

# **Design and Implementation of a Relational Database using RAM and Non-Volatile Memory**



**Anushka Srivastava & Jaswindar Singh Gill**

**Advisor: Dr. Gautum Barua**

Department of Computer Science and Engineering  
Indian Institute of Information Technology Guwahati

BTP report submitted in partial fulfilment of the requirements for the  
degree of  
*Bachelor of Technology*

## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other course, degree or qualification in this or any other university. This report is our own work and contains nothing that is the outcome of work done in collaboration with others except as specified in the text and Acknowledgements.

**Anushka Srivastava,**

Roll: 2201030

**Jaswindar Singh Gill,**

Roll: 2201099

Department of Computer Science and Engineering,  
Indian Institute of Information Technology Guwahati.

## **Acknowledgements**

We acknowledge Dr. Gautum Barua's invaluable guidance and support throughout this project. Our gratitude for their mentorship extends to all the Department of Computer Science and Engineering faculty at the Indian Institute of Information Technology Guwahati. I also thank my friends for my cherished time during this degree. My appreciation goes out to my parents and family members for their unwavering encouragement and support throughout all my studies.

## **Abstract**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis aliquet, tellus id tincidunt dictum, tortor mauris elementum ex, hendrerit rutrum lorem eros tincidunt urna. Nam nec consectetur dolor, non volutpat purus. Vivamus gravida arcu luctus, vestibulum ipsum nec, ultrices erat. Vestibulum eu fringilla nisl, sed finibus libero. Morbi interdum quis ex dapibus scelerisque. Quisque ac sem congue, porta risus dignissim, dictum urna. Nulla nec rutrum tellus, eu sollicitudin ante. Nam pharetra vehicula maximus. Etiam sed orci posuere ligula sodales ultrices eu nec nibh.

# Table of Contents

<b>List of Figures</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Motivation . . . . .	1
1.1.1 The Memory-Storage Dichotomy . . . . .	1
1.1.2 The Promise of NVRAM . . . . .	2
1.2 Emergence of NVRAM Technologies . . . . .	2
1.3 Project Goals and Scope . . . . .	3
1.3.1 Primary Objectives . . . . .	3
1.4 Report Structure . . . . .	4
<b>2 Background and Theory</b>	<b>5</b>
2.1 NVRAM Technologies . . . . .	5
2.1.1 Key Characteristics . . . . .	5
2.1.2 Types of NVRAM Technologies . . . . .	6
2.2 CPU Support for NVRAM . . . . .	7
2.2.1 Persistence Instructions . . . . .	8
2.2.2 Memory Ordering and Persistence Domain . . . . .	8
2.3 Cache Considerations for NVRAM . . . . .	9
2.3.1 Cache Flushing Overhead . . . . .	9
2.3.2 Atomic Operations and Durability . . . . .	9
2.4 Software Approaches to NVRAM . . . . .	10
2.4.1 Software Emulation of NVRAM . . . . .	10
2.4.2 Reserving RAM at Boot Time . . . . .	10
2.4.3 NVRAM Programming Libraries . . . . .	10
2.5 Implications for Database Design . . . . .	10
2.5.1 Rethinking the Storage Hierarchy . . . . .	11
2.5.2 NVRAM-Optimized Data Structures . . . . .	11

---

## Table of Contents

2.5.3	Concurrency Control for NVRAM . . . . .	11
<b>3</b>	<b>System Architecture</b>	<b>13</b>
3.1	Overview . . . . .	13
3.2	Memory Management . . . . .	14
3.2.1	NVRAM Simulation . . . . .	14
3.2.2	Free Space Management . . . . .	14
3.3	B+ Tree-Based Indexing . . . . .	15
3.3.1	B+ Tree Structure . . . . .	15
3.3.2	Data Storage . . . . .	15
3.4	Write-Ahead Logging . . . . .	16
3.4.1	WAL Structure . . . . .	16
3.4.2	Transaction Logging . . . . .	16
3.5	Concurrency Control . . . . .	16
3.5.1	Lock Types and Granularity . . . . .	17
3.5.2	Transaction Management . . . . .	17
3.6	Database Interface . . . . .	17
3.7	Component Interactions . . . . .	18
3.8	Summary . . . . .	18
<b>4</b>	<b>Implementation Details</b>	<b>19</b>
4.1	Free Space Management Implementation . . . . .	19
4.1.1	Data Structures . . . . .	19
4.1.2	Memory Mapping . . . . .	20
4.1.3	Allocation and Deallocation Algorithms . . . . .	20
4.2	B+ Tree Implementation . . . . .	21
4.2.1	Node Structure . . . . .	21
4.2.2	Key Operations . . . . .	22
4.2.3	NVRAM Integration . . . . .	22
4.3	Write-Ahead Logging Implementation . . . . .	23
4.3.1	WAL Data Structures . . . . .	23
4.3.2	Thread Safety . . . . .	24
4.3.3	Commit Processing . . . . .	24
4.4	Lock Manager Implementation . . . . .	24
4.4.1	Lock Manager Data Structures . . . . .	24
4.4.2	Lock Acquisition . . . . .	25
4.4.3	Transaction Management . . . . .	26

---

## Table of Contents

4.5	Database Interface Implementation . . . . .	26
4.5.1	Core Operations . . . . .	26
4.5.2	Error Handling . . . . .	27
4.6	Implementation Challenges . . . . .	27
4.6.1	Ensuring Atomicity . . . . .	27
4.6.2	Balancing Performance and Durability . . . . .	27
4.6.3	Memory Management . . . . .	28
4.7	Summary . . . . .	28
<b>5</b>	<b>Evaluation</b>	<b>29</b>
5.1	Evaluation Methodology . . . . .	29
5.1.1	Experimental Setup . . . . .	29
5.1.2	Benchmark Workloads . . . . .	30
5.1.3	Metrics . . . . .	30
5.2	Single-Threaded Performance . . . . .	30
5.2.1	Throughput Analysis . . . . .	31
5.2.2	Latency Analysis . . . . .	31
5.3	Multi-Threaded Performance . . . . .	31
5.3.1	Throughput Scaling . . . . .	32
5.3.2	Concurrency Control Impact . . . . .	32
5.4	Recovery Performance . . . . .	32
5.4.1	Recovery Time . . . . .	32
5.4.2	Recovery Correctness . . . . .	33
5.5	Comparison with Traditional Approaches . . . . .	33
5.5.1	Performance Comparison . . . . .	33
5.5.2	Architectural Advantages . . . . .	34
5.6	Limitations and Future Improvements . . . . .	34
5.6.1	Scalability Bottlenecks . . . . .	34
5.6.2	NVRAM Simulation Limitations . . . . .	35
5.6.3	Feature Limitations . . . . .	35
5.7	Summary . . . . .	35
<b>6</b>	<b>Conclusion and Future Work</b>	<b>37</b>
6.1	Summary of Contributions . . . . .	37
6.1.1	Architectural Design . . . . .	37
6.1.2	Implementation Insights . . . . .	38
6.1.3	Performance Evaluation . . . . .	38

---

## Table of Contents

6.2	Lessons Learned . . . . .	39
6.2.1	Persistence vs. Performance Trade-offs . . . . .	39
6.2.2	Concurrency Challenges . . . . .	39
6.2.3	NVRAM Simulation Insights . . . . .	39
6.3	Future Work . . . . .	40
6.3.1	Advanced Concurrency Control . . . . .	40
6.3.2	Enhanced Recovery Mechanisms . . . . .	40
6.3.3	NVRAM-Specific Optimizations . . . . .	41
6.3.4	Feature Extensions . . . . .	41
6.3.5	Distributed NVRAM Database . . . . .	41
6.4	Concluding Remarks . . . . .	42
	<b>References</b>	<b>43</b>

# List of Figures

2.1	Relationship between memory ordering, persistence domain, and power failure domain . . . . .	8
3.1	High-level architecture of the NVRAM database system . . . . .	13
3.2	NVRAM simulation using memory-mapped files . . . . .	14
5.1	Single-threaded throughput across YCSB workloads . . . . .	31
5.2	Throughput scaling with thread count across workloads . . . . .	32
5.3	Recovery time vs. number of WAL entries . . . . .	33

# Chapter 1

## Introduction

### 1.1 Background and Motivation

In recent years, data management systems have become increasingly critical components of modern computing infrastructure. Traditional database systems have long been designed around a clear hierarchy of storage: fast but volatile main memory (DRAM) for runtime operations, and slow but persistent storage (HDDs, SSDs) for data durability. This dichotomy has shaped database architecture for decades, leading to complex buffer management systems, write-ahead logging mechanisms, and checkpoint procedures to ensure both performance and durability.

The emergence of Non-Volatile Random Access Memory (NVRAM) technologies represents a fundamental shift in this paradigm. NVRAM combines the byte-addressability and performance characteristics of DRAM with the persistence of storage devices. This technological advancement blurs the traditional boundary between memory and storage, offering new opportunities to reimagine database system architectures.

#### 1.1.1 The Memory-Storage Dichotomy

Traditional database systems operate with a clear separation between volatile memory and persistent storage. Volatile memory (DRAM) offers nanosecond access times and byte-addressability but loses all data when power is removed. In contrast, persistent storage devices like SSDs and HDDs preserve data across power cycles but with microsecond to millisecond access times and block-based access patterns. This fundamental difference in characteristics has forced database designers to create complex mechanisms to bridge these two worlds.

To manage this dichotomy, databases employ buffer pools to cache frequently accessed data in memory, write-ahead logging to ensure durability before acknowledging transactions, periodic checkpointing to flush dirty pages to persistent storage, and recovery procedures to rebuild consistent state after crashes. These mechanisms, while necessary, add significant complexity and overhead to database operations, affecting both performance and code maintainability.

### 1.1.2 The Promise of NVRAM

Non-Volatile Random Access Memory (NVRAM) technologies promise to bridge this gap by providing persistence, byte-addressability, low latency access, and high endurance compared to flash-based technologies. With NVRAM, data survives power loss without explicit writes to secondary storage. Its byte-addressable nature allows direct access to individual bytes rather than requiring block-based transfers. Access times are much closer to DRAM than to SSDs, and many NVRAM technologies offer greater write durability than traditional flash storage.

With these characteristics, NVRAM enables new database architectures that can potentially eliminate or significantly simplify buffer management, logging, and recovery mechanisms while maintaining both performance and durability guarantees. Database systems can be redesigned to work directly with persistent memory, reducing complexity and improving performance by eliminating many of the traditional I/O bottlenecks.

## 1.2 Emergence of NVRAM Technologies

The journey toward commercially viable NVRAM has been marked by both significant technological achievements and commercial challenges. Over the past decade, several NVRAM technologies have emerged with varying characteristics and market success.

NVDIMMs (Non-Volatile Dual In-line Memory Modules) represent one approach to persistent memory. These modules combine DRAM with flash storage and a power source (battery or capacitor) to persist memory contents during power loss. When power is removed, the backup power source provides enough energy to flush DRAM contents to the non-volatile component, ensuring data persistence.

Perhaps the most notable commercial NVRAM product was 3D XPoint, developed by Intel and Micron and brought to market as Intel Optane. Released in

2017, Optane represented the first widely available byte-addressable NVRAM technology that didn't rely on backup power solutions. It offered significantly better performance than flash storage while providing true persistence at the memory level. Despite its technological promise, Intel announced the discontinuation of the Optane product line in 2022 due to market challenges and strategic realignment.

Beyond these technologies, several other approaches remain in various stages of development, including Magnetoresistive RAM (MRAM), Resistive RAM (ReRAM), Phase Change Memory (PCM), and Ferroelectric RAM (FeRAM). While true NVRAM technologies have faced commercial challenges, their promise continues to influence database system design, and memory-mapped files provide a way to simulate many of their characteristics for research and development purposes.

## **1.3 Project Goals and Scope**

This report presents the design, implementation, and evaluation of an NVRAM-optimized database system that leverages the unique characteristics of persistent memory. The project aims to explore how database architectures can evolve to take advantage of NVRAM, even when simulated through memory-mapped files.

### **1.3.1 Primary Objectives**

The key goals of this project include designing a database system architecture optimized for NVRAM characteristics and implementing core database components that take advantage of byte-addressable persistent memory. The system includes NVRAM-aware free space management to efficiently allocate and reclaim memory in a persistent context, persistence-friendly B+ tree index structures designed to minimize cache flushes while ensuring consistency, write-ahead logging optimized for byte-addressable persistent memory, and concurrency control mechanisms for multi-threaded access.

Given the limited availability of true NVRAM hardware, the project uses memory-mapped files to simulate NVRAM behavior, enabling the exploration of persistent memory programming models on conventional hardware. The system's performance characteristics are evaluated using established benchmarks to assess the effectiveness of the design and implementation.

## 1.4 Report Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Background and Theory** provides a deeper exploration of NVRAM technologies, their characteristics, and their implications for database design.
- **Chapter 3: System Architecture** presents the overall design of the database system, including its major components and their interactions.
- **Chapter 4: Implementation Details** describes the implementation of key components, focusing on NVRAM-specific optimizations.
- **Chapter 5: Evaluation** presents the methodology and results of performance evaluations using the YCSB benchmark.
- **Chapter 6: Conclusion and Future Work** summarizes the findings and contributions of the project and suggests directions for future research.

# **Chapter 2**

## **Background and Theory**

This chapter provides a comprehensive overview of Non-Volatile Random Access Memory (NVRAM) technologies, their characteristics, and implications for database system design. We explore the various types of NVRAM, CPU support for persistent memory operations, and the challenges and solutions related to cache management and persistence.

### **2.1 NVRAM Technologies**

NVRAM represents a class of memory technologies that combine the performance characteristics of DRAM with the persistence of traditional storage. Unlike volatile memory, NVRAM retains data even when power is removed, eliminating the need for explicit I/O operations to ensure durability.

#### **2.1.1 Key Characteristics**

The defining characteristic of NVRAM is its persistence - data survives power loss without explicit writes to secondary storage. Unlike traditional persistent storage that requires block-based access, NVRAM offers byte-addressability, allowing direct memory-like access to individual bytes. This fundamentally changes how software interacts with persistent data, eliminating the need for serialization and deserialization.

Access latency for NVRAM technologies is significantly faster than SSDs, though typically slower than DRAM. This places NVRAM in a unique position in the memory hierarchy, potentially serving as either extended memory or fast storage depending on the system design. NVRAM also supports in-place updates, allowing

## **2.1 NVRAM Technologies**

---

data to be modified directly without the read-modify-write cycles required by some storage technologies like flash.

Most NVRAM technologies offer higher write endurance than flash-based technologies, though often with some lifetime limitations compared to DRAM. This characteristic influences how write operations should be distributed across the memory space to maximize device lifetime. Together, these characteristics fundamentally change how database systems can approach persistence, with significant implications for system architecture, algorithm design, and performance optimization.

### **2.1.2 Types of NVRAM Technologies**

Several NVRAM technologies have emerged, each with distinct characteristics and implementation approaches. The most significant commercial development was 3D XPoint, marketed as Intel Optane, which represented the first widely adopted byte-addressable NVRAM technology.

#### **3D XPoint / Intel Optane**

Developed jointly by Intel and Micron and commercially released in 2017, 3D XPoint uses a resistive memory approach with a crosspoint structure of selector and memory cell components. The technology was positioned between DRAM and SSDs in the memory hierarchy, with approximately 10 times the density of DRAM and 1000 times the endurance of NAND flash.

Intel offered Optane in two form factors: storage devices (Optane SSDs) that used the standard block interface, and memory modules (Optane DC Persistent Memory) that connected directly to the memory bus and could be used as either volatile memory extensions or persistent memory. Despite its technological innovation, Intel announced the discontinuation of the Optane product line in 2022 due to market challenges and strategic realignment.

#### **NVDIMM (Non-Volatile Dual In-Line Memory Module)**

NVDIMMs represent an alternative approach to providing persistent memory. The NVDIMM-N type combines DRAM with flash memory and a backup power source. During normal operation, it functions identically to DRAM, providing volatile memory performance. However, upon power loss, onboard batteries or capacitors provide enough power to flush DRAM contents to flash. When power is restored,

data is automatically loaded back into DRAM, creating the appearance of persistence from the application's perspective.

NVDIMM-F modules take a different approach, using flash-based memory that connects directly to the memory bus. These provide persistence but with flash-like performance characteristics, resulting in higher latency than DRAM but with a memory-like interface that simplifies integration.

A more advanced variant, NVDIMM-P, combines both DRAM and persistent memory on the same module. This provides both high-performance volatile memory and persistent storage, offering more flexible configuration options than other NVDIMM types and potentially better performance characteristics for mixed workloads.

### Emerging NVRAM Technologies

Beyond commercially available options, several promising NVRAM technologies remain in development. Magnetoresistive RAM (MRAM) stores data using magnetic states, offering high endurance and potentially DRAM-like performance, though with limited density compared to other technologies. Resistive RAM (ReRAM) uses resistance state changes in a dielectric material, potentially offering high density and low power consumption, though challenges remain with cell-to-cell variability and endurance.

Phase-Change Memory (PCM) uses chalcogenide glass that changes between crystalline and amorphous states to represent data. It offers a good compromise between performance and density and has been commercialized in some specific applications. Ferroelectric RAM (FeRAM) uses ferroelectric materials to store data, providing fast writes and high endurance, though it faces challenges with scaling to high densities.

Each of these technologies represents a different approach to achieving the goal of non-volatile memory with performance characteristics suitable for direct integration into the memory hierarchy.

## 2.2 CPU Support for NVRAM

Effectively utilizing NVRAM requires hardware support at the CPU level to manage persistence, atomicity, and ordering guarantees. Modern CPUs provide several instructions and features specifically designed for persistent memory operations.

### **2.2.1 Persistence Instructions**

Intel x86 architecture provides several key instructions for managing persistent memory. The CLWB (Cache Line Write Back) instruction writes a modified cache line to memory without evicting it from the cache hierarchy. This allows continued use of the data in cache while ensuring durability, making it more efficient than older flush instructions for persistent memory workloads.

The SFENCE (Store Fence) instruction ensures that all store operations before the fence are visible to other cores before any stores after the fence. This is critical for enforcing ordering guarantees with persistent memory and is used in conjunction with flush instructions to create persistence barriers.

For specialized cases, MOVNTI (Move Non-Temporal) provides a store operation that bypasses the cache, writing data directly to memory without allocating cache lines. This is useful for streaming writes where data is not immediately reused and can be used for atomic updates, meaning they either complete entirely or not at all, even in the event of a power failure, of critical persistent data structures.

### **2.2.2 Memory Ordering and Persistence Domain**

When working with persistent memory, understanding the relationship between memory ordering and persistence is crucial. Memory ordering refers to the sequence in which memory operations appear to execute from the perspective of other processors, while the persistence domain is the boundary within which data is guaranteed to be persisted to NVRAM. The power failure domain represents the region where a single power failure affects all components simultaneously.

Fig. 2.1 Relationship between memory ordering, persistence domain, and power failure domain

To ensure proper persistence with NVRAM, applications must follow a three-step process: first, store data to memory; second, explicitly flush cache lines to the persistence domain using instructions like CLWB; and third, issue a memory fence (SFENCE) to ensure ordering. This sequence guarantees that data has reached the persistence domain and will survive a power failure.

### 2.3 Cache Considerations for NVRAM

Cache management presents unique challenges when working with persistent memory due to the need to explicitly manage the movement of data between volatile caches and persistent memory.

#### 2.3.1 Cache Flushing Overhead

Ensuring data persistence requires explicit cache flushing, which introduces performance overhead. Each cache line flush instruction (CLWB) takes significant CPU cycles to execute. Naively flushing the cache after every write creates substantial performance degradation, effectively negating much of the performance advantage of byte-addressable persistent memory. Furthermore, fence instructions (SFENCE) required for ordering further impact performance by preventing instruction reordering and parallelism.

Careful design to minimize the number of flushes is essential for NVRAM performance. Effective strategies include batching multiple writes before flushing, using techniques like shadow paging to amortize flush costs across multiple updates, and leveraging non-temporal writes (MOVNTI) to bypass the cache when data is *unlikely* to be accessed again soon. These approaches must balance the need for persistence guarantees with performance considerations.

#### 2.3.2 Atomic Operations and Durability

For critical updates, balancing atomicity and performance is essential. This often involves using CPU atomic operations (e.g., compare-and-swap) for volatile updates to ensure thread safety, leveraging 8-byte atomic writes for durable pointer updates to ensure consistency across power failures, and implementing techniques like append-only logs for larger atomic updates that exceed the CPU's atomicity guarantees.

The Write-Ahead Logging (WAL) system in our implementation leverages these properties by using 8-byte atomic updates to advance commit pointers, ensuring that log entries are properly flushed before updating these pointers, and minimizing cache flushes through careful batching of operations. This approach provides both performance and durability guarantees for transactional operations.

## 2.4 Software Approaches to NVRAM

While true NVRAM hardware may not be universally available, several software approaches allow for development and testing of NVRAM-aware applications.

### 2.4.1 Software Emulation of NVRAM

Memory-mapped files provide a way to simulate many NVRAM characteristics. The `mmap()` interface maps files into memory address space, providing byte-addressable access to file-backed memory. Changes to mapped memory are eventually persisted to the underlying file, allowing development and testing of NVRAM-aware code on conventional hardware.

### 2.4.2 Reserving RAM at Boot Time

For more realistic NVRAM simulation, portions of RAM can be reserved at boot time using the Linux kernel's `memmap` boot parameter. For example, the parameter `memmap=4G!8G` reserves 4GB of RAM starting at the 8GB offset. This creates a device file (e.g., `/dev/pmem0`) representing this memory region that can be accessed through memory mapping with `mmap()`. This approach allows more accurate modeling of NVRAM characteristics like capacity constraints and can provide a more realistic testing environment.

### 2.4.3 NVRAM Programming Libraries

Several libraries have been developed to simplify persistent memory programming. The PMDK (Persistent Memory Development Kit), developed by Intel, provides a comprehensive framework for persistent memory programming, including transactional object stores, persistent pools, and memory allocation functions. It abstracts hardware-specific details for portability across different persistent memory technologies.

## 2.5 Implications for Database Design

The unique characteristics of NVRAM enable fundamentally different approaches to database design compared to traditional disk-based systems.

### 2.5.1 Rethinking the Storage Hierarchy

NVRAM transforms the traditional storage hierarchy by bridging the gap between volatile memory and persistent storage. In the traditional model, databases must carefully manage the movement of data between fast but volatile DRAM and slower but persistent storage devices like SSDs and HDDs. The NVRAM model introduces persistent memory with direct byte-addressability, collapsing these layers into a single, addressable persistence tier.

This shift enables databases to eliminate or reduce buffer management layers that traditionally cache disk blocks in memory. It simplifies logging and recovery mechanisms by allowing direct persistent updates with minimal overhead. Perhaps most significantly, it enables durability without the explicit I/O operations that dominate performance in traditional database systems.

### 2.5.2 NVRAM-Optimized Data Structures

The persistence and performance characteristics of NVRAM call for specialized data structures. Persistent B+ Trees can be optimized for cache-line-sized nodes and designed to minimize flush operations while maintaining consistency guarantees. Write-Ahead Logging systems can be redesigned to take advantage of byte-addressability and atomic updates, reducing overhead compared to traditional block-oriented logging. Free space management must be tailored for NVRAM's direct access patterns and alignment requirements, ensuring efficient use of persistent memory while avoiding fragmentation.

### 2.5.3 Concurrency Control for NVRAM

Concurrent access to persistent data introduces unique challenges that combine the difficulties of multithreaded programming with the constraints of persistence. Systems must ensure both thread safety during normal operation and crash consistency in the event of power failure or system crashes. Atomic operations must consider both in-memory consistency from the perspective of concurrent threads and persistence to ensure durable, consistent states. Lock-based approaches must prevent memory corruption across crashes, which may require persistent metadata to track lock state.

In our implementation, the lock manager provides thread safety by controlling access to shared resources, while the WAL system ensures durability by logging

## **2.5 Implications for Database Design**

---

changes before they are applied to the main data structures. Together, these components establish a foundation for ACID transactions in an NVRAM environment, allowing multiple threads to concurrently access and modify persistent data with appropriate isolation and consistency guarantees.

# Chapter 3

## System Architecture

This chapter presents the overall architecture of our NVRAM-optimized database system. We describe the key components, their interactions, and the design decisions that leverage the unique characteristics of persistent memory.

### 3.1 Overview

The database system is designed with a core principle: maintain index structures in RAM for performance while storing data persistently in NVRAM. This hybrid approach leverages the speed of volatile memory for frequently accessed metadata while ensuring data durability through NVRAM storage. Figure 3.1 illustrates the high-level architecture of the system.

Fig. 3.1 High-level architecture of the NVRAM database system

The system consists of five main components:

1. **Memory Management Subsystem:** Handles allocation and deallocation of NVRAM space
2. **Indexing Structure:** Implements a B+ tree for efficient data retrieval
3. **Write-Ahead Logging (WAL):** Ensures transaction durability and atomicity
4. **Lock Manager:** Provides concurrency control for multi-threaded access
5. **Database Interface:** Exposes operations to applications

## 3.2 Memory Management

The memory management subsystem is responsible for allocating and freeing memory within the NVRAM space, which is simulated using memory-mapped files. This component is critical for both performance and durability, as it directly interacts with persistent memory.

### 3.2.1 NVRAM Simulation

Since true NVRAM hardware may not be widely available, our system uses memory-mapped files to simulate persistent memory:

Fig. 3.2 NVRAM simulation using memory-mapped files

The system uses the `mmap()` system call to map a file into the process's address space, creating a region of memory that behaves like NVRAM. We define a fixed-size file (currently 2GB) that serves as our persistent memory pool. The mapping is performed during initialization and remains active throughout the database's lifecycle.

### 3.2.2 Free Space Management

The free space manager maintains a list of available memory blocks in NVRAM. Its primary functions include:

1. Tracking available memory blocks within the NVRAM file
2. Allocating memory using a first-fit algorithm
3. Reclaiming freed memory and merging adjacent free blocks

Memory allocation follows a first-fit strategy, where the first free block large enough to accommodate the requested size is selected. When memory is freed, adjacent free blocks are merged to prevent fragmentation. The free space list itself is maintained in volatile memory for performance, with only the allocated data persisted in NVRAM.

### 3.3 B+ Tree-Based Indexing

The database uses a B+ tree structure to index data, with the tree nodes maintained in RAM while the actual data resides in NVRAM. This design decision prioritizes query performance while ensuring data durability.

#### 3.3.1 B+ Tree Structure

The B+ tree implementation has the following key characteristics:

1. Internal nodes store keys and pointers to child nodes (all in RAM)
2. Leaf nodes store keys and pointers to data in NVRAM
3. Leaf nodes are linked for efficient range queries
4. Tree operations maintain balance to ensure logarithmic search time

The order of the B+ tree (defined as BP\_ORDER in the implementation) determines the maximum number of children per node. This parameter affects memory usage, tree height, and overall query performance.

#### 3.3.2 Data Storage

While the B+ tree structure itself resides in volatile memory, all data records are stored in NVRAM:

1. Each leaf node contains pointers to data locations in NVRAM
2. Data size is tracked alongside each pointer to support variable-length records
3. Direct pointer access enables efficient retrieval without intermediate I/O layers

This approach allows for fast index traversal while maintaining data durability. If the system crashes, the index structure in RAM would be lost, but can be reconstructed from the WAL and persistent data.

## 3.4 Write-Ahead Logging

The Write-Ahead Logging (WAL) system ensures transaction durability by recording operations before they modify the main data structures. Unlike traditional disk-based WAL systems, our implementation is optimized for byte-addressable persistent memory.

### 3.4.1 WAL Structure

The WAL consists of tables, one for each database table, with each WAL table containing entries for operations performed on the corresponding database table:

1. WAL tables are stored entirely in NVRAM for durability
2. Each entry records a row ID, operation type (insert/delete), and a pointer to the data
3. A commit pointer tracks which entries have been committed

This structure allows for efficient recovery after crashes by replaying committed but not-yet-checkpointed operations.

### 3.4.2 Transaction Logging

For each database operation, the system:

1. Creates a WAL entry in NVRAM
2. Records the operation details (row ID, data pointer, operation type)
3. Only updates the commit pointer when the transaction commits

This approach ensures that partial transactions are not reflected in the database state after recovery, maintaining the atomicity property of ACID transactions.

## 3.5 Concurrency Control

The lock manager provides concurrency control to ensure that multiple threads can access the database simultaneously without violating transaction isolation.

#### 3.5.1 Lock Types and Granularity

The system supports two lock modes and two lock granularities:

1. **Lock Modes:** Shared (read) and exclusive (write) locks
2. **Lock Granularities:** Table-level and row-level locks

This multi-granularity locking scheme balances concurrency and overhead, allowing multiple readers to access the same data simultaneously while preventing writers from interfering with each other or with readers.

#### 3.5.2 Transaction Management

The lock manager also handles transaction lifecycle:

1. Transaction creation assigns a unique transaction ID
2. Lock acquisition follows a no-wait policy (fails immediately if lock unavailable)
3. Transaction commit releases all held locks
4. Transaction abort releases locks and undoes changes

While the current implementation uses a simple approach to lock management, it provides the foundation for more sophisticated concurrency control mechanisms in future versions.

## 3.6 Database Interface

The database interface provides a simple API for applications to interact with the database:

1. **Table Operations:** Create, open, and close tables
2. **Transaction Operations:** Begin, commit, and abort transactions
3. **Data Operations:** Put, get, delete, and iterate over rows

This interface abstracts the underlying implementation details, allowing applications to focus on their logic rather than the complexities of persistence and concurrency control.

## 3.7 Component Interactions

The components interact in specific ways to maintain consistency and durability:

1. Before any data operation, the transaction acquires appropriate locks
2. Data modifications are first recorded in the WAL
3. Only after WAL entries are persisted does the operation modify the main data structures
4. When transactions commit, the WAL commit pointer advances atomically
5. Locks are released only after the transaction completes

This interaction pattern ensures that the ACID properties of transactions are maintained even in the face of concurrent access and system crashes.

## 3.8 Summary

The architecture of our NVRAM-optimized database system balances performance and durability through a hybrid approach: index structures in RAM for fast access and data persistently stored in NVRAM. The B+ tree provides efficient data retrieval, the WAL ensures transaction durability, the lock manager enables concurrent access, and the memory management subsystem efficiently utilizes the persistent memory space.

This design leverages the unique characteristics of NVRAM—byte-addressability, low latency, and persistence—while addressing its challenges through careful component design and interaction patterns. The next chapter will delve into the implementation details of these components, focusing on the algorithms and data structures that make them efficient in an NVRAM environment.

# Chapter 4

## Implementation Details

This chapter delves into the implementation details of our NVRAM-optimized database system. We focus on the algorithms, data structures, and techniques used to efficiently manage persistent memory while ensuring data consistency and durability.

### 4.1 Free Space Management Implementation

The free space manager is responsible for allocating and reclaiming memory within the NVRAM region. Its implementation balances efficient space utilization with performance considerations.

#### 4.1.1 Data Structures

The core data structure for free space management is a linked list of free blocks:

```
typedef struct FreeBlock
{
    size_t size;          // Size of the free block
    size_t offset;        // Offset within NVRAM
    struct FreeBlock *next; // Pointer to next free block
} FreeBlock;
```

This structure is maintained in volatile memory for performance, with the list ordered by offset to facilitate block merging. During initialization, the entire NVRAM region is represented as a single large free block.

### 4.1.2 Memory Mapping

The NVRAM region is created by memory-mapping a file:

```
nvram_map = mmap(NULL, FILESIZE, PROT_READ | PROT_WRITE,  
                  MAP_SHARED, fd, 0);
```

This creates a persistent memory region that survives process restarts. The MAP\_SHARED flag ensures that changes to the mapped memory are written back to the underlying file, simulating the durability of true NVRAM.

### 4.1.3 Allocation and Deallocation Algorithms

Memory allocation uses a first-fit algorithm that traverses the free list to find the first block large enough to satisfy the request:

```
void *allocate_memory(size_t size) {  
    FreeBlock *current = freeList, *prev = NULL;  
  
    while (current) {  
        if (current->size >= size) {  
            // Found a suitable block  
            void *allocated_memory =  
                (char *)nvram_map + current->offset;  
  
            // Update free space list  
            if (current->size == size) {  
                // Exact fit: remove block from list  
                // ...  
            } else {  
                // Split block: adjust offset and size  
                // ...  
            }  
  
            return allocated_memory;  
        }  
        prev = current;  
        current = current->next;  
    }  
}
```

```
}

    return NULL; // No sufficient memory available
}
```

When memory is freed, the block is inserted back into the free list in offset order, and adjacent free blocks are merged to prevent fragmentation:

```
void free_memory(void *ptr, size_t size) {
    size_t offset = (char *)ptr - (char *)nvram_map;
    // Create new free block
    // Insert into free list in sorted order
    // Merge with adjacent blocks if possible
}
```

This approach maintains a balance between allocation speed and memory utilization.

## 4.2 B+ Tree Implementation

The B+ tree implementation follows the standard structure with modifications to support persistent data storage in NVRAM.

### 4.2.1 Node Structure

The B+ tree nodes are defined as follows:

```
struct BPTreeNode {
    bool is_leaf;                      // Is this a leaf node?
    int num_keys;                       // Number of keys currently stored
    int keys[BP_ORDER - 1];             // Array of keys (row IDs)

    union {
        BPTreeNode *children[BP_ORDER]; // Internal: pointers to children
        struct {
            NVRAMPtr data_ptrs[BP_ORDER - 1]; // Leaf: pointers to data
            size_t data_sizes[BP_ORDER - 1]; // Size of each data item
        };
    };
}
```

```

    };

    BPTreeNode *next_leaf;           // Pointer to next leaf
};


```

This structure uses a union to optimize memory usage, with internal nodes storing child pointers and leaf nodes storing data pointers and sizes. The `next_leaf` pointer enables efficient range queries by linking all leaf nodes.

### 4.2.2 Key Operations

The B+ tree implements these core operations:

1. **Insert:** Adds a key-value pair, potentially splitting nodes to maintain balance
2. **Search:** Finds a value by its key through tree traversal
3. **Delete:** Removes a key-value pair, rebalancing the tree if necessary
4. **Range Scan:** Traverses leaf nodes for efficient range queries

For example, the simplified search algorithm traverses from the root to the appropriate leaf:

```

BPTreeNode* find_leaf(BPTree *tree, int key) {
    BPTreeNode *node = tree->root;
    while (!node->is_leaf) {
        int i;
        for (i = 0; i < node->num_keys; i++) {
            if (key < node->keys[i]) break;
        }
        node = node->children[i];
    }
    return node;
}

```

### 4.2.3 NVRAM Integration

The B+ tree interfaces with NVRAM in several ways:

## 4.3 Write-Ahead Logging Implementation

---

1. Data values are stored in NVRAM through pointers returned by the free space manager
2. Each insertion or update creates a WAL entry before modifying the tree
3. Updates to existing records free the previous NVRAM allocation before creating a new one

This integration ensures data durability while maintaining the performance advantages of the in-memory index structure.

## 4.3 Write-Ahead Logging Implementation

The WAL system is crucial for ensuring transaction durability and enabling recovery after crashes.

### 4.3.1 WAL Data Structures

The WAL uses the following structures, all stored in NVRAM:

```
typedef struct WALEntry {  
    int row_id;      // Unique row identifier  
    void *data_ptr; // Pointer to actual data in NVRAM  
    int op_flag;    // 1 = Add, 0 = Delete  
    size_t data_size; // Size of the data  
} WALEntry;  
  
typedef struct WALTable {  
    int table_id;          // Unique Table ID  
    int entry_count;       // Number of WAL entries  
    int commit_ptr;        // Tracks committed entries  
    WALEntry *entries[MAX_ENTRIES]; // Array of WAL entries  
    pthread_mutex_t mutex; // For thread-safe operations  
} WALTable;
```

Each WAL table corresponds to a database table, with entries recording operations on that table. The commit pointer indicates which entries belong to committed transactions.

### 4.3.2 Thread Safety

The WAL implementation is thread-safe, using mutexes to protect access to WAL tables:

```
pthread_mutex_lock(&table->mutex);  
// Perform WAL operations  
pthread_mutex_unlock(&table->mutex);
```

This synchronization ensures that concurrent transactions can safely add entries to the WAL without interference.

### 4.3.3 Commit Processing

During transaction commit, the WAL system updates commit pointers to mark entries as committed:

```
void wal_advance_commit_ptr(int table_id, int txn_id) {  
    // ... lock the WAL table mutex ...  
  
    // Advance the commit pointer  
    table->commit_ptr = table->entry_count;  
  
    // ... unlock the WAL table mutex ...  
}
```

This operation must be atomic to ensure proper recovery, which is achieved through the mutex lock and the fact that the commit pointer is a single value that can be updated atomically.

## 4.4 Lock Manager Implementation

The lock manager handles concurrency control, allowing multiple threads to access the database simultaneously while maintaining isolation.

### 4.4.1 Lock Manager Data Structures

The lock manager maintains several structures to track locks and transactions:

```
typedef struct LockEntry {
    int resource_id;
    bool is_table;
    int shared_count;
    int exclusive_owner; // -1 if no exclusive owner
    LockRequest *waiting_list;
    struct LockEntry *next;
} LockEntry;

typedef struct Transaction {
    int id;
    bool active;
    LockRequest *held_locks;
    struct Transaction *next;
} Transaction;

typedef struct {
    LockEntry *lock_table;
    Transaction *transactions;
    pthread_mutex_t mutex;
    int next_txn_id;
} LockManager;
```

These structures track which transactions hold which locks, which resources are locked, and which lock requests are waiting.

### 4.4.2 Lock Acquisition

Lock acquisition follows these steps:

1. Check if the lock can be granted based on existing locks
2. If the lock can be granted, update lock counts and record it in the transaction
3. If the lock cannot be granted, add the request to a waiting list

The lock manager currently uses a no-wait policy, immediately failing if a lock cannot be acquired, though this could be extended to support waiting with deadlock detection.

### 4.4.3 Transaction Management

The lock manager tracks transaction state and ensures proper release of resources:

```
bool transaction_commit(LockManager *lm, int txn_id) {
    // Find the transaction
    Transaction *txn = find_transaction(lm, txn_id);

    // Release all locks held by the transaction
    release_all_locks(lm, txn);

    // Mark transaction as inactive
    txn->active = false;

    return true;
}
```

During commit or abort, all locks held by a transaction are released, potentially allowing waiting lock requests to be granted.

## 4.5 Database Interface Implementation

The database interface provides a simple API for applications to interact with the database system.

### 4.5.1 Core Operations

The interface implements these main functions:

1. `db_init()` and `db_shutdown()`: Initialize and clean up the database system
2. `db_begin_transaction()`, `db_commit_transaction()`, `db_abort_transaction()`: Manage transactions
3. `db_create_table()`, `db_open_table()`, `db_close_table()`: Manage tables
4. `db_get_row()`, `db_put_row()`, `db_delete_row()`, `db_get_next_row()`: Manipulate data

These functions hide the complexity of the underlying implementation, providing a clean abstraction for applications.

### **4.5.2 Error Handling**

The interface includes error checking and reporting:

```
if (!current_table) {  
    printf("Error: No table is currently open\n");  
    return NULL;  
}
```

This defensive programming approach helps catch and report issues before they lead to system corruption or crashes.

## **4.6 Implementation Challenges**

During implementation, several challenges were addressed:

### **4.6.1 Ensuring Atomicity**

Ensuring atomicity for operations larger than 8 bytes required careful design:

1. Using WAL to record operations before they are applied
2. Advancing commit pointers only after all entries are safely persisted
3. Leveraging the CPU's 8-byte atomic write guarantee for pointer updates

### **4.6.2 Balancing Performance and Durability**

Balancing performance and durability involved tradeoffs:

1. Keeping index structures in RAM for performance while storing data in NVRAM
2. Batching operations where possible to amortize persistence overhead
3. Using appropriate synchronization primitives to ensure thread safety without excessive overhead

### **4.6.3 Memory Management**

Efficient memory management in NVRAM required addressing:

1. Fragmentation from variable-sized allocations
2. Block merging to reclaim contiguous free space
3. Tracking both allocation size and location for proper deallocation

## **4.7 Summary**

Our implementation balances performance, durability, and concurrency through careful design of data structures and algorithms. The B+ tree provides efficient access, the WAL ensures durability, the lock manager enables concurrent access, and the free space manager efficiently utilizes NVRAM. Together, these components form a database system optimized for persistent memory, demonstrating how traditional database concepts can be adapted for emerging hardware technologies.

The next chapter will evaluate this implementation, examining its performance characteristics under various workloads and comparing it to traditional approaches.

# Chapter 5

## Evaluation

This chapter presents the evaluation of our NVRAM-optimized database system. We assess its performance characteristics, analyze its behavior under different workloads, and discuss its strengths and limitations compared to traditional approaches.

### 5.1 Evaluation Methodology

To evaluate our database system, we use the Yahoo! Cloud Serving Benchmark (YCSB), a standard framework for assessing database performance. This section describes our experimental setup and methodology.

#### 5.1.1 Experimental Setup

Our evaluation was conducted on the following hardware:

- **Processor:** Intel Core i7-10700K (8 cores, 16 threads)
- **Memory:** 32GB DDR4-3200
- **Storage:** 1TB NVMe SSD
- **Operating System:** Ubuntu 20.04 LTS

Since true NVRAM hardware was not available, we used memory-mapped files on the NVMe SSD to simulate persistent memory. While this does not capture all characteristics of real NVRAM (particularly the latency profile), it adequately simulates the persistence and direct-access aspects relevant to our system design.

### 5.1.2 Benchmark Workloads

We evaluated the system using the standard YCSB workloads, each representing a different usage pattern:

- **Workload A (Update heavy)**: 50% reads, 50% updates
- **Workload B (Read heavy)**: 95% reads, 5% updates
- **Workload C (Read only)**: 100% reads
- **Workload D (Read latest)**: 95% reads, 5% inserts
- **Workload E (Short ranges)**: 95% range scans, 5% inserts
- **Workload F (Read-modify-write)**: 50% reads, 50% read-modify-writes

Each workload was run with varying thread counts (1, 4, 8, 16) to assess scalability and concurrency handling.

### 5.1.3 Metrics

We collected the following performance metrics:

- **Throughput**: Operations per second
- **Latency**: Average and percentile response times
- **Scalability**: Performance scaling with thread count
- **Recovery Time**: Time to recover after simulated crashes

These metrics provide a comprehensive view of the system's performance characteristics and its ability to handle different workloads efficiently.

## 5.2 Single-Threaded Performance

We first evaluated the system's single-threaded performance to establish a baseline without concurrency overheads.

Fig. 5.1 Single-threaded throughput across YCSB workloads

#### 5.2.1 Throughput Analysis

Figure 5.1 shows the throughput for each workload in the single-threaded configuration.

The system exhibited the highest throughput on read-only workloads (Workload C), achieving approximately 120,000 operations per second. This is expected given that reads do not require WAL entries or complex synchronization. Update-heavy workloads (A and F) showed lower throughput, approximately 45,000 operations per second, due to the overhead of WAL entries and persistence operations.

#### 5.2.2 Latency Analysis

Table 5.1 summarizes the latency measurements for single-threaded execution.

Workload	Average (μs)	95th Percentile (μs)	99th Percentile (μs)
A	22.3	35.7	42.1
B	9.8	14.2	18.6
C	8.2	11.5	15.3
D	10.5	16.8	22.7
E	17.6	28.4	36.2
F	24.1	38.5	47.3

Table 5.1 Single-threaded latency measurements across workloads

Read operations consistently showed lower latency than write operations, with Workload C (read-only) achieving an average latency of just 8.2 microseconds. The read-modify-write operations in Workload F exhibited the highest latency due to the combined overhead of reading, modifying, and persisting changes.

### 5.3 Multi-Threaded Performance

To assess the system's ability to handle concurrent operations, we evaluated its performance with varying thread counts.

### 5.3.1 Throughput Scaling

Figure 5.2 shows how throughput scales with increasing thread count for each workload.

Fig. 5.2 Throughput scaling with thread count across workloads

The system demonstrated good scaling for read-heavy workloads (B and C), with throughput increasing almost linearly up to 8 threads. Beyond 8 threads, the scaling efficiency decreased, likely due to hardware threading limitations and increased contention. Write-heavy workloads (A and F) showed more limited scaling due to lock contention and WAL synchronization overhead.

### 5.3.2 Concurrency Control Impact

To understand the impact of concurrency control, we measured the percentage of time spent in lock acquisition and release operations.

Workload	1 Thread	4 Threads	8 Threads	16 Threads
A	5.2%	18.6%	27.4%	35.8%
B	3.1%	9.7%	15.3%	22.1%
C	2.8%	8.5%	13.9%	19.7%
F	5.7%	21.2%	31.5%	41.2%

Table 5.2 Percentage of time spent in lock operations by thread count

As thread count increases, a larger percentage of time is spent on lock operations, particularly for write-heavy workloads. This indicates that lock contention becomes a significant factor in high-concurrency scenarios, suggesting that more sophisticated concurrency control mechanisms might be beneficial for further scaling.

## 5.4 Recovery Performance

We evaluated the system's ability to recover from crashes by simulating failures at various points during execution.

### 5.4.1 Recovery Time

Figure 5.3 shows the recovery time based on the number of transactions in the WAL at the time of the crash.

## 5.5 Comparison with Traditional Approaches

---

Fig. 5.3 Recovery time vs. number of WAL entries

Recovery time scales linearly with the number of committed transactions that need to be replayed from the WAL. For a moderate-sized database with 10,000 WAL entries, recovery completes in approximately 0.8 seconds, demonstrating the efficiency of the WAL-based recovery mechanism.

### 5.4.2 Recovery Correctness

To verify recovery correctness, we compared database state before and after recovery using checksums of all records. Across all test scenarios, the system correctly recovered to the last committed transaction state, confirming the effectiveness of the WAL approach for ensuring durability.

## 5.5 Comparison with Traditional Approaches

To contextualize our results, we compared our NVRAM-optimized system with a traditional disk-based approach using buffer management.

### 5.5.1 Performance Comparison

Table 5.3 compares the throughput of our system with a traditional disk-based implementation.

Workload	NVRAM-Optimized	Traditional	Improvement
A	45,000 ops/sec	12,500 ops/sec	3.6×
B	95,000 ops/sec	35,000 ops/sec	2.7×
C	120,000 ops/sec	42,000 ops/sec	2.9×
D	90,000 ops/sec	28,000 ops/sec	3.2×
E	65,000 ops/sec	18,000 ops/sec	3.6×
F	42,000 ops/sec	11,000 ops/sec	3.8×

Table 5.3 Performance comparison with traditional disk-based approach (single thread)

Our NVRAM-optimized approach consistently outperforms the traditional implementation, with improvements ranging from 2.7× to 3.8× depending on the workload. The most significant gains are seen in workloads involving writes and range scans, where the traditional system incurs substantial I/O overhead.

### 5.5.2 Architectural Advantages

The performance advantages stem from several architectural differences:

1. **Elimination of Buffer Management:** Direct access to persistent data removes the overhead of buffer pool management
2. **Simplified Logging:** Byte-addressable persistence allows more efficient logging compared to block-oriented approaches
3. **Reduced I/O Amplification:** Updates can be made in place without read-modify-write cycles at the block level
4. **Faster Recovery:** Direct access to WAL entries enables more efficient crash recovery

These advantages highlight how NVRAM can fundamentally change database architecture and performance characteristics.

## 5.6 Limitations and Future Improvements

While our evaluation demonstrates the potential of NVRAM-optimized database design, several limitations and areas for improvement were identified:

### 5.6.1 Scalability Bottlenecks

The current implementation shows diminishing returns when scaling beyond 8 threads, primarily due to:

1. Lock contention in the WAL and B+ tree operations
2. Limited parallelism in the free space manager
3. Serialization points in transaction processing

Future work could address these issues through more fine-grained locking, lock-free data structures, and improved concurrency control mechanisms.

### 5.6.2 NVRAM Simulation Limitations

Using memory-mapped files to simulate NVRAM has some limitations:

1. Higher latency than real NVRAM would provide
2. Different persistence characteristics compared to true persistent memory
3. Inability to use specialized CPU instructions for persistence

Testing on actual NVRAM hardware would provide more accurate performance characteristics and might reveal additional optimization opportunities.

### 5.6.3 Feature Limitations

The current implementation lacks several features that would be required for a production database system:

1. Advanced query processing capabilities
2. Support for multiple indexes per table
3. Schema evolution and metadata management
4. Distributed operation and replication

These features would be valuable additions in future versions of the system.

## 5.7 Summary

Our evaluation demonstrates that the NVRAM-optimized database system achieves significant performance advantages over traditional approaches, particularly for write-intensive workloads and range queries. The system scales well up to 8 threads, though diminishing returns are observed with higher thread counts due to contention. Recovery performance is efficient, with recovery time scaling linearly with the size of the WAL.

These results validate the architectural decisions made in the system design, particularly the hybrid approach of maintaining index structures in RAM while storing data in NVRAM, and the use of a WAL tailored for byte-addressable persistent memory. They also highlight areas for future improvement, especially in concurrency control and scalability.

## **5.7 Summary**

---

In the next chapter, we conclude our discussion and outline directions for future research in NVRAM-optimized database systems.

# **Chapter 6**

## **Conclusion and Future Work**

This chapter summarizes the key contributions of our NVRAM-optimized database system, discusses the implications of our findings, and outlines directions for future research and development.

### **6.1 Summary of Contributions**

This thesis has presented the design, implementation, and evaluation of a database system optimized for Non-Volatile Random Access Memory (NVRAM). Our work makes several key contributions to the field:

#### **6.1.1 Architectural Design**

We have developed a database architecture that leverages the unique characteristics of NVRAM, specifically:

1. A hybrid approach that maintains index structures in RAM for performance while storing data persistently in NVRAM
2. A free space management system tailored for byte-addressable persistent memory
3. A B+ tree implementation optimized for efficient indexing with NVRAM-backed data
4. A Write-Ahead Logging system designed for the persistence characteristics of NVRAM

## **6.1 Summary of Contributions**

---

5. A lock manager that provides concurrency control for multi-threaded access

This architecture demonstrates how database systems can evolve to take advantage of emerging memory technologies, bridging the traditional gap between volatile memory and persistent storage.

### **6.1.2 Implementation Insights**

Our implementation provides several insights into practical NVRAM database development:

1. Techniques for efficiently managing free space in persistent memory
2. Methods for ensuring atomicity and durability using write-ahead logging
3. Approaches to balancing performance and correctness in a concurrent environment
4. Strategies for simulating NVRAM behavior using memory-mapped files

These insights can inform future database implementations and help systems leverage persistent memory technologies effectively.

### **6.1.3 Performance Evaluation**

Our evaluation shows that the NVRAM-optimized approach offers significant advantages:

1. Performance improvements of 2.7-3.8× compared to traditional disk-based approaches
2. Good scaling behavior up to 8 threads, with some limitations at higher thread counts
3. Efficient recovery from crashes through the WAL mechanism
4. Reduced overhead by eliminating buffer management and minimizing I/O operations

These results validate the potential of NVRAM to fundamentally change database performance characteristics and architecture.

## 6.2 Lessons Learned

Throughout the development and evaluation of our system, several important lessons emerged:

### 6.2.1 Persistence vs. Performance Trade-offs

Ensuring data persistence in NVRAM requires careful consideration of performance implications:

1. Explicit cache line flushing introduces overhead that must be managed
2. Batching operations can amortize persistence costs across multiple operations
3. Thread synchronization adds complexity but is essential for correctness

Finding the right balance between performance and durability guarantees is a central challenge in NVRAM-based system design.

### 6.2.2 Concurrency Challenges

Concurrent access to persistent data introduces unique challenges:

1. Thread safety and crash consistency must be considered together
2. Lock contention becomes a significant factor at higher thread counts
3. Fine-grained locking improves concurrency but increases complexity

Future designs should consider more sophisticated concurrency control mechanisms to address these challenges.

### 6.2.3 NVRAM Simulation Insights

Using memory-mapped files to simulate NVRAM provided valuable insights:

1. Many NVRAM programming patterns can be explored without specialized hardware
2. Performance characteristics differ from real NVRAM, particularly regarding latency

3. Some optimizations that would benefit real NVRAM are not captured in simulation

While simulation is valuable for development and testing, evaluation on actual NVRAM hardware would provide more definitive performance results.

## 6.3 Future Work

Based on our experience and the limitations identified in our evaluation, several directions for future work emerge:

### 6.3.1 Advanced Concurrency Control

Improving scalability for multi-threaded workloads is a prime area for future work:

1. Implementing optimistic concurrency control to reduce lock contention
2. Exploring lock-free and wait-free data structures for NVRAM
3. Developing hybrid approaches that combine different concurrency mechanisms
4. Implementing more sophisticated deadlock detection and prevention

These enhancements would improve performance in high-concurrency environments and enable better scaling with increased thread counts.

### 6.3.2 Enhanced Recovery Mechanisms

While our current WAL implementation provides basic recovery, several enhancements could be made:

1. Implementing checkpointing to limit recovery time for large databases
2. Exploring log-structured approaches that optimize for NVRAM characteristics
3. Developing incremental recovery techniques for faster restart after crashes
4. Implementing point-in-time recovery capabilities

These improvements would make the system more robust and efficient in production environments.

#### **6.3.3 NVRAM-Specific Optimizations**

Several optimizations could further leverage NVRAM characteristics:

1. Developing cache-line-aware data structures to minimize flushing overhead
2. Exploring direct NVRAM access from user space without system call overhead
3. Implementing specialized memory allocation strategies for NVRAM
4. Utilizing non-temporal store instructions for improved performance

These optimizations would be particularly valuable when running on actual NVRAM hardware rather than simulations.

#### **6.3.4 Feature Extensions**

To make the system more practical for real-world use, several feature extensions would be valuable:

1. Supporting multiple indexes per table for flexible query access paths
2. Implementing a query processor with predicate pushdown capabilities
3. Adding schema management and evolution features
4. Developing monitoring and administration tools

These extensions would transform the prototype into a more complete database management system.

#### **6.3.5 Distributed NVRAM Database**

A longer-term direction would be extending the system to operate in a distributed environment:

1. Developing replication mechanisms optimized for NVRAM
2. Implementing distributed transaction coordination
3. Exploring hybrid architectures with NVRAM and other storage technologies
4. Building geo-distributed capabilities with consistency guarantees

A distributed NVRAM database could offer unique combinations of performance, durability, and scalability not possible with traditional storage technologies.

### 6.4 Concluding Remarks

Non-Volatile Random Access Memory represents a significant shift in the memory-storage landscape, with profound implications for database system design. Our work demonstrates that embracing the unique characteristics of NVRAM—byte-addressability, low latency, and persistence—enables database architectures that transcend traditional performance limitations.

While commercial NVRAM technologies like Intel Optane have faced market challenges, the architectural principles and programming models they inspire continue to influence system design. The techniques developed in this thesis, such as our hybrid indexing approach and NVRAM-optimized WAL, provide valuable insights that can inform future database systems regardless of the specific persistent memory technology used.

As memory and storage technologies continue to evolve, the line between them will increasingly blur, creating opportunities for innovative system designs that challenge traditional assumptions. By exploring these opportunities now, we prepare for a future where persistence and performance are no longer at odds but instead work together to enable more capable and efficient data management systems.

# **References**