Hekaton [40], 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 4.1 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 4.1, 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

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

Figure 4.1: 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 4.2: Structure of WAL Record – Structure of the log record constructed by the DBMS while using the WAL protocol.

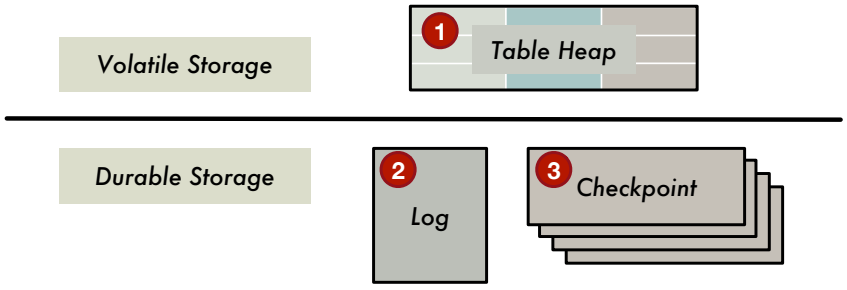


Figure 4.3: WAL Commit Protocol – The ordering of writes from the DBMS to durable storage while employing the WAL protocol.

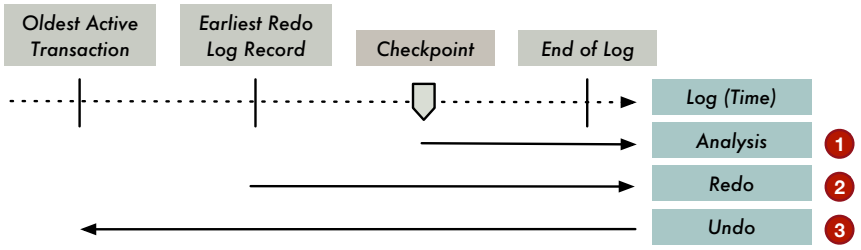


Figure 4.4: WAL Recovery Protocol – The phases of the recovery protocol.

identifier 305 holds a latch on the tuple with ID 103. See [14, 59] 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. Later, in Section 4.2, we present our WBL protocol for NVM systems.

4.1.1 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 4.2, 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. 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.

With an in-memory DBMS, transactions access tuples through pointers without indirection through a buffer pool [14]. 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 [79]. 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.

4.1.2 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 4.3. 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 [39]. 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.

4.1.3 Recovery Protocol

The traditional WAL recovery algorithm (see Figure 4.4) comprises of three phases: (1) analysis, (2) redo, and (3) undo. In the *analysis phase*, the DBMS processes the log starting from the latest

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 4.5: WAL Example – Contents of the WAL during recovery.

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 4.5 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 [10, 45].

Although WAL supports efficient transaction processing when memory is volatile and durable storage cannot support fast random writes, it is inefficient for NVM storage [13]. 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.

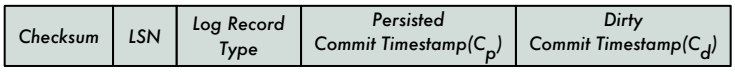


Figure 4.6: Structure of WBL Record – Structure of the log record constructed by the DBMS while using the WBL protocol.

4.2 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. Although our description of WBL is for MVCC DBMSs, we also discuss how to adapt the protocol for a single-version system.

4.2.1 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 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.

4.2.2 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

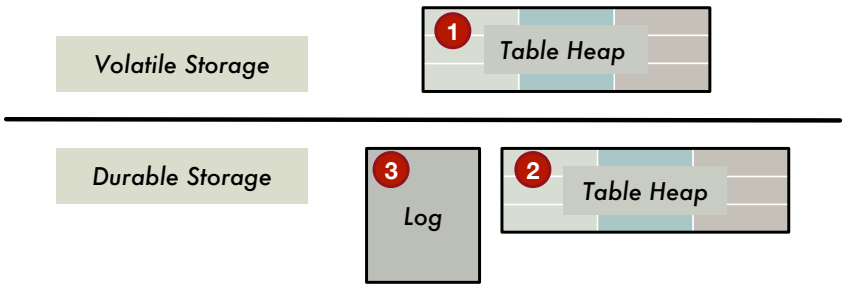


Figure 4.7: WBL Commit Protocol – The ordering of writes from the DBMS to durable storage while employing the WBL protocol.

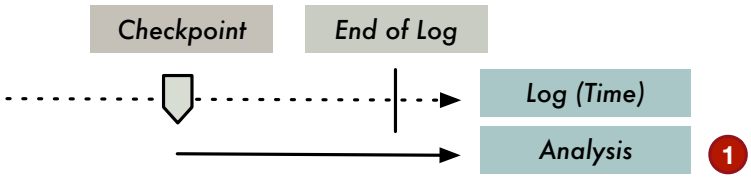


Figure 4.8: WBL Recovery Protocol – The phases of the recovery protocol.

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, 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. 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 4.6) 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.

The diagram in Figure 4.7 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,

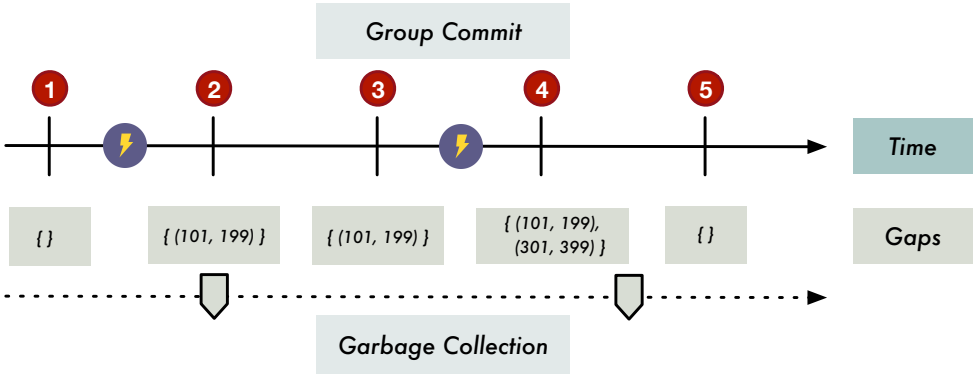


Figure 4.9: WBL Commit Timestamp Gaps – An illustration of successive system failures resulting in multiple commit timestamp gaps. The effects of transactions in those gaps are eventually undone by the garbage collector.

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.

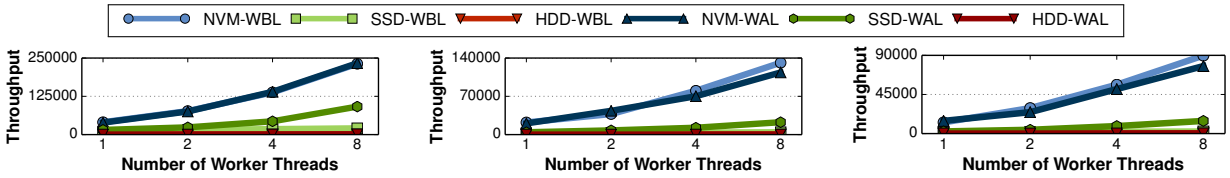
4.2.3 Recovery Protocol

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

The example in Figure 4.9 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

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 4.10: WBL Example – Contents of the WBL during recovery.



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

Figure 4.11: YCSB Throughput – The throughput of the DBMS for the YCSB benchmark with different logging protocols and durable storage devices.

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 4.8, 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 4.10 shows the contents of the log after a system failure. This example is based on the same the workload used in Figure 4.5. 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.

4.3 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 [3]. 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 [48, 50]. 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 hardware emulator. 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 [13]. 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 emulator's allocator and filesystem interfaces to store its logs, checkpoints, and table heap on NVM. 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 [25, 27]. 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.

4.3.1 Benchmarks

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

YCSB: This is a widely-used key-value store workload from Yahoo! [32]. 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).

4.3.2 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 4.11a. 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 4.11b and 4.11c. 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.

4.3.3 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 4.1 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 Section 4.3.3 present the recovery measurements for the YCSB benchmark. The recovery times of the WAL-based configurations grow linearly in proportion to the number of

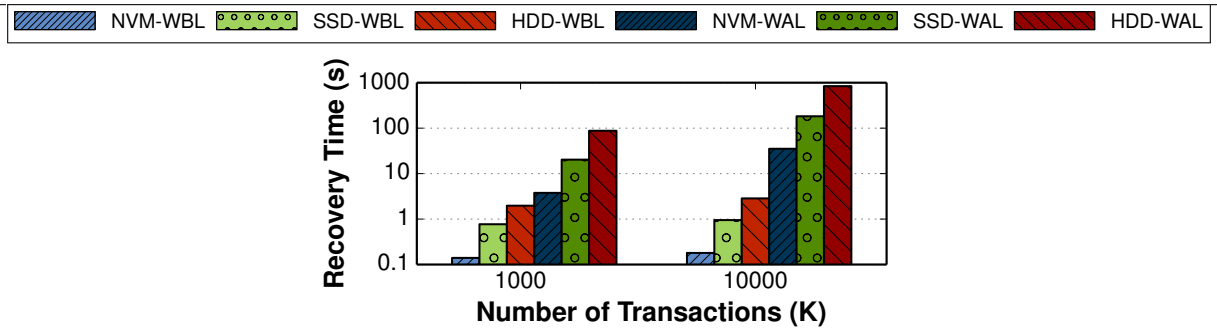


Figure 4.12: Recovery Time – The time taken by the DBMS to restore the database to a consistent state after a restart with different logging protocols.

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.

4.4 Summary

This chapter 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×.

The dissertation has so far focused on a two-tier storage hierarchy comprising of DRAM and NVM. We expect the cost of first-generation NVM devices to be significantly higher than that of SSD. Thus, we suspect that SSD will continue to remain in the storage hierarchy, at least in the near future, in order to shrink the operational cost of the DBMS. We next plan to examine the impact of introducing NVM in a multi-tier storage hierarchy comprising of DRAM, NVM, and SSD.

Chapter 5

Buffer Management (Proposed Work)

In this chapter, we present our proposed work on the design and evaluation of a cross-media buffer manager. Our goal is to develop new buffer management policies that maximize the utility of NVM in a multi-tier storage hierarchy. In addition to guiding *what* data should be moved across the storage hierarchy, these policies will dictate *where* and *when* the data should be moved. The introduction of NVM into the storage hierarchy necessitates these decisions.

5.1 Introduction

Storage technology has evolved into a diversified set of offerings that each occupy different points in a multi-dimensional design space defined by performance, cost, and capacity. These include: (1) high-density hard disk drives (HDDs), (2) solid-state drives (SSDs), and (3) emergent NVM technologies. Table 1.1 presents the characteristics of these different durable storage technologies and their location within the design space. Given this diversity, future DBMSs will likely need to manage data across several co-existing storage technologies in order to reduce capital and operational expenditures, while still satisfying the performance requirements of the applications.

Figure 5.1 presents the data flow paths in a multi-tier storage hierarchy comprising of DRAM, NVM, and SSD. Unlike HDD and SSD, the processor can *directly* access data stored on NVM without requiring to first copy over the data to DRAM. This new data flow path increases the degrees of freedom in the storage hierarchy. The buffer management policies of our cross-media buffer manager will make use of this data flow path to maximize DBMS performance. They will dictate *where* and *when* the data should be moved across the storage hierarchy.

Existing DBMSs make tradeoffs that are appropriate only for a specific storage technology. This limits them from leveraging the characteristics of different storage media. For example, disk-oriented DBMSs assume that the bulk of the data is stored on a HDD. Since retrieving the needed data from disk typically incurs a long delay, these DBMSs employ legacy techniques such as heavyweight concurrency-control schemes to overcome this limitation [75]. In contrast, the architecture of memory-oriented DBMSs assumes that all data fits in main memory, and it does away with the slower, disk-oriented components from the system [40, 53, 55, 84]. But, they still have to employ components that can recover the database after a restart because DRAM is volatile.

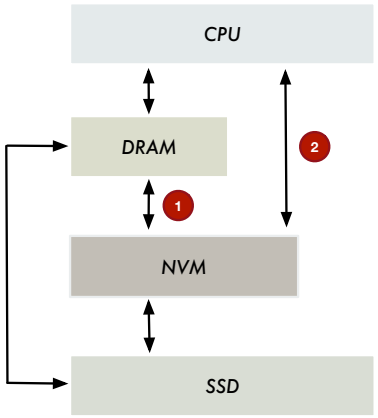


Figure 5.1: Data Flow Paths – Data flow paths in a multi-tier storage hierarchy comprising of DRAM, NVM, and SSD.

We propose to build a cross-media buffer manager that leverages the strengths and works around the limitations of each storage technology. Our objective is to minimize write-amplification and eliminate the performance overhead of legacy techniques.

To give a concrete example, consider the device characteristics of modern SSDs and HDDs. These devices require multi-block I/O operations for maximal storage efficiency. In case of SSDs, this is because of the underlying device characteristics that prevents efficient in-place updates, even at a block level. SSDs typically require a multi-block erasure in order to rewrite physical blocks, and the size of the erasure region has grown larger over time in order to increase storage density. Traditionally, HDDs support efficient sector overwrites. However, recently disks have started adopting a shingle write pattern to increase storage density, and require an entire region of disk sectors to be re-written in order to update a single sector.

In order to support fast updates, these devices often absorb writes in a sequential manner at the physical level irrespective of the write pattern at the virtual level. This involves repeated movement and re-writing of data in order to create empty regions for sequential writes. In case of SSDs, this *device-level write amplification* reduces the device lifetime. Our buffer manager will address these limitations by periodically coalescing the cold data on NVM and migrating them from NVM to SSD in full erasure block chunks. In this manner, it only stores the frequently updated data on NVM and shrinks device-level write amplification.

Many DBMSs coalesce updates to a uniform block size (e.g., updating a field in a tuple will write an entire block). As with device-level write amplification, *DBMS write amplification* hurts application performance, especially for NVM devices that can support efficient small writes. Our buffer manager will absorb fine-grained updates at the NVM layer, and later coalesce them into block updates for SSD. Such an efficient aggregation of repeated data updates lowers the DBMS write amplification.

5.2 Proposed Work

This work is in its beginning stages and we propose work along three thrusts:

- Evaluate the performance impact of NVM technologies with varying performance, cost, and capacity settings in a multi-tier storage hierarchy. We will focus on a set of metrics, including the operational throughput per dollar, and the number of read and write accesses to a particular tier in the storage hierarchy.
- Build a trace-driven simulator that addresses the limitations of the hardware emulator, and perform a cross-validation against the emulator in order to ensure the validity of the metrics collected from the simulator.
- Construct a class of buffer management policies that maximize the utility of NVM using the insights derived from experiments conducted on the simulator, and characterize the utility of NVM for different OLTP, OLAP, and HTAP benchmarks.

5.3 Evaluation Plan

Trace-driven Simulation: Currently, the NVM hardware emulator, presented in Section 2.1, only supports symmetric read and write latency settings. However, in certain NVM technologies, writes are significantly more costly than reads, and suffer from higher latency, lower per-chip bandwidth, higher energy costs, and endurance problems [19]. In order to better understand the impact of this read-write asymmetry on the DBMS performance, we will build a trace-driven simulator that models a multi-tier storage hierarchy. This will allow us to independently control the read and write latency settings of the NVM device in the storage hierarchy.

Tracing Framework: We will collect I/O traces by instrumenting a DBMS to record read, write, and flush operations. Additionally, we plan to collect the state of the DBMS buffer pool after it is warmed up. We will run these traces on the simulator and measure a set of metrics, including the operational throughput per dollar, and the number of read and write accesses to a particular tier in the storage hierarchy. Our goal is to use the insights derived from this evaluation to guide the design of a class of buffer management policies that maximize the utility of NVM in a multi-tier storage hierarchy. We expect these insights to be useful for NVM device manufacturers and database administrators (DBAs).

5.4 Related Work

We conclude with a discussion of related work and highlight the primary contributions of this proposed project.

Caching Systems: Prior research efforts have focused on using caching among different storage technologies. FlashStore is a key-value store that uses a SSD as a fast cache between DRAM and HDD and minimizes the number of SSD accesses [36]. RAMCloud is a sharded data storage system that uses disk as a back up for data stored on replicated DRAM [74]. It improves the DRAM utilization by employing a log-structured design on both DRAM and disk [72]. Nitro is an SSD caching system that relies on data compression and deduplication to maximize storage utilization [62]. RIPQ is a novel caching layer that shrinks write amplification by using the local SSD as a read-only cache for remote storage [86].

NVM/SSD optimized block storage & file systems: Several recent projects have focused on specialized storage solutions for emerging NVM technologies. 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 [31]. PMFS is another filesystem from Intel Labs that is designed for byte-addressable NVM [43]. It relies on a WAL for meta-data and uses shadow paging for data. It assumes a simpler hardware barrier primitive than epochs. Further, it optimizes memory-mapped I/O by mapping the persistent memory to the application's address space. EXT4 DAX extends the EXT4 file system in order to support direct mapping of NVM by bypassing the buffer cache [34]. Aerie provides direct access for file data I/O using user-level leases for NVM updates [89]. NOVA is a novel per-inode log-structured file system that provide synchronous file system semantics on NVM, but requires system calls for every operation [92]. F2FS is a SSD-optimized log-structured file system that sorts data to reduce file system write amplification [60]. Strata is a cross-media file system that supports performance-isolated access to NVM using a per-application log by efficiently operating on SSDs and HDDs [57].

Unlike these research efforts, we focus on the construction of a class of buffer management policies that maximize the utility of NVM technologies with varying performance, cost, and capacity settings. Furthermore, we tackle the buffer management problem within the context of a DBMS. Operating inside a DBMS allows us to support, and requires us to handle, a broader class of application access patterns.

Chapter 6

Related Work

In this chapter, we provide a discussion of related work. We begin with a discussion of work related to the general themes in this dissertation, then examine specific areas in depth.

The design of a DBMS's architecture is predicated on the target storage hierarchy. There are essentially two types of DBMS architectures: disk-oriented [16, 84] and memory-oriented systems [6, 40, 47, 53, 55]. The former is exemplified by the first DBMSs, such as IBM's System R [16], where the system is predicated on the management of blocks of tuples on disk using an in-memory cache; the latter by IBM's IMS/VS Fast Path [47], where the system performs updates on in-memory data and relies on the disk to ensure durability. The need to ensure that all changes are durable has dominated the design of systems with both types of architectures. This has involved optimizing the layout of data for each storage layer depending on how fast it can perform random accesses [46]. Further, updates performed on tuples stored in memory need to be propagated to an on-disk representation for durability. Previous studies have shown that the overhead of managing this data movement for OLTP workloads is considerable [51]. The advent of NVM offers an intriguing blend of the two storage mediums. This dissertation explores the changes required in DBMSs to leverage the properties of NVM. It has benefited from prior work and in some cases built upon it.

Persistent Programming: There were several efforts in the late 1980s and early 1990s on developing support for persistent memory and language features for database applications [11, 17, 82]. The basic premise was that the DBMS's query language was integrated with the host language and the application could make changes to records in the database as if they were stored locally. This idea was also explored in persistent object stores for object-oriented DBMSs and filesystems [23, 68]. The language integration efforts provide strongly-typed programming primitives, but allowed the possibility of programming errors that corrupt the database. Further, it was non-trivial to optimize queries with these languages. There are multiple proposals for application APIs for programming with persistent memory. Mnemosyne and NV-heaps use software transactional memory to support transactional updates to data stored on NVM [30, 90]. While the former supports word-based transactions, the latter supports node-based transactions. The primitives provided by these systems allow programmers to use NVM in their applications but do not provide the transactional semantics required by a DBMS.

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 [37]. 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 [77]. Their work is based on Shore-MT [52], 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 [91]. 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 [44]. Both of these projects rely on software-based NVM simulation.

MARS [29] is an in-place updates engine optimized for NVM that relies on a hardware-assisted primitive that allows multiple writes to arbitrary locations to happen atomically. MARS does away with undo log records by keeping the changes isolated using this primitive. Similarly, it relies on this primitive to apply the redo log records at the time of commit. In comparison, our WBL approach is based on non-volatile pointers, a software-based primitive, and uses existing (or upcoming) synchronization instructions. It removes the need to maintain redo information in the WAL, but still needs to maintain undo log records until the transaction commits.

SOFORT [73] 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 [4], 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 [25]. FOEDUS is a scalable OLTP engine designed for a hybrid storage hierarchy [56]. 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 [94]. 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 [70]. 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. Baker et al. evaluate the utility of battery-backed DRAM as a client-side file cache in a distributed filesystem to reduce write traffic to file servers [18]. Rio is a persistent file cache that relies on uninterruptible power supply to provide a safe, in-memory buffer for filesystem data [63]. It reduces transaction latency by absorbing synchronous writes to disk without losing data during system crashes. 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 [31]. PMFS is another filesystem from Intel Labs that is designed for byte-addressable NVM [43]. It relies on a

WAL for meta-data and uses shadow paging for data. It assumes a simpler hardware barrier primitive than epochs. Further, it optimizes memory-mapped I/O by mapping the persistent memory to the application's address space.

Replication: Mojim provides reliable and highly-available NVM by using a two-tier architecture that can support additional level of redundancy and efficiently replicates the data stored on NVM [93]. It allows programmers to use fault-tolerant memory storage in their applications but does not provide the transactional semantics required by a DBMS. RAMCloud is a DRAM-based storage system that can be used as a low-latency key-value store [72]. While both Mojim and RAMCloud provide reliable memory-based storage systems, the former exports a memory-like interface to the applications and the latter provides a key-value interface. Mojim relies on fail-over to recover from failures instead of adopting RAMCloud's approach that relies on sharded data storage to achieve fast recovery.

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 [48, 50]. 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.

Simulator-Driven Studies: SafeRAM is one of the first projects that explored the use of NVM in software applications [33]. Using simulations, they evaluated the improvement in throughput when disk is replaced by battery-backed DRAM. More recently, Kannan et al. explore the performance implications of using NVM in end client devices [54]. Qureshi et al. use an architecture level model of PCM to examine the trade-offs for a main memory system consisting of PCM storage coupled with a small DRAM buffer [80]. These efforts use simulators for running traces that are unable to accurately model real hardware. We use a hardware emulator in our experiments, as it provides a more accurate environment with actual hardware and real-time benchmarking.

Chapter 7

Timeline

We conclude with a proposed timeline.

- 2017 November: Submit thesis proposal.
- 2017 December: Present thesis proposal.
- 2017 October: Build the trace-driven simulator for the buffer management project.
- 2017 November: Construct the tracing framework by instrumenting PostgreSQL.
- 2017 December: Collect traces from a set of OLTP, OLAP, and HTAP benchmarks.
- 2018 January: Develop a class of buffer management policies based on the insights derived from the experiments conducted on the simulator.
- 2018 February: Document the insights and policies in a technical report.
- 2018 January~March: Interviews.
- 2017 November~2018 March: Write thesis.
- 2018 April~2018 May: Defend and graduate.

Bibliography

[1] H-Store. http://hstore.cs.brown.edu.

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

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

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

[5] VoltDB. http://voltdb.com.

[6] Oracle TimesTen Products and Technologies. Technical report, February 2007.

[7] Intel's 3d memory is 1,000 times faster than modern storage. https://www.engadget.com/2015/07/28/intel-3d-memory-1000-times-faster/, July 2015.

[8] Hpe unveils computer built for the era of big data. https://news.hpe.com/a-new-computer-built-for-the-big-data-era/, May 2017.

[9] A new breakthrough in persistent memory gets its first public demo. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/, May 2017.

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

[11] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. Database Syst.*, 1985.

[12] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, 2017.

[13] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

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

[15] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. In *VLDB*, 2017.

[16] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM TODS*, 1(2):97–137, June 1976.

[17] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Readings in Object-oriented Database Systems*, 1990.

[18] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *ASPLOS*, pages 10–22, 1992.

[19] N. Ben-David and *et al.* Parallel algorithms for asymmetric read-write costs. In *SPAA*, 2016.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[21] T. Bingmann. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/.

[22] 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.

[23] M. J. Carey, D. J. DeWitt, J. F. Richardson, and E. J. Shekita. Object-oriented concepts, databases, and applications. chapter Storage Management for Objects in EXODUS. 1989.

[24] J. Chang, P. Ranganathan, T. Mudge, D. Roberts, M. A. Shah, and K. T. Lim. A limits study of benefits from nanostore-based future data-centric system architectures. In *CF*, CF '12, pages 33–42, 2012.

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

[26] F. Chen, M. Mesnier, and S. Hahn. A protected block device for persistent memory. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2014.

[27] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[28] H. Chu. MDB: A Memory-Mapped Database and Backend for OpenLDAP. Technical report, OpenLDAP, 2011.

[29] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, 2013.

[30] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, pages 105–118. ACM, 2011.

[31] 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.

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

[33] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. VLDB, pages 327–335. Morgan Kaufmann Publishers Inc., 1989.

[34] J. Corbet. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, 2014.

[35] J. Dean and S. Ghemawat. LevelDB. http://leveldb.googlecode.com.

[36] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *VLDB*, pages 1414–1425, 2010.

[37] 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.

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

[39] 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.

[40] 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*, pages 1243–1254, 2013.

[41] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[42] A. Driskill-Smith. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[43] 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.

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

[45] M. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997.

[46] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE TKDE*, pages 509–516, Dec. 1992.

[47] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS Fast Path. Technical report, Tandem, 1985.

[48] 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.

[49] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system R database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

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

[51] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[52] 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.

[53] 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. In *VLDB*, pages 1496–1499, 2008.

[54] S. Kannan, A. Gavrilovska, and K. Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *HPCA*, 2014.

[55] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, ICDE, pages 195–206, 2011.

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

[57] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[58] D. Laney. 3-D data management: Controlling data volume, velocity and variety. Feb. 2001.

[59] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB*, 2011.

[60] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *FAST*, pages 273–286, 2015.

[61] LevelDB. Implementation details of LevelDB. https://leveldb.googlecode.com/svn/trunk/doc/impl.html.

[62] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *ATC*, 2014.

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

[64] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[65] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM J. Res. Dev.*, 46(2-3).

[66] 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 TODS*, 17(1):94–162, 1992.

[67] I. Moraru, D. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, 2013.

[68] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Softw. Eng.*, 1992.

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

[70] 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.

[71] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[72] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[73] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. DaMoN, pages 8:1–8:7, 2014.

[74] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 2015.

[75] A. Pavlo. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. PhD thesis, Brown University, 2013.

[76] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

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

[78] T. Perez and C. Rose. Non-volatile memory: Emerging technologies and their impact on memory systems. *PURCS Technical Report*, 2010.

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

[80] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[81] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[82] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Trans. Program. Lang. Syst.*, 1993.

[83] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

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

[85] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, (7191):80–83, 2008.

[86] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *FAST*, pages 373–386, 2015.

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

[88] V. Viswanathan, K. Kumar, and T. Willhalm. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[89] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[90] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 91–104. ACM, 2011.

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

[92] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

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

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