The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Systems and Society

ARITHMETY: Adaptive Real-Time Text Inversion over Hybrid DRAM and Persistent Memory

— Honours project (S1/S2 2025)

A thesis submitted for the degree
Bachelor of Advanced Computing (Research and Development) (Honours)

By:
Kshama Patel

Supervisor:
Dr. Shoaib Akram

March 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

March (2026), Kshama Patel

Acknowledgements

I would like to sincerely thank my supervisor, Shoaib Akram. I could not have made the progress or achieved the outcomes of my thesis without his guidance, feedback, and support. Our meetings together were both productive and enjoyable. I deeply appreciate the time and effort he has dedicated to helping me reach my full potential and produce a thesis that I am proud of. This journey has been invaluable, allowing me to develop both my technical and perseverance skills.

I would also like to thank my family and friends for their support and motivating me during tough times. I could not have achieved this without them, and I am truly grateful for everything they have done for me.

Abstract

Persistent memory (PM) reduces the overhead of accessing persistent data by offering byte addressability, non-volatility, and scalable capacity. It is especially suited to data-intensive applications, such as search and analytic engines, that perform frequent I/O operations. Unfortunately, unlike volatile main memory, Dynamic Random Access Memory or DRAM, writes to PM require special treatment. The key reason is the presence of volatile caches in the memory hierarchy. On a shutdown, data may reside in the caches, and it may reach PM in an order different from what the programmer desires. Today, developers use special instructions for flushing and ordering writes from caches to PM. Specifically, using a combination of *clflush* and *mfence* instructions guarantees persistence in a crash-consistent manner, but these instructions introduce significant overhead. Also, PM has low read and bandwidth compared to DRAM. Therefore, when used in the data centre where user latency is the determinant for the success of an online service, there is a risk of violating service-level agreements if PM's bandwidth is under pressure. This work focuses on enabling (1) low-overhead crash consistency using application-specific logging, and (2) guaranteeing bounded tail latency by adapting the service to PM pressure. We define PM pressure to be an aggregation of the read and write PM bandwidth.

Recent work explores exploiting PM for real-time search and analytic engines. The state-of-the-art proposal, namely SPIRIT, builds an inverted index over hybrid DRAM and PM (i.e., performs text inversion), but exhibits two drawbacks. (1) Merging of terms across search indices requires several high-overhead, *cflush* and *mfence* instructions per term, and (2) the system cannot adapt to client priorities (e.g., bounded tail latency at the expense of weaker crash consistency guarantees). In this work, we first optimise the merge operation by proposing a crash consistency mechanism that reduces the total number *cflush* and *mfence* instructions required per term. Our optimisation reduces the merge time by 4×, and consequently, the total indexing time by 2×. Next, we rigorously analyse search indexing and query workloads, and find variations in PM pressure. Our findings show that (1) increased reads and writes to PM can degrade query performance (reads) by up to 2.9×, (2) high-frequency, multi-term queries are the slowest, being 61.7× slower than low-frequency, single-term queries with single-threaded querying, and (3) multi-threaded querying degrades query performance (reads) across all query types,

with a maximum of 3.7×.

To adapt to various scenarios and respect client priorities, we propose ARITHMETY, a new search engine that introduces a controller service to continuously monitor variations in PM pressure. It dynamically adapts execution to deliver the specified client priority. Specifically, the client specifies fast query response (with bounded query latency as a threshold), strong crash consistency or data freshness as a priority. ARITHMETY adapts to workload and client priorities using four execution phases, (1) profiling the workload in real-time, (2) detecting drift in query response times, (3) deciding to maintain or mitigate PM pressure, and (4) adapting indexing as per the decision. Our evaluation shows that ARITHMETY reduces tail latency by 3.5×, in a system evaluating high-frequency, multi-term queries, with a bounded query latency of 40 ms. ARITHMETY ensures that tail latency remains below the bounded latency, if possible, or at the lowest achievable latency for a query workload. Moreover, ARITHMETY adapts its behaviour to continue indexing during periods of low pressure, while reducing latencies by up to 1.3× under high pressure conditions with slow queries.

Table of Contents

Chapter 1

Introduction

Today, real-time search and analytics are critical to the success of enterprises, such as Google, Twitter, and LinkedIn. These companies invest heavily in optimising response times and enhancing the accuracy of search results. As the web continuously evolves with new content being created and made available, search engines face constant pressure to provide users with the most up-to-date information with minimal delay.

To meet this demand, search engines employ specialised software known as web crawlers. Web crawlers navigate and collect newly generated content, in real-time, from the World Wide Web (WWW). They feed this new (fresh) content to a real-time search engine forming a crawl-index-search pipeline. Indexing organises the content in a way that supports efficient look-ups for serving user search queries. A fundamental data structure used by search engines is an inverted index (Zobel and Moffat (2006)), which maps each term to the web pages in which it appears.

The storage of inverted indices poses challenges. Service providers have considered hosting indices in DRAM to deliver fast real-time responses for search (Gupta et al. (2023)). However, as the number of web pages continues to grow, the index size scales proportionally (Gupta et al. (2023)). Since DRAM is expensive and available only in limited capacity, it poses scalability challenges. It also lacks persistence, such that in the event of a crash, the index on DRAM is lost. Hosting inverted indices on Solid State Drives (SSDs) has proven insufficient, as heavy processing in the kernel delays query responses (Akram (2021)).

The challenges posed by DRAM-only or SSD-only systems have driven research into hybrid memory systems that employ a two-tier memory hierarchy. In such systems, DRAM typically serves as the primary tier, offering fast but limited capacity storage. SSDs, or another slower memory technology, function as the secondary tier. This approach allows the index to scale across both tiers, overcoming scalability limits while also providing persistence (on secondary storage).

Prior work proposes a real-time search engine over hybrid memory. The state-of-the-art proposal, SPIRIT, is the first hybrid memory search engine that combines DRAM and an emerging memory technology, persistent memory (PM), to build an inverted index (i.e., perform text-inversion) (Hasnat and Akram (2025)). PM is designed to bridge the gap between DRAM and SSDs. It combines near-DRAM access speeds with the persistence capabilities of SSDs. The evaluation of SPIRIT by Hasnat and Akram (2025) demonstrates a promising direction for hybrid memory search engines that integrate DRAM and PM, while providing crash consistency guarantees. By eliminating system calls and kernel entry during latency-sensitive operations, such as query search, SPIRIT delivers true real-time responses, unlike existing search engines that are classified as near real-time (Elastic (2025)).

However, SPIRIT exhibits two drawbacks. (1) Merging of terms across inverted indices requires several high-overhead, *cflush* and *mfence* instructions per term, and (2) SPIRIT cannot adapt to client priorities (e.g., bounded tail latency at the expense of weaker crash consistency guarantees).

Crash consistency on PM is challenging. The primary reason is due to the presence of volatile caches within the memory hierarchy. During a shutdown, data may still reside in these caches, potentially reaching PM in an order different from that intended by the developer. To address this, developers use a combination of *clflush* and *mfence* instructions to ensure persistence in a crash-consistent manner. However, these instructions incur significant overhead (Wu et al. (2019)). Nonetheless, to ensure crash consistency, SPIRIT issues *clflush* and *mfence* instructions to persist recovery data.

Particularly, the overhead of *clflush* and *mfence* instructions is noticeable during the merge operation of SPIRIT. This overhead stems from the fact that several instructions are issued for every term that is merged. In this work, our first contribution is proposing, implementing and evaluating an optimised crash consistency mechanism that reduces total number of *clflush* and *mfence* instructions issued. Our evaluation results show that SPIRIT's merge time is reduced by 4×, and consequently, the total time for indexing a workload by 2×.

Our next contribution addresses the second drawback of SPIRIT, i.e., it does not adapt to client priorities. For example, in a real-world scenario, clients may value fast query response times at the expense of weaker crash consistency guarantees, or vice versa (Jiang et al. (2023)). Existing data-intensive applications for hybrid memories, including SPIRIT, always prioritises strong crash consistency guarantees. In this work, we focus on making SPIRIT an adaptive application that reacts to client priorities. To optimise it for fast query responses, we first analyse SPIRIT to identify variations in PM pressure. We hypothesise that concurrent indexing (writes to PM) and query evaluation (reads to PM) increase PM pressure and slow down reads, as work of Fedorova et al. (2022) shows. This is due to the limited PM bandwidth.

In our analysis, we observe variations in PM pressure by observing the impact on per-query latency (reads). We define PM pressure to be an aggregation of the read and write

PM bandwidth. Our findings indicate that increased reads and writes in the system negatively impact per-query latency (reads). Specifically, we observe an increase of up to 2.9× in query latency with mixed reads and writes in the system. Moreover, high-frequency, multi-term queries exhibit the highest latency, being 61.7× slower compared to low-frequency, single-term queries with a single query evaluator. Additionally, increasing the number of query evaluators results in an increase of up to 3.7× in per-query latency. These results strongly motivate the need for an adaptive system capable of (1) detecting drifts in per-query latency (or PM pressure) and (2) dynamically adjusting SPIRIT's execution to maintain or mitigate PM pressure based on the client priority.

In this work, we propose ARITHMETY, a system that executes over hybrid DRAM and persistent memory and adapts to client priorities. When clients favour fast query responses, ARITHMETY bounds tail query latency by pausing indexing (writes), and resuming indexing when PM pressure decreases. Conversely, when strong crash consistency is favoured, the system continues both indexing (writes) and query evaluation (reads), regardless of variations in PM pressure. ARITHMETY is built on a client-server architecture. We design and implement a server that hosts a controller service. The controller is responsible for monitoring and detecting variations in PM pressure, and deciding to maintain or mitigate pressure based on the client priority. The implementation of ARITHMETY extends SPIRIT to communicate with the server and send real-time measurements of per-query latency and query workload. SPIRIT dynamically adapts indexing (writes to PM) as per the decision of the controller. Our design and implementation is generalisable to other applications.

Evaluating real-time search engines is challenging as enterprises do not make their workloads (i.e., trace of client requests) publicly available. Therefore, we design realistic workloads that simulate varying levels of pressure on PM to effectively evaluate ARITHMETY. Our results show that ARITHMETY adapts to reduce tail latency by up to 3.5×. This result is observed in a system executing high-frequency, multi-term queries, while gradually increasing pressure via mixed reads and writes to PM. ARITHMETY ensures that tail latency remains below a bounded latency, if possible, or at the lowest attainable latency for a query workload. Furthermore, ARITHMETY maintains a balance between data ingestion and query response times by dynamically adapting under mixed query workloads. It intelligently delays indexing to lower the latency for slower queries, by up to 1.3×, and restarts writes when PM pressure is low.

The main contributions of this work are summarised as follows:

- Proposing and implementing an optimised crash consistency mechanism for real-time full-text search that minimises the use of high-overhead *clflush* and *mfence* instructions.
- Rigorous analysis of real-time search engine execution on hybrid DRAM-PM systems to determine variations in pressure on PM, and uncovering a new opportunity to lower the pressure on PM to bound tail latency, when faster query response times are favoured.

- Designing and implementing ARITHMETY, which provides a controller service for an existing real-time search engine, capable of detecting changes in PM pressure, and adapting to maintain or mitigate PM pressure based on the client priority.

1.1 Thesis Structure

The thesis is structured into two main parts. The first part provides the necessary background in Chapter 2, including the explanation of SPIRIT required for understanding our key contributions. Chapter 3 discusses the prior literature. We then discuss our first contribution, proposing and implementing an optimised crash consistency mechanism in Chapter 4. We discuss its evaluation in Chapter 9. Chapter 5 outlines the experimental methodology for both contributions of the thesis. The second part of the thesis focuses on our second contribution, the motivation, design, implementation, and evaluation of ARITHMETY. Chapter 6 motivates the need for ARITHMETY by presenting the analysis of variations in PM pressure. Chapter 7 details the design and implementation of ARITHMETY. We present the evaluation of ARITHMETY in two parts. Chapter 8 presents a sensitivity analysis of configurable parameters. Chapter 9 presents evaluation results for our workloads. Finally, Chapter 10 summarises the thesis and outlines directions for future work.

Chapter 2

Background

2.1 Search Engines

A text search engine retrieves a list of documents to respond to a user query (Schütze et al. (2008)). For example, if a user searches for the term *dog*, the search engine returns links to documents containing the word *dog*.

A text search engine implements a two-step process to deliver results to users (Halavais (2017)). ①**Indexing** involves ingesting new documents and storing them in an inverted index (discussed in Section 2.1.2), which acts as the database for the search engine. Building an inverted index is commonly referred to as text inversion. ②**Querying** is initiated by the user's query. In this step, the search engine retrieves the most relevant documents from its index based on the user's query and returns them to the user. We delay the full explanation of indexing and querying till Section 2.1.2.

2.1.1 Real-Time Search

Unlike traditional search, real-time search engines index recently generated content and make it searchable for user queries without noticeable delay (Busch et al. (2012a)). This ensures the search results reflect the most up-to-date information. This contrasts with search engines that retrieve results only from pre-indexed or periodically updated data.

2.1.2 Indexing and Query Evaluation

In this section, we describe indexing and query evaluation, specific to real-time search, along with relevant data structures.

An inverted index is the underlying data structure many search engines use (Busch et al. (2012a)). At a high level, it maintains a mapping between each term and the list

of documents containing that term. In other words, it lists the documents per term. This is different to a forward index, which lists the terms per document (see Figure 2.1).

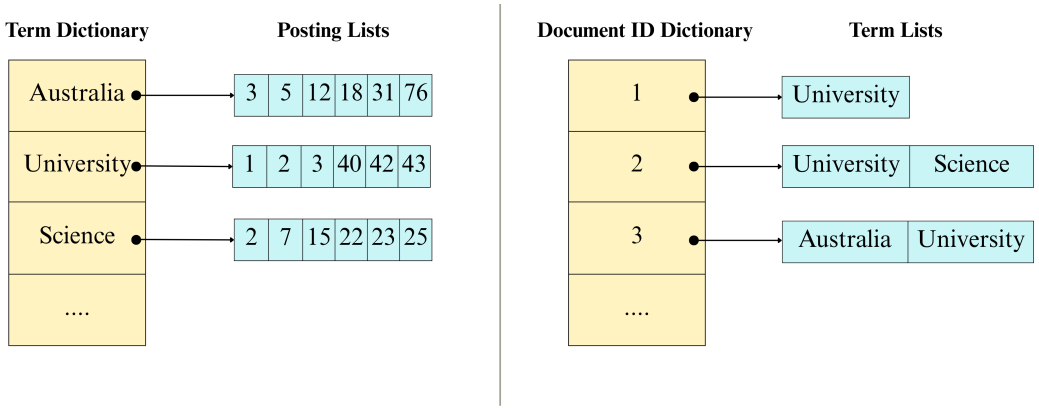


Figure 2.1: A simplified diagram of an inverted index (left) versus a forward index (right) data structure.

Indexing builds an inverted index as new documents are ingested. The inverted list consists of a term dictionary and posting lists. The **posting lists** are stored in a dedicated postings file. Each posting includes the document ID, an integer value, and metadata, e.g. term position and frequency. The **term dictionary** is a separate file that contains a list of unique terms, along with their mappings to the postings file. For each term, this mapping specifies the offset into the postings file, where its corresponding posting list begins.

At a high level, query evaluation involves processing a user query, which may contain a single term, or multiple terms. The results returned by a search engine depend on the type of query. For instance, an AND query only retrieves documents containing all terms specified in the user query. Query evaluation typically involves three main steps. ①**Parsing** the user query. ②**Dictionary lookup**, which involves checking whether each query term is present in the inverted index's term dictionary, and then, retrieving the corresponding offsets into the postings file. Many implementations use a hashtable to implement an inverted index for efficient lookups (Thai and Akram (2022)). ③**Posting list traversal**, which involves simply copying the posting list for a single-term query, but, for an multi-term AND query, this is more complex. A naive approach scans the posting lists for all query terms from left to right, comparing document ID candidates to find matches (Manning et al. (2008)).

2.2 Memory Technologies

Memory technologies use different storage media for storing data, each differing in cost, performance, capacity and their ability to retain data. Memory technologies like Random Access Memory (RAM) – Dynamic Random Access Memory (DRAM), and Static

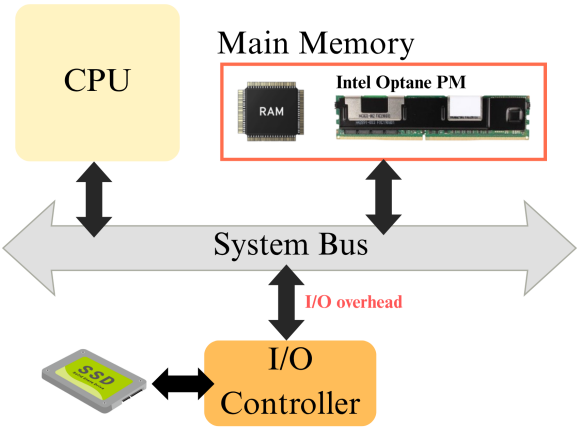


Figure 2.2: The read and write access paths to different types of memory and I/O in a computer system (RAM image: Supiandi, Intel Optane PM image: LenovoInc).

Random Access Memory (SRAM), are types of volatile memory, which lose data when powered off. Whereas Non-Volatile Memory (NVM), like Hard Disk Drive (HDD) and Solid State Drive (SSD), retain data even after the system is powered off, offering persistence. While DRAM serves as short-term data storage, it functions as the computer's main memory, connected directly via the system bus. This placement enables fast data access. However, due to its hardware complexity, fast access speeds and high demand, DRAM is expensive and typically limited in capacity (Zolenko (2024)). Conversely, SSDs serve as long-term, secondary storage. SSDs are not connected directly via the system bus, and consequently incur I/O overhead from system calls, additional copying of data and slower access speeds (see Figure 2.2). However, SSDs are significantly more affordable than DRAM, making them a cost-effective solution for large-capacity storage (Mead (2003)).

2.2.1 Persistent Memory

Persistent memory (PM) is a new type of NVM that offers access times on the order of DRAM. PM modules connect to the system bus (as shown in Figure 2.2) and form part of the main memory, similar to DRAM. In addition, PM is byte-addressable storage, enabling read and write operations at the granularity of individual bytes, similar to DRAM.

PM can be considered as a memory tier in-between DRAM and SSDs. It provides the persistence capabilities of SSDs, retaining data even when powered off. It is more cost-effective than DRAM, and it is offered in larger storage capacities similar to SSDs (Izraelevitz et al. (2019); Yang et al. (2020)).

We consider a specific type of persistent memory – Intel's real-world NVM prototype, known as Intel Optane DC Persistent Memory (Optane PM). The key features of Optane

PM are as follows:

- Optane PM modules are available in capacities of 128 GB, 256 GB, and 512 GB (Sun et al. (2022)).
- Optane PM enables reads/writes to occur, internally at 256-byte granularity.
- In comparison to DRAM, Optane PM shows 2× to 3× higher read latencies, and in comparison to SSDs, it shows 10× lower read latencies (Yang et al. (2020)).
- Unlike DRAM, Optane PM bandwidth is asymmetric, this means that the rate at which data can be read from/written to memory varies. A single Optane PM module has a maximum read bandwidth of 6.6 GB/s, and a maximum write bandwidth of 2.3 GB/s (Izraelevitz et al. (2019)).
- Sun et al. (2022) report Optane PM to be 3× to 4× less expensive than DRAM per GB, which makes it a promising alternative for scaling-up systems.

In this work, our focus is on Intel Optane PM.

PM can be configured in two modes: Memory (cached) and App Direct (uncached) mode (Izraelevitz et al. (2019)). Memory mode extends the memory capacity available by using PM as an additional cache, as shown in Figure 2.3. This improves system performance by providing more space to store frequently accessed (hot) data. In App Direct mode, PM is simply used as an additional storage device, which is an affordable expansion to the existing memory, with low-latency access to a persistent storage medium.

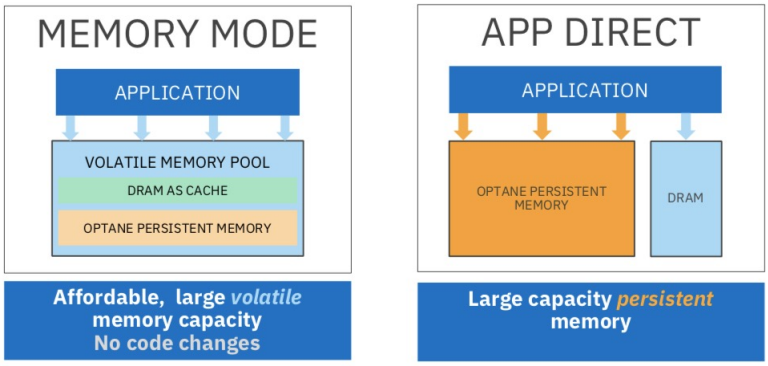


Figure 2.3: Comparison of PM memory (cache) mode versus app direct (uncached) mode (McLellan (2020)).

2.2.2 Hybrid Memory Systems

Hybrid memory systems exploit different memory technologies to leverage the benefits of each memory type. These systems typically consist of two memory tiers: a fast, low-capacity primary tier, and a slower, high-capacity secondary tier. DRAM is a common choice for the primary tier, while NVM is a common choice for the secondary tier, to

guarantee persistence and scalability. The properties of PM, an NVM memory technology, make it well-suited for use as secondary storage in hybrid memory systems (Kassa et al. (2021)). This enables the construction of large-scale, high-performance memory systems that can also provide data persistence (Hasnat and Akram (2025)).

2.3 Crash Consistency

Crash consistency refers to ensuring that data remains in a consistent state after an unexpected system crash (Pillai (2017)). A system crash can occur from a sudden power loss or even a software bug that renders the system unusable. This requires a restart of the system to resume normal operation. Since system crashes are inevitable, it is crucial to ensure the integrity of both in-memory and on-disk data. This is so that a system continues to function as expected, without data loss or corruption. Therefore, crash consistency is a property of many database systems.

Database systems typically rely on a Database Management System (DBMS), which enforces Atomicity, Consistency, Isolation, and Durability (ACID) properties, to ensure reliable and correct database transactions take place (Pillai (2017)). A database transaction is a sequence of operations to apply to the database, treated as a single unit of operation. The ACID properties ensure that the sequence of operations is executed as a single database transaction. **Atomicity** guarantees that each transaction is an all-or-nothing operation, meaning it either completes fully or has no effect at all. **Consistency** preserves data integrity by ensuring transactions take the database from one valid state to another. **Isolation** prevents concurrently executing transactions from interfering with each other's intermediate states. **Durability** guarantees that committed changes remain permanent despite system failures. Together, these properties provide strong crash consistency guarantees (Banothu et al. (2016)).

Crash consistency protocols are part of many applications to ensure that data remains in a consistent state following an unexpected system crash. These protocols can be considered in two parts: (1) the update protocol and (2) the recovery protocol (Pillai (2017)). The update protocol outlines the steps an application follows to modify data during execution time. In contrast, the recovery protocol specifies the actions taken after restarting the system from a crash.

Approaches to Guarantee Crash Consistency

Write-Ahead Logging (WAL) is a standard technique used in database systems to revert the database to a consistent state following a crash (Hu et al. (2019)). The core principle is to record all intended changes to a separate log before applying them to the database itself (see Figure 2.4). In the event of a crash, the system can be restored to a consistent state by replaying the log. Typically, this log is stored as a file on persistent storage. Each DBMS implements WAL to ensure crash-consistent transactions take place.

As explained by Lindsay et al. (2005), there are two main WAL mechanisms, redo

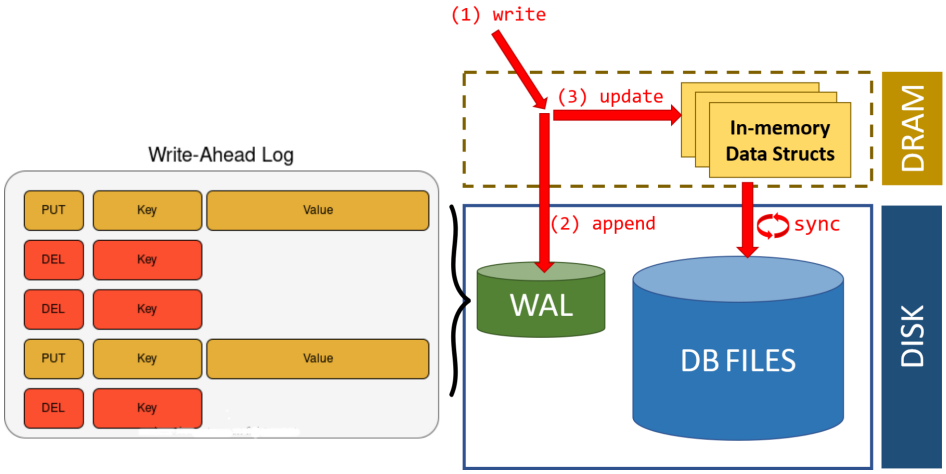


Figure 2.4: Database update process that uses Write-Ahead Logging (WAL) to guarantee crash-consistent updates (IntelCoporation (2020); Hagy (2020)).

and undo logging, that DBMS systems implement to ensure crash consistency. During recovery, the former re-applies transactions to persist new updates, while the latter rolls back transactions to revert partial (new) updates. Consequently, redo logs store a copy of the new update, which is reapplied during recovery to maintain durability. In contrast, undo logs record the old value (before the new update), enabling the system to roll back partial transactions and thereby ensure atomicity.

2.3.1 Crash Consistency on Persistent Memory

PM's crash consistency demands complex cooperation between application code and CPU caches for ordering write operations (Oliveira et al. (2025)). Since volatile caches are orders of magnitude faster than PM, caches are utilised to store frequently accessed data and to temporarily store writes to PM. Cache lines are evicted and written to PM based on predefined eviction policies. For instance, an eviction policy may only write-back cache lines when the cache is full. Data may reside in the caches (not persisted) on a crash, and it may reach PM in an order different from what the programmer desires (Oliveira et al. (2025)).

Unfortunately, an atomic write to PM can only be guaranteed for up to 8 bytes in a 64-bit CPU (Lee et al. (2019)). To guarantee atomicity of larger writes to PM, and ensure durability of writes, *clflush* instructions are used to move data from the cache to PM combined with WAL (Zhang et al. (2022)). However, a *clflush* alone is insufficient as *clflush* instructions can be reordered. To ensure that a write is persisted to PM before execution continues, an *mfence* instruction must be issued after *clflush* (Neal et al. (2020)).

Common Persistent Memory Bugs

The use of *clflush* and *mfence* instructions can lead to persistent memory bugs. Oliveira et al. (2025) reports the common PM errors made by developers.

Ordering bugs. These types of bugs stem from the reordering of writes done by modern CPUs. Let's consider when adding a key-value pair to a database. The developer first stores the value, and then the key. This ordering ensures that, in the event of a crash after the first operation takes place, the database will appear to have an empty slot. This is because the key is never written, and so, there is no pointer to the value. However, due to potential reordering of writes, the key might be persisted before the value. This results in a corrupt pointer, leading to an inconsistent database state.

Atomicity bugs. These bugs arise from the incorrect use of *clflush* and *mfence* instructions. Consider a program that swaps two objects stored in PM. This operation is intended to be atomic i.e., the two objects are swapped or they are not. To achieve this, the developer issues two *clflush* instructions (one for each PM object) followed by a single *mfence*. While the *mfence* ensures that the *clflush* instructions are completed before continuing execution, it does not guarantee that both writes are persisted atomically, as a single transaction. If a crash occurs between the two *clflush*s, only update of one object is persisted. This leaves the database in an inconsistent state, where both objects now point to the same value and the other object's value is lost. Thus, WAL should be used in conjunction.

2.4 SPIRIT

We discuss the state-of-the-art real-time, text-inversion search engine, SPIRIT (Hasnat and Akram (2025)). We begin with a high-level overview, then describe SPIRIT's memory organisation and the two core processes of search, indexing and query evaluation. Lastly, we discuss the execution modes of SPIRIT, dwelling specifically into the crash consistent recovery mode of SPIRIT.

2.4.1 Overview

SPIRIT: Scalable and Persistent In-Memory Indices for Real-Time Search is a real-time text-inversion search engine over hybrid memory (Hasnat and Akram (2025)). It leverages a two-tier hybrid memory architecture. DRAM is used for fast, primary storage, whereas PM serves as a larger, persistent secondary tier. Inverted indices are built across both tiers, with the most recent data held in DRAM. Inverted indices in DRAM are copied from DRAM to PM periodically. Query evaluation searches the indices on both DRAM and PM to serve user queries.

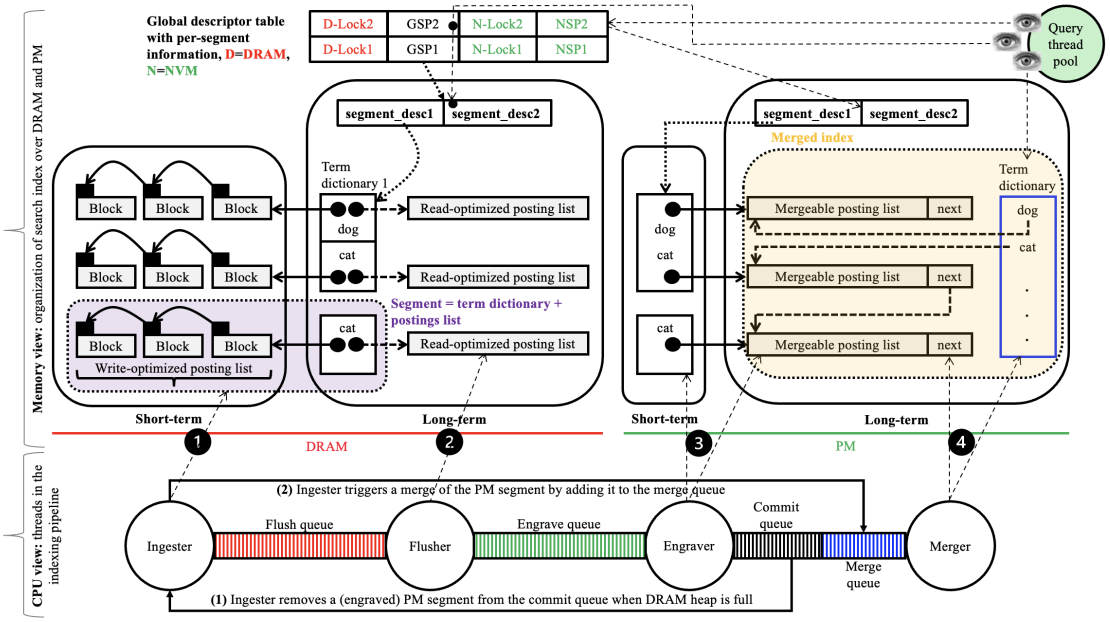


Figure 2.5: Overview of SPIRIT (Hasnat and Akram (2025)). The diagram shows a CPU and memory view of the system. This diagram is a slightly modified version of the one provided in Hasnat and Akram (2025).

2.4.2 Memory Organisation

SPIRIT divides both volatile (DRAM) and non-volatile (PM) memory into short-term and long-term heap regions (see Figure 2.5). The short-term DRAM heap stores posting lists as linked lists to optimise writes for new documents to be indexed. In SPIRIT, a segment is a self-contained inverted index, composed of a term dictionary and a posting list for each term. Once a segment is full, the posting lists are copied to long-term DRAM, and its short-term DRAM memory is reclaimed. In the long-term DRAM heap, posting lists of a segment are stored as contiguous arrays to optimise read performance. Thus, query evaluation threads only ever read from the long-term DRAM heap. Short-term PM is used to store the term dictionary of a segment, not its posting lists. Posting lists are written directly to the long-term PM heap when a segment is copied from DRAM to PM. This is because posting lists of a segment are not re-copied on merging, as merging occurs in place. During merging, the merged index on long-term PM is updated with new terms from the segment being merged. Existing terms have their posting lists extended through a sequence of pointer updates. After a successful merge, the space occupied by the segment's term dictionary on short-term PM is reclaimed. We use inverted index and merged index interchangeably.

2.4.3 Indexing

The indexing pipeline of SPIRIT consists of four stages: (1) ingest, (2) flush, (3) engrave, and (4) merge. Each stage in the indexing pipeline is handled by a dedicated thread (ingester, flusher, engraver and merger), with work passed between each thread via flush, engrave, commit, and merge queues. **Figure 2.5** shows the indexing pipeline of SPIRIT.

The ①**ingester** thread processes new documents in segment sizes. Once a segment is full, the ingester thread enqueues a task for the flusher thread via the flush queue. The ②**flusher** thread's role is to convert write-optimised posting lists into a read-optimised format. Subsequently, the ③**engraver** thread's role is to copy the read-optimised segment from DRAM onto PM. This process is started when the flusher thread enqueues a task via the engrave queue. Similarly, once a segment has been successfully engraved, the engraver thread enqueues a task onto the commit queue. Tasks on the commit queue are removed and immediately enqueued onto the merge queue in two scenarios: (1) when the ingester thread detects that DRAM space is running out, or (2) when the engraver thread has completed copying all segments in DRAM. Finally, the ④**merger** thread's job is to merge all engraved segments on the merge queue (one-by-one). SPIRIT performs an in-place merge, updating the merged index's term dictionary with new terms, and for existing terms, adding the new posting list to the existing list.

2.4.4 Query Evaluation

In SPIRIT, query evaluation is performed by dedicated threads that operate independently of the indexing pipeline threads discussed earlier. The Global Descriptor Table (GDT) grants query evaluation threads full visibility of the search index residing across DRAM and PM. Each GDT entry maintains per-segment metadata comprising of (1) a Global Segment Pointer (GSP) pointing to the descriptor of a DRAM or PM segment, (2) a NVM Segment Pointer (NSP) to an engraved but unmerged segment, (3) a lock to prevent concurrent updates to a DRAM segment during query evaluation, and (4) a lock for preventing concurrent access to segments (see Figure 2.5).

2.4.5 Execution Modes

SPIRIT supports three execution modes: (1) volatile, (2) graceful shutdown, and (3) crash consistent recovery mode. Modes (2) and (3) allow the search index to be recovered on restart, whereas this is not the case in volatile mode. Graceful shutdown recovery relies on the persistence guarantees of PM (i.e., data is retained on restart). Upon receiving a shutdown signal, ingestion is stopped, and all segments in DRAM are copied to PM and merged. Crash consistent recovery is more involved, as it requires persisting enough information to be able to recover from an unexpected crash at any point in time. We describe SPIRIT's crash consistency mechanism in detail next.

2.4.6 Crash Consistency

In the event of a system crash, SPIRIT's crash consistency is able to recover the index from both DRAM and PM. Since a crash can occur at any stage of SPIRIT's indexing pipeline, the crash consistency mechanism is designed to handle failures during three key phases: (1) while updating segments in DRAM, (2) while engraving segments from DRAM to PM, and (3) while merging segments. The mechanism combines WAL-redo logs with Intel 64-bit atomic writes to ensure reliable recovery. In the event of a crash, redo logs are replayed to re-execute updates. This ensures the system is not left in an in-between, corrupt state on restart.

To handle failures across the three key phases, SPIRIT maintains three separate redo logs: transaction log (*tLog*), engrave log (*eLog*), and merge log (*mLog*), all residing on PM. Both *tLog* and *eLog* are implemented as circular buffers, each with a head and tail pointer. In contrast, *mLog* is not circular. Its single entry is overwritten after a segment is successfully merged. The *tLog* captures documents ingested by the ingester thread, with each entry including a document ID. The *eLog* stores metadata, along with the old head pointer of *tLog*, to enable recovery of partially copied segments from DRAM to PM. Lastly, *mLog* records both segment-level and term-level information, enabling recovery of a partially merged segment.

Each log is updated and cleared by threads in the indexing pipeline. The ingester thread appends entries to *tLog* to indicate the ingestion of a new document in DRAM. This *tLog* entry is later cleared by the engraver thread by moving the *tLog* head pointer, indicating the engraving of that segment has started. The engraver thread, however, first creates an *eLog* entry for that segment before any pointer updates to *tLog*. An *eLog* entry can be cleared by the merger thread when an engraved segment has been merged. The merger thread is responsible for writing entries to *mLog*, and clearing them. We delay the full explanation of *mLog* till Chapter 4. Specific details of *tLog* and *eLog* are omitted as they fall outside the scope of this work.

Chapter 3

Related Work

In this chapter, we discuss prior work on hybrid memory search engines that leverage persistent memory (PM) to expand capacity, as well as work focused on achieving crash consistency on PM. We discuss verification frameworks, hardware optimisations, and software libraries developed for achieving crash consistency on PM. Not much prior work investigates adaptive real-time text inversion on hybrid DRAM-PM systems. We discuss prior literature on adaptive PM systems focusing on other applications.

3.1 Persistent Memory-backed Search Engines

Recent work has explored NVM-backed databases, key-value stores, and filesystems (Kannan et al. (2018); Kaiyrakhmet et al. (2019); Zheng et al. (2019)). However, there is limited work on leveraging the emerging storage technology PM for storing search indices. Akram (2021)'s work motivates rethinking incorporating PM hardware into the software stack for search engines. Their evaluation shows that directly constructing indices on PM reduces indexing throughput due to PM's higher latency and limited bandwidth. Nonetheless, they demonstrate that a PM-backed search engine, where the index is first built in DRAM, offers a promising approach. The most recent and relevant work, by Hasnat and Akram (2025), builds upon the findings of Akram (2021) and proposes a real-time text-inversion search engine, SPIRIT. SPIRIT uses both DRAM and PM to construct an inverted index across the two storage technologies. Our work specifically focuses on SPIRIT's open research problems to optimise its crash consistency and introduce adaptivity into the PM-based search engine.

3.2 Crash Consistency on Persistent Memory

3.2.1 Verification Frameworks

PM is a transformative storage technology, however, ensuring crash consistency on PM requires developers to have a deep understanding of CPU caches and memory interactions. Missing or misordered *clflush* and *mfence* instructions can result in PM bugs that go undetected (Oliveira et al. (2025)). To reduce this burden on developers, frameworks (Jiao et al. (2025); LeBlanc et al. (2025)) have been proposed to verify crash consistency guarantees.

Silhouette is a bug detection framework for PM filesystems (Jiao et al. (2025)). It operates on execution traces of the filesystem that capture stores to PM and persist instructions, such as *clflush* and *mfence*. From these traces, it extracts invariants (e.g., "Store 1 must occur before Store 2") and simulates crash scenarios to verify that the invariants hold and data is correctly updated. However, false positives may occur, when data mismatches arise from unprotected stores that do not affect the crash consistency guarantees.

In contrast, the formal verification framework *PoWER* extends Hoare logic with crash conditions (LeBlanc et al. (2025)). Unlike prior work, PoWER does not rely on specialised tools or external frameworks. It provides a write API in which each store must be accompanied by a pre- and post-condition. The precondition specifies the system state before a write, such as $key = oldValue$ in a key-value store. This forces developers to reason about all possible crash states before performing subsequent writes.

3.2.2 Hardware Optimisations

Crash consistency on PM presents an opportunity to rethink traditional methods for ensuring crash consistency. Traditional methods, such as write-ahead logging (WAL), rely on using persist instructions that incur high overhead (Wu et al. (2019)). To address this, *Hercules* introduces a hardware-assisted approach that leverages an eADR-supported CPU cache technology (Ye et al. (2024)). In this design, cache lines act as transient recovery logs, storing new updates. The eADR-supported hardware automatically flushes cache lines to an ADR-protected domain upon a crash. Upon recovery, the persisted recovery logs can be replayed from the ADR-protected domain. This eliminates the need for explicitly persisting recovery data to PM using high-overhead persist instructions.

3.2.3 Software Libraries

Hardware technologies such as eADR, which optimise crash consistency on PM, are costly and not commonly available in most systems (Oliveira et al. (2025)). As a result, many systems instead rely on software libraries designed to achieve crash consistency on PM. Intel's PMDK (Intel) library abstracts the intricacies of persisting data to PM. It offers a set of API calls that allow users to interface with PM. For example, *persist()*

that internally issues *clflush* and *mfence* instructions to ensure that a write reaches PM before continuing execution. However, ensuring the use and correct ordering of persist() calls is left up to the developer. PMDK does not provide built-in checks for preventing common (PM) developer errors.

The NV-Heaps (Coburn et al. (2011)) library includes explicit checks. It is a library that offers an allocator for PM configured in direct access (DAX) mode. This mode bypasses the kernel to provide fast, byte-addressable access to PM. NV-Heaps perform invariant checks on recovery to detect any errors. An example of an invariant check is that all non-null pointers in the heap must point to valid objects. Corundum (Hoseinzadeh and Swanson (2021)) is a Rust-based library, and is designed on similar principles to NV-Heaps to prevent common PM errors. However, both libraries internally rely on high-overhead *clflush* and *mfence* instructions to persist recovery data. Pronto (Memaripour et al. (2020)) introduces Asynchronous Semantic Logging (ASL) to move the overhead of persist instructions off the critical path. This reduction in overhead is achieved via a background thread for logging, which runs alongside the foreground thread that updates volatile data. The background thread synchronises with the foreground thread at the end to mark the operation as committed. This synchronisation introduces overhead, particularly when the foreground thread completes early. Therefore, our work focuses on reducing the total number of high-overhead persist instructions in SPIRIT's crash consistency, rather than introducing a background thread for logging.

3.3 Adaptive Persistent Memory Systems

Dapper is an adaptive management system for PM in non-uniform memory access (NUMA) servers (Xue et al. (2019)). In NUMA architecture, each server has its own local memory, and accessing memory on a remote server incurs higher latency. It is used in multi-server systems to provide scalable memory bandwidth. Dapper dynamically allocates PM within each server, adjusting memory capacity to match the workload demand. On the other hand, *PMSort* is an adaptive sorting engine for organising data on PM, that chooses the most suitable sorting algorithm based on the workload characteristics (Hua et al. (2021)). However, there is no prior literature on PM-backed search engines that can dynamically adapt.

Chapter 4

Crash Consistency Optimisation

In this chapter, we introduce an optimised crash consistency mechanism for SPIRIT, designed to reduce merge time. We minimise the number of high overhead, *clflush* and *mfence* instructions issued for every term merged, at the cost of increased PM usage. We begin by discussing the sources of overhead in the original crash consistency. We then present the optimised approach, comparing and contrasting the new merge log (*mLog*) structure with the original one. We describe the most relevant *mLog* fields, outlining their roles in the original mechanism and in the optimised one (if applicable). We conclude with a discussion of the optimised mechanism's recovery and the key takeaway.

4.1 Sources of Overhead

Hasnat and Akram (2025)'s results show that, in crash consistent recovery mode, merge time is 19× slower than in no-recovery (volatile) mode, and 34× slower compared to other stages of SPIRIT's indexing pipeline. We believe this overhead stems from several updates to *mLog* made using high overhead, *clflush* and *mfence* instructions. One such example is *mLog*'s *termCommit* field, discussed in detail later. The value of *termCommit* is updated three times per term merged: (1) to indicate the start of the merge, (2) to indicate in-between merge, and (3) to indicate the end of the merge. To guarantee successful recovery, each update to *termCommit* is persisted (on PM) before execution proceeds. This means high overhead, *clflush* and *mfence* instructions are issued after each of the three updates, for every term that is merged.

4.2 Structure of Merge Log

Figure 4.1 compares the original and optimised *mLog* structure. Logically, the *mLog* fields can be grouped into two categories: (1) per-segment fields and (2) per-term fields. Per-segment fields are written to PM once for each segment merged, whereas per-term

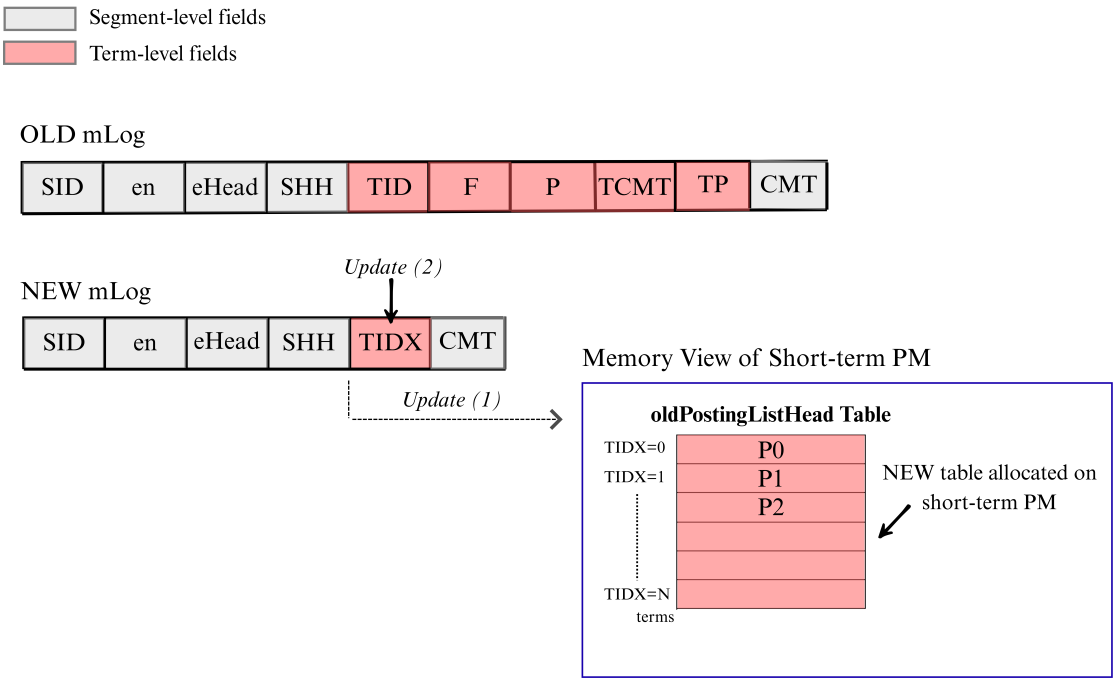


Figure 4.1: Comparison of the *mLog* structure and update process for the optimised crash consistency, with the original (old) *mLog* structure.

fields (highlighted in red in Figure 4.1) are persisted for every term merged within a segment. This design enables recovery to resume from the last successfully merged term instead of re-merging the entire segment. We now discuss each category, focusing primarily on term-level fields, as they are key to the optimisation. We say persist barrier to mean the combination of *clflush* and *mfence*.

4.2.1 Segment-Level Fields

The role of each segment-level *mLog* field is as follows.

- *SID.* The segment ID of the segment being merged.
- *en.* A pointer to the engraved state information of the segment being merged. This includes a pointer to the term dictionary, on short-term PM, of the segment.
- *eHead.* A copy of the previous *eLog* head pointer to ensure that, in the event of a crash, the *eLog* entry for the segment can be recovered.
- *SHH.* A pointer to the oldest allocation in the short-term PM heap.
- *CMT.* The state of the merge upon crash. A value of 0 denotes the start of a segment merge (no terms merged), 1 indicates an in-progress merge, and 2 indicates a successfully merged segment. This field is used during recovery to determine if, and how, merging of a segment should be completed.

In the optimised crash consistency, all segment-level fields remain part of *mLog*, and their role remains unchanged.

4.2.2 Term-Level Fields

TCMT

The *termCommit* (*TCMT*) field is an integer that can take values 0, 1, or 2 to determine the point in time of crash upon recovery. It is analogous to *CMT* field. A value of 0 indicates that the crash occurred before merging a term. A value of 1 indicates a crash occurred while merging a term. A value of 2 indicates that the crash occurred after merging a term. For every term merged, *termCommit* must be updated from $0 \rightarrow 1 \rightarrow 2$. Original implementation combines *clflush* and *mfence* instructions to guarantee each *termCommit* update is persisted to PM before continuing execution. Thus, for every term merged, a total of three persist barriers are issued.

These persist barriers are necessary to make assumptions about the possible state of the database on recovery, especially if the database could be corrupted. If *termCommit* is 1, there is a possibility, the crash occurred while updating the database, leaving it in a corrupt state. Therefore, the field *P* (discussed later) must be used to redo the merge and bring the database to a consistent state. Whereas, this is not necessary when *termCommit* is 0 or 2 on recovery.

In the optimised crash consistency, *termCommit* (*TCMT*) is **removed** and replaced with *termIdx* (*TIDX*). This is to eliminate 2 out of the 3 persist barriers issued per term. *TIDX* by itself is sufficient to indicate whether a crash could leave the database in a corrupt state. Explanation of *TIDX* is delayed until its respective section.

TID

termId (*TID*) stores the index of the term currently being merged in the segment. By using *termId*, recovery can resume from the term after the last successfully merged term, avoiding the need to re-merge the entire segment.

In the optimised crash consistency, *termId* (*TID*) is **removed** and replaced with *TIDX*. *TID* is persisted using a *clflush* instruction, which we eliminate from removing *TID*. *TIDX* by itself is sufficient to track where to resume the merge from. This is elaborated on in the *TIDX* section.

F

The *termFound* (*F*) indicates whether a term already exists in the merged index (*F*=1) or not (*F*=0). It is persisted with a persist barrier before setting $TCMT = 1$. Thus, we can guarantee that if $TCMT = 1$, then the *termFound* field is valid for that term.

In the optimised crash consistency, the *termFound* (*F*) field is **removed**. This eliminates another persist barrier issued per term. Whether a term exists in the merged index is now implicitly encoded in the *P* field. If the term is not found, *P* is set to NULL, otherwise, it is a valid memory pointer.

TP

termPosition (*TP*) indicates the term's position in the merged index. This field is only set when a term already exists in the index.

In the optimised crash consistency, the *termPosition* (*TP*) is **removed**. We observe that a term's position in the index is deterministic and can be recomputed during recovery. It is important to note that the merged index hash table size must be the same on recovery.

TIDX

termIdx (*TIDX*) is a new field that is introduced. It stores the index of the current term being merged in the segment. The key distinction from *TID* is that *termIdx* refers specifically to the index among non-empty terms in the segment. This distinction is essential for correctly retrieving the corresponding *P* field during recovery (discussed in the *P* section). Briefly, the *P* field is a pointer to the head of a linked list of posting lists, for a term in the merged index. It must be reliably persisted before a database update takes place for a term, to ensure we can recover from a corrupted state.

termIdx is set to -1 at the start of merging a new segment, and to -2 upon completion of a merge. These are two special cases. We need a way to determine whether a valid P field has been logged for the first term, corresponding to $termIdx = 0$. If *termIdx* is set to -1, it indicates that merging has not yet begun (database not updated), and the first term can be merged safely on recovery. Similarly, $termIdx$ = -2 confirms that the merge of the last term was completed. This prevents the recovery process from re-merging terms after the de-allocation of segment data has begun. These two cases must be handled explicitly, since *TCMT* and *TID* are no longer available to indicate this information. To be specific, $TCMT = 1$ and $TID = 0$, indicated a valid P field had been logged for the first term on crash.

To ensure that *termIdx* points to the P field of the correct term being merged, it is important to log *termIdx* and P at an appropriate point. A key aspect of the original crash consistency is that a *TCMT* value of 1 indicates that the P field is valid for the term being merged, and can be used to recover from a corrupt database state. Our design follows the same principle, in which *termIdx* is updated and persisted, only after the term's corresponding P field has been persisted. It is important to note that P is set to NULL if the term is not found in the merged index.

Logging *termIdx* and P require a persist barrier each. A persist barrier for the P field ensures that the pointer is persisted before updating *termIdx*. Without this barrier, *termIdx* can be updated before P, and the new *termIdx* could now be pointing to an invalid P. A persist barrier after *termIdx* ensures that, if a crash occurs after updating *termIdx*, the previously merged term is not re-merged.

Since *TCMT* is no longer present, if a crash occurs before updating *termIdx*, we can only be certain that the persisted P field belongs to the previously merged term. Consequently, if that term was already merged, it will be re-merged during recovery. However, this re-merge does not affect the correctness of the database state. Either the term and its posting list are re-copied, or if the term already exists, the pointer is updated using the logged P field, preventing circular references.

P

The *oldPostingListHead* (P) field plays a critical role in recovering from unexpected crashes and must be persisted reliably. The *oldPostingListHead* is a pointer to the head of a linked list of posting lists, for a given term in the merged index. If a crash occurs during a term merge (of a term that already exists) the database update to link the posting lists may be left in an inconsistent state. This is because an atomic update cannot be guaranteed for a two-step pointer update of (1) updating the head pointer of the term's entry in the merged index, and (2) linking the new posting list with existing posting lists. A crash occurring after (1) but before (2) could result in losing reference to the existing posting lists. Therefore, to ensure correctness, *oldPostingListHead* is persisted before a database update, with a persist barrier. It is restored on recovery when a crash occurs while a term merge was in progress. Note that this recovery step

applies only to terms that already exist in the merged index. In the case that the term does not exist, merging does not update the *oldPostingListHead* field. Instead, we rely on the F (found) field to decide whether the *oldPostingListHead* field is read during recovery.

In the optimised crash consistency, *oldPostingListHead* pointer (P) has the same purpose, however, it is not stored as a single entry that is overwritten on every term merge. In the previous design, the *TCMT* field indicated the crash point. This allowed us to determine whether the *oldPostingListHead* entry was valid for a given term. For example, if a crash occurred after a successful merge (but before the start of the next merge), i.e. $TCMT = 2$, we knew the *oldPostingListHead* had not been updated for the next term. However, in the new design, this information is no longer available. A crash could now occur after setting the *oldPostingListHead*, but before updating the *TIDX* to reflect the next term. This leads to an ambiguous state where the *oldPostingListHead* corresponds to the next term, while the *TIDX* still refers to the previously merged term. On recovery, there is no way to determine whether the pointer actually corresponds to the current *TIDX*. Thus, the new design must reconsider the method for storing *oldPostingListHead* pointers.

oldPostingListHead pointers are now stored in a table on PM, replacing the previous approach of writing them as part of *mLog* (see Figure 4.1). The table is allocated on short-term PM at engraving time, and its space is reclaimed once the segment merge is complete. The table only allocates entries for non-empty terms in the segment to optimise PM usage. A pointer to the table is passed as part of the engraved state information (*en*) of a segment, via the commit/merge queue. At merge time, persisting an *oldPostingListHead* pointer is a two-step process. (1) Indexing into the table using the *TIDX* field to locate the term's entry for logging the pointer. (2) Issuing a persist barrier to ensure the pointer is persisted to the table (on PM) before any database update. In the original crash consistency, *oldPostingListHead* pointer is written to *mLog* for every term found in the merged index. Thus, redirecting these writes to a separate table should not impact overall performance. However, there is an increase in PM usage due to a separate table entry allocated per term to store P.

4.3 Crash Recovery

On recovery, segment merge is restarted from *TIDX* assuming $CMT = 1$. There is no need to explicitly distinguish between partially merged terms and those at the start or end of a merge. Instead, the last persisted *TIDX* is treated as the (next) term to merge, with a valid P field recorded in the table. Recovery then uses *TIDX* to index into the *oldPostingListHead* table and do the merge for that term. Whether the term was found in the index (originally) is determined from its P field. The trade-off of removing the *TCMT* field is the possibility of re-merging a term that has been already merged, but this is an acceptable trade-off to reduce the merge time. The remaining terms in the segment are merged starting from *TIDX*. As in the original mechanism,

crash consistency during recovery is ensured by writing to *mLog*. Once the segment merge completes, the *oldPostingListHead* table's memory is reclaimed. Special handling is required when *TIDX* = -2, which indicates that the segment merge was completed and the *oldPostingListHead* table memory may have been reclaimed. This prevents the recovery process from attempting to access freed memory.

4.4 Key Takeaway

The optimised crash consistency eliminates a total of 3 persist barriers (out of 5) and 1 *clflush* instruction, for every term that is merged.

Chapter 5

Experimental Methodology

We now describe the experimental methodology used across four key aspects of this thesis. (1) Evaluation of the optimised crash consistency mechanism. (2) Analysis of PM pressure, motivating the design of ARITHMETY. (3) Evaluation of ARITHMETY. (4) Sensitivity analysis of ARITHMETY's parameters. We begin by describing the evaluation platform, common datasets, and configurations, followed by the specific methodology and measurement metrics for all four.

5.1 Platform, Datasets, and Configurations

Evaluation Platform. For all experiments in this thesis, we use one of the sockets of a server with two CPU sockets. Our server has two datacenter-grade Intel Xeon (6252N) processors with each each CPU running at the peak frequency of 2.30 GHz. Our servers run Ubuntu 20.04.4 Linux OS (5.4.0 kernel). Each CPU has 24 physical cores and 48 logical cores (two threads per core), and 96 cores across the two CPUs. Our server has 12 32 GB DIMMs and 12 128 GB Intel Optane NVDIMMs. We use Intel Optane PM in App Direct mode so as to bypass the page cache and expose it as byte-addressable memory to SPIRIT. We disable NUMA for predictable execution.

Ingestion Datasets. The ingestion datasets are derived from Wikipedia's English corpus (McCandless (2021)). From the corpus file, we generate multiple datasets varying in the number of documents. Each document is limited to 1 KB in size and is line-terminated. Specifically, all our experiments consider a subset of ingestion datasets of 1M, 5M, 10M, and 25M documents. The specific ingestion datasets used are detailed in the respective methodology sections.

Querying Datasets. We use 50K terms from topTerms20120502.txt (McCandless (2021)). The terms vary in frequency and are categorised into three types: (1) low-frequency (L), (2) medium-frequency (M), and (3) high-frequency (H). The distribution

of terms is as follows:

L:	49K
M:	< 1K
H:	100

We form two types of queries: (1) single-term queries, and (2) two-term queries. Two-term queries are evaluated as AND queries. We generate 6 query workloads: low-frequency, single-term (L), medium-frequency, single-term (M), high-frequency, single-term (H), low-frequency, two-term (LL), medium-frequency, two-term (MM), and high-frequency, two-term (HH). Each query workload consists of 5M queries. Terms may appear multiple times, particularly in the H and HH query workloads. This reflects real-world behaviour, where users are likely to query the same term multiple times.

Default Parameters and Settings. Unless otherwise specified, we use the following default parameters and settings of SPIRIT. SPIRIT is compiled in crash consistency recovery mode, with a path to the recovery file on PM provided. SPIRIT supports multiple I/O options. In our setup, we use file I/O to provide indexing and querying workloads. All indexing and querying workloads are on a Western Digital HDD (15 TB total capacity). We configure SPIRIT with the settings specified in Table 5.1, unless otherwise specified. We run SPIRIT in four modes:

- *Indexing-only.* In this mode, an indexing workload is specified, and the index is built on PM by providing the path to a directory on PM. Recovery is not attempted, only recovery data is written to a recovery file on PM. No querying workload is run.
- *Querying-only.* In this mode, no indexing workload is specified. Instead, a path to a prebuilt index on PM (to query) is provided. Recovery is first attempted by reading the recovery file on PM associated with the prebuilt index. The querying workload that is specified, runs after recovery. By default, a single query thread evaluates queries sequentially.
- *Recovery-only.* In this mode, no indexing workload is specified. Instead, a path to an index on PM (to recover) is provided. Recovery is attempted by reading the recovery file on PM associated with the index to recover. No querying workload is run after.
- *Concurrent.* In this mode, an indexing workload is specified, the index is built on PM, and a recovery file is written to while indexing. The querying workload specified runs alongside indexing. By default, a single query thread evaluates queries sequentially.

5.2 Crash Consistency Optimisation

Configurations for Evaluation. We compile and run the optimised and unoptimised (state-of-the-art) version of SPIRIT under the same experimental configurations. We

Table 5.1: Default SPIRIT configurations.

Configuration Option	**Default**
Short-term DRAM	1 GB
Long-term DRAM	5 GB
Short-term PM	5 GB
Long-term PM	20 GB
Single segment size	128 MB
Maximum segments	1000
Merged segment hash table size	128 MB for 1M, 512 MB for 5M, 1024 MB for 10M, 3024 MB for 25M
tlog size	Scaled to # documents to index for an indexing workload
elog size	Scaled to total # segments (reported by SPIRIT) for an indexing workload

run 4 trials for each experimental configuration to reduce noise in results.

To evaluate the overhead on indexing time, we run SPIRIT in *Indexing-only* mode with indexing workloads of 1M, 5M, and 10M documents. We configure SPIRIT with the settings specified in Table 5.1.

To measure query throughput, we run SPIRIT in *Concurrent* mode. This is to assess the impact of indexing and writing recovery data to PM, on query performance. We run all six querying workloads, as separate experiments, recording the queries per second (QPS) for each. We fix the indexing workload to 1M. To avoid empty query results at the start (before sufficient documents are ingested), querying begins after 300K documents have been ingested. We configure SPIRIT with the settings specified in Table 5.1. With the **exception** of long-term DRAM. We experiment with the following configurations of long-term DRAM.

- *Tight.* 0.5 GB
- *Medium.* 2 GB
- *Loose.* 4 GB

To measure recovery time, we first run SPIRIT in *Indexing-only* mode, crash the SPIRIT instance, and then run it in *Recovery-only* mode. We avoid crashing SPIRIT at random intervals and attempting recovery. Since inconsistent crash points lead to different recovery times, and cannot be compared directly. We try to mitigate this by crashing SPIRIT at fixed ingestion timelines: (1) 30%, (2) 50%, (3) 70%, and (4) 100% of 1M documents have been ingested.

Measurement Metrics. To measure the overhead on indexing time of SPIRIT's crash consistency recovery mode, we use both the total indexing time and the per-stage in-

dexing pipeline times reported by SPIRIT. We measure query throughput in queries per second (QPS). To measure recovery time, we use the recovery time reported by SPIRIT when SPIRIT is run in *Recovery-only* mode.

5.3 PM Pressure Analysis

Configurations for Evaluation. We use the optimised crash consistency version of SPIRIT to ensure there is minimal overhead when it comes to indexing. We compile and run a further modified version of SPIRIT. SPIRIT is modified to not ingest any stopwords to reduce the index size (discussed later). The list of terms that are classified as stopwords is from NLTK's list of English stopwords (Bleier (2010)).

To design a workload that simulates pressure on PM, we mix indexing (writes to PM) and query evaluation (reads to PM) together. We avoid running SPIRIT in *Concurrent* mode. This is to avoid query evaluation becoming slower later in execution, as more documents are indexed. This approach does not allow us to clearly assess the impact on query performance, whether it is caused by increased PM pressure or the execution time of the query. We run SPIRIT in *Indexing-only* mode to build an index on PM, and then run it in *Querying-only* mode to query it. We simulate increased pressure in the system by running additional SPIRIT instances (at the same time) in *Indexing-only* mode. Each builds a separate index on PM, which is possible with a smaller index size (after removing stopwords). We fix the indexing workload to 25M, including the index to query. 25M workload ensures indexing and querying do not complete instantly, and can be run concurrently. We configure SPIRIT with the settings specified in Table 5.1. With the **exception** of the following:

- Long-term DRAM: 128 MB, to force reads/writes to PM for *Indexing-only* instances, simulating pressure on PM.
- Long-term PM: 22 GB, scaled up to run 25M workload.
- Maximum segments: 2000, scaled up to run 25M workload.

We regenerate the querying workloads, H and HH, replacing any queries with stopwords with non-stopword queries. This is to be consistent with SPIRIT not indexing stopwords and keep H and HH as high-frequency query workloads. The total number of queries to evaluate per query type is summarised in Table 5.2. We size the query workload to complete within a reasonable time, and finish with the indexing workload to fully evaluate mixing reads and writes.

The final set of experiments run for the analysis is summarised in Table 5.3. Each experiment is run 4 times to reduce noise in results. A SPIRIT instance always runs in *Querying-only* mode. The number of queries to evaluate, per query thread, is kept constant (see Table 5.2 for the number of queries per thread). To clarify, we run the six querying workloads one after another, as separate experiments.

Table 5.2: Number of queries evaluated per query type and per query thread.

Query Type	Number of Queries
L	450000
M	140000
H	100000
LL	240000
MM	70000
HH	30000

Table 5.3: Experiment combinations for analysis of PM pressure.

Query Threads	Number of Indexing Instances					
	0	1	4	8	16	32
1	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH
4	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH
8	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH
16	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH
32	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH	L, M, H, LL, MM, HH

Measurement Metrics. We use the average per-query latency reported by SPIRIT to quantify read performance. When multiple query threads are running, we take the average of the average per-query latency reported by each. In conjunction, we use PM read and write bandwidth derived from Intel Xeon CPU performance counters. We use Intel's Performance Counter Monitor (PCM) command-line utility to monitor the PM bandwidth (Intel (2025)). We record PM read and write bandwidth every second to accurately reflect the system's behaviour. We report the average read and write bandwidth over the duration of the SPIRIT instance, from the start of querying to the end. Together, these metrics allow us to analyse the pressure on PM.

5.4 ARITHMETY Evaluation

Implementation. ARITHMETY is implemented in C/C++. We use the POSIX sockets API from the standard C library for client-server interactions. We use the cJSON library for de-serialisation and serialisation, the Pthreads library for multithreading, and a concurrent queue from oneAPI for thread-safe data access.

Workload Generation. We design four workloads to simulate different levels of pressure on PM in the system. These workloads emulate real-world scenarios and evaluate ARITHMETY's ability to adapt based on client priorities. For each workload, we run 4 trials to reduce noise in results. The workloads are as follows.

- Concurrent-indexing (CI) and single-threaded-querying (ST) — CIST. In this workload, we progressively increase pressure on PM by introducing additional reads and writes in the system. We run one SPIRIT instance in *Querying-only*

mode and one in *Indexing-only* mode. To increase system pressure, the number of *Indexing-only* instances are increased from 1 → 4 → 8 → 16. The indexing workload is fixed at 25M, and the querying workload consists only of HH queries – the slowest query type identified in Chapter 6. This setup allows us to evaluate ARITHMETY under conditions of high query latency. A single query thread evaluates 45K queries. A querying workload of 45K HH queries is generated to run concurrently up to 16 *Indexing-only* instances.

- Concurrent-indexing (CI) and multi-threaded-querying (MT) — CIMT. This workload is identical to CIST, except that 32 query threads evaluate queries in parallel. The number of queries to evaluate per thread remains constant. 45K queries are evaluated per thread with a total of 1,440,000 queries.
- Query-type (QT) and single-threaded-querying (ST) — QTST. In this workload, we vary the pressure on PM by switching between fast and slow query types while mixing reads and writes from concurrent indexing and querying. One SPIRIT instance runs in *Querying-only* mode, alongside eight *Indexing-only* instances. The indexing workload is fixed at 25M. The querying workload consists of H and HH queries. The distribution and ordering of H and HH queries for a 30K-query workload, is as follows, given a 1:1:2 ratio:

  (1) H: 7.5K
  (2) HH: 7.5K
  (3) H: 15K

- Query-type (QT) and multi-threaded-querying (MT) — QTMT. This workload is identical to QTST, except that 32 query threads evaluate queries in parallel. 30K queries (with above distribution) are evaluated per thread with a total of 960,000 queries.

Configurations for Evaluation. We compile and run ARITHMETY, which consists of the server application (providing the controller service) and SPIRIT (modified to interact with the server), running concurrently.

We run the server application under two configurations as follows.

- *Adaptive.* In this mode, the client priority specified is fast query response.
- *Baseline.* In this mode, the client priority specified is strong crash consistency.

We configure SPIRIT with the settings specified in Table 5.1, with **exceptions** listed in Section 5.3 to run a 25M indexing workload.

We summarise the parameter values used for evaluating ARITHMETY under both the *Adaptive* and *Baseline* configurations in Table 5.4. The two threshold parameters are chosen based on our sensitivity analysis results in Chapter 8. We vary the moving window size and select a size of 3 for evaluation, as we find it has a negligible effect on the system's ability to respond to variations in PM pressure. The profiling frequency and duration

Table 5.4: Parameter values used for evaluating ARITHMETY.

Parameter	**Value**
Moving average window size	3
Circular latency buffer size	50
Reaction threshold (query latency)	40 ms
Positive gradient threshold (for drift detection)	1×10^{-4}
Profiling frequency	3 secs
Profiling duration	30 secs

are set to 3 seconds and 30 seconds, respectively, as these values are found suitable for our workloads. Likewise, we use a circular latency buffer size of 50. Automatically determining the correct parameters for arbitrary workloads is left for future work.

Measurement Metrics. We use the average per-query latency and individual per-query latencies reported by SPIRIT to understand the latency distribution over time and quantify read performance.

5.5 Sensitivity Analysis

Configurations for Evaluation. We compile and run ARITHMETY in *Adaptive*, defined in Section 5.4. To perform sensitivity analysis for ARITHMETY's parameters: reaction threshold (query latency) and positive gradient threshold for drift detection, we run the CIST workload defined in Section 5.4. We select these two parameters as they are crucial for drift detection and to trigger controller action to mitigate PM pressure. We configure ARITHMETY with the parameter values specified in Table 5.4, with the **exception** of the reaction threshold and the positive gradient threshold, see below.

We conduct the following sensitivity analysis experiments. The server application is relaunched for every new threshold value to test, and the same threshold experiment is run 4 times to reduce noise in results.

- *Reaction-threshold-varied* (RV). In this experiment, we vary the reaction threshold as summarised in Table 5.5. Thresholds are selected having insight from Chapter 6. We bound the reaction threshold to 30 ms to evaluate ARITHMETY's behaviour, when the threshold is below the minimum average latency of HH queries. The positive gradient threshold is fixed to 1×10^{-4} as in Table 5.4.
- *Gradient-threshold-varied* (GV). In this experiment, we vary the positive gradient threshold for detecting an increasing trend in query latencies, as part of drift detection. The thresholds tested are summarised in Table 5.6. The reaction threshold

Table 5.5: Reaction thresholds

Reaction threshold (query latency)	**Value (ms)**
tr1	30
tr2	40
tr3	50
tr4	60

Table 5.6: Positive gradient thresholds

Positive gradient threshold	**Value (% of gradient of 1)**
tg1	1×10^{-4} (0.01%)
tg2	1×10^{-3} (0.1%)
tg3	1×10^{-2} (1%)
tg4	1×10^{-1} (10%)

is fixed to 40 ms as in Table 5.4.

Measurement Metrics. We use the average per-query latency and individual per-query latencies reported by SPIRIT to understand the latency distribution over time and quantify read performance.

Chapter 6

Analysis of Pressure on Persistent Memory

In this chapter, we analyse the pressure on PM during SPIRIT's key operations of indexing and query evaluation. This analysis motivates the design of our adaptive system, ARITHMETY, described in Chapter 7. We execute SPIRIT in two modes: (1) concurrent execution (indexing and query evaluation together) and (2) isolated execution (indexing-only or querying-only). Indexing-only, or also referred to as an indexer is a SPIRIT instance that is exclusively constructing an index of 25M documents (i.e., a 25M indexing workload). Whereas, querying-only is a SPIRIT instance that is strictly serving user queries by searching a pre-built (25M documents) index. Our analysis focuses on two complementary metrics: (1) a software-level metric – average per-query latency reported by SPIRIT, and (2) a hardware-level metric – PM read and write bandwidth obtained from hardware performance counters. We use PM read and write bandwidth to gain a clearer understanding of PM pressure. We begin by analysing the impact of query type, followed by the impact of concurrent indexing on per-query latency. We finish by summarising the key findings. The impact of multi-threaded querying (i.e., multiple query evaluation threads) is discussed throughout the chapter.

6.1 Impact of Query Type

The querying workload can be categorised based on (1) term frequency and (2) whether queries consist of one or two terms. The term frequency is classified as low, medium, or high, depending on how often a term appears across the documents indexed. We present results for six query types: low-frequency single-term (L), medium-frequency single-term (M), high-frequency single-term (H), low-frequency two-term (LL), medium-frequency two-term (MM), and high-frequency two-term (HH). We first present results for the isolated execution mode, where a single querying-only SPIRIT instance runs with no

other concurrent indexing-only instances. We discuss results for a single query evaluation thread experiment before discussing multi-threaded querying. Figure 6.1 compares the average per-query latency when running 1, 4, 8, 16 or 32 query evaluation threads. Figure 6.2 shows the average per-query latency results from Figure 6.1, normalised to 1 query evaluation thread.

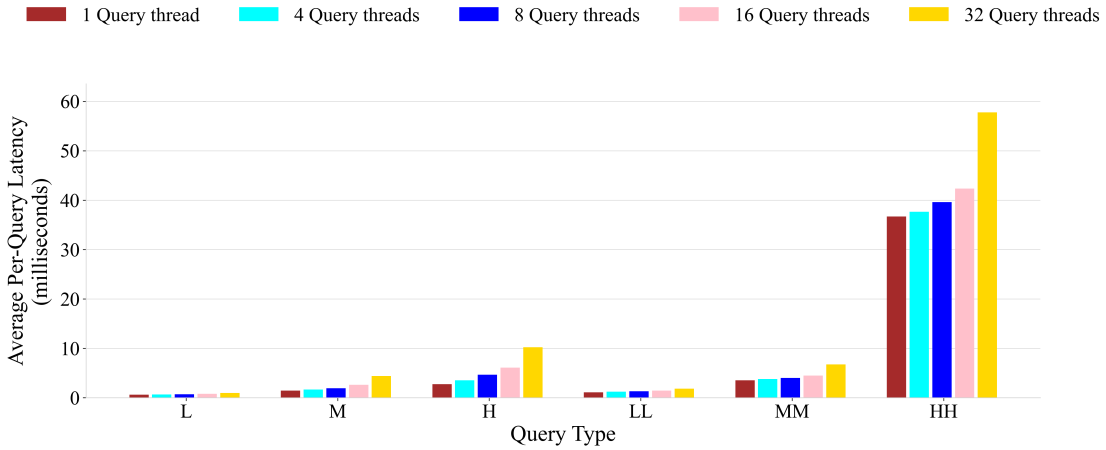


Figure 6.1: Average per-query latency compared with 1, 4, 8, 16, and 32 query evaluation threads, across six query types. Experiment conducted in isolated execution mode, with a single querying-only SPIRIT instance.

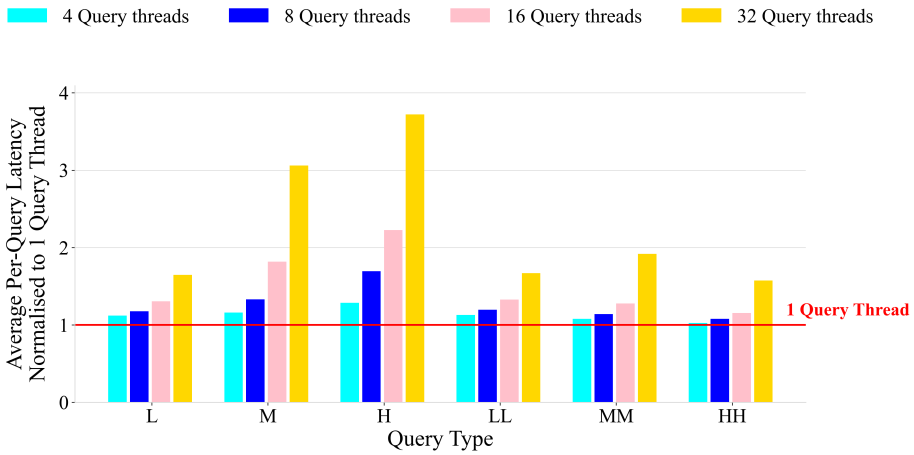


Figure 6.2: Average per-query latency, normalized to 1 query thread, compared with 4, 8, 16, and 32 query evaluation threads, across six query types. Experiment conducted in isolated execution mode, with a single querying-only SPIRIT instance.

Figure 6.1 shows that HH queries are the slowest, with an average per-query latency of 37 ms with one query thread. On the other hand, L queries are the fastest, with an

average per-query latency of 0.6 ms, making them 61.7× faster than HH queries. We expect low-frequency single-term (L) queries to be evaluated more quickly because the merged index either contains the term with a small-sized posting list to copy or does not contain the term at all. The latter only constitutes a single, constant-time hashtable lookup of the merged index. Evaluating a single-term query terminates once the term is found and its posting list has been copied. In contrast, high-frequency two-term (HH) queries require the query evaluator to search posting lists to identify documents containing ALL query terms. This is because two-term queries are evaluated as AND queries. The evaluation step incurs an $O(n)$ cost, where n is the length of the smaller posting list among the two terms. The evaluation time is further affected because both terms are high-frequency, resulting in longer posting lists to be traversed.

6.1.1 Impact of Multi-threaded Querying

We now discuss the impact of multi-threaded querying. We draw our attention to Figure 6.2, which provides a clear comparison between the latencies of 4, 8, 16 and 32 query threads versus 1 query thread. The overall trend across all query types is that the average per-query latency increases as the number of query evaluation threads increases. Several factors may contribute to this, including (1) synchronisation overhead caused by multiple query threads reading from the same input queries file, (2) synchronisation overhead from writing per-query latencies to a shared output file, and (3) increased utilisation of PM bandwidth. We explore increased utilisation of bandwidth in detail in Section 6.2.

For 4 and 32 query threads, we observe that H queries incur an increase in latency of 1.29× and 3.72×, respectively, compared to 1 query thread. Whereas, for HH queries, the increase in latency is 1.03× and 1.57×, respectively. Figure 6.2 shows that HH queries are the least impacted by multi-threaded querying, whereas H queries are the most impacted. We believe this is because HH queries spend most of their time performing comparisons, thereby primarily utilising CPU time. In contrast, H queries are single-term and involve minimal comparisons, spending most of their time reading from PM due to the high frequency of terms. Consequently, when the interference on PM increases with more query evaluators concurrently reading from PM, H queries are the most affected. As shown in Figure 6.1, HH queries still remain the slowest at 32 query threads. HH queries are 5.6× slower than H queries.

6.2 Impact of Concurrent Indexing

Our analysis so far shows that query type impacts per-query latency, with HH queries being the slowest. We now analyse how pressure on PM varies, as additional indexing-only SPIRIT instances run along side a querying-only instance serving user queries. The goal is to identify opportunities for our adaptive system to alleviate increased pressure on PM by adjusting SPIRIT's execution.

We recall that SPIRIT consists of two key processes, indexing and query evaluation. Before analysing the impact of concurrent execution (indexing and querying together), we first analyse the read and write PM bandwidth for isolated execution (indexing-only and querying-only). Figure 6.3 presents a comparison of PM bandwidth for an indexing-only and querying-only instance (run in insolation). For querying-only, the average write bandwidth is 1.75 MB/s while the read bandwidth is close to 1474 MB/s. This aligns with our expectations, since serving user queries primarily involves reading from the merged index on PM. The behaviour of indexing-only is less intuitive as both reads and writes to PM occur. Writes to PM are present as the merged index is being built on PM. While reads to PM occur because merging of a segment during indexing, first checks whether terms exist in the merged index on PM.

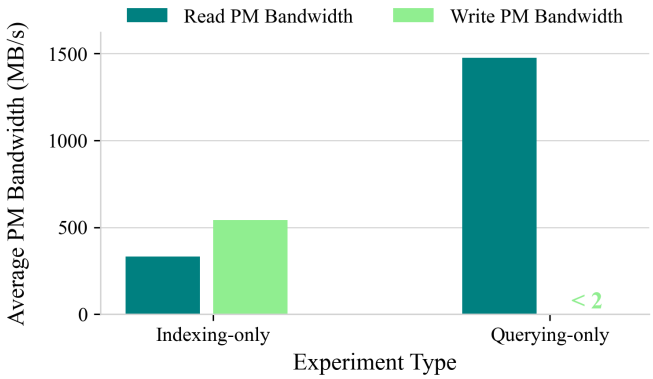


Figure 6.3: Comparison of average read and write PM bandwidth for an indexing-only, versus querying-only SPIRIT instance (in isolated execution mode). The indexing-only instance indexes 25M documents. The querying-only instance evaluates HH queries with a single query thread.

Now, we investigate how running indexing-only instances, alongside a querying-only instance, affects per-query latency. The workloads we execute are designed to simulate increasing pressure on PM, by increasing the number of indexing-only instances that are concurrently reading and writing to PM. Figure 6.4 summarises the results for single query thread experiments, with 0 (i.e., querying-only instance in isolation), 1, 4, 8 and 16 concurrent indexing-only SPIRIT instances. Figure 6.5 presents results from Figure 6.4, normalised to 0 concurrent indexing-only instances for ease of comparison.

Figure 6.5 clearly shows that the average per-query latency increases across all query types, as the number of concurrent indexing-only instances increases. This is relative to when 0 indexing-only instances are running in the system. For 1 versus 16 indexing-only instances, we observe that HH queries incur an increase in latency of 1.03× and 1.42×, respectively, compared to 0 instances. The increase in latency is less compared to other query types, for example, L queries. For 1 versus 16 instances, L queries incur an increase in latency of 1.16× and 1.97×, respectively. HH queries remain low-impacted as they

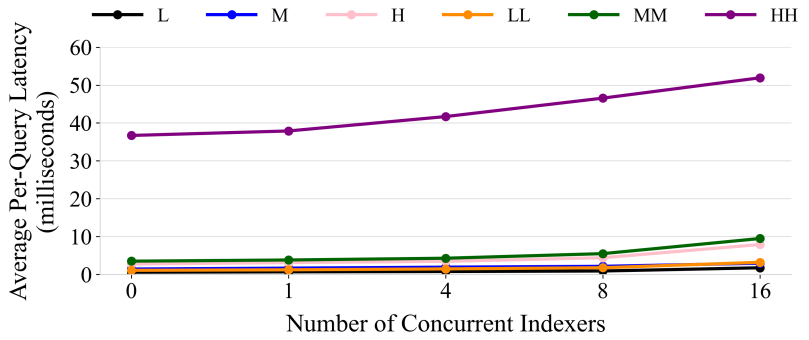


Figure 6.4: Average per-query latency measured with 0, 1, 4, 8, and 16 concurrent indexing-only (indexer) SPIRIT instances, across six query types with a single query thread.

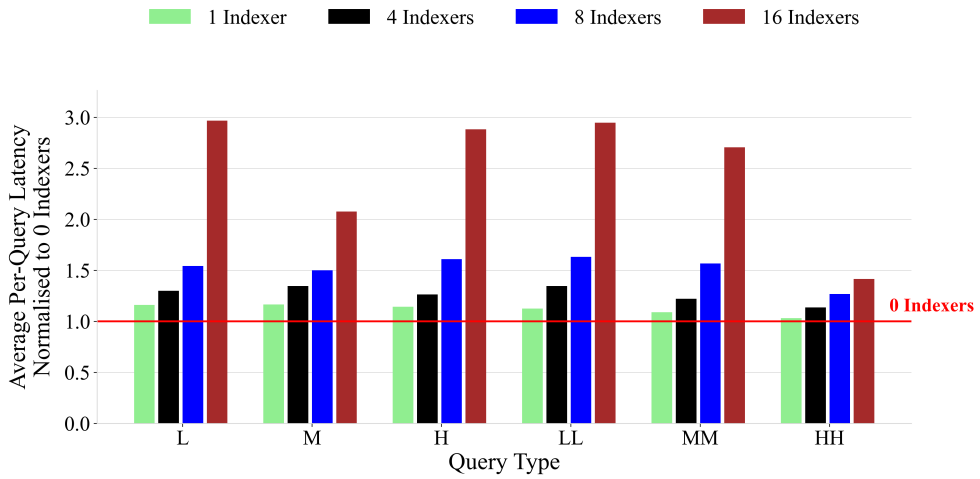


Figure 6.5: Average per-query latency, normalised to 0 concurrent indexing-only (indexer) instances, compared with 1, 4, 8, and 16 indexing-only SPIRIT instances, across six query types with a single query thread.

spend most of their time performing comparisons and thus primarily utilise CPU time. In contrast, L queries involve no comparisons and spend most of their time reading from PM. Consequently, when interference from concurrent indexing increases (i.e., additional reads and writes to PM), L queries are the most affected. Despite HH queries being the least affected out of all, Figure 6.4 shows that HH queries remain the slowest, reaching an average per-query latency of 52 ms at 16 concurrent instances, for 1 query thread. HH queries still remain 28.9× slower than L queries, despite the 1.97× increase in latency observed for L queries at 16 instances.

6.2.1 Impact of Multi-threaded Querying

Now, we analyse how multi-threaded querying affects per-query latency in workloads with concurrent indexing-only instances. Figure 6.10 provides a breakdown of results for 4, 8, 16 and 32 query evaluation threads. Overall, the average per-query latency increases across query types with the number of query evaluation threads. This is observed across 0, 1, 4, 8 and 16 indexing-only instance workloads. Figure 6.6 shows per-query latency for 32 query threads, normalised to 1 query thread. For 32 query threads, the average per-query latency is consistently higher (i.e., greater than the baseline of 1) across all indexer workloads, compared to 1 query thread. The observed increase in latency with multi-threaded querying can be attributed to synchronisation overhead among concurrent query evaluation threads, as well as higher PM bandwidth utilisation (discussed next). Figure 6.7 shows per-query latency for 32 query threads, normalised to 0 indexers. This figure highlights that latency continues to increase proportionally to the number of indexers, similar to the experiments with 1 query thread. For 4 versus 16 indexer instances with 32 query threads, the increase in latency for HH queries is 1.16× and 1.33×, respectively, relative to 0 indexers.

6.2.2 Utilisation of PM Bandwidth

Figure 6.8 shows the PM bandwidth utilisation for 1 query thread. Both the read and write bandwidth increases with the number of concurrent indexing-only instances. The read bandwidth reaches just above 2750 MB/s, while the write bandwidth reaches close to 1500 MB/s at 16 instances. As our earlier analysis indicates, a single indexing-only instance contributes to both reads and writes, so the proportional increase observed in bandwidth utilisation is expected as more instances run. Fedorova et al. (2022) reports that increased writes in a system using Intel Optane PM can slow down reads. With a single query thread, the read bandwidth continues to increase. However, we note that the overall utilisation of PM bandwidth increases with more concurrent (indexing-only) instances, indicating that there is more work in the system. Therefore, the increase in per-query latency observed in Figure 6.5, with more concurrent instances, is justified.

Figure 6.9 shows the PM bandwidth utilisation for 32 query threads. Unlike the increasing trend in read bandwidth observed with 1 query thread, we see a 1.36× drop when scaling from 0 to 16 concurrent indexing-only instances. This suggests that the read

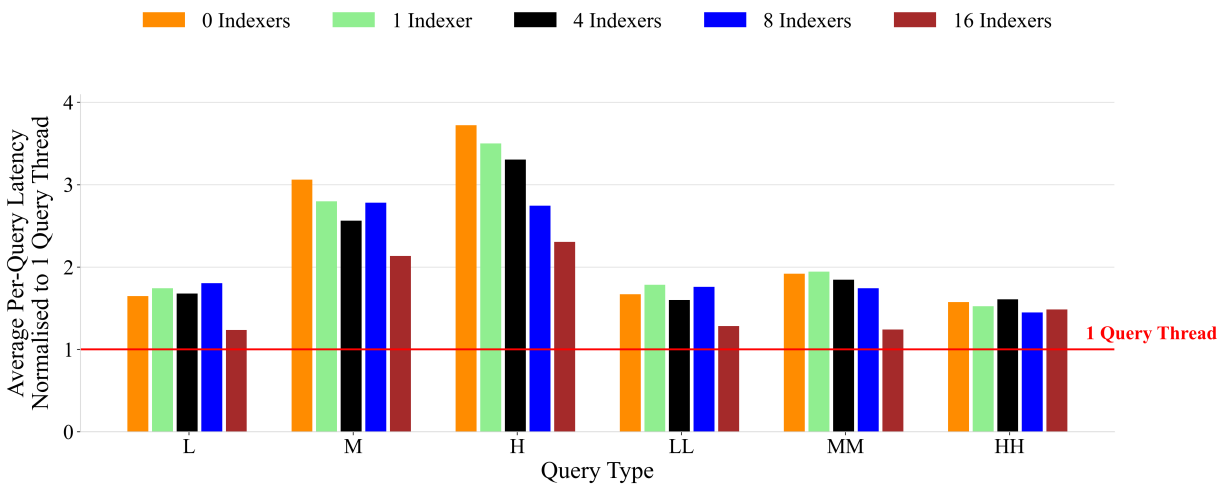


Figure 6.6: Average per-query latency with 32 query threads, normalised to 1 query thread, across six query types, measured with 0, 1, 4, 8, and 16 concurrent indexing-only (indexer) SPIRIT instances.

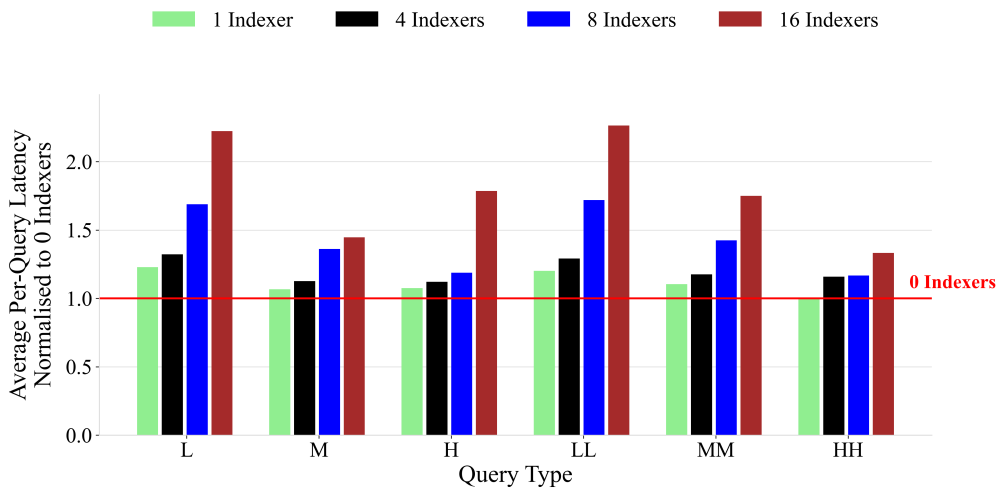


Figure 6.7: Average per-query latency with 32 query threads, normalised to 0 indexing-only (indexer) instances, compared with 1, 4, 8, and 16 concurrent indexing-only SPIRIT instances, across six query types.

bandwidth of Intel Optane PM has reached its maximum, aligning with the results of Fedorova et al. (2022)'s work. The average read bandwidth for 0 concurrent instances and 1 query thread is 1480 MB/s, and 25770 MB/s for 32 query threads. This is a 17.4× increase, indicating that increasing the number of query threads increases PM bandwidth utilisation. This, in turn, likely contributes to increased per-query latency seen in Figure 6.2 with more query threads. In contrast, the write bandwidth remains unchanged from 1 to 32 query threads. This is expected as the number of indexing-only instances remains the same, which are contributing writes to PM.

6.3 Key Takeaways

Our key takeaways from the analysis of PM pressure are as follows.

- Query type affects per-query latency, with HH queries having the highest latency, even when there are no additional reads and writes in the system from concurrent indexing.
- The presence of mixed reads and writes to PM, due to concurrent indexing and query evaluation, negatively affects read performance, resulting in higher per-query latency among all query types.
- A large number of concurrently running query evaluation threads adversely impacts per-query latency (read performance) due to increased PM bandwidth utilisation and possible synchronisation overheads.

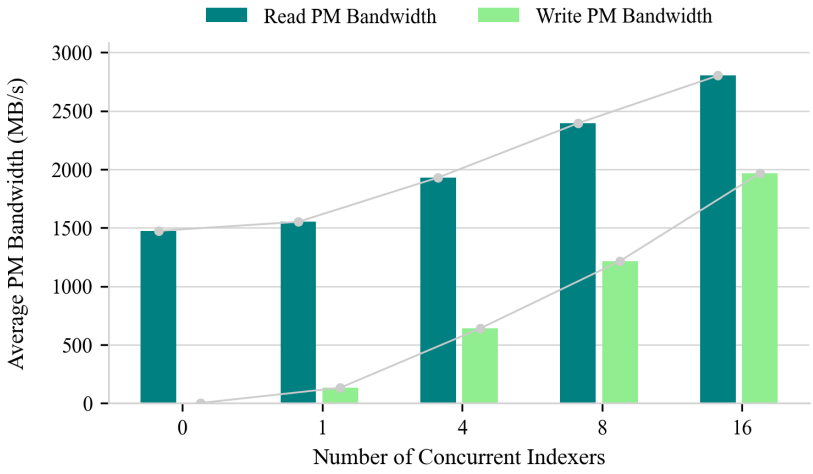


Figure 6.8: Average read and write PM bandwidth with 1 query thread, for HH query workload, measured with 0, 1, 4, 8, and 16 concurrent indexing-only SPIRIT instances.

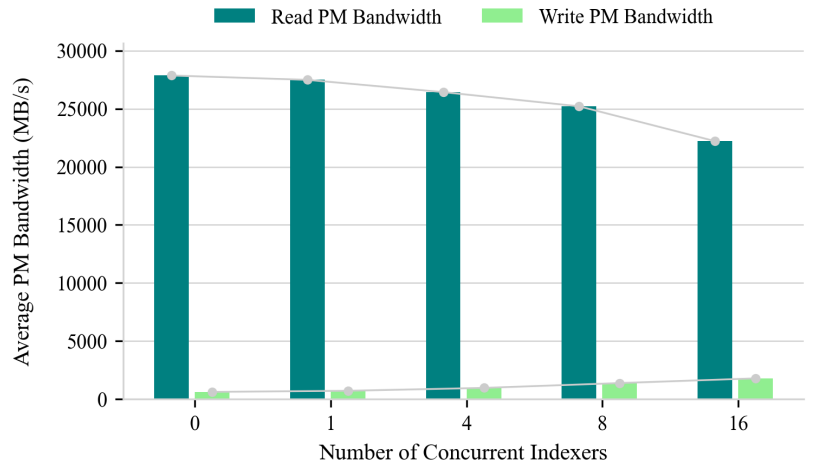
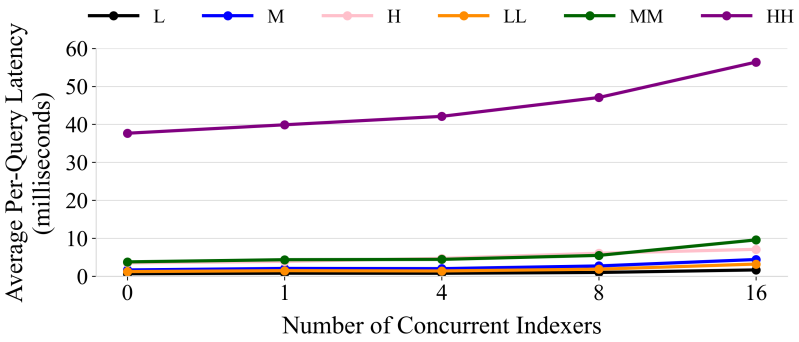
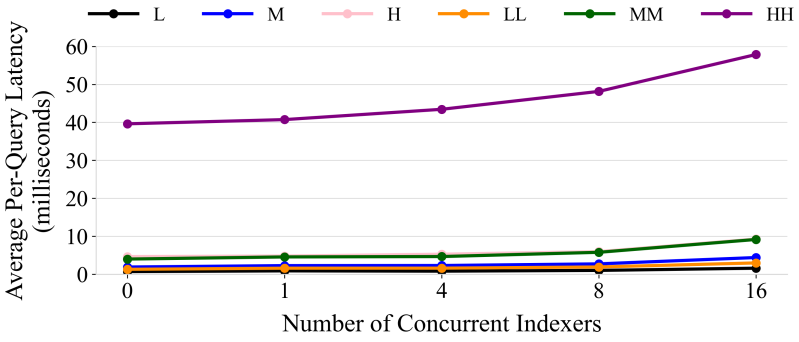


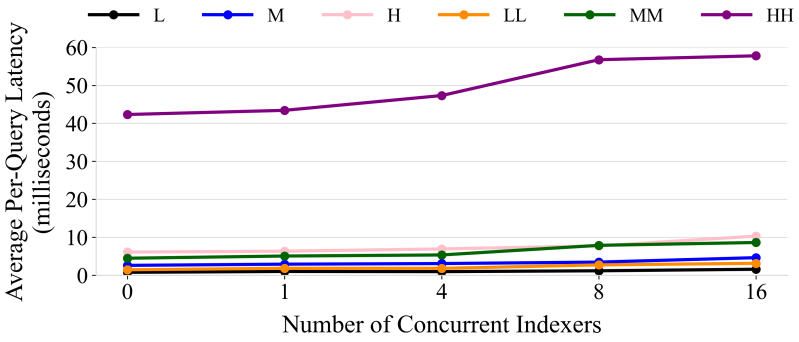
Figure 6.9: Average read and write PM bandwidth with 32 query threads, for HH query workload, measured with 0, 1, 4, 8, and 16 concurrent indexing-only SPIRIT instances.



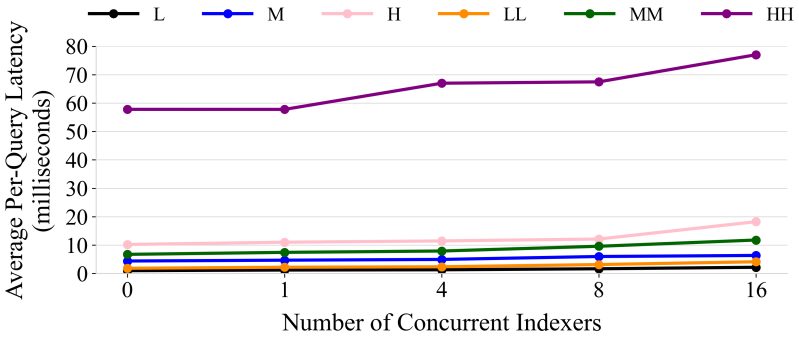
(a) 4 Query threads



(b) 8 Query threads



(c) 16 Query threads



(d) 32 Query threads

Figure 6.10: Average per-query latency measured with 0, 1, 4, 8, and 16 concurrent indexing-only (indexer) SPIRIT instances, across six query types and a varied number of query evaluation threads.

Chapter 7

Design and Implementation of ARITHMETY

We now discuss the design and implementation of ARITHMETY, which satisfies fast query response, crash consistency guarantees or ingestion of new data by adapting to workload characteristics. We implement ARITHMETY by extending the state-of-the-art text-inversion engine, SPIRIT. Also, we build upon the optimised crash consistency mechanism described in Chapter 4. Its design considerations are guided by key takeaways from Chapter 6.

We restate the key takeaways here for clarity.

1. Query type affects per-query latency, with HH queries having the highest latency, even when there are no additional reads and writes in the system from concurrent indexing.
2. When PM is under pressure from both indexing (reads and writes) and query evaluation (reads), reads slow down, increasing query latency. Higher PM bandwidth utilisation also results from a large number of query evaluation threads running concurrently, leading to increased query latency.

We begin by outlining the design considerations that underpin the system, and then discuss ARITHMETY's architecture, which consists of: (1) a server offering a controller service, and (2) implementation in SPIRIT.

7.1 Design Considerations

The design of ARITHMETY is based on the following considerations:

(A) Be aware of the current pressure on PM. ARITHMETY exploits the analysis that per-query latency increases with mixed reads and writes in the system from

concurrent indexing and query evaluation. It introduces a profiling phase in SPIRIT to capture per-query latency measurements and forward them to the controller service. The controller takes action to maintain or mitigate pressure on PM, informed by per-query latency measurements.

(B) Be aware of changes in PM pressure over time. ARITHMETY monitors per-query latencies over time and detects drift by analysing the trend of the most recent n latencies. By identifying whether the trend is consistently increasing, decreasing, or steady, it informs the controller's action.

(C) Be aware of query workload. ARITHMETY exploits the analysis that query type affects per-query latency, where L queries are 61.7× faster than HH queries. SPIRIT captures the query workload type and forwards it alongside per-query latencies to the controller. This is to capture scenarios in which the query workload type is fast, regardless of the pressure on PM. This informs the controller's action.

(D) Be aware of client priority. To optimise for priorities of real-time search engines, such as fast query response, crash consistency or ingestion of new data, the controller takes action based on the client priority.

7.2 Architecture of ARITHMETY

ARITHMETY is designed on a client-server architecture. In our design, we introduce a new server component, with each SPIRIT instance acting as a client. Instead of modifying SPIRIT to communicate with other instances, to allow adaptive behaviour, we chose to separate this logic into a dedicated server. This approach provides modularity. The server can be extended to offer various services beyond a service specific to SPIRIT. Additionally, this separation allows SPIRIT to remain a real-time search engine, while the server/service is reserved for centralised decision-making. The separation achieved via a client-server architecture enables SPIRIT to scale independently of the server.

We begin by discussing the server and the controller (service), followed by implementation in SPIRIT to interact with the server, and conclude with a step-through of the end-to-end integration.

7.2.1 Server

We design and implement a server that (1) handles client connections, and (2) provides a service, referred to as the controller. Hereafter, we use server to refer to the process that listens for and handles client connections, and controller to mean the specific service that responds to changes in PM pressure. The term "server application" refers to the complete system responsible for handling client connections and providing the controller service, as shown in Figure 7.1.

The server acts as the entry point to the controller. Its ability to handle simultaneous connections is a critical requirement. To achieve this, we use I/O multiplexing to handle

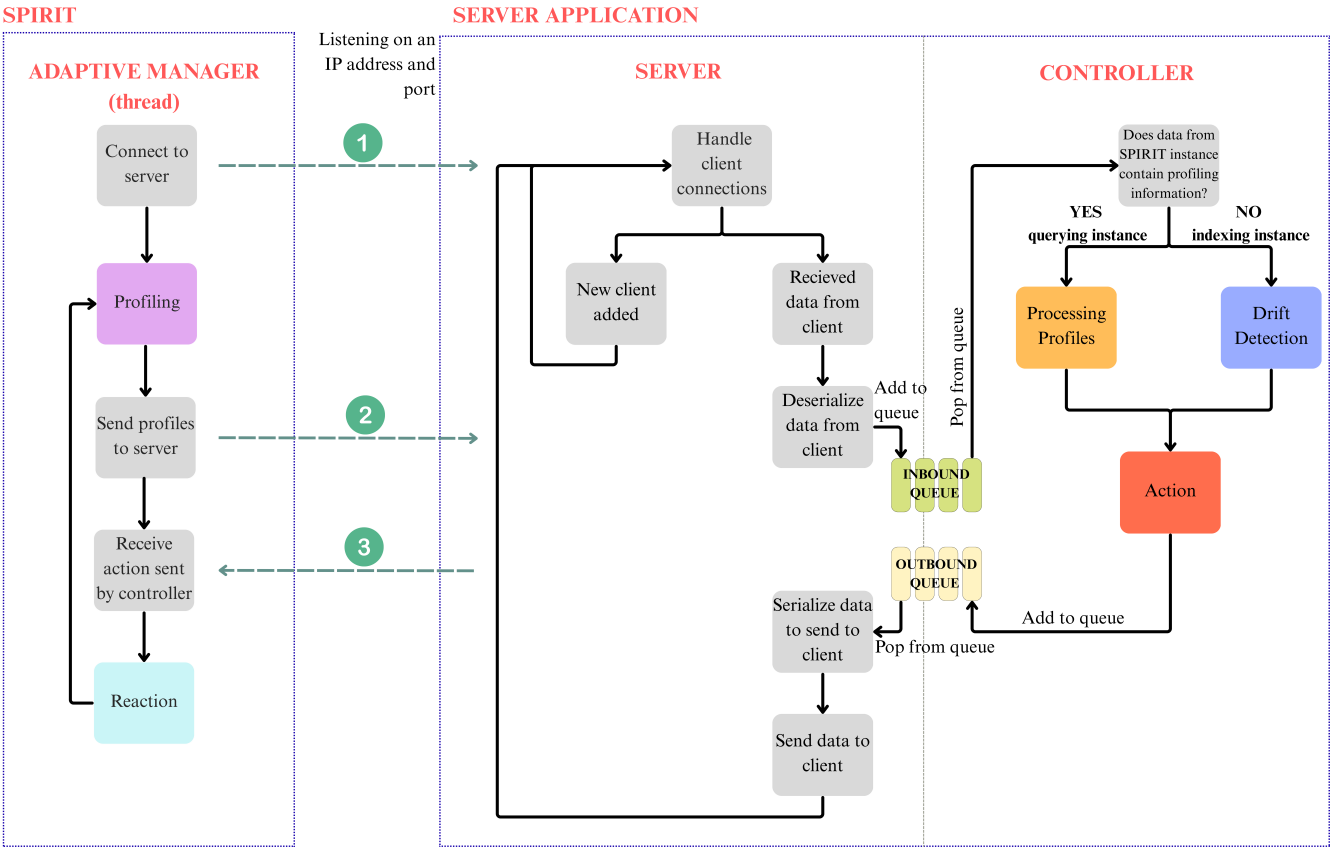


Figure 7.1: End-to-end flow of ARITHMETY.

concurrent connections (Shichao (n.d.)).

The application interacts through a generic server interface, which provides flexibility to support different server implementations. Our implementation of the server (1) handles new client connections, (2) receives and deserialises data, (3) forwards the deserialised data to the controller, and (4) sends the controller's response serialised back to the client. To maintain flexibility, the application is exposed through generic serialisation and deserialisation interfaces. Our implementation exchanges data between the client and the server using the easily extensible JSON format. The server communicates with the controller through a concurrent queue, ensuring independence between the two. This allows the server to enqueue tasks without waiting for the controller to complete processing them. Likewise, the controller can dequeue and process tasks independently.

7.2.2 Controller

Once again, we expose the controller as a generic interface to allow flexibility. Our controller implementation is guided by the four design considerations of ARITHMETY. We build ARITHMETY on all four, for it to automatically adapt to client priorities. If the system is aware of pressure on PM and the client prioritises fast query responses, ARITHMETY can stop additional reads and writes to PM when the pressure is high. By also being aware of the current query workload, ARITHMETY resumes indexing (reads and writes) once faster queries are encountered, that remain within the client-specified latency bounds. Conversely, when strong crash consistency guarantees are prioritised, new data must be persisted promptly, and thus writes to PM cannot be delayed. So, the controller is designed with three main execution phases: Processing Profiles, Drift Detection, and Action. We describe each phase and the motivation for each in detail next.

Profiling Metrics

SPIRIT sends profiling metrics to the controller on regular intervals (discussed further in Section 7.2.3). We select these metrics based on design considerations (A) and (B). We observe that per-query latency increases (for all query types) when PM is under pressure from mixed reads and writes. We select per query latency (Q_L) as a profiling metric, as it is directly impacted when there is increased PM pressure. Additionally, it is a user-level metric that reflects what services care about – query response times. We know there is variation in per-query latency across query types, with L queries being 61.7× faster than HH queries. Based on this, we also choose the query type (Q_T) as a profiling metric. These two metrics allow the controller to automatically adapt, based on changes in PM pressure and query workload. Our design defines a *profile* that is easily extensible to incorporate additional metrics in the future.

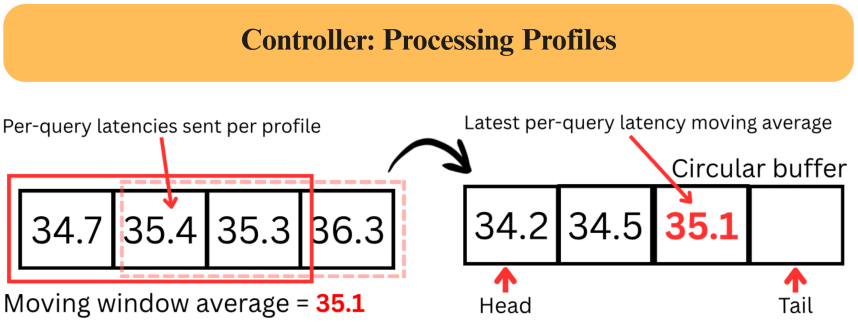


Figure 7.2: Processing Profiles phase of the controller. Two-step process: (1) computing moving window averages, and (2) storing these averages in a circular buffer.

Processing Profiles

A SPIRIT instance (that is serving user queries) regularly sends a list of profiles, each consisting of profiling metrics, Q_L and Q_T, as discussed earlier. To provide the controller with visibility of the current query type, the latest query type sent is stored on the controller. In contrast, storing per-query latencies is more involved.

A key principle for the controller is to be aware of changes in pressure on PM over time. To satisfy this, the profile processing step is necessary to aggregate and store per-query latencies over time. Q_L values are aggregated by taking a moving window average to smooth out noise (see Figure 7.2). The size of the moving window is a configurable parameter. To maintain visibility of the moving averages of query latencies, the controller uses a circular buffer to track the averages (see Figure 7.2). While it is essential for the controller to be aware of changes in pressure on PM over time, only the recent changes in PM pressure are critical to inform the Drift Detection step. Consequently, older latency averages become outdated. Using a circular buffer allows memory to be efficiently reused to store only the recent averages. The size of the circular latency buffer is a configurable parameter.

Drift Detection

Drift detection is a fundamental step for determining the current PM pressure by analysing the overall trend in latencies (increasing, decreasing or steady) at a given time. This stems from the design consideration of (B). The results from the Drift Detection step inform the Action step of the controller. A naive approach is to consider only the most recent average per-query latency and check whether it exceeds a specified threshold. If the most recent average latency is greater than or equal to the threshold, the current PM pressure is classified as increasing, otherwise it is not increasing. However, this method fails to capture whether latencies are increasing, decreasing or steady overall, across all averages stored in the circular buffer. The most recent average latency may contain a temporary spike that exceeds the threshold, even though the overall trend is not increasing.

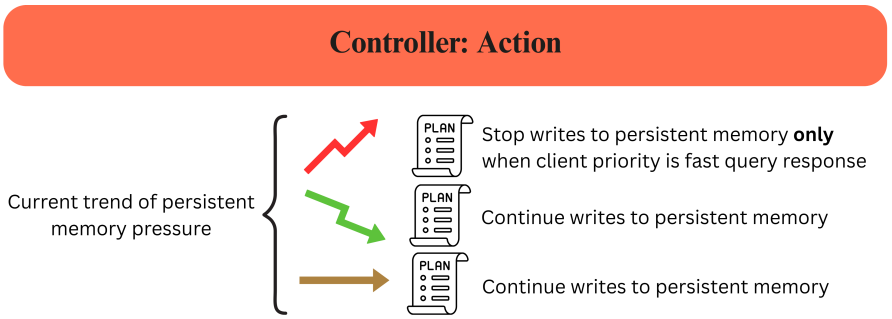


Figure 7.3: Action phase of the controller. This phase depends on the current PM pressure identified by the Drift Detection phase.

Therefore, alongside checking the average against a threshold, linear regression analysis is performed to determine the trend across all the averages stored. The controller calculates the regression slope and compares it against a configurable parameter, referred to as the positive gradient threshold. The line of best fit is only classified as increasing when its slope or gradient is greater than the positive gradient threshold. The regression slope (m) is calculated using the following formula (Montgomery et al. (2012)):

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

Where:

- n = number of latency averages in buffer
- $x_i = 0, 1, 2, \ldots, n-1$ (the index of each latency average in buffer)
- y_i = latency average from buffer

In summary, the controller sets the current PM pressure to increasing, only if the regression slope is greater than the positive gradient threshold, **and** the recent average per-query latency is greater than or equal to a specified threshold. The latter threshold is called the reaction threshold for query latency, and is provided by the client to bound tail latency when possible. Chapter 8 reports results of varying the two threshold parameters.

Additionally, a trend is classified as not increasing after previously being classified as increasing under two conditions. (1) When the query workload changes, indicated by a change in the current query type to a faster workload, and the trend in PM pressure is decreasing. (2) When the trend in current PM pressure is decreasing consistently, and the average latency falls below the reaction threshold.

Action

Figure 7.3 provides a high-level overview of the controller's action phase. From Chapter 6 we know that per-query latency increases when PM is under pressure from both writes and reads. This insight forms the basis for the controller's action. A naive approach is to stop all SPIRIT instances that are not serving user queries but still contributing reads and writes to PM. Instead, we implement a more advanced mechanism that temporarily halts only the engraver and merger threads of SPIRIT's indexing pipeline. These two threads read and write to PM, as they are responsible for copying segments to PM and merging segments in PM, respectively. Temporarily stopping these threads allows the controller to resume their operation when, for example, the query type changes to a faster query type.

Once the controller's drift detection step identifies the current PM pressure as either it is increasing or not increasing, a *plan* is constructed. The plan in our implementation represents the four threads of SPIRIT's indexing pipeline: ingester, flusher, engraver, and merger. The controller specifies the execution status of each as execute or stop executing.

Nonetheless, the controller's design does not yet consider being aware of client priority. In real-world systems, a client may not always value low query latency and thus fast query response times. Instead, they may prioritise strong crash consistency guarantees or freshness (or ingestion) of new data (Tomsic et al. (2018)). If strong crash consistency or freshness of data is a priority, the controller does **not** halt the engraver and merger threads, even when drift detection detects an increasing trend. This ensures that segments in DRAM are copied to the persistent domain (PM). Additionally, the ingester thread is not halted when DRAM space runs out, as it can free segments in DRAM once merged on PM, avoiding compromising data freshness.

Our controller supports three client priorities:

- Fast query response
- Strong crash consistency
- Freshness of Data

The two system guarantees that ARITHMETY provides at any given time, based on the client priority, are discussed in detail in Section 7.3.

7.2.3 Implementation in SPIRIT

We now describe the implementation in SPIRIT to enable an instance to interact with the server application. To incorporate adaptive behaviour into SPIRIT without affecting querying and indexing operations of real-time search, a separate Adaptive Manager thread runs concurrently.

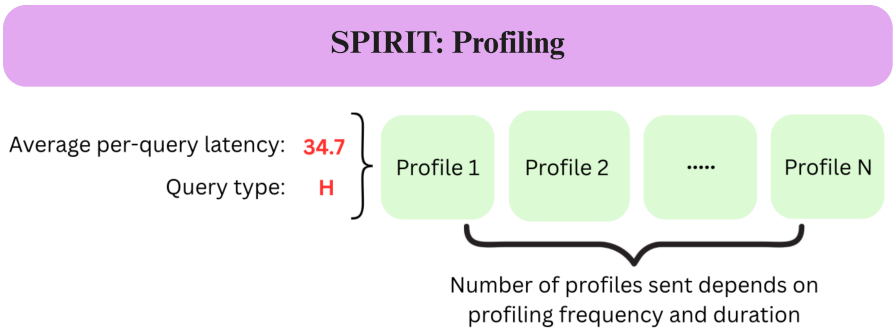


Figure 7.4: Profiling phase of the adaptive manager thread of SPIRIT. In this phase, an SPIRIT instance that is querying periodically sends profiles to the controller.

We expose the adaptive manager to a set of APIs. This provides a high level of abstraction. The adaptive manager calls APIs to do the following.

1. Establish a connection to the server.
2. Collate profiling metrics, Q_L and Q_T into profiles.
3. Send profiles to the server
4. Process the action received from the server/controller.

Note, steps (2) and (3) only apply to SPIRIT instances that are serving user queries (querying), while step (4) only applies to instances that are ingesting new content and building an index (indexing). We now discuss the two key execution phases of the adaptive manager.

Profiling

The profiling phase collects profiling metrics from a SPIRIT instance (querying) to send to the server application. SPIRIT gathers the per-query latency and query type measurements into a profile. Several profiles are sent at once to amortise the cost of network API calls (Borisov et al. (2025)). Figure 7.4 provides a high-level overview of the profiling phase. The total number of profiles sent at once is based on:

$$\text{Total number of profiles} = \frac{\text{profiling duration}}{\text{profiling frequency}}$$

The choice of profiling duration (how long we profile) and profiling frequency (how often we profile) are critical in obtaining representative measurements. As discussed earlier, these metrics are important in detecting pressure on PM, and underpin the controller's Drift Detection and Action phase. We determine suitable profiling duration and frequency parameters for our workloads (ref. Table 5.4), however, automatically tuning these parameters for arbitrary workloads is left for future work.

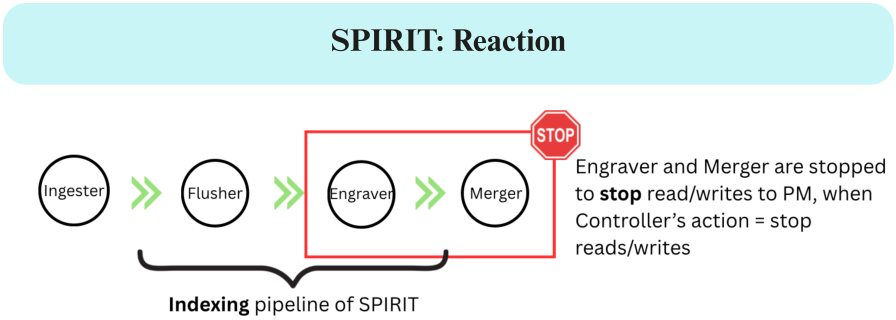


Figure 7.5: Reaction phase of the Adaptive Manager thread of SPIRIT. In this phase, an SPIRIT instance that is indexing reacts to the controller action sent.

In SPIRIT, query evaluation thread(s) evaluate queries and individually record per-query latencies. An important consideration is how to aggregate the query latencies reported by multiple query threads. For each profile, the per-query latency (Q_L) is determined by taking the average across all query threads, of the average per-query latency over the last n queries. This is given by:

$$\bar{L}_{\text{profile}} = \frac{1}{m} \sum_{i=1}^{m} \bar{L}_i$$

Where:

- m = number of query threads
- $\bar{L}_i$ = average per-query latency for last n queries for thread i
- $\bar{L}_{\text{profile}}$ = per-query latency (Q_L) for a given profile

The alternative is to send a list of latency averages to the server application per query thread. We avoid this approach due to scalability limitations. As the number of query threads increases, the total number of profiles to send would grow proportionally. To address this, we perform aggregation in SPIRIT before sending data to the server application.

Reaction

Figure 7.5 provides a high-level overview of the reaction phase. The reaction phase follows on from the controller's action sent (ref. Section 7.2.2), specifying the execution status of each thread in the indexing pipeline. Internally, our design provides an API to process the action by issuing a RUN (execute) or PAUSE (stop executing) command to the corresponding indexing pipeline thread. To implement this, we use the POSIX library, which provides efficient event-driven synchronisation guarantees (Thomadakis and Liu (1997)). We recall that this step is only for instances indexing.

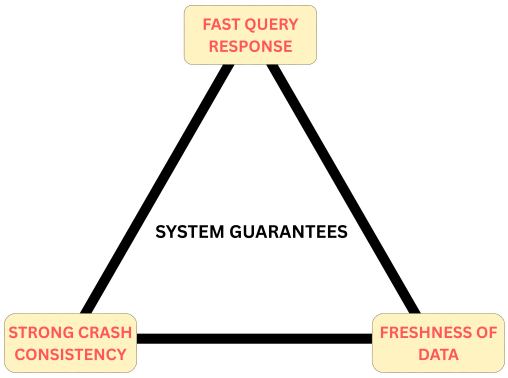


Figure 7.6: Triad of ARITHMETY's guarantees.

7.2.4 End-to-End Integration

Figure 7.1 shows the end-to-end flow of ARITHMETY. To summarise, an SPIRIT instance first connects to the server. If the instance is querying, it continuously sends profiles containing (aggregated) per-query latency and query type measurements. These profiles are then forwarded to the controller and processed. The current trend in PM pressure is identified through drift detection, which then determines the appropriate action to take. The controller decides either to maintain or mitigate PM pressure based on the client priority. Controller sends the action back, and the SPIRIT instance (indexing) reacts by stopping engraver and merger threads (if actioned) to mitigate pressure on PM. This interaction with the server continues until the instance terminates, i.e after indexing and/or querying completes. We observe that ARITHMETY shows higher CPU utilisation than the state-of-the-art, SPIRIT, which is a trade-off for a system that dynamically adapt to variations in PM pressure.

7.3 System Guarantees

Figure 7.6 illustrates the guarantees that ARITHMETY can provide. These three guarantees correspond to the client priorities that can be specified in ARITHMETY. We analyse the total number of guarantees ARITHMETY provides at any given time, similar to the CAP theorem (Brewer (2000)).

- When fast query response is prioritised, the system halts the engraver and merger threads, i.e. writes to PM, when PM pressure is high. There are two possible scenarios, depending on whether there is available DRAM capacity or not. (1) If there is available DRAM capacity, ingestion of new content continues, preserving the freshness of data. However, newly ingested content remains in DRAM and is not immediately persisted to PM, which compromises the strong crash consistency guarantee. (2) If there is no available DRAM, ingestion of new content is stalled, compromising the freshness of data. Since no new content is ingested in DRAM, no new content needs to be persisted, thereby maintaining the strong crash consistency

guarantee. In both scenarios, ARITHMETY holds two out of its three guarantees.

- When the freshness of data is prioritised, engraver and merger threads are not halted. So, new content continues being ingested, with DRAM space reclaimed via merging on PM when necessary. This also means that newly ingested content in DRAM is persisted, and the strong crash consistency guarantee is maintained. However, under high PM pressure, query response times increase, compromising the fast query response guarantee.
- When strong crash consistency is prioritised, engraver and merger threads are not halted, ensuring that data is persisted promptly. The system keeps ingesting new content. When DRAM space is exhausted, merging segments on PM frees up DRAM, maintaining the freshness of data. However, under high PM pressure, query response times increase, compromising the fast query response guarantee.

Therefore, at any given time, ARITHMETY can provide two of the three guarantees.

Chapter 8

Sensitivity Analysis

In this chapter, we present the results of our sensitivity analysis of the reaction threshold and drift detection (i.e., positive gradient threshold) with respect to query latency.

8.1 Sensitivity of Reaction Threshold to Query Latency

Figure 8.1 shows the sensitivity of reaction threshold to query latency for an Adaptive system that prioritises fast query response, and a Baseline system that prioritises strong crash consistency. The results indicate that a greater reaction threshold delays the controller's response to mitigate PM pressure and reduce per-query latencies in the Adaptive system. We observe a drop in latencies past 11,300 queries with $tr2 = 40$ ms (Figure 8.1 (b)). In contrast, for $tr4 = 60$ ms (Figure 8.1 (d)), latencies cross the threshold of 60 ms past 34,000 queries, triggering the controller to take action at that point. We recall that the reaction threshold is part of the controller's drift detection mechanism. Specifically, the average query latency must exceed the threshold (as well as be increasing over time) for the controller to take action to stop additional reads and writes to PM. Latencies sorted by the Baseline latencies in Figures 8.2 and 8.3 clearly show the difference at which the controller intervenes to mitigate pressure and lower latency for $tr2 = 40$ versus $tr4 = 60$.

We observe that ARITHMETY starts to reduce PM pressure just past the reaction threshold, and continues to effectively monitor and manage increasing PM pressure throughout the query workload. In Figure 8.1, for the Adaptive configuration, we observe effective mitigation of PM pressure, as latencies consistently remain below the reaction threshold throughout the query workload. This is in all cases except $tr1 = 30$ ms. Our analysis from Chapter 6 reports that HH queries exhibit a minimum average per-query latency of 37 ms, which exceeds the 30 ms threshold, explaining this exception. This also explains why we see comparable latencies for Baseline and Adaptive for the first

11,000 queries in Figure 8.1 (a). We observe that the tail latency of 137 ms (at 33,100 queries) is reduced by 4.1× for $tr1 = 30$ ms, $tr2 = 40$ ms and $tr3 = 50$ ms. Whereas, latency is reduced by 2.7× for $tr4 = 60$, as the controller's response is delayed due to the higher reaction threshold and pressure on PM is not mitigated instantly.

There are two distinct peaks in latency around 25,500 and 33,100 queries in the Baseline system, across all four reaction thresholds. This behaviour can be attributed to the workload progressively increasing the number of (indexing) SPIRIT instances that concurrently perform reads and writes to PM, along side querying. The workload transitions from 1 to 4 to 8 to 16 concurrent instances. The first peak corresponds to the point when 8 instances start running. The second, tail-latency peak of 137 ms at 33,100 queries corresponds to when 16 instances begin executing, further increasing pressure on PM.

There is a peak in query latency at the start (during the first query), which can be attributed to startup overheads, such as an initially empty CPU caches and translation lookaside buffer (TLB). The TLB, which caches recent virtual-to-physical page mappings, affects performance because the search index resides in PM.

In summary, we observe that a higher reaction threshold delays the controller's response to mitigate pressure and favour fast query responses. These results are observed for a system where PM pressure gradually increases due to mixed reads and writes from concurrent indexing and querying. To evaluate ARITHMETY, we set the reaction threshold to 40 ms to ensure the system promptly responds to increases in PM pressure and prioritises faster query responses (early on) for already slow queries.

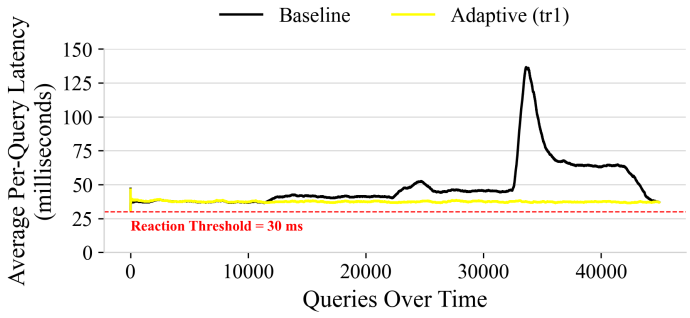
8.2 Sensitivity of Drift Detection to Query Latency

Figure 8.4 shows the sensitivity of drift detection, i.e. positive gradient threshold to query latency for the Adaptive and Baseline systems. The positive gradient threshold defines the minimum slope or gradient of the line of best fit for latencies required to classify it as increasing. We observe a similar impact on query latency as that observed when varying the reaction threshold. As the gradient threshold increases, the controller's action to mitigate pressure and favour fast query responses is delayed. This is expected because a larger/stricter threshold for gradient means a steeper increase in average query latency is necessary for the controller to mitigate pressure. We know that the highest pressure on PM is observed closer to the end, at 33,100 queries with 16 concurrent instances and a tail latency of 111 ms.

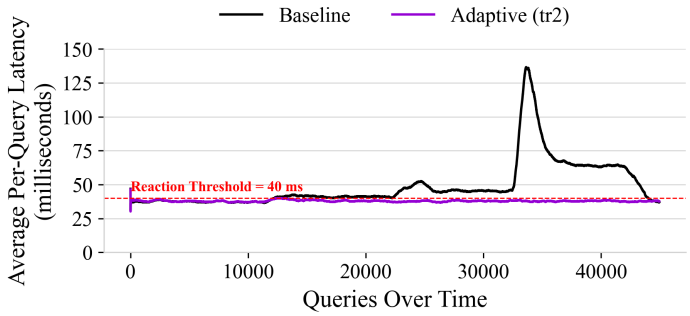
A weaker, gradient threshold of $tg1 = 1 \times 10^{-4}$ (Figure 8.4 (a)) results in a reduction in per-query latency at approximately 11,300 queries. Whereas, a stricter threshold of $tg2 = 1 \times 10^{-3}$ (Figure 8.4 (b)) delays reduction in latency to around 32,000 queries, during which a steeper increase in latencies is observed. Even though the reaction threshold is fixed at 40 ms, the controller only takes action past 48 ms (for $tg2 = 1 \times 10^{-3}$) as latencies begin to increase at a greater rate. There is no adaptive action is observed

with the strictest threshold of $tg4 = 1 \times 10^{-1}$ (Figure 8.4 (d)), despite the increasing PM pressure. This can be attributed to the trend in latencies not increasing steeply enough over time. Furthermore, Adaptive exhibits higher tail latency than Baseline for $tg4$. We attribute this to the higher CPU utilisation of ARITHMETY (than Baseline which does not consider drift detection), which particularly affects HH queries since they also incur high CPU utilisation from intensive comparisons.

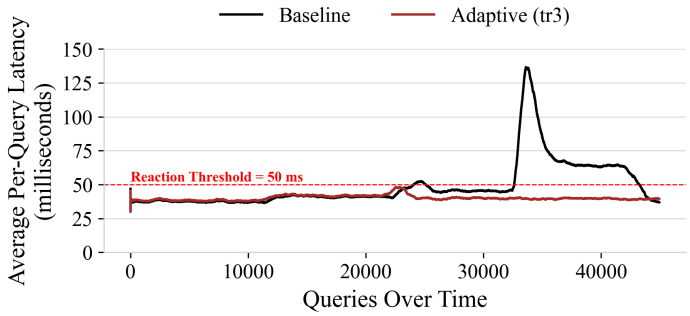
In summary, we observe that drift detection with a stricter/larger positive gradient threshold delays the controller's response to mitigate PM pressure and favour fast query responses. These results are observed for a system where PM pressure gradually increases due to mixed reads and writes from concurrent indexing and querying. To evaluate ARITHMETY, we set the positive gradient threshold (of drift detection) to 1×10^{-4} to ensure the system promptly responds to increases in PM pressure and prioritises faster query responses (early on) for already slow queries.



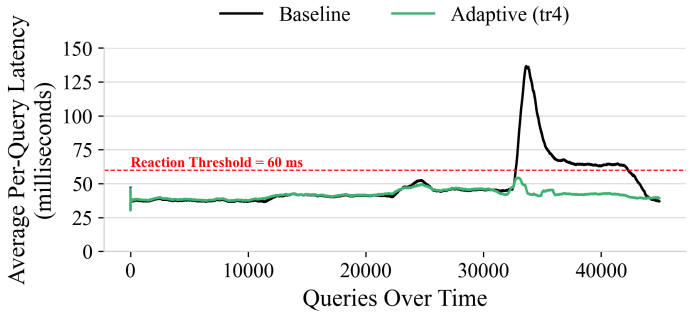
(a)



(b)



(c)



(d)

Figure 8.1: Query latencies over time for Adaptive versus Baseline under varying reaction thresholds: (a) $tr1 = 30$ ms, (b) $tr2 = 40$ ms, (c) $tr3 = 50$ ms, and (d) $tr4 = 60$ ms. The positive gradient threshold is fixed at 1×10^{-4}.

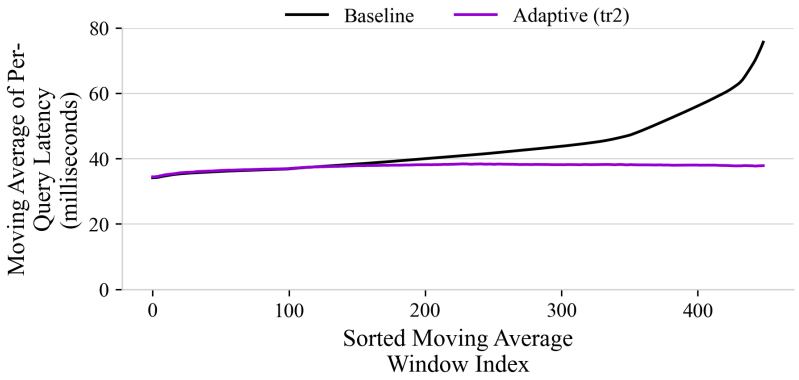


Figure 8.2: The moving window averages of query latency for reaction threshold $tr2 = 40$ ms, sorted by Baseline latencies from lowest to highest.

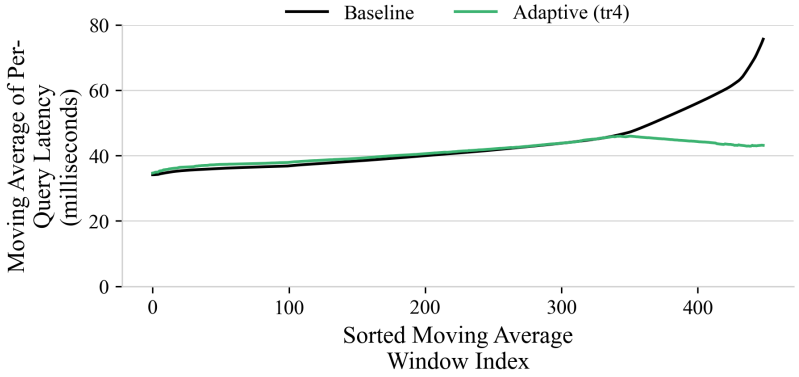
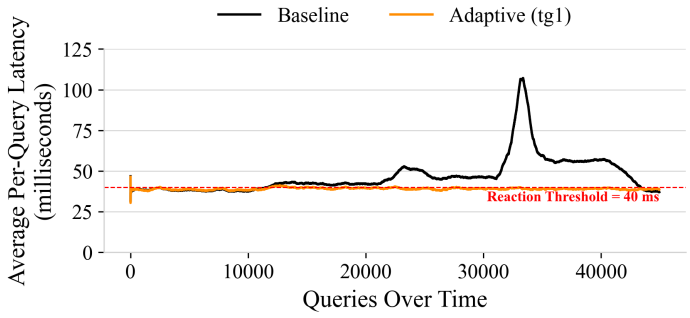
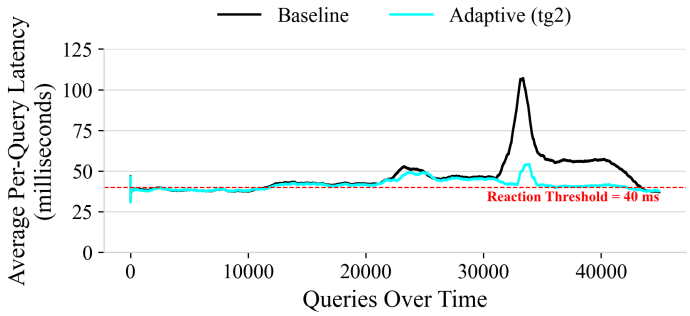


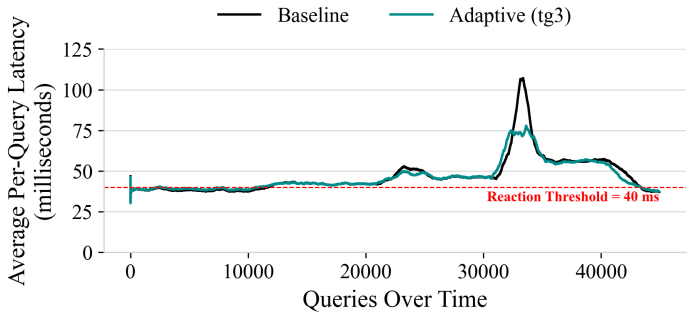
Figure 8.3: The moving window averages of query latency for reaction threshold $tr4 = 60$ ms, sorted by Baseline latencies from lowest to highest.



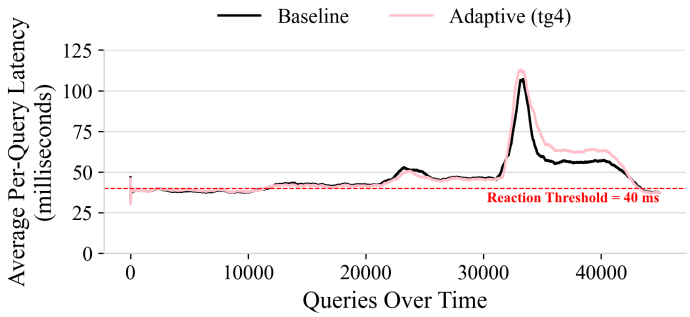
(a)



(b)



(c)



(d)

Figure 8.4: Query latencies over time for Adaptive versus Baseline under varying positive gradient (drift detection) thresholds: (a) $tg1 = 1 \times 10^{-4}$, (b) $tg2 = 1 \times 10^{-3}$, (c) $tg3 = 1 \times 10^{-2}$, and (d) $tg4 = 1 \times 10^{-1}$. The reaction threshold is fixed at 40 ms.

Chapter 9

Evaluation Results

In this chapter, we present the evaluation results for the two main contributions of this work: (1) crash consistency optimisation and (2) ARITHMETY. Lastly, we summarise the key takeaways from our evaluation of the two.

9.1 Evaluation of Crash Consistency Optimisation

In this section, we present and analyse the results evaluating the crash consistency optimisation discussed in Chapter 4.

9.1.1 Indexing Time

Figure 9.1 compares the total indexing time between SPIRIT's optimised and baseline crash consistency, across indexing workloads of 1M, 5M, and 10M documents. We observe a 2× reduction in total indexing time for the optimised versus the baseline, for both the 1M and 5M workloads. It is worth noting that the total indexing time for baseline and optimised increases going from 1M to 5M, as more documents have to be indexed. The 10M workload is not evaluated for the state-of-the-art SPIRIT (baseline), as there is a bug causing the search engine to hang.

To identify the factors contributing to the reduction in total indexing time, Figure 9.2 provides a breakdown by SPIRIT's indexing pipeline stages. The results show that the reduction in total indexing time is primarily due to the 4× reduction in merge time. In contrast, the time spent in the ingest, flush, and engrave stages is unchanged with the optimisation. This is expected, as the crash consistency optimisation specifically targets removing persist barriers (*clflush* and *mfence*) during the merge stage. By eliminating three of the five high-overhead persist barriers, the optimisation effectively reduces number of barriers issued for every term merged.

Additionally, the optimisation reduces the merge time to be less than the ingestion time, by 1.4×. This makes ingestion the primary contributor to the overall indexing time. We evaluate next how this change impacts query throughput when indexing and querying occur concurrently.

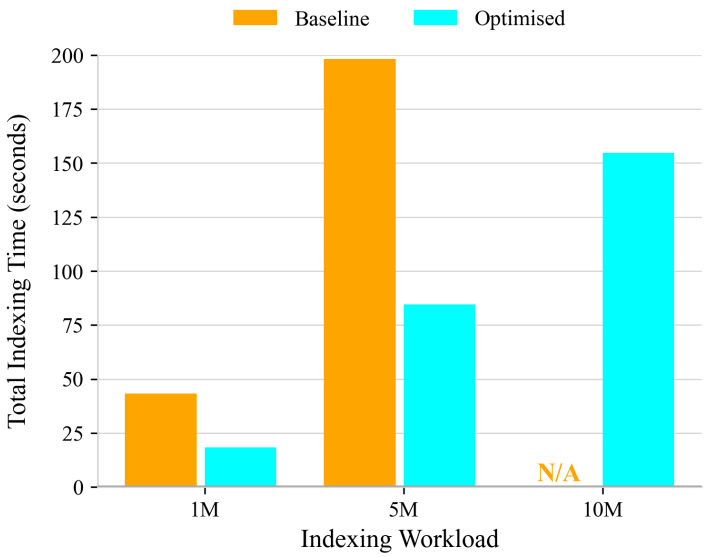


Figure 9.1: Total indexing pipeline time, comparing SPIRIT's optimised and baseline crash consistency across different indexing workloads (**1M** documents, **5M** documents and **10M** documents).

9.1.2 Query Throughput

Figure 9.3 shows the percentage change in average queries per second (QPS) (– averaged across all query types) when comparing the optimised crash consistency with the baseline. We observe a positive increase in percentage change of average QPS, for each DRAM configuration (*loose*, *medium*, and *tight*). A maximum percentage change of 103% is observed under the *tight* DRAM configuration (0.5 GB long-term DRAM). This improvement is because ingester is forced to perform more frequent merges to free up space in long-term DRAM. Since our optimisation reduces the number of persist barriers issued for every term merged, the merge time is reduced (Observation 9.1.1) and thus, segments are merged more quickly. In contrast, the baseline crash consistency experiences slower merges. This results in query evaluators stalling for longer periods during query evaluation (and thus, serving fewer queries per second) while waiting to acquire a read lock for the newly merged segment. The *loose* DRAM configuration (4 GB long-term DRAM) shows a percentage change of 1.3% in QPS, as fewer merges are necessary, due to the larger DRAM capacity. SPIRIT's policy of keeping segments visible in long-term DRAM for as long as possible works favorably under the *loose* configuration.

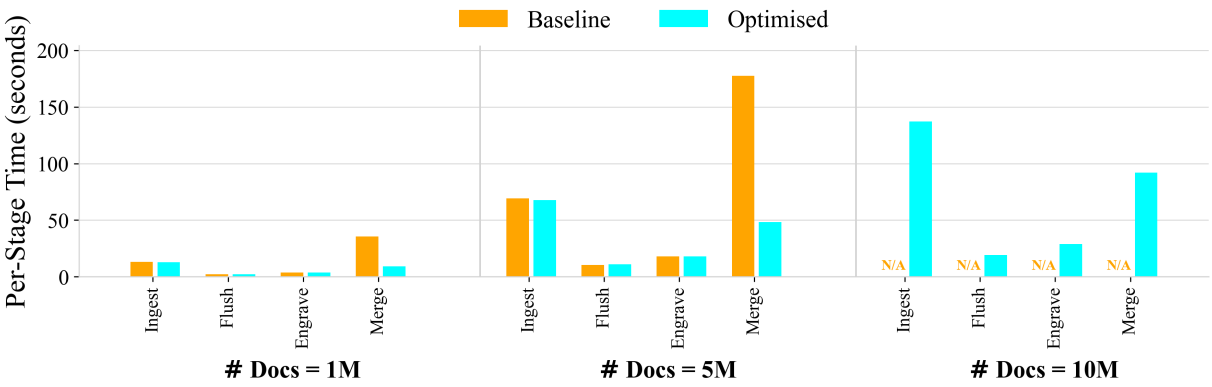


Figure 9.2: Breakdown of indexing pipeline time by four stages (ingest, flush, engrave and merge) of the pipeline, comparing SPIRIT's optimised and baseline crash consistency across different indexing workloads (**1M**, **5M**, **10M**). # Docs indicates the number of documents to be ingested for each indexing workload.

9.1.3 Recovery Time

Figure 9.4 shows the percentage change in average recovery time (– averaged across crash points, 30%, 50%, 70%, and 100%) when comparing the optimised crash consistency with the baseline. The percentage change in the average recovery time is negative for the two indexing workloads tested, i.e. indicating a reduction in the recovery time for the optimised version. A percentage change of -0.45% is observed for 1M, and -1.36% for 5M. The 10M workload is not compared due to a bug in the baseline SPIRIT implementation. The reduction time is attributed to the fact that *mLog* is updated during recovery to maintain crash consistency. This is true for both the baseline and optimised versions. Consequently, using fewer persist barriers to update *mLog* on recovery (in optimised), while merging the remainder of the terms in a segment, contributes to reduced recovery time. A large reduction in recovery time is not observed because the reported recovery time only reflects the time taken to complete an in-progress merge of a segment.

9.2 Evaluation of ARITHMETY

This section presents and analyses the results of the four workloads established in Chapter 5, Section 5.4, to evaluate how ARITHMETY performs across varying levels of pressure on PM.

9.2.1 Concurrent Indexing and Single-Threaded Querying

Figure 9.5 shows the query latencies over time for the Concurrent Indexing and Single-Threaded Querying (CIST) workload. We observe that past 11,000 queries, the average per-query latency exceeds the reaction threshold of 40 ms. Under the *Adaptive* configu-

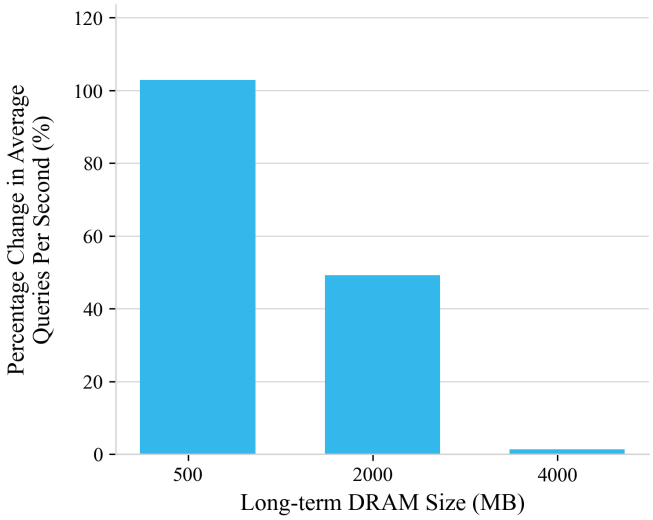


Figure 9.3: Percentage change in average queries per second (QPS), averaged across six query types, comparing SPIRIT's optimised and baseline crash consistency. DRAM size configurations are *tight* (500 MB), *medium* (2000 MB), and *loose* (4000 MB).

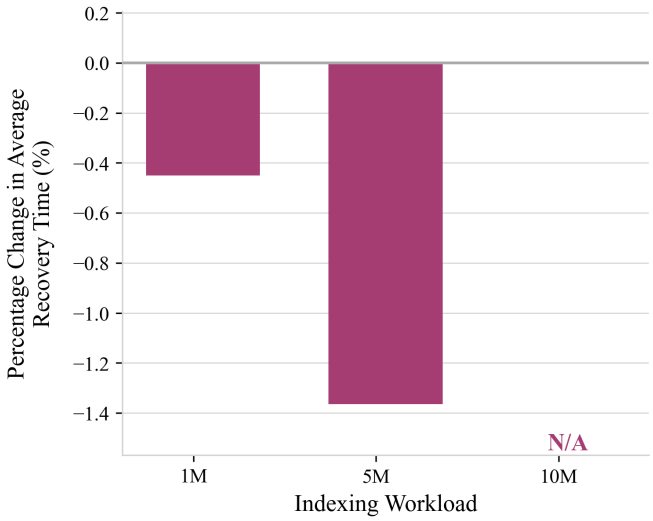


Figure 9.4: Percentage change in average recovery time, averaged across four crash points (30%, 50%, 70% and 100% of documents ingested), comparing the SPIRIT's optimised and baseline crash consistency for different indexing workloads (**1M** documents, **5M** documents and **10M** documents).

ration, where the client prioritises fast query responses, ARITHMETY adapts to reduce query latency to be below this threshold. A tail latency of 137 ms is observed at 33,100 queries. This can be attributed to 16 indexing instances contributing concurrent reads and writes to PM, thereby increasing PM pressure. The 8 (previously running) instances also result in a smaller peak of 52 ms in latency at 25,500 queries. ARITHMETY detects increased pressure in the system, and keeps the tail latencies within the 40 ms bound. This results in a 3.5× reduction in tail latency of 137 ms with the *Baseline* configuration.

Figure 9.6 presents moving window averages of query latency sorted by the Baseline. The difference in average latencies between Adaptive and Baseline, beyond a moving window index of 160 clearly shows that Adaptive effectively mitigates increasing pressure to favour faster query responses.

We calculate the average per-query latency across the 45,000 HH queries evaluated, to be 37 ms for Adaptive, and 49 ms for Baseline, representing a 1.3× reduction. This reduction is observed in a system where pressure on PM progressively increases due to an increasing number of concurrent instances. An average per-query latency of 37 ms aligns with our earlier analysis in Chapter 6. The analysis indicates that for the HH query type with a single query thread and no additional reads or writes to PM, the minimum attainable query latency is 37 ms.

There is a peak in query latency at the start (during the first query), which can be attributed to startup overheads, such as an initially empty CPU caches and translation lookaside buffer (TLB). The TLB, which caches recent virtual-to-physical page mappings, affects performance because the search index resides in PM.

In summary, the Baseline configuration prioritises strong crash consistency guarantees and so continues additional reads and writes to PM from indexing. ARITHMETY in Adaptive, however, reacts to mitigate PM pressure early on given the reaction threshold and increasing pressure in the system.

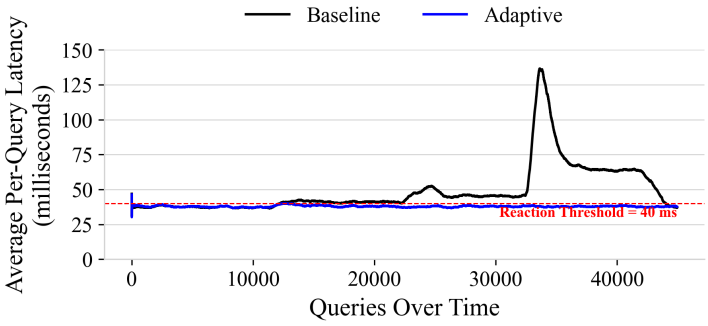


Figure 9.5: Query latencies over time for HH queries with a single query thread. Latencies are measured over the progression of 1 → 4 → 8 → 16 concurrent indexing-only SPIRIT instances. The reaction threshold is set to 40 ms.

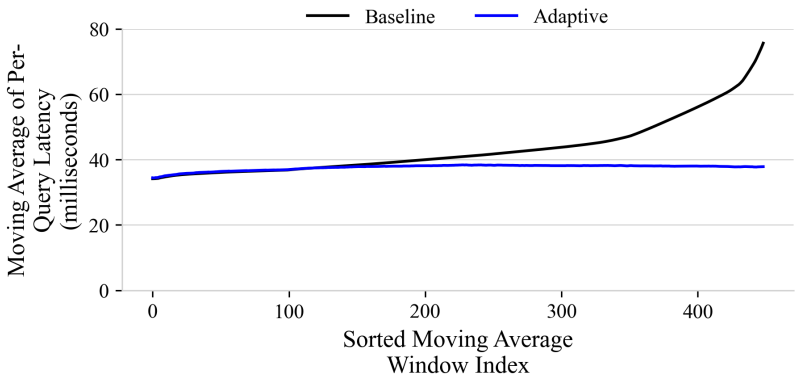


Figure 9.6: Moving window averages of query latency (window size = 100), sorted by the Baseline averages, for HH queries with a single query thread. Raw latencies are measured over the progression of $1 \rightarrow 4 \rightarrow 8 \rightarrow 16$ concurrent indexing-only SPIRIT instances.

9.2.2 Concurrent Indexing and Multi-Threaded Querying

Figure 9.7 shows a latency distribution over time, for 32 query threads. Overall, ARITHMETY detects increasing pressure in the system and reacts to reduce per-query latency past 9,500 queries. For the first 9,500 queries, latencies are similar to Baseline because detecting drift requires an increasing trend in latencies. Comparing the (sorted) Adaptive results in Figure 9.8 and Figure 9.6, we observe that ARITHMETY reacts past a window index of 160 (1 thread) versus 100 (32 threads). This means for 32 query threads action to mitigate pressure is taken earlier on. The difference occurs because 32 query threads exceed the reaction threshold immediately, while a single query thread reaches it only at 11,000 queries. Furthermore, the average per-query latency for multi-threaded querying is 1.6× higher than single-threaded. This aligns with our earlier analysis that multi-threaded querying increases the average per-query latency across all query types.

ARITHMETY under Adaptive, reacts to bound the tail latency at 58 ms, which is a 1.91× reduction in tail latency compared to the Baseline. This shows ARITHMETY's ability to detect high pressure and mitigate it. The tail latency appears at 39,100 queries versus 33,100 (1 query thread). We believe this to be because of higher bandwidth utilisation and thus, slower indexing (reads), which delays the 16 concurrent instances (and high PM pressure).

9.2.3 Query Type and Single-Threaded Querying

Figure 9.9 shows that under Adaptive with mixed query workloads, ARITHMETY reacts to reduce query latency only when it exceeds the reaction threshold of 40 ms (i.e., when HH queries are encountered). On average, we observe a 1.1× reduction in latency compared to the Baseline for HH queries. This improvement is significant for very slow,

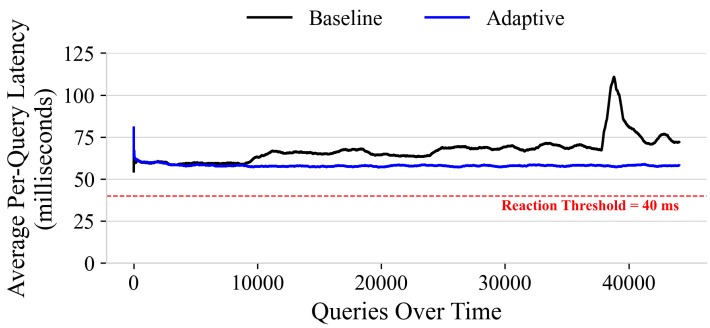


Figure 9.7: Query latencies over time for HH queries with 32 query threads. Latencies are measured over the progression of $1 \rightarrow 4 \rightarrow 8 \rightarrow 16$ concurrent indexing-only SPIRIT instances. The reaction threshold is set to 40 ms.

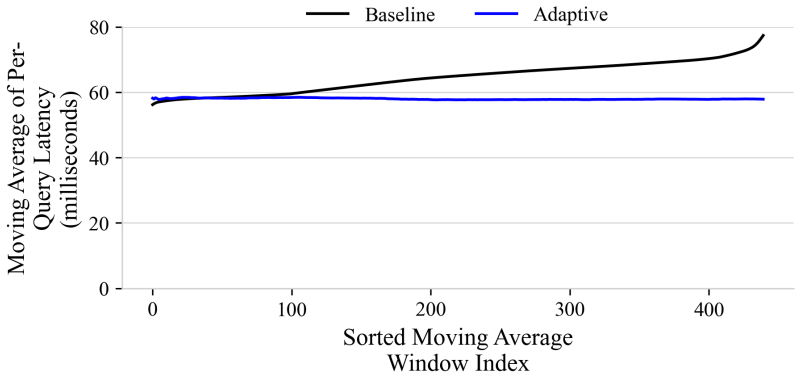


Figure 9.8: Moving window averages of query latency (window size = 100), sorted by the Baseline averages, for HH queries with 32 query threads. Raw latencies are measured over the progression of $1 \rightarrow 4 \rightarrow 8 \rightarrow 16$ concurrent indexing-only SPIRIT instances.

HH queries, when fast query responses are prioritised. The reduction can be explained by ARITHMETY temporarily stopping reads and writes to PM, incoming from concurrent indexing, to favour faster responses for slow queries. We observe that all query latencies (for H and HH) remain below 50 ms for this workload. This is lower than in the CIST workload because only 8 concurrent indexing instances are utilising the PM bandwidth, compared to up to 16 in CIST.

Importantly, even when fast responses are favoured, ARITHMETY resumes indexing during periods of low pressure on PM. Specifically, between 0–7,500 and 15,700–30,000 queries. When pressure is low, ARITHMETY maintains a similar, average latency of 4.5 ms, to Baseline (due to restarting additional reads and writes to PM), as clearly shown in Figure 9.10. The steep increase, and decrease in latency, around 7,500 and 15,700 queries, respectively, is caused by the shift in query workload from fast (H) queries to slow (HH) queries, and vice versa.

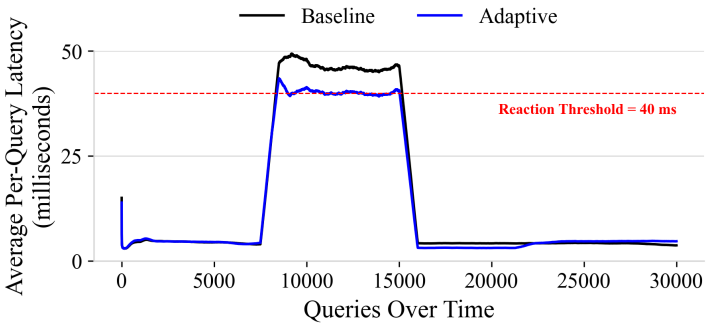


Figure 9.9: Query latencies over time for mixed queries (H and HH) with a single query thread. Latencies are measured with 8 concurrent indexing-only SPIRIT instances. The reaction threshold is set to 40 ms.

9.2.4 Query Type and Multi-Threaded Querying

Figure 9.11 shows a latency distribution over time, for 32 query threads that closely resembles the distribution of single-threaded querying in Figure 9.9. However, with 32 query threads, the average per-query latency is higher than for 1 query thread. The averages are 22 ms (32 threads) and 13 ms (1 thread), respectively, representing a 1.7× increase. This observation supports our earlier analysis, which shows that multi-threaded querying increases average per-query latency across all query types.

For 32 query threads, we observe a 1.3× reduction in average latency, under the Adaptive configuration between 7,500 and 15,700 queries. This reduction highlights that ARITHMETY effectively detects and mitigates PM pressure affecting HH (slow) queries. However, in Adaptive, the latencies for slower HH queries remain above the 40 ms reaction threshold, bounded at around 61 ms. Whereas, for the single-threaded case, latencies are bounded at around 40 ms. This observation is explained by our earlier analysis, which

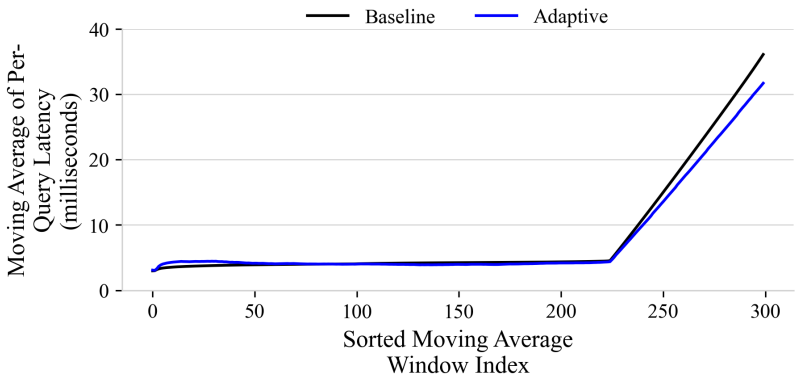


Figure 9.10: Moving window averages of query latency (window size = 100), sorted by the Baseline averages, for mixed queries (H and HH) with a single query thread. Raw latencies are measured with 8 concurrent indexing-only SPIRIT instances.

shows that for 32 query threads and HH query type, the minimum attainable average latency is 58 ms. The average per-query latency in ARITHMETY is higher than in the state-of-the-art SPIRIT, primarily due to its higher CPU utilisation impacting HH queries.

Overall, ARITHMETY effectively responds to high system pressure by reducing latency for slower queries, and when the pressure is low, it maintains latencies comparable to the Baseline of 15 ms. Figure 9.12 shows this behaviour, with latencies sorted relative to the *Baseline* latencies. We observe an increasing difference in latency averages between Adaptive and Baseline after a moving window index of 220.

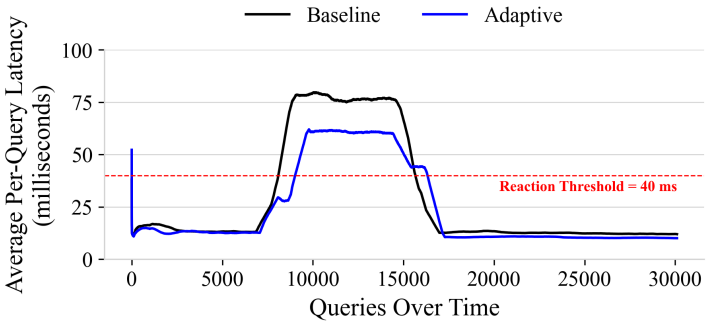


Figure 9.11: Query latencies over time for mixed queries (H and HH) with 32 query threads. Latencies are measured with 8 concurrent indexing-only SPIRIT instances. The reaction threshold is set to 40 ms.

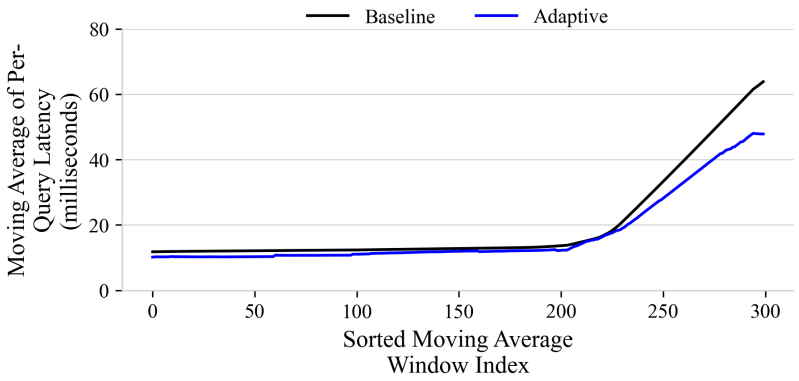


Figure 9.12: Moving window averages of query latency (window size = 100), sorted by the Baseline averages, for mixed queries (H and HH) with 32 query threads. Raw latencies are measured with 8 concurrent indexing-only SPIRIT instances.

9.3 Key Takeaways

The key takeaways from our evaluation of the crash consistency optimisation and ARITHMETY are as follows.

- Reducing the number of high-overhead, persist barriers (*clflush* and *mfence*) significantly reduces the overhead of writing to PM to guarantee crash consistency.
- It is possible to construct a system that can detect and respond to drifts in query latency when the client prioritises fast query responses over strong crash consistency or ingestion of new data.
- Variations in PM pressure can stem from different workloads that differ by the number of concurrent indexing (reads and writes to PM) instances, query type and number of query evaluation threads.
- ARITHMETY adapts to all four workload types, CIST, CIMT, QTST and QTMT.
- ARITHMETY promptly responds to PM pressure, ensuring that tail latency remains within the specified (reaction) bound or at the lowest attainable latency for the given query type.
- ARITHMETY effectively adapts by continuing indexing (reads and writes) and ingesting new content when PM pressure is low, and stopping writes under high pressure to prioritise serving slow queries and ensuring faster query responses.

Chapter 10

Concluding Remarks

10.1 Conclusion

In this thesis, we demonstrate that data-intensive applications, such as real-time search, can fully leverage hybrid DRAM-PM systems by utilising an optimised crash consistency mechanism and adapting to workload characteristics. The state-of-the-art proposal, SPIRIT offers strong crash consistency guarantees, however its merge time is affected by the overhead of *cflush* and *mfence* instructions used to persist recovery data. Our work addresses this limitation by reducing the number of these high-overhead instructions, and improves merge time by up to 4×. Furthermore, our analysis of pressure on PM shows that the average per-query latency increases by up to 2.9× under mixed reads and writes from concurrent indexing and query evaluation. In addition, the average per-query latency varies by the query type, with L queries being 61.7× faster than HH queries. ARITHMETY leverages these insights to monitor PM pressure and decide whether to maintain or mitigate it based on the client priority. When client priority favours fast query responses, our results show that ARITHMETY reduces the tail latency of HH queries by up to 3.5×. Additionally, ARITHMETY adapts to mixed query workloads by continuing indexing (writes) under low pressure, while maintaining up to 1.3× lower latency for slow, HH queries. ARITHMETY ensures that tail latency remains within the client-specified bound whenever possible, or otherwise, achieves the lowest attainable latency for a given query workload. Overall, our research concludes that mixed reads and writes degrade read performance (query latency) for a real-time, text-inversion search engine. It also concludes the need for adaptive systems that can balance performance trade-offs based on variations in pressure: optimising between fast query responses, strong crash consistency guarantees and freshness of data.

10.2 Future Work

We state several directions for future work to address ARITHMETY's limitations, and generalise our research to other storage technologies and applications.

10.2.1 Multi-Tenant Workloads

In our research, the analysis that underpins the design of ARITHMETY focuses on a single-tenant setup (one SPIRIT instance) serving user queries. This approach simplifies the analysis of PM pressure. However, the design and implementation of ARITHMETY can be extended to handle multiple tenants. Achieving this would require maintaining a separate latency buffer, per tenant, to detect drift in latencies (and PM pressure), and introducing analytical variables, e.g., to track the current query workload per tenant. Nonetheless, our analysis, design, and evaluation consider multiple query evaluation threads. This analysis reflects a real-world scenario, where multiple query evaluation threads run in parallel to serve user queries (Busch et al. (2012b)). Our results provide the insight that concurrently executing query threads increases PM bandwidth utilisation, leading to higher query latencies.

10.2.2 More Rigorous Sensitivity Analysis

In our research, we perform a sensitivity analysis of the reaction threshold and drift detection with respect to query latency. However, automatically determining the best value for parameters such as moving window size, latency buffer size, profiling frequency, and duration across arbitrary workloads is left for future work. Our evaluation selects these parameters to be best suited to our workloads.

10.2.3 Adaptive Ingestion

ARITHMETY mitigates PM pressure by halting engraving and merging operations of SPIRIT. Currently, when long-term DRAM space is exhausted, data ingestion is stalled, introducing a trade-off. While this approach preserves crash consistency (as no new data is in DRAM to be persisted), it compromises data freshness since new documents are not ingested. In adaptive ingestion, ARITHMETY would dynamically request additional DRAM when the client prioritises both data freshness and fast query responses. To extend ARITHMETY with adaptive ingestion is left for future work.

10.2.4 Generalisation to Other Storage Media

In our research, we focus on Intel Optane PM, however, this technology has been discontinued by Intel. Several alternatives offer promising solutions for secondary-tier memory. One such option is disaggregated memory using the Compute Express Link (CLX) interface, which provides a high-speed interconnect accessible across multiple CPUs. This technology can be leveraged to host inverted indices, enabling fast data transfers.

10.2.5 Generalisation to Other Data-Intensive Applications

We demonstrate that ARITHMETY effectively adapts to variations in PM pressure using SPIRIT. ARITHMETY is designed to monitor and adapt to variations in PM pressure and can be extended to other data-intensive systems that perform concurrent read and write operations, such as key-value stores. While we implement ARITHMETY's controller service to interact with and adapt SPIRIT's execution, its modular interface allows for easy extensibility to other systems. In the future, we will build a stand-alone API to plug the controller service into an arbitrary data-intensive application.

Bibliography

Akram, S., 2021. Exploiting intel optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21)* (Virtual, Canada, 2021), 14. ACM, New York, NY, USA. doi:10.1145/3459898.3463906. https://shbakram.github.io/assets/papers/ismm-2021-search.pdf. [Cited on pages 1 and 15.]

Banothu, N.; Bhukya, S.; and Sharma, K. V., 2016. Big-data: Acid versus base for database transactions. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 3704–3709. doi:10.1109/ICEEOT.2016.7755401. [Cited on page 9.]

Bleier, S., 2010. Nltk's list of english stopwords. https://gist.github.com/sebleier/554280. [Cited on page 30.]

Borisov, A.; Amit, N.; and Tsafrir, D., 2025. Batching with end-to-end performance estimation. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, 136–143. [Cited on page 52.]

Brewer, E. A., 2000. Towards robust distributed systems. In *PODC*, vol. 7, 343–477. Portland, OR. [Cited on page 54.]

Busch, M.; Gade, K.; Larson, B.; Lok, P.; Luckenbill, S.; and Lin, J., 2012a. Earlybird: Real-time search at twitter. In *2012 IEEE 28th International Conference on Data Engineering*, 1360–1369. doi:10.1109/ICDE.2012.149. [Cited on page 5.]

Busch, M.; Gade, K.; Larson, B.; Lok, P.; Luckenbill, S.; and Lin, J., 2012b. Earlybird: Real-time search at twitter. In *2012 ieee 28th international conference on data engineering*, 1360–1369. IEEE. [Cited on page 74.]

Coburn, J.; Caulfield, A. M.; Akel, A.; Grupp, L. M.; Gupta, R. K.; Jhala, R.; and Swanson, S., 2011. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39, 1 (2011), 105–118. [Cited on page 17.]

Elastic, 2025. Elastic enterprise search. https://www.elastic.co/elasticsearch/. [Cited on page 2.]

Fedorova, A.; Smith, K. A.; Bostic, K.; LoVerso, S.; Cahill, M.; and Gorrod, A., 2022. Writes hurt: Lessons in cache design for optane nvram. In *Proceedings of the 13th Symposium on Cloud Computing*, 110–125. [Cited on pages 2, 40, and 42.]

Gupta, S.; Oh, Y.; Yan, L.; Sutherland, M.; Bhattacharjee, A.; Falsafi, B.; and Hsu, P., 2023. Astriflash a flash-based system for online services. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 81–93. IEEE. [Cited on page 1.]

Hagy, M., 2020. Write-ahead logging for durable key/value persistence. https://dbfromzero.com/write_ahead_log.html. [Cited on page 10.]

Halavais, A., 2017. *Search engine society*. John Wiley & Sons. [Cited on page 5.]

Hasnat, A. and Akram, S., 2025. Spirit: Scalable and persistent in-memory indices for real-time search. *ACM Transactions on Architecture and Code Optimization*, 22, 1 (2025), 1–26. [Cited on pages 2, 9, 11, 12, 15, and 19.]

Hoseinzadeh, M. and Swanson, S., 2021. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 429–442. [Cited on page 17.]

Hu, Y.; Zhu, Z.; Neal, I.; Kwon, Y.; Cheng, T.; Chidambaram, V.; and Witchel, E., 2019. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM Trans. Storage*, 15, 2 (May 2019). doi:10.1145/3318159. https://doi.org/10.1145/3318159. [Cited on page 9.]

Hua, Y.; Huang, K.; Zheng, S.; and Huang, L., 2021. Pmsort: An adaptive sorting engine for persistent memory. *Journal of Systems Architecture*, 120 (2021), 102279. [Cited on page 17.]

Intel. Persistent memory development kit (pmdk). https://pmem.io/pmdk/. [Cited on page 16.]

Intel, 2025. Intel® performance counter monitor (intel® pcm). https://github.com/intel/pcm. [Cited on page 31.]

IntelCoporation, 2020. Optimizing write ahead logging with intel® optane™ persistent memory. https://www.intel.cn/content/www/cn/zh/developer/articles/technical/optimizing-write-ahead-logging-with-intel-optane-persistent-memory.html. [Cited on page 10.]

Izraelevitz, J.; Yang, J.; Zhang, L.; Kim, J.; Liu, X.; Memaripour, A. S.; Soh, Y. J.; Wang, Z.; Xu, Y.; Dulloor, S. R.; Zhao, J.; and Swanson, S., 2019. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714 (2019). http://arxiv.org/abs/1903.05714. [Cited on pages 7 and 8.]

Jiang, X.; Wei, H.; and Huang, Y., 2023. Tunable causal consistency: Specification and implementation. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, 169–176. IEEE. [Cited on page 2.]

Jiao, B.; Goel, A.; and Wang, A.-I. A., 2025. Silhouette: Leveraging consistency mechanisms to detect bugs in persistent {Memory-Based} file systems. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 407–423. [Cited on page 16.]

Kaiyrakhmet, O.; Lee, S.; Nam, B.; Noh, S. H.; and Choi, Y.-r., 2019. {SLM-DB}:{Single-Level}{Key-Value} store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 191–205. [Cited on page 15.]

Kannan, S.; Bhat, N.; Gavrilovska, A.; Arpaci-Dusseau, A.; and Arpaci-Dusseau, R., 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 993–1005. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/kannan. [Cited on page 15.]

Kassa, H. T.; Akers, J.; Ghosh, M.; Cao, Z.; Gogte, V.; and Dreslinski, R., 2021. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 821–837. USENIX Association. https://www.usenix.org/conference/atc21/presentation/kassa. [Cited on page 9.]

LeBlanc, H.; Lorch, J. R.; Hawblitzel, C.; Huang, C.; Tao, Y.; Zeldovich, N.; and Chidambaram, V., 2025. Power never corrupts: Tool-agnostic verification of crash consistency and corruption detection. *OSDI (to appear)*, (2025). [Cited on page 16.]

Lee, S. K.; Mohan, J.; Kashyap, S.; Kim, T.; and Chidambaram, V., 2019. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 462–477. [Cited on page 10.]

LenovoInc. Intel optane dc persistent memory – thinksystem server options. https://www.lenovo.com/us/en/p/servers-storage/servers/options/thinksystem-options/intel-optane-dc-persistent-memory/wmd00000388?srsltid=AfmBOop6MOyRhvjQWhLOvbv-1KSNtALMZPxkC1f9YbC7sbojj29ChT0N. [Cited on page 7.]

Lindsay, B.; PIRAHESH, H.; and SCHWARZ, P., 2005. Write-ahead logging. *Readings in Database Systems*, 17, 1 (2005), 355. [Cited on page 9.]

Manning, C. D.; Raghavan, P.; and Schütze, H., 2008. *Introduction to Information Retrieval*. Cambridge University Press. https://nlp.stanford.edu/IR-book/html/htmledition/processing-boolean-queries-1.html. [Cited on page 6.]

McCandless, M., 2021. Open-source collaboration, or how we finally added merge-on-refresh to apache lucene. https://blog.mikemccandless.com. [Cited on page 27.]

McLellan, P., 2020. Persistent memory at twitter. https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/pm-twitter. [Cited on page 8.]

Mead, M., 2003. History of costs of ram vs. hard drives vs. ssds. https://azrael.digipen.edu/~mmead/www/Courses/CS180/ram-hd-ssd-prices.html. [Cited on page 7.]

Memaripour, A.; Izraelevitz, J.; and Swanson, S., 2020. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 789–806. [Cited on page 17.]

Montgomery, D. C.; Peck, E. A.; and Vining, G. G., 2012. *Introduction to Linear Regression Analysis*. John Wiley & Sons, 5th edn. [Cited on page 50.]

Neal, I.; Reeves, B.; Stoler, B.; Quinn, A.; Kwon, Y.; Peter, S.; and Kasikci, B., 2020. {AGAMOTTO}: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 1047–1064. [Cited on page 10.]

Oliveira, J.; Gonçalves, J.; and Matos, M., 2025. Rethinking pm crash consistency in the cxl era. *arXiv preprint arXiv:2504.17554*, (2025). [Cited on pages 10, 11, and 16.]

Pillai, T. S., 2017. *Application Crash Consistency*. Ph.D. thesis, University of Wisconsin–Madison. [Cited on page 9.]

Schütze, H.; Manning, C. D.; and Raghavan, P., 2008. *Introduction to information retrieval*, vol. 39. Cambridge University Press Cambridge. [Cited on page 5.]

Shichao, n.d. Chapter 6. i/o multiplexing: The select and poll functions. https://notes.shichao.io/unp/ch6/. [Cited on page 48.]

Sun, J.; Qiu, Z.; Egan, R.; Ho, H.; Li, Y.; and Wang, Z., 2022. Persistent memory as an effective alternative to random access memory in metagenome assembly. *BMC bioinformatics*, 23, 1 (2022), 513. [Cited on page 8.]

Supiandi, Y. A close up view of a ram memory chip – cut out. https://www.vecteezy.com/png/55660717-a-close-up-view-of-a-ram-memory-chip-cut-out. [Cited on page 7.]

Thai, A. and Akram, S., 2022. Optimizing a hash table-based inverted index for persistent memory. *Proceedings of the 2022 ACM Conference on Information and Knowledge Management (CIKM)*, (2022). https://shbakram.github.io/assets/papers/comp3770-2022-S1-VSB.pdf. [Cited on page 6.]

Thomadakis, M. E. and Liu, J.-C., 1997. On the real-time service quality of posix support in solaris and irix. (1997). [Cited on page 53.]

Tomsic, A. Z.; Bravo, M.; and Shapiro, M., 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Proceedings of the 19th International Middleware Conference*, 120–133. [Cited on page 51.]

Wu, K.; Ren, J.; and Li, D., 2019. Architecture-aware, high performance transaction for persistent memory. *arXiv preprint arXiv:1903.06226*, (2019). [Cited on pages 2 and 16.]

Xue, D.; Huang, L.; Li, C.; and Wu, C., 2019. Dapper: An adaptive manager for large-capacity persistent memory. *IEEE Transactions on Computers*, 68, 7 (2019), 1019–1034. doi:10.1109/TC.2019.2899836. [Cited on page 17.]

Yang, J.; Kim, J.; Hoseinzadeh, M.; Izraelevitz, J.; and Swanson, S., 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 169–182. [Cited on pages 7 and 8.]

Ye, C.; Chen, M.; Jiang, Q.; and Wang, C., 2024. Hercules: Enabling atomic durability for persistent memory with transient persistence domain. *ACM Trans. Embed. Comput. Syst.*, 23, 6 (Sep. 2024). doi:10.1145/3607473. https://doi.org/10.1145/3607473. [Cited on page 16.]

Zhang, M.; Hua, Y.; Li, X.; and Xu, H., 2022. Scalable crash consistency for secure persistent memory. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 817–822. [Cited on page 10.]

Zheng, S.; Hoseinzadeh, M.; and Swanson, S., 2019. Ziggurat: A tiered file system for {Non-Volatile} main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 207–219. [Cited on page 15.]

Zobel, J. and Moffat, A., 2006. Inverted files for text search engines. *ACM Comput. Surv.*, 38, 2 (Jul. 2006), 6–es. doi:10.1145/1132956.1132959. https://doi.org/10.1145/1132956.1132959. [Cited on page 1.]

Zolenko, M., 2024. How does cheap ram differ from expensive one? https://e-catalog.com/post/5815/188-how-does-cheap-ram-differ-from-expensive-ram. Accessed: 2025-07-12; updated 14 November 2024. [Cited on page 7.]