SPIRIT: Scalable and Persistent In-Memory Indices for Real-Time Search

ADNAN HASNAT, Australian National University, Australia
SHOAIB AKRAM, Australian National University, Australia

Today, real-time search over big microblogging data requires low indexing and query latency. Online services, therefore, prefer to host inverted indices in memory. Unfortunately, as datasets grow, indices grow proportionally, and with limited DRAM scaling, the main memory faces high pressure. Also, indices must be persisted on disks as building them is computationally intensive. Consequently, it becomes necessary to frequently move on-heap index segments to storage, slowing down indexing. Reading storage-resident index segments requires filesystem calls and disk accesses during query evaluation, leading to high and unpredictable tail latency.

This work exploits hybrid DRAM and scalable non-volatile memory (NVM) to offer dynamically growing and instantly searchable (i.e., real-time) persistent indices in on-heap memory. We implement SPIRIT, a real-time text inversion engine over hybrid memory. SPIRIT exploits the byte-addressability of hybrid memory to enable direct access to the index on a pre-allocated heap, eliminating expensive block storage accesses and filesystem calls during live operation. It uses an in-memory segment descriptor table to offer: ❶ instant segment availability to query evaluators upon fresh ingestion, ❷ low-overhead segment movement across memory tiers transparent to query evaluators, and ❸ decoupled segment movement into NVM from their visibility to query evaluators, enabling different policies for mitigating NVM latency. SPIRIT accelerates compaction with zero-copy merging. It supports volatile, graceful shutdown, and crash-consistent indexing modes. The latter two modes offer instant recovery using persistent pointers.

SPIRIT with hybrid memory and strong crash consistency guarantees exhibits many orders of magnitude better tail response times and query throughout than the state-of-the-art Lucene search engine. Compared against a highly optimized non-real-time evaluation of Lucene with liberal DRAM size, on average, across six query workloads, SPIRIT still delivers 2.5× better (real-time) query throughput. Our work applies to other services that will benefit from direct on-heap access to large persistent indices.

CCS Concepts: • **Information systems** → **Search engine indexing**; **Main memory engines**; • **Hardware** → **Memory and dense storage**.

Additional Key Words and Phrases: Real-time search, hybrid memory, non-volatile memory, data-intensive

ACM Reference Format:
Adnan Hasnat and Shoaib Akram. 2024. SPIRIT: Scalable and Persistent In-Memory Indices for Real-Time Search. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2024), 25 pages. https://doi.org/10.1145/3703351

1 INTRODUCTION

Today, enabling fast real-time search over social media content is critical to the success of many enterprises, including Linkedin, Meta, and X. Social media content is either queried explicitly for *relevance search*, similar to static web content, or implicitly by the service for *timeline retrieval* to populate a user's home feed. The latter generates queries based on the user's preferred topics or followers and is more frequent on social media platforms — the source of 50% of tweets recommended

Authors' addresses: Adnan Hasnat, Adnan.Hasnat@anu.edu.au, Australian National University, Canberra, ACT, Australia; Shoaib Akram, Shoaib.Akram@anu.edu.au, Australian National University, Canberra, ACT, Australia.

1544-3566/2024/1-ART1
https://doi.org/10.1145/3703351

on X's `For You` and `Following` tabs [37]. The queries run concurrently with an indexing engine that builds indices in real-time, coping with a massive volume of data, e.g., 500 million tweets per day for X [37]. Consequently, real-time indexing puts high pressure on the aggregate CPU and memory of the real-time cluster, and concurrent queries exacerbate the pressure further.

The critical data structure search engines use for locating documents (web pages, social media posts, or tweets) matching a word (term) is an inverted index [49]. Offering real-time response times requires hosting the index in memory [14, 42]. Unfortunately, indices grow proportional to datasets, and large indices put pressure on DRAM. However, DRAM scaling cannot cope with the growth in datasets [40], increasing infrastructure cost [6, 40]. Scaling in-memory indices to large datasets demands dense memory technologies, complementing DRAM.

Hosting indices in memory to deliver low response times is also at odds with persisting them on storage. Popular search engines, e.g., Apache Solr and Elasticsearch, use a segmented index, and each fixed-size segment resides on the heap before being moved to the page cache. Segments are buffered in the page cache to amortize the I/O overhead. Ultimately, calling `fsync` to bulk-persist segments on storage, an operation called *flush* in Elasticsearch, becomes necessary to limit DRAM requirements and avoid losing significant index updates on a crash. Unfortunately, an `fsync` is expensive regarding CPU and device bandwidth [9], hurting tail latency. Existing systems either risk losing a considerable state or paying a performance penalty. Per-segment synchronized I/O preserves index updates proactively but under-utilizes device bandwidth. Finally, if the OS drops portions of the index from memory, reading the storage-resident segments later generates I/O. In prior art, SSDs struggle to deliver the tail response times required by real-time search [2].

A storage-centric design exposes the overheads of filesystem operations even to search queries resolved in memory. For example, popular real-time search engines do not make on-heap segments visible to query evaluators [9]. Instead, query evaluators use filesystem calls to access new segments from the OS cache or storage. (In Elasticsearch, one set of calls reads the commit point, and another reads the index segment.) These calls incur high overhead, as our evaluation with near real-time API of a popular search library [11] shows, prohibiting real-time operation [9].

Memory-to-storage transfers also require segment reformatting, which incurs a serialization cost. This reformatting makes graceful shutdowns and restarts, which are becoming increasingly common in data centers, slow and inconvenient. Finally, merging storage-resident index segments costs extra CPU and I/O overheads due to the need to sort individual segments and copy them out of place. A new hardware and software stack explicitly optimized for in-memory operation is needed for scalable real-time indexing and search.

Non-volatile memory (NVM) or persistent memory (PM) technologies, such as 3D XPoint (Intel) [3, 44], carbon nanotubes (Nantero) [12], and Spin-transfer Torque MRAM (STT-MRAM) devices (Everspin) [1], offer high density, potentially mitigating DRAM pressure. Their byte-addressability and persistent nature precludes expensive reformatting required for memory to storage transfers [7]. Filesystem calls turn into regular load/store accesses, albeit slower than DRAM [44].

We contribute **SPIRIT**, an engine for **S**calable, **P**ersistent, and **I**n-memory, **R**eal-time **I**nversion of **T**ext over large multicore servers with **hybrid** DRAM-NVM memory. We design it from first principles, including user-level management of split heap over a hybrid address space, a segmented (on-heap) persistent index architecture that is amenable to instant restart, a software-pipelined indexing engine with separate threads for ingestion, engraving (DRAM to NVM copy), and compaction via in-place merging, full visibility of index segments (at least one version, DRAM or NVM) to query evaluators through fast persistent pointers, mechanisms and policies for intelligently migrating index segments across memory tiers, and efficient utilization of (limited) DRAM capacity and NVM bandwidth. SPIRIT offers volatile, graceful shutdown, and crash-consistent indexing modes. It can recover the NVM index after a graceful shutdown or an abrupt crash.

The fundamental principle in SPIRIT is that a segmented index is hosted in a pre-allocated heap over hybrid DRAM-NVM memory. The on-heap segments are engraved (moved to persistent NVM) without excessive CPU overhead. Both in-flight DRAM and persistent NVM segments are exposed to the query evaluators through a global descriptor table (GDT). SPIRIT retains segments on the DRAM heap until it runs out of DRAM. When SPIRIT eventually drops segments from DRAM, the pre-engraved segments are visible to query evaluators transparently with a single pointer update in the GDT and lightweight synchronization. A specific heap partition stores the merged segment (MS) in persistent form, and each segment is eventually merged into it. SPIRIT provides users precise control over DRAM/NVM capacity caps.

SPIRIT exploits NVM's characteristics in various ways. It offers fast in-place merging, precluding expensive, out-of-place copying and preserving NVM bandwidth. Slow merging leads to index throttling in existing data-intensive systems [9, 17]. Meta-data sharing among merged segments further improves memory (NVM) efficiency. Both the DRAM and NVM segment formats in SPIRIT are amenable for on-heap storage. Thus, it avoids interacting with the filesystem API or kernel page cache during live operation. SPIRIT's architecture enables forming policies for making NVM segments visible based on the type of query workload and DRAM pressure. Finally, SPIRIT exploits NVM to offer crash consistency guarantees stronger than the state of the art.

Compared to the Lucene search engine library [11], SPIRIT improves the tail latency (P99) and QPS of popular disjunctive queries by 53× and 210×, respectively, and delivers 4× higher indexing throughput. Relaxing Lucene's real-time query evaluation by refreshing the index reader every 1000 queries still enables SPIRIT to deliver 2.4× faster P99 response times and 4.37× higher QPS. Other key results from our extensive evaluation of SPIRIT include: ❶ An unmerged NVM index delivers 50% (on average) of the QPS of an (ideal) DRAM-Only system (non-real-time). The slowdown is only 18% with a merged index due to meta-data sharing. ❷ Concurrent query evaluation slows ingestion by 35%. ❸ Crash consistency increases the cost of in-place merging by 10×, impacting query evaluation if DRAM is scarce. ❹ Background NVM operations (e.g., merging) slow down queries by 20%, indicating room for policies to manage NVM bandwidth better.

Overall, our work makes the following contributions:

- design and implementation of SPIRIT, a real-time search engine that hosts crash-consistent, persistent, indices in two-tiered, on-heap memory, simplifying the storage stack;
- mechanisms and policies for efficiently managing the index in hybrid memory, including fast index visibility for search without I/O, cross-tier movement transparent to query evaluators, in-place merging to improve speed and economy, and native support for crash consistency;
- extensive evaluation on real hardware, showing the tail latency, indexing throughput and QPS benefits of SPIRIT relative to state-of-the-art Lucene baselines, and results of sensitivity studies that open up new research problems in real-time search over hybrid memories.

2 RELATED WORK

Comparison to Lucene-based engines. SPIRIT is the first real-time search engine in literature to exploit DRAM-NVM hybrid memory for building large (persistent) indices. The widely used Lucene-based engines, Apache Solr [10] and Elasticsearch [9] flush the on-heap segments to storage using buffered I/O. Index visibility to query evaluators incurs extra overhead due to filesystem calls and kernel-specific policies for synchronizing page cache and storage. These systems do not use a second memory tier and put pressure on DRAM. In contrast, SPIRIT retains the index in (scalable) on-heap memory, and visibility is controlled via in-memory data structures. Similar to other engines, SPIRIT uses a segmented index but attains higher performance by operating on

segments in memory. Finally, Earlybird [37] is Twitter's real-time search engine based on Lucene, dividing the index into real-time (write-optimized) and immutable (read-optimized) segments.

NVM-backed inverted index in prior art. Prior work explores NVM and NVMe SSDs for hosting search indices [2]. They conclude that NVM for indexing alone is extremely slow, and the primary cause is backend operations, such as sorting, flushing, and merging. Further, tail latency is up to 1.5 seconds with Optane SSDs. We verify their results and observe that some queries take seconds even after warm-up. Their work motivates rethinking real-time search over hybrid memory, but their scope is limited to performance analysis. Another prior work uses performance counters to observe where queries spend their time, and their access patterns, motivating slow (scalable) memories for hosting search indices [6].

NVM-backed data stores. Recent work proposes NVM-based indices, including hash tables [15, 29, 30, 38] and B trees [34, 45], NVM-based key-value stores [4, 5, 13, 16, 18, 19, 25, 39, 46, 47], in-memory caches [20], and filesystems [23, 48]. They enable crash consistency, while few use NVM as a DRAM extension. Inverted indices have distinctive characteristics that demand NVM-optimized organizations. For example, inserts append new doc IDs to existing lists, and merging creates increasingly large-sized values (posting lists). Unlike SPIRIT, Tair-PMem [13] is a KV store that ingests directly in PM, but prefetches data into DRAM during query evaluation. Bullet [16] ingests in DRAM, but optimizes for PM writes. Other LSM-tree-based key-value stores [4, 18, 19, 25, 46] also establish NVM as a new tier in the storage hierarchy. NoveLSM [19] uses an NVM-backed Memtable, while SLM-DB [18] places a global index in NVM and maintains single-level SSTables, unlike traditional LSM. MatrixKV [46] exploits NVM for fine-grained column compaction. SpanDB [4] exploits heterogeneous storage to offer cost efficiency. Our work uses NVM for capacity and persistence, modifying the software stack to offer on-heap indices. SPIRIT uses DRAM and NVM-backed hash tables as index segments. Prior work optimizes hash tables for NVM [29, 30, 33, 38, 50]. They mitigate NVM writes, and some store meta-data in DRAM. Finally, KVell [26] is a key-value store aiming to saturate NVMe SSD bandwidth using direct I/Os, unsorted data on disk (precluding sorting and merging), and a shared nothing architecture.

3 BACKGROUND

We provide background on indexing and search using inverted indices, and hybrid memory.

Indexing and query evaluation. The key data structure search engines use for speeding up query evaluation is an inverted index [49]. It maps words (terms) to document locations. It consists of posting lists and a dictionary. Typically, the posting lists are stored in a postings file, and the dictionary is stored in a separate term dictionary file. A posting list contains postings, identifying documents as sorted integer IDs, and meta-data, such as term frequency and position. The term dictionary maps each term to an offset into the postings file where the posting list for that specific term starts. The mapping from term to postings list is kept in SSTable-like sorted files merged in the background as needed [22]. The basic idea is similar to a log-structured merge tree [35].

Evaluating search queries require finding document IDs matching query terms. The parsing step identifies query terms, including any conditional operators. Single-term queries are straightforward, while boolean queries use conjunctions (*AND*) and disjunctions (*OR*) operators. The next step is the dictionary lookup. The dictionary provides offsets into the postings file. Terms and offsets are stored in the same file sorted alphabetically. Since the sorting is alphabetical, terms are recursively split into prefixes and suffixes, and common prefixes are only stored once. Typically, the dictionary is a hash table or a key-value store. Once the offsets into the postings file are available, the next step performs set operations (multi-term queries) to find matching IDs. Posting list traversal is a linear function of the posting list size and index size.

The naive approach for evaluating an AND (intersection) query is to scan the posting lists from left to right and match candidate IDs across the lists. On a mismatch, the *naive* algorithm advances one of the lists past the maximum document ID seen so far. Faster approaches use binary or finger search [49]. Using skip lists to minimize comparisons across postings lists is also typical.

Hybrid memory. Hybrid memory consists of fast DRAM and high-capacity but slow secondary byte-addressable memory. The secondary tier is either remote (disaggregated [27, 28]) DRAM or another scalable technology [21, 24]. Although future technologies are possible, we consider NVM the second tier and use Intel Optane DC Persistent Memory (PM) as a real-world NVM prototype. It connects to the memory bus, similar to DRAM. It exhibits 10× lower latency than NVMe SSDs, and 2–3× higher latency than DRAM [44]. Media read/writes take place at 256-byte granularity, and 16 KB write combining buffer merges adjacent lines to mitigate high media latency [3, 44]. Small (< 256 B) and random PM accesses are slower than large and sequential ones. A lightweight filesystem abstracts PM media [41, 43].

4 DESIGN AND IMPLEMENTATION

4.1 Principles

The design of SPIRIT follows six principles.

Make indexed data instantly searchable. Ingest in DRAM to minimize indexing latency; make index updates instantly visible.

Operate nonstop from a user-space heap. At all times, retain the entire index in scalable, preallocated, on-heap DRAM-NVM memory, and avoid external memory allocators and block storage accesses for low-latency in-memory operation.

Perform macro-management. SPIRIT operates on segments (bulk persist and merge) with minimum locking.

Persist proactively but control visibility. NVM is fast, so segments persist immediately without write buffering, but visibility of NVM segments to query evaluators is delayed until necessary.

Maximize memory economy. Limit DRAM capacity via tunable parameter. Multiple NVM segments share metadata (e.g., term statistics) to preserve space; merge in place to preserve NVM capacity and bandwidth.

Allow multiple operational modes. SPIRIT runs in volatile, graceful shutdown, and crash-consistent indexing modes, enabling different performance-recovery design points based on needs.

4.2 Scope and Assumptions

Social media platforms maintain two search indices: ❶ real-time index serves recent content (e.g., seven days), and ❷ archive index serves all content to date. Memory-to-storage transfers in the real-time cluster occur when the system runs out of DRAM or when the index needs to be persisted. We focus on the real-time index, exploiting NVM to mitigate DRAM pressure and avoid block I/O, and also keeping the index persistent and crash-consistent in NVM. Archiving happens in a separate cluster [36], and we leave it to future work. When SPIRIT runs out of the NVM heap, index segments must be deleted for continuing ingestion, similar to services running out of disk capacity in servers today. Swapping NVM segments to disk is possible, but we leave it to future work.

4.3 High-Level Architecture

Figure 1 shows the high-level architecture of SPIRIT with both the memory (index) view (top) and the CPU (indexing pipeline) view (bottom). Each SPIRIT instance is an independent process (executor). Memory view includes the heap organization and index segments in four heap partitions. The multithreaded indexing pipeline consists of four stages interconnected by concurrent queues.

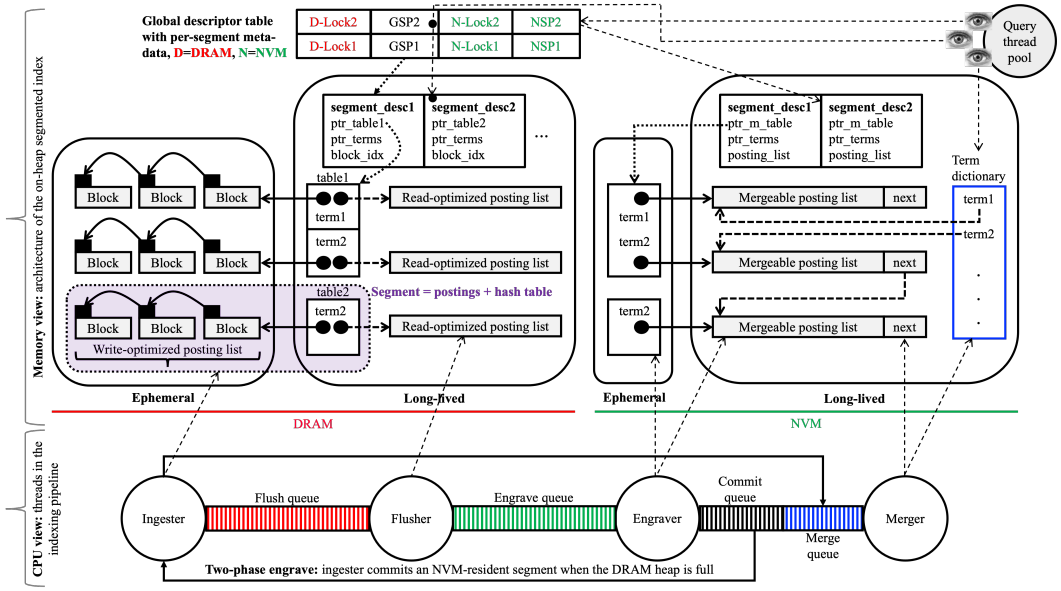


Fig. 1. The indexing pipeline of SPIRIT (bottom, CPU view), and the structure of the on-heap inverted index (top). Query evaluators (top right) are directed to index segments via the global descriptor table. They always follow the GSP, which points to either DRAM or NVM segment. In the rare case, a query evaluator follows NSP to mitigate DRAM pressure.

Ingestion begins in DRAM as populating new segments in NVM results in prohibitive indexing latency [2]. The data moves from left (DRAM) to right (NVM) in SPIRIT during indexing (Figure 1).

SPIRIT has three modes of operation: ❶ volatile mode, in which the index is not recoverable on a shutdown or a crash, ❷ graceful shutdown mode, in which the index is recoverable after a graceful shutdown, and ❸ crash consistent mode, in which the index is fully recoverable after a crash. The volatile mode is the fastest, followed by graceful shutdown, and then crash consistent mode. Depending on the required consistency requirements, a service provider can pick a suitable mode. Conveniently offering different operational modes is an advantage of byte-addressable NVM. We first discuss the index organization and indexing operation in volatile mode, then discuss indexing for graceful shutdown and crash consistent modes.

On-heap index organization. The heap is first divided into DRAM and NVM. Each sub-heap is further divided into ❶ ephemeral or short-term and ❷ long-term. The short-term DRAM heap stores postings as a linked list of blocks (write-optimized), while the long-term heap stores postings in a read-optimized manner. A (hash table) dictionary is associated with the posting lists stored in the long-term DRAM heap. The ingestion of fresh documents populates the short-term heap with postings and the long-term heap with the dictionary. (Segment is the combination of postings and dictionary and the unit of heap allocation.) Once a freshly ingested segment reaches a threshold size, it becomes immutable and is transformed from a write-optimized segment (WOS) format to a read-optimized (ROS) one. The DRAM-resident ROS is copied into NVM using direct access. After the copy, and appropriate updates to specific log entries, the segment is recoverable in the event of a crash. The short-term NVM heap stores the hash table for the NVM-resident ROS segment (pre-merging), and the long-term NVM heap stores the posting lists in a format amenable to in-place merging. Segments are eventually merged into a final merged persistent segment (MS). The

long-term NVM heap stores the dictionary for the MS. The MS dictionary can be placed in DRAM (space permitting) for faster access.

Document processing pipeline. At a high level, SPIRIT's indexing pipeline (Figure 1) consists of four threads ❶ **ingester** absorbs new documents into a front-end hash table, ❷ **flusher** transforms the hash table into an immutable, read-optimized segment, ready to be written to NVM ❸ **engraver** writes the flushed segment to NVM, and at some stage, depending on the specific policy, makes the engraved segment visible to the query evaluators and ❹ **merger** merges the engraved segments into the persistent index one after the other. Stages are single-threaded to simplify concurrency, and scaling up the service requires multiple executors.

Query evaluation pool. Separate from the indexing pipeline, a pool of query evaluator threads resolve queries (top-right of Figure 1). Since query evaluators access (read) immutable index segments, using a pool of query evaluation threads does not complicate synchronization significantly. A query evaluator has complete visibility over the entire on-heap index. This full visibility is a key strength of SPIRIT, as no off-heap access is required to service a query, obviating user-kernel boundary cross-over, both in terms of the CPU mode switch and off-heap memory accesses (OS page cache). A lightweight concurrency protocol switches the visibility of index segments across different formats, e.g., DRAM WOS to ROS, and different memory technologies, DRAM to NVM. This switching happens instantly and transparently to query evaluators.

Global Descriptor Table (GDT). GDT is a crucial data structure to enable complete index visibility to query evaluators. (See top-left of Figure 1). GDT is a linked list of segment descriptors that serve as *entrypoints* for searchable segments. Each GDT entry contains: ❶ Global Segment Pointer (GSP) points to *either* the descriptor of a DRAM segment in ingested, flushed, or engraved state; *or* the descriptor of an NVM segment in commit or merged state, ❷ NVM Segment Pointer (NSP) points to the segment descriptor of the engraved (not yet committed) segment (mainly for convenience and making specific policies that mitigate DRAM pressure), ❸ a DRAM lock (D-Lock) for locking the DRAM segment during query evaluation, and ❹ an NVM lock (N-Lock) for locking the NVM segment. The GDT is a pointer-based replacement for the *commit point* — a file that lists all known segments — in Elasticsearch.

4.4 Memory Management

In SPIRIT, all memory is pre-allocated. This pre-allocated memory includes volatile (DRAM) and non-volatile (NVM) heaps. NVM is exposed as a memory-mapped file. We use anonymous `mmap` for allocating the DRAM heap and file-backed `mmap` for allocating the NVM heaps. Pre-allocation precludes external allocator calls during live operation (critical for bounding tail latency).

Heap management. As discussed above, we divide the DRAM and NVM heaps into short-term and long-term partitions. The short-term DRAM heap stores postings as a linked list of blocks when the segment is fresh (before flushing). The data structures on the long-term DRAM heap survive the flush operation: ❶ hash table (dictionary), ❷ a sequential term log, ❸ segment descriptor, and ❹ contiguous posting arrays. The structures on the short-term heap are reclaimed after the flush operation, while ones on the long-term heap are reclaimed after the GSP is updated from the DRAM version of the segment to the NVM (permanent) version, i.e., when evicted from DRAM to be committed and merged into MS.

The structures on the short-term NVM heap include: ❶ a mergeable table (more on this later) and ❷ sequential term log. The long-term NVM heap contains the postings and the engraved segment descriptor. The segment data on the short-term NVM heap is recycled after merging. The MS has a hash table-based dictionary and an in-place merged list of postings.

The heap partition sizes impact indexing throughput and query latency. The short-term DRAM heap is sized to allow the ingester to allocate a new segment while the flusher transforms WOS into

ROS. A short-term DRAM heap setting of twice the segment size works best. The long-term DRAM heap affects the query latency. The shorter the long-term DRAM heap, the more the likelihood that query evaluators traverse the NVM heap. Since NVM is slower than DRAM, query evaluation is slower with limited DRAM. Regarding NVM heap sizing, if merging lags behind engraving, the short-term NVM heap is not recycled fast enough to keep the back end of the indexing pipeline running, eventually stalling the engraver. Most NVM belongs to the long-term NVM heap.

Segmented index architecture. The index is organized into several segments. The life cycle of a segment begins in DRAM as an in-memory hash table. A segment goes through different stages; as it goes through the stages, its state and organization change, both in DRAM and NVM. A freshly allocated segment consists of a bucket table, and each bucket contains the term and pointer to a posting block. A new segment uses a linked list of (fixed or variable-sized) blocks, and each block has a pointer to a list of pre-allocated postings. When the segment is full (according to a heuristic), the posting list is transformed into a read-optimized contiguous array structure with no internal pointers. When this transformation (flushing) is complete, the memory consumed by postings allocated on the short-term heap is recycled. Transparently to query evaluators, the bucket meta-data is updated to point to the read-optimized array. Once a segment is flushed, it becomes immutable and accepts no more updates.

Each segment can be in one of four states: ❶ ingested, ❷ flushed, ❸ engraved, ❹ committed, and ❺ merged. The state of each segment is stored in the least significant three bits of the GSP. Segments are engraved into NVM eagerly immediately upon flushing. An engraved segment has an NVM copy, but query evaluators see the DRAM version of the segment. Eventually, when SPIRIT runs out of DRAM, it frees the DRAM segment. The query evaluators now use the *committed* NVM segment for resolving queries. A committed NVM segment is a candidate for merging in MS.

Each segment has a unique monotonically increasing segment Id. A single document is never split across segments. This property simplifies query evaluation, especially intersection queries, as each segment is searched independently. On a system shutdown, the last segment ID is preserved.

Segment allocator. We use a custom allocator for efficient macro management of segments encouraged by allocating and freeing index segments in strict FIFO order. For example, in the ingestion stage, SPIRIT allocates posting blocks on the short-term DRAM heap that is recycled in the same order after flushing. Similarly, the data structures on the long-term DRAM heap are deleted in the allocation order. Circular heap allocators incur a small fragmentation when allocating a buffer bigger than the available space between the oldest allocated address and the heap end.

SPIRIT carefully allocates structures on the heap in strict segment order. For example, when creating a new segment, it pre-allocates on the long-term DRAM heap ❶ a specific number of bytes for storing a sequential term log and ❷ uninitialized postings. Before enqueuing the segment for flushing, it reallocates the log and postings since their exact size is now known. The FIFO property is not maintained if the flusher allocates the structures on the long-term DRAM heap instead.

Per-query allocation. Queries require memory for storing the result. We allocate memory up-front for the query evaluators on the long-term DRAM heap and reclaim it after evaluating each query.

4.5 Hash Table

SPIRIT uses a custom hash table to store index segments at different indexing stages and an MS dictionary. Both the freshly ingested segment and the MS dictionary are performance critical. Each query searches both segments. The MS dictionary is either volatile or persistent. Existing NVM-aware key-value stores incur high latency [8] when used as dictionaries. Our hash table uses open-addressing. Object or bulk chaining is ineffective as the term space is small. The hash bucket contains the term, a pointer to the posting lists, and some meta-data (e.g., total count).

The table on the ingestion side uses a linked list of a variable-sized block (VSB) of postings. A fixed-size block (FSB) incurs high search latency because traversing a long chain of blocks is costly. FSBs also incur high meta-data overhead due to additional links (pointers). Both FSB and VSB result in fragmentation. However, if we set the initial block capacity for VSB blocks to one, the fragmentation is minimized, and utilization is around 70%. The performance of VSB is always better than FSB, and searching the VSB is still slower than searching a contiguous array of postings.

The flushed and engraved hash tables are similar except for posting storage. In the flushed segment, the posting list is a contiguous array; in an engraved segment, it is a mergeable list.

4.6 Fresh and Flushed Segments

The initial DRAM segment is write-optimized for quickly writing new postings. The state of the segment is encapsulated in a segment descriptor. The main elements of this state are ❶ pointer to a hash table of buckets, ❷ pointer to a structure with all unique words encountered so far `term log`, ❸ a pointer to blocks of postings, and ❹ pointer to `blocks`. SPIRIT uses these pointers in the segment descriptor for convenience at various points during operation.

Each bucket has a pointer to the word. The reason `term log` is maintained separately is to store variable-sized words efficiently. The bucket contains a pointer to a linked list of blocks (pre-flushing) or arrays (post-flushing). Query evaluators traverse the appropriate structure by looking up the state of the ingested segment in the low-order bits of the pointer. Blocks store postings in a non-contiguous fashion (fast writes), and the array contains postings in contiguous memory (fast reads). Traversing the postings in blocks is slow, as it requires chasing pointers in the linked list of blocks. Flushing puts all blocks' postings together in a contiguous array to improve locality (read performance). Each block contains a postings array and a pointer to the next block.

4.7 Engraved and Mergeable Segment

The engraver transforms the DRAM segment (table of buckets) into an NVM segment (table of mergeable buckets). The key idea of a mergeable bucket is that postings from different segments can be linked/merged in place without requiring out-of-place copies each time a new segment is inducted into NVM. Specifically, a mergeable bucket has a pointer to the term and a structure called a mergeable posting list. A mergeable posting list has a pointer to the actual posting for a specific segment and a pointer to the following mergeable posting list belonging to a different segment. The mergeable posting list keeps the segment ID of a specific segment and other related meta-data.

Engraving. Engraving in SPIRIT is a two-phase process: ❶ **engrave:** transforming, copying, and persisting the DRAM segment to NVM, and ❷ **commit:** updating GSP in the GDT to point to the NVM segment. Committing occurs only when SPIRIT runs out of DRAM, and it is implemented in the context of the ingest stage (when the ingester cannot find DRAM space for a new segment).

Engraving follows our macro-management principle, i.e., copying segments in bulk favors NVM bandwidth. Engraver also creates a new mergeable bucket and posting list as required, fixing the pointers in the mergeable bucket to point to the correct word and posting lists. Finally, the engraver enqueues the engraved segment in the commit queue. We use `pmem_memcpy_persist` for a bulk copy. Following our principle of macro management, we free the segment in bulk post commit.

Eager Engrave, Lazy Pointer Update. The segment is engraved (i.e., copied to NVM) as soon as it is flushed. However, SPIRIT does not immediately update the global segment pointer in the GDT to the NVM segment descriptor. Instead, it uses the *eager engrave, lazy pointer update* policy. Specifically, as soon as the segment is flushed, it is engraved into NVM. However, the GSP points to the DRAM copy of the segment. The key idea is to resolve queries from DRAM segments until they are evicted from DRAM. When SPIRIT runs out of DRAM, it updates the GSP to point to the engraved NVM segment. Indeed, the design space is rich, and one could conceive other policies for

engraving and pointer updating: eager engrave, eager pointer update, lazy engrave, lazy pointer update, and even opportunistic engrave and lazy pointer update. Opportunistic policies help when the ingestion rates vary over time (e.g., diurnal patterns).

In-Place Merging. SPIRIT uses byte-addressable NVM to speed up merging, a critical bottleneck in existing engines [9, 31]. Merging two index segments requires concatenating their posting lists. Merging on traditional storage exhibits two inefficiencies: ❶ efficient merging demands term-sorted segments, and sorting incurs a high cost, ❷ merging requires out-of-place copying leading to a new segment file. SPIRIT links together posting lists in two segments with a single pointer update. The data structure that enables this merging without incurring any data movement is a mergeable posting list. Although one could conceive an in-place merge on block storage with some effort, it would lead to excessive page faults during query evaluation. Even with NVM, reading an in-place merged index results in random accesses, but their impact is less than random SSD or disk accesses.

Merged Persistent Segment (MS). All segments are finally merged into a merged persistent segment (MS). Merging is initiated after the segment descriptor reaches the head of a commit queue (see Figure 1). Recall that committing happens in the context of the ingester when SPIRIT runs out of DRAM. The merger takes a segment to be merged, reads terms in the segment, and checks to see if the terms already exist in the MS. If it exists, the merger appends the mergeable posting list of the candidate segment to the beginning of the linked list of postings. Merging posting lists from engraved segments involves only a single pointer update.

We explore three data structures to serve as the MS dictionary. We find the lookup latency of a Berkeley DB (BDB) dictionary and a pmemkv [8] storage engine to be high. Therefore, we use a DRAM or NVM-backed hash table-based dictionary. We place it in DRAM during live operation, copying it into NVM upon shutdown.

4.8 Graceful Shutdown and Recovery

So far, we have discussed indexing in volatile mode. Graceful shutdowns happen when applying binary updates and security patches and replacing devices. SPIRIT exploits NVM for fast, graceful shutdown and restart. On receiving a shutdown signal, SPIRIT stops ingesting documents, flushing in-flight segments, and merging them into the persistent index. Proper shutdown demands flushing CPU caches using special instructions. SPIRIT uses specialized *memcpy* operations that flush the relevant cache lines during engraving. Therefore, most segment data resides in NVM upon shutdown. Hash table-based MS dictionary is an exception, but other dictionaries, e.g., those based on the *pmemkv* library, are crash-consistent by design.

For a proper restart, SPIRIT persists meta-data in a recovery file: ❶ valid field to indicate if the recovery file is valid for restart, ❷ the last engraved document/segment ID, ❸ the tail pointer of the long-term NVM heap, and ❹ static index parameters such as dictionary type, segment hash table, and heap sizes. On recovery, SPIRIT copies the dictionary into DRAM (unless an NVM-backed dictionary is used) and recovers the index meta-data for restart. We use custom persistent pointers to store index offsets from the heap start instead of absolute virtual addresses to account for ASLR.

4.9 Crash Consistency

We now discuss building a crash-consistent index such that, after an abrupt crash, the index is fully recoverable upon restarting SPIRIT. A crash can occur while in-flight (ingested or flushed) segments are in DRAM or when a segment is being engraved from DRAM into NVM. Second, a crash can occur during merging. We combine undo/redo logs, persist barriers, and 64-bit atomic updates in Intel x86-64 ISA to make SPIRIT's index crash consistent. We allocate the persistent logs in NVM and control their sizes. A 64-bit log entry (e.g., log valid and commit bits) is written atomically, while multi-word log entries require a persist barrier (`clflust_opt + sfence`). When

running SPIRIT in crash-consistent mode, it is also feasible to gracefully shut down the service. We use a recovery file similar to graceful shutdown (GS) mode discussed in the previous section. This file contains metadata as in GS mode with the addition of the short-term PM heap pointers. In crash-consistent mode, we set the file's valid bit before ingestion begins, and allocate space for logs.

We use a transaction log (`tLog`) of new document URLs ingested but not yet engraved to redo the work upon recovery. The `tLog` is a circular log that contains 64-byte URLs. The ingester adds new URLs to the log tail (`tTail`). The head of the `tLog` (`tHead`) is adjusted by the engraver in bulk to account for all the documents in the engraved segment. When the tail and head meet, either the engraver is too slow, and ingestion stalls or there are no more documents to index. Two persist barriers are needed when writing new data at the tail (one for persisting URL, and one for the tail), and one persist barrier when moving `tHead` forward.

The engraver uses another circular log (`eLog`) that records engraved but unmerged segments and serves as the GDT and merge queue on recovery. On a successful merge of an engraved segment, the merger moves the head of the `eLog` (`eHead`) forward. The per-segment `eLog` entry contains the segment ID, short-term heap meta-data (i.e., the tail pointer of the heap before adding new data), long-term heap meta-data (i.e., the tail of the heap before adding new data), and the old `tHead` (before segment engraving). It also contains a valid bit to identify if this undo log entry is valid (i.e., a log entry is persisted properly without a crash). Furthermore, the `eLog` contains a persistent pointer to segment data (stored on a short-term heap), valid only if the segment is engraved. The commit field in the log identifies if the segment is successfully engraved. Two persist barriers are needed when initializing a segment's log entry, and one when moving `eHead` forward. On commit, first we write the state of the engraved segment, new heap meta-data, and other meta-data (e.g., last segment ID and DID), followed by a persist barrier. All this meta-data can also be used for graceful shutdown. We then write the commit field in the log, followed by a persist barrier.

The merger uses a log (`mLog`) that logs the merging progress on per-term and per-segment levels. In this way, SPIRIT offers stronger consistency guarantees than Lucene, which cannot recover partially merged segments. The `mLog` is divided into two portions. The first portion contains segment-level data: ❶ the segment ID being merged, ❷ `eHead` that is advanced if the merge is successful, ❸ short-term heap tail metadata (heap head is advanced as data becomes redundant during merge), and ❹ commit field (CMT) to inform SPIRIT if the segment is successfully merged. If CMT is 0, the entire log entry is invalid, and SPIRIT crashes at the start of the merge. If it is 1, this log entry is valid, and SPIRIT crashes during merging. If CMT is 2, this log entry is valid, and the merging of the segment finishes successfully (and `eHead` is up to date). The CMT field helps SPIRIT on recovery in repairing the persistent index. This portion of the log is recycled after a successful merge. Then, the other portion of `mLog` consists of ❶ the term ID in the segment being merged, ❷ the term found in the merged dictionary or not, and ❸ the old head pointer of the term's mergeable posting list in the dictionary, and commit entry for the individual term (TCMT). TCMT records if the log specific to the current term is valid or invalid and if the term has been successfully merged. Persist barriers are needed at per-segment log initialization and commit. Three persist barriers are needed for each term's merging. One at the start (TCMT is set to 0), one after looking up the term and setting TCMT to 1, and one after merging the term and setting TCMT to 2.

On recovery, SPIRIT reads the recovery file and first correctly recovers the logs, including the head and tail pointers. It also recovers general metadata (which may need to be repaired from logs). Then, it starts analyzing the logs to enable recovery. It first analyzes `mLog` and finishes merging the segment if one is in progress (CMT in `mLog` is 1). It then recovers the head of `eLog` as follows. If CMT is 1, finish merging and move the head of `eLog` forward to match the segment ID in both logs. If CMT is 0 or 2, the head in the `eLog` is correctly stored. Next, it goes over the `eLog` entries individually and checks for the commit fields in the log. If the commit field is 0, it repairs the heap

meta-data and old tHead to remove the partially engraved segment (docs to be reindexed from tLog). Otherwise, if the commit field is 1, it enqueues the segment into the merge queue and adds it to the GDT. Then, it recovers tHead. The correct tHead is stored in the eLog entry corresponding to the last valid entry. It then analyzes the tLog, and instructs the ingester to begin by re-indexing the documents in the tLog. Finally, indexer threads are restarted, and recovery is complete.

4.10 Scalability and Concurrency

SPIRIT's four-stage indexing pipeline is single-threaded to keep synchronization lightweight. Multiple indexing and query threads operating simultaneously on the index increase complexity. SPIRIT uses a query thread pool because they only perform read accesses.

Scaling SPIRIT. Scaling to large datasets and ingestion rates requires multiple executors. Each executor ingests a document subset. Integrating query evaluators in the same address space as the indexing pipeline gives them complete visibility of the on-heap index. Since executors do not share physical address ranges, scaling via multiple processes incurs no synchronization overhead.

Concurrency management. SPIRIT uses lightweight concurrency to manage concurrent index access. Write locking rarely blocks query evaluators. When SPIRIT runs out of DRAM, it must commit a segment residing in DRAM and NVM. SPIRIT uses write locking to ensure no query evaluator traverses the segment during the commit. As another example, the space occupied by merged segments must be reclaimed. It first changes the segment's state to merged, then checks to see if any query evaluator is reading the DRAM or NVM segments by acquiring a write lock on both, only then freeing both segments. Atomic pointer updates avoid race conditions when transitioning from write-optimized blocks to read-optimized arrays and introducing new postings.

4.11 Query evaluation

We discuss the evaluation of intersection queries. (Other queries are simpler to resolve.) SPIRIT's segmented index enables intersection on a per-segment basis. Also, document IDs are sorted, and an ID is guaranteed not to appear in multiple segments. This property enables optimizations, such as per-segment caching, compression, and skipping. SPIRIT resolves queries in five phases: ❶ read each segment's status bits and acquire DRAM/NVM locks in GDT, ❷ decide whether to read the segment pointed to by GSP (default) or NSP, ❸ identify segments that potentially have matching IDs, ❹ perform the list intersection. SPIRIT can handle deletes and updates similar to Lucene using per-segment bit vectors. Search queries can omit results marked as deleted in the bit vector.

Query evaluators only require read locks, and they release locks on a per-segment basis. In ❷, our existing heuristic favors DRAM segments and only reads NVM segments when committed. Indeed, more complicated heuristics that balance query speed and indexing throughput are possible. For ❹, SPIRIT implements multiple list intersection algorithms, including a naive approach (it visits every ID in all lists), binary search, and finger search. The benefit of more advanced algorithms depends on the corpus' specific properties and term frequency. Although SPIRIT can use a skip list in the MS to speed up segment advances, we find its utility low.

Query caching. For hot queries, SPIRIT uses an LRU query (result) cache of previously computed results. Constant ingestion in real-time search requires computing over the most recent index updates and simultaneously reusing old results. We opt for a result cache instead of a postings cache because, interestingly, an NVM-backed result cache improves QPS over no caching. We cache the intersection results (multi-term AND queries) but do not cache the postings of individual words. The cache contains meta-data that identifies newer segments in the index. SPIRIT computes over these newer segments the default way. Specifically, storing the largest segment ID observed by the cache informs the query evaluators of the segments that do not require retrieval. The cache is organized into buckets (one bucket per query) and posting blocks. We allocate blocks carefully to

minimize fragmentation. For example, queries with few results do not allocate an entire block. We cache results from committed segments only to avoid computations over partial segments. SPIRIT combines cached results with newly computed ones before responding to the user. The allocated space for the cache is divided between the hash table and the data blocks. Each cache bucket is approximately 350 bytes, allowing close to 3000 cache buckets in a 1 MB hash table.

5 EXPERIMENTAL METHODOLOGY

Implementation. SPIRIT is implemented in C/C++. It uses Intel's PMDK library for accessing NVM. We use Pthreads for multithreading and the `concurrent queue` from `oneAPI Threading Building Blocks` library for pipelining.

Platform. We use a dual-socket Dell PowerEdge R740 server running the Ubuntu 18.04.1 Linux OS (5.4.0 kernel). Each CPU is an Intel Xeon Gold 6252N (2.3 GHz) with 48 physical cores (96 logical) and 36 MB L3 cache. Each host iMC supports six memory channels. Each channel attaches to a 32 GB Micron DDR4 DIMM (400 GB total DRAM) and a 128 GB Intel Optane NVDIMM (1.5 TB total PM). We use `numactl` to bind executors to a socket. We use best practices for Optane PM use [44], including using PM in interleaved mode and avoiding NUMA accesses. We use PM in *App Direct* mode, exposing it to SPIRIT via a light-weight filesystem [41, 43].

Datasets and Ingestion. We use Wikipedia's English corpus from the Luceneutil website [32]. They provide a line-terminated (1 KB each) file of the corpus. We extract lines to individual files, which SPIRIT and our baseline Lucene reads from a storage directory. We evaluate SPIRIT using 1 M (default), 5 M, and 10 M documents. Our experiments require 64 GB capacity for hosting the index, using the default dataset and 16 executors, stressing DRAM and NVM.

Measurement Metrics and Methodology. We use execution time and documents ingested/merged per second to quantify the indexer's performance. Query throughput is quantified by queries per second (QPS) and the tail latency. We run search benchmarks in two ways: ❶ non-concurrent (**NC**) mode, with indexing and query evaluation in isolation to each other, and ❷ windowed real-time (**WRT**), in which both indexing and query evaluation happens concurrently, and we time queries in a selected window of the total execution. In WRT, we run the system long enough to index a fixed number of documents. For example, if we run SPIRIT with 1 M documents, then the query evaluators have enough queries to keep busy during the time it takes for the indexer to process 1 M documents. In WRT, we start our QPS measurement after ingesting a threshold (200 K) of documents. The intuition for WRT is that initially, the index is empty, and all queries, popular and others, are resolved instantly, inflating QPS.

Query Formation. We use two query types: ❶ single-word and ❷ multi-word conjunctive queries. Our workloads are homogeneous. We use terms from `topTerms20120502.txt` available on the Luceneutil website. The terms are divided into low (L), medium (M), and high (H) categories. We form a query workload suite consisting of six workloads: single-term L, M, and H; 2-term LL, MM, and HH. Each query workload consists of 100 K queries. We use a query thread pool for resolving queries, and a single query is resolved sequentially.

Default Parameters. Unless otherwise stated, we use the following default parameters. We index 1 M documents, measure QPS for 100 K queries, and use six query workloads. We vary the number of executors from 1 to 16 and use one query thread per executor. The segment size is 128 MB. The ephemeral and long-lived DRAM and NVM heaps are 1 GB, 5 GB, 5 GB, and 20 GB, respectively. We bound DRAM to the heap size, stressing DRAM and NVM in hybrid memory experiments. We evaluate different advanced algorithms for intersection queries (naive per-element comparison, binary search, and finger search) and use the (overall) fastest algorithm. We use the hash table DRAM-resident dictionary. In WRT mode, the window is sized as follows: first, the indexer ingests

800 K documents, and the QPS is measured while the queries execute concurrently with ingestion of 200 K additional documents.

Lucene and JVM Settings. Lucene is an industrial-strength search engine library [11] widely used in the real world. We use Lucene 9.6.0 running on top of OpenJDK 17 for comparison with SPIRIT. Lucene offers two index formats [6]: an on-heap uncompressed index, namely, DirectPostingsFormat (DPF), and an off-heap compressed (default) index, namely, LucenePostingsFormat (LPF). With LPF, compressed posting lists are read via memory-mapped I/O on demand into the OS page cache, and then decompressed and copied into the managed heap. DPF decompresses postings and other data structures into Java `byte` arrays and `int` arrays and stores them on the heap on start-up. The query evaluator resolves queries by directly accessing postings on the heap. The on-heap index is closer to our proposal, whereas the off-heap index is Lucene's default. We configure Lucene for an apples-to-apples comparison with SPIRIT, turning off early termination and query caching and using SPIRIT's reverse chronological ranking function. We use one executor and query thread and build Lucene's index with forced merging. We run each query workload five times to warm up the JVM and JIT. We use JVM's `allocateHeapAt` option to store the on-heap index in NVM. We use Linux cgroups to experiment with tight, moderate, and loose DRAM sizes. With DPF, DRAM is required for the OS page cache as the heap is mapped over NVM, and the index is stored on the heap. We use the default JVM's G1 GC and young/old heap settings to ensure the heap is large enough to fit the index. The off-heap Lucene index (LPF) requires DRAM partitioning between the OS page cache and the in-memory JVM heap. We even partition DRAM between the heap (default nursery) and OS cache. To reduce JIT warmup time, we use the OpenJDK compiler options -XX:-TieredCompilation and -server. These flags remove tiered compilation and use the strongest (server) compiler during JIT compilation, enabling a more even comparison between SPIRIT (C) and Lucene (Java).

Configurations for Evaluation. We evaluate and compare 22 hybrid memory/storage configurations, including nine SPIRIT and 13 Lucene configurations (see Table 1). We run most configurations with tight (T), moderate (M), and loose (L) DRAM. The SPIRIT configurations are distinguished according to modes, volatile (SP-V), graceful shutdown (SP-G), and crash-consistent (SP-C), all running in real-time. Next, we create two search applications using the Lucene near real-time (NRT) and non-NRT APIs, ❶ a real-time (similar to WRT setup described above) application that opens a directory reader while the index writer is concurrently adding documents to the index, and ❷ a traditional search application that evaluates queries against a fixed (static) snapshot of the index. The NRT application uses a parameter called *refresh* interval, which determines how frequently the reader is refreshed. We use a refresh interval of 1 (the reader is refreshed before each query), 10 (every 10 queries), 100, 1000, and infinity (refreshed once during initialization). These configurations are labeled as NRT-1, NRT-10, NRT-100, NRT-1000, and NRT-∞, respectively. We run Lucene NRT configuration without imposing DRAM limits (i.e., unlimited page cache and large JVM heap) to show real-time search performance in state-of-the-art Lucene without conflating storage technology impacts. Unlike Lucene's query evaluation, which uses memory-mapped I/O, Lucene's indexing is not optimized for Optane PM's direct-access (DAX) mode. Therefore, we run the remaining configurations against an NVM-backed static snapshot of the index we build in a separate step on SSD. In the three DAX configurations, we place the managed heap in DRAM and index in NVM. We also include two other Lucene configurations for comparison. The NODAX configuration uses Optane PM without the DAX mode (i.e., we mount ext4 without the DAX option). The SSD configuration is similar, except the index is backed by NVMe SSD instead of NVM. In NODAX and SSD, a DRAM division problem exists, i.e., DRAM must be divided between the managed heap and page cache. This division problem is complex and requires tedious hand-tuning. Therefore, we show the best result across thousands of experimental runs. The NRT, DAX, NODAX,

and SSD configurations use Lucene's default LPF posting format. Finally, we include (NVM-based) DPF configurations that use Lucene's DPF postings format, storing postings directly on the heap. DPF allocates the managed heap in NVM and uses DRAM as a page cache. Our configurations represent the myriad of ways search applications can exploit today's hybrid memory and storage.
DRAM Limit. The expectation is that with current DRAM scaling trends, we can only fit a certain percentage of the index in DRAM. ❶ *loose* equals index size, ❷ *moderate* equals 55% of index size, and ❸ *tight* equals 15% of index size. Thus, with 16 executors, DRAM equals 64 GB, 32 GB, and 8 GB. The index size with SPIRIT is 3.6 GB for the default dataset, approximately the same as the uncompressed (on-heap) Lucene index. The index size with Lucene's LPF format is 320 MB. However, we use the same DRAM sizes across all configurations we evaluate for a fair comparison.
Statistical Significance Our results are statistically significant. The coefficient of variation (COV) for (average) indexing time across eight experimental runs is 0.25%, 0.69%, and 0.38% for 1, 4, and 16 executors. In RT and WRT modes, the variation is higher, and the COV is up to 1.6%. For QPS, across all configurations, the COV is between 0.09% (HH, one executor) and 2.6% (M, 16 executors).

6 EVALUATION RESULTS

Using DRAM alone in a real-time cluster increases DRAM cost, but it is necessary today to avoid storage latency. *Our evaluation aims to establish if we can use NVM to mitigate DRAM pressure and still deliver real-time operation.* We first rigorously compare SPIRIT to widely used Apache Lucene search engine library [11]. We then characterize the indexing and query behavior of SPIRIT in non-concurrent and real-time modes, varying a range of parameters. Specifically, we ask the following key questions.

- How does SPIRIT compare to Lucene in indexing and query performance with in-memory and storage-resident indices?
- How slow is NVM-Only compared to DRAM-Only? What is the impact of DRAM size?
- Where does indexing spend most cycles, and what is the impact of storage technology and concurrent queries?
- What is the cost of enabling graceful shutdown and crash consistent modes on indexing throughput and QPS?

6.1 Comparison to Lucene

Tail and average latencies. We show the P50, P95, P99, and average latencies (microseconds) for our six query workloads and 22 search engine configurations in Table 1. Most importantly, we notice that the near real-time configurations of Lucene result in a very long tail latency. If we enable each query always to observe the most recent updates to the index (NRT-1), the P99 latency is more than 500 ms. The slowest SPIRIT queries (HH), on the other hand, only take 11.8 ms (tight DRAM) and 9.6 ms (loose DRAM).

As we increase the refresh interval in Lucene from one to 100 queries, the search results exclude the most recent updates to the index, but the P99 drops by 38×. For Lucene, reopening an index is extremely time-consuming as it involves traversing and opening multiple file descriptors and other operations. On the other hand, in SPIRIT, each query by design always observes the most recent updates to the index, as the query evaluator traverses all in-memory data structures. For the HH queries (loose DRAM), the best-performing SP-V, SP-G, and SP-C configurations of SPIRIT are 4× faster than NRT-100. We note that NRT-∞ that takes a snapshot of the index once after 800 K documents are ingested, and then changes to the index are ignored, is only 1.6× slower than SPIRIT. Overall, we conclude that Lucene's near real-time search is extremely slow compared to SPIRIT, even when the index is entirely stored in DRAM.

	L				M				H				LL				MM				HH			
	P50	P95	P99	Avg	P50	P95	P99	Avg	P50	P95	P99	Avg	P50	P95	P99	Avg	P50	P95	P99	Avg	P50	P95	P99	Avg
SP-V-T	26	41	52	27	59	137	211	68	228	1834	2126	398	42	85	116	43	215	426	599	226	1467	7421	11227	2245
SP-V-M	26	38	46	27	48	101	152	55	166	1061	1446	277	42	77	99	43	183	378	559	195	1369	6361	10502	2034
SP-V-L	26	36	42	27	41	75	109	46	125	1000	1094	213	42	72	85	43	159	334	507	174	1281	5601	9697	1878
SP-G-T	26	41	51	27	59	135	207	68	229	1829	2176	397	48	92	123	49	230	468	713	244	1721	8479	13020	2606
SP-G-M	26	38	46	27	48	101	152	55	165	1061	1463	273	48	80	103	49	192	403	603	207	1477	6576	11053	2174
SP-G-L	26	36	42	27	41	75	110	46	126	1005	1107	210	48	72	88	49	162	339	521	176	1282	5548	9505	1876
SP-C-T	67	91	105	65	96	172	247	105	258	1854	2197	425	114	177	210	111	289	511	719	301	1609	8003	11832	2410
SP-C-M	45	62	71	45	65	116	167	71	179	1074	1471	288	78	121	143	77	217	422	604	232	1436	6421	10476	2125
SP-C-L	28	36	41	28	41	75	108	45	126	1007	1109	211	48	71	87	47	158	331	509	172	1277	5549	9580	1868
NRT-1	42815	306709	542697	83717	45131	297782	477702	82816	63795	377984	577864	111173	47487	318163	545872	89286	51384	322060	556774	92507	61659	384500	564877	112931
NRT-10	358	53637	242252	10226	671	56869	281596	11354	2703	115079	362296	21704	811	61355	292364	12055	1420	74535	315342	14356	3976	120219	411704	21891
NRT-100	109	340	12849	1305	331	1028	29172	2028	1772	13164	80319	6862	230	720	25458	1854	597	1249	25846	2190	2822	9592	56248	7504
NRT-1K	67	154	223	229	208	663	1396	602	1663	12489	39404	5336	127	287	573	351	329	779	1346	769	2441	7708	16988	5114
NRT-∞	12	47	88	19	127	546	1044	219	1526	12086	35025	3456	27	56	108	33	185	515	830	225	2163	7011	14286	2930
DAX-T	11	51	102	19	157	713	1500	290	1884	24902	45781	5024	24	54	98	27	214	608	947	241	2252	8102	24595	3029
DAX-M	11	53	105	19	156	694	1446	248	1776	14349	44533	3783	25	56	100	27	223	625	976	245	2245	7661	14616	2918
DAX-L	11	53	110	17	155	701	1232	238	1462	12373	25403	3062	26	57	102	27	233	642	1005	266	2344	7660	13894	2882
DPF-T	146	448	717	167	1134	4277	7182	1651	7967	40797	58671	11598	170	431	582	172	183	641	1178	225	2928	13610	24914	4367
DPF-M	16	266	465	76	326	2351	5076	358	4002	25989	48293	8182	13	47	193	18	174	553	942	206	1845	6845	12226	2327
DPF-L	9	164	381	15	287	2300	5000	350	3697	25234	48000	5996	12	39	96	17	174	537	901	201	1840	6830	12003	2536
NODAX	8	43	87	16	134	614	1395	205	1507	10882	30731	3137	19	47	102	24	232	661	1069	264	2479	7878	12702	2964
SSD	9	45	92	16	132	619	1424	204	1301	8667	26119	2663	19	46	93	24	229	659	1114	259	2532	7803	12257	2996

Table 1. Showing the P50, P95, P99, tail, and average latencies (microseconds) for SPIRIT and Lucene.

L and LL queries are the fastest to resolve across all configurations. The P50, P95, P99, and average latencies for SP-V and SP-G are less than 50 μs, whereas for SP-C, we observe an increase in all latencies for tight DRAM. For example, the P99 for L queries increases from 52μs to 105μs. We see a similar increase for P50 that increases by 2.6×. Crash-consistent indexing slows down query evaluation, so optimizing this overhead in the future is important. If we give up the real-time search in Lucene and use NRT-∞, then for L queries, Lucene delivers, on average, an improvement in tail latency over SPIRIT. The P99, however, is still better with SPIRIT. Certain filesystem-related operations in Lucene make specific queries take much longer than average queries. The DAX, NODAX, and SSD configurations also deliver better or competitive tail latencies for L queries, but DPF results in much higher tail latency than SPIRIT.

For H queries, P50 with NRT-1 is 279× higher than SPIRIT (SP-V-T). On the other hand, NRT-∞ is 6.7× higher than SP-V-T. The remaining Lucene configurations also result in higher P50 latencies than SPIRIT. The results are similar for P95 and P99 latencies. SSD is the best-performing Lucene configuration, slightly better than NRT-∞ because it does not incur the overhead of concurrent indexing. On average, DAX and NODAX behave similarly.

The slowest queries across all configurations are HH queries. The P50 and P99 with SPIRIT (SP-C-L) are 1.3 ms and 9.6 ms, respectively, 45× and 233× higher than L queries, showing the wide variability in resolving search queries. Among the Lucene configurations, the fastest P99 response time is provided by DPF-L (around 12 ms). The tail latency with SSD is almost similar. As with other configurations, we observe that P95 is higher than P50 but lower than P99 by a substantial margin. This difference emphasizes the focus on optimizing the P99 tail response time. On average, Lucene is closer to SPIRIT regarding tail response time for HH queries. We note that all DAX (hybrid DRAM-NVM) Lucene configurations exhibit a higher tail latency than SPIRIT, but the DRAM heap makes a slight difference in response time.

We conclude that SPIRIT is much faster than Lucene for real-time search. For some Lucene configurations, L queries are faster in batch mode. Our tail latency results validate our premise that filesystem and related operations slow down query evaluation in existing real-time search engines.

QPS. We show the harmonic mean of QPS across six query workloads for all configurations in Figure 2. The SPIRIT configurations (left-most) in the figure deliver the highest QPS. With loose DRAM, all three SPIRIT modes deliver the same QPS. However, when DRAM is scarce, SP-V-T is

the fastest, followed by SP-G-T, and then SP-C-T. Crash-consistent indexing mode incurs a QPS loss of 12% relative to SP-V-T. Crash-consistent indexing slows down merging, and query evaluation over an unmerged index is slow due to many dictionary lookups. Tight DRAM heaps are slower than moderate and loose heaps.

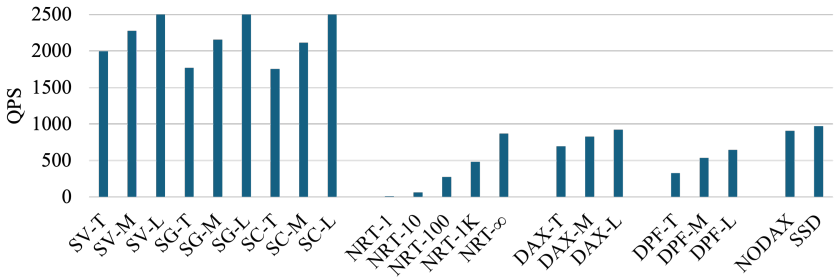


Fig. 2. Showing the QPS for all the SPIRIT (left-most) and Lucene configurations.

The NRT configurations deliver QPS many orders of magnitude lower than SPIRIT. The reason is the overhead of filesystem-related operations when reading the index segments. In SPIRIT, accessing the most recent index segments involves simple pointer manipulations over memory-resident data structures, the key point of this work. Opening the index reader once during initialization (NRT-∞) results in an 83× speedup over NRT-1, which opens the index reader before every query. Therefore, Lucene has a tradeoff between better QPS and returning the most recent documents to the user. For example, NRT-∞ cannot return any results containing the 200 K documents indexed after the index reader is opened. Nevertheless, even NRT-∞ is almost 3× slower than SPIRIT. Not surprisingly, the best-performing Lucene configuration is SSD, as Lucene is optimized for block devices. Fully crash-consistent SPIRIT delivers 2.6× better QPS than this fastest configuration. We also note that NRT-∞ is slower than SSD as NRT incurs the overhead of concurrent index writing.

The DAX configurations deliver better QPS with more DRAM, as more memory is available for the managed heap. The best-performing DAX configuration is still slower 2.7× slower than SPIRIT. From tight (DAX-T) to loose (DAX-L) DRAM heap, there is a 32% increase in QPS. Surprisingly, the NVM-based non-DAX configuration is, on average, 7% slower than SSD. We suspect non-DAX NVM accesses via page cache are not optimized and involve expensive memory copies. The best-performing DAX configuration with loose DRAM is faster than NODAX by a small margin.

The uncompressed on-heap postings format (DPF) also delivers lower QPS than SPIRIT. The best-performing DPF configuration is DPF-L, and it is 3.9× slower than crash-consistent SPIRIT (SP-C-L). It is also slower than LPF configurations (by up to 42%) because it maps the heap over NVM, and heap accesses are critical for fast response times. It uses DRAM as a page cache, but the coarse granularity of transfers between NVM and DRAM slows down search queries.

Indexing with SPIRIT versus Lucene. We compare the indexing throughput of SPIRIT and Lucene for similar datasets and parameters. Lucene's indexing is optimized for SSD, so we build the index in DRAM, and force a final commit and merge on storage. With one executor, SPIRIT ingests 2.5× more documents per second than Lucene. Other operations, such as flushing and engraving, happen concurrently, and does not slow down ingestion with proper memory and storage provisioning. We find that when running SPIRIT in volatile mode ❶ in-place merging of NVM-resident segments in SPIRIT is 6.74× faster than Lucene's out-of-place merging of SSD-resident segments, ❷ flushing and engraving (committing in Lucene) is 3.78× faster than Lucene. A final commit operation moves the segment to SSD. SPIRIT 's flushing is an in-DRAM operation, while its engraving is a DRAM-NVM copy. Despite our best efforts in a fair comparison, Lucene's

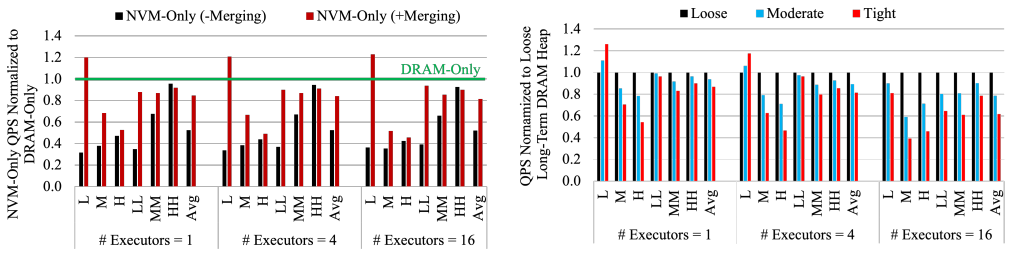
ingestion likely involves more expensive analysis of text documents than that of SPIRIT. We observe negligible GC overhead in Lucene indexing, using large DRAM.

Without a graceful shutdown, the index is not recoverable. The youngest segments are non-engraved, and the DRAM-resident dictionary is unrecoverable. CPU caches are also not flushed to NVM in our Optane PM generation. Indexing time incurs a tiny increase with recovery after graceful shutdown is enabled. This increase is only 1.3% due to persistent pointers (that store only offsets instead of virtual addresses) for managing the NVM heap. The overhead of persisting the hash table involves copying it from DRAM and ensuring it is flushed to NVM. This overhead is 3%. An NVM-resident dictionary speeds up flushing to NVM but increases the shutdown time by 1.6× due to slow merging. Recovery with SPIRIT is instant. The main overhead is mapping the long-term NVM heap into virtual memory, which requires reading metadata from the recovery file containing heap metadata and last IDs and copying the dictionary into DRAM. For 1 M and 10 M documents, the restart time is 224 ms and 900 ms, respectively.

We use default parameters to understand the overhead of crash-consistent indexing. The ingest, engrave, and merge threads manipulate log entries, thus incurring the cost of extra NVM writes and persist barriers. Compared to volatile mode, ingestion and engraving are 8% and 21% slower, respectively. Merging takes much longer than the baseline volatile mode, and the time to fully merge segments for the default corpus increases from 2 s to 34 s. The reason is that SPIRIT provides strong recovery guarantees compared to Lucene and similar data-intensive applications. Partially merged segments can be recovered. Each merge operation for a term in a segment (a single pointer update in volatile mode) requires at least three persist barriers in crash-consistent mode. This worst-case overhead can be optimized by relaxing the consistency guarantees.

6.2 Detailed Query Performance Analysis

We now evaluate SPIRIT's query behavior in detail, first reporting the QPS for DRAM-Only and NVM-Only in NC mode with and without merging, varying the executor count. We then show the impact of heap sizing on QPS in WRT mode, and the impact of query caching.



(a) NVM-Only QPS

(b) QPS Sensitivity to DRAM Size

Fig. 3. Showing the QPS with NVM-Only relative to DRAM-Only for single and multi-term query workloads (a), and QPS normalized to loose long-term DRAM heap for different heap sizes with merging (b).

DRAM-Only versus NVM-Only. We compare QPS with DRAM-Only and NVM-Only. For a fair comparison against DRAM-Only, we first disable merging in NVM, placing non-merged segments in DRAM or NVM. We execute queries after indexing and show the results of our evaluation in Figure 3(a). The figure shows the QPS with NVM-Only against a DRAM-Only baseline. (DRAM-Only is one, and higher is better.) We first focus on results without merging. We observe that across a different number of executors, on average, across all workloads, NVM-Only is 48% slower than DRAM-Only. Specifically, L queries are up to 68% slower with NVM-Only. The DRAM-NVM gap

opens up with single-term and MM queries. The HH queries are less sensitive to NVM latency (only 4% slower with NVM). We observe a trend: as the CPU overhead per memory access increases, the query workload is less sensitive to memory latency. Most of the time, L queries are so unpopular that they do not even traverse the posting lists and merely perform a dictionary lookup. Therefore, a single lookup is slower in NVM than in DRAM. At the other end of the extreme, the HH queries perform a compute-intensive intersection operation, and therefore, for each index (memory) access, they incur a substantial CPU overhead. The MM (32% slower) queries lie between the two extremes.

Impact of merging. We now discuss the normalized QPS of NVM-Only with merging enabled. Many non-merged segments increase the dictionary lookup latency because each non-merged segment is an independent index. With 16 executors, on average, across six query workloads, QPS with merging is 33% higher than no merging. We observe two surprising results in Figure 3(a). First, the QPS of L/LL queries with NVM-Only is even better than DRAM-Only (up to 23%). This better QPS is because the cost of many hash table lookups slows down L queries much more than the slow access latency of NVM. Second, HH queries are slower (4%) with merging compared to no merging. For HH queries, the cost of a dictionary lookup is only a fraction of the total query latency. Therefore, searching over a merged segment does not speed up HH queries. The slightly worst performance with merging is likely due to locality effects.

Heap sensitivity. Figure 3(b) (with merging enabled) shows the QPS of SPIRIT in WRT mode for different heap configurations. Specifically, we vary the long-term DRAM heap and observe the QPS. The long-term DRAM impacts QPS as SPIRIT retains segments in DRAM upon engraving and only makes them visible to query evaluators when it runs out of DRAM. The more the DRAM capacity available to SPIRIT, the better the QPS, especially for L, M, LL, and MM query workloads.

We observe that QPS is low with tight heaps, and this trend worsens with increasing executor count. With 16 executors, on average, the QPS of M queries is 61% lower compared to a loose DRAM heap (the whole index resides in DRAM). The HH queries are less sensitive to DRAM capacity and incur a slowdown of 10% with one executor and 21% with 16 executors. On average, a moderate heap slows query evaluation by 6% (one executor) and 21% (16 executors). On the other hand, a tight heap results in a QPS drop of 38% on average with 16 executors. These results show that DRAM heap sizing is critical for fast response times, and real-time search favors more DRAM capacity.

Time spent searching DRAM segments. We measure the time query evaluators spent traversing the segment in DRAM versus NVM. Across the query workloads and different executor counts, we find that ❶ with tight DRAM long-term heap, queries spend an average of 10% time in DRAM segments, and the rest (90%) in NVM, ❷ with moderate DRAM heap, queries spend 52% of the time in DRAM versus 48% in NVM, and ❸ with loose DRAM heaps, queries spend all of the time searching the DRAM segments.

Multi-Executor scalability. We show the scalability of query evaluation with increasing executor and thread count. Although SPIRIT can use a query thread pool, the expectation is that the incoming document stream is partitioned across multiple executors for better scaling of the ingestion and query evaluation pipeline. We run multiple executors and a single executor with multiple query threads and measure the QPS for three workloads (Figure 4). We observe in Figure 4 that QPS for L queries drops relative to using multiple executors with increasing thread count, and QPS with 16 threads is 50% of the QPS with 16 executors, each running with one query thread. This result shows that contention for the dictionary and the resulting synchronization hurts the query latency of L queries significantly. Each query evaluator spends less time reading individual segments and more time in synchronization because it performs more lock operations on segments. HH queries benefit from a query thread pool relative to multiple processes. QPS with 16 threads (one executor) is 31% higher than 16 executors. For HH queries, locality matters more because of the complex intersection they perform, and shared LLC helps multiple query threads significantly.

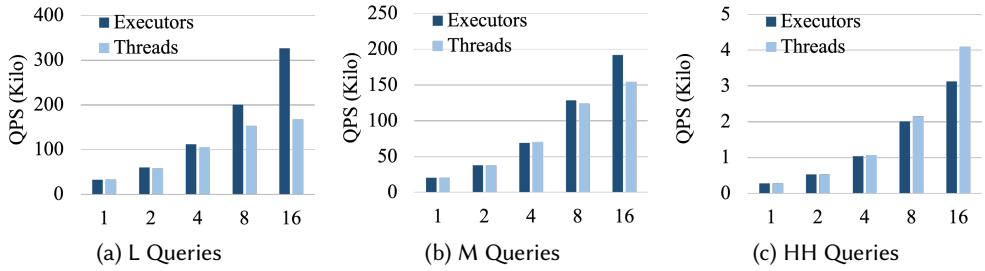


Fig. 4. Showing QPS scalability with multiple executors and threads.

Impact of query caching. With query caching, SPIRIT combines old and newly computed results over fresh postings to improve QPS. We also show the QPS impacts of query cache placement in DRAM and NVM. Figure 5(a) shows the QPS with a query cache of different sizes relative to no caching. We size the cache relative to the index size. We show QPS both with a DRAM-backed and NVM-backed cache. We note that depending on the cache size, QPS with a DRAM cache improves between 18% (2% of index size) and 48% (50% of index size). Beyond a specific size, QPS plateaus. The figure shows that a result cache can be backed by slower technology and improve performance. For example, backing the result cache by NVM improves QPS by up to 44% over no caching, and it is only slightly slower (up to 4%) over a DRAM cache. Overall, for hot queries, SPIRIT uses its result cache efficiently to improve their average latency.

Figure 5(b) shows the cache hit rate and the percentage of segments with cached and uncached results. We notice the hit rate of the cache lies between 26% (smallest cache size) and 60% (largest cache size) for our workloads. Next, we show the percentage of segments that require fresh computation. We notice counter-intuitively that the percentage of segments that require fresh computation increases as the cache size increases. The reason is that as the cache size increases, it can retain results for a greater number of queries. Some of the queries may have been cached a long time in the past. Consequently, when such a query arrives, a substantial number of new segments must be searched, creating a significant increase in uncached segments. This trend also explains the plateauing of QPS with increasing cache size. In our workloads, queries have a repetition frequency of approximately 15%. Increasing the repetition frequency to 25% makes the cache even more effective (70% improvement in QPS). Finally, assigning more memory to the hash table diminishes QPS returns. We use the optimal size of 1 MB.

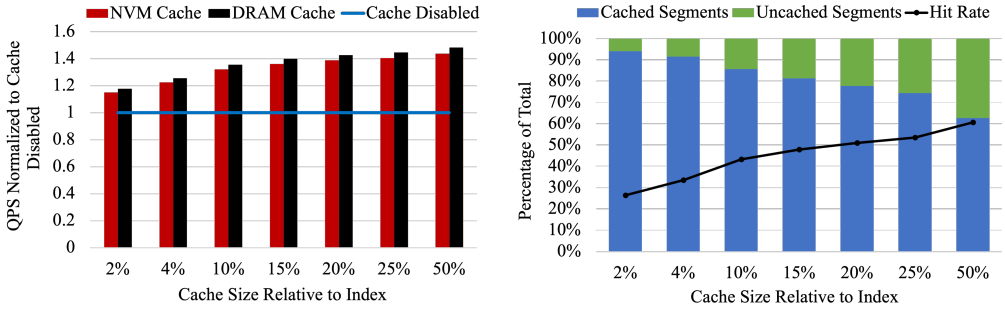
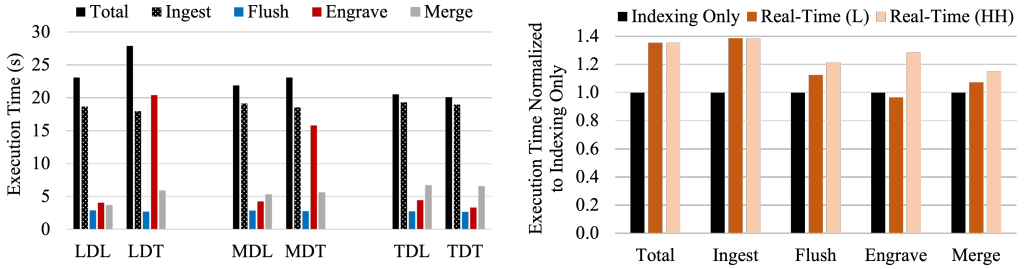


Fig. 5. Showing the QPS with NVM and DRAM-backed query cache of different sizes (a), and insight into cached versus uncached (computed) segments (b).

6.3 Detailed Indexing Performance Analysis

One executor indexes 72 K documents per second in NC mode. Indexing throughput scales with increasing executor count. With a loose heap, indexing throughput drops by only 30% (per executor) as we increase the executor count from one to 16 (50 K documents per second). The throughput drop is due to limited memory (especially NVM) bandwidth. Next, surprisingly, as we reduce the DRAM capacity (i.e., moderate and tight heaps), the indexing throughput increases (up to 18.4%). This effect is because more segments are left unmerged with a loose heap at the end of ingestion. Therefore, merging takes additional time to finish after ingestion. With a tight heap, merging begins early, and only a few segments require merging at the end of ingestion.

Figure 6(a) shows the sensitivity of indexing performance to different heaps and characterizes different pipeline stages (16 executors). The figure shows the total execution time and the time each stage is busy. (We normalize to total execution time.) We use three DRAM long-term heaps, loose (L), moderate (M), and tight (T), and two short-term NVM heaps, loose (L) and tight (T). In the figure, LDL represents a loose long-term DRAM heap and a loose short-term NVM heap. Heap size does not impact the pipeline's front end much, and we observe little change in ingest time across configurations. The total indexing time still decreases as DRAM gets tight. First, the merge time increases (1.82×) with tight DRAM as merging begins early, overlapping with the rest of the pipeline operation. With a loose DRAM, merging happens mostly in isolation (during shutdown). Second, with tight DRAM, loose and tight short-term NVM heaps lead to the same engraving time. However, with loose DRAM, engrave time increases dramatically (5×) from loose NVM (LDL) to tight NVM (LDT). With tight NVM, the engraver risks running out of NVM space to copy segments. As the long-term DRAM heap has more space for retaining segments, they are not committed and, hence, not merged. Therefore, unmerged segments occupy the short-term NVM space, eventually slowing down and blocking the engraver. These results again show proper heap sizing is important.



(a) Breakdown in Indexing Times (b) Impact of Concurrent Indexing

Fig. 6. Showing the total indexing time and its breakdown for different heap configurations (a), and the impact of concurrent query evaluation on indexing (b).

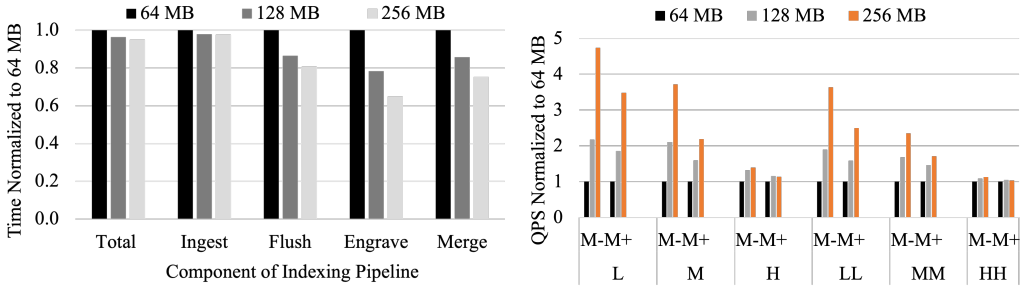
Figure 6(b) shows the increase in indexing time and the time different pipeline stages are busy when queries run concurrently in RT mode (16 executors). We observe an increase in indexing time of 35% with both L and HH queries. The ingester is the most sensitive to concurrent queries (L and HH), followed by the engraver (HH). Flushing and merging are only slightly impacted by RT mode.

6.4 Sensitivity to Segment Size

The unit of index management in SPIRIT is a segment. We evaluate the impact of segment size by running SPIRIT in RT mode with HH queries, one executor and query evaluator, and a DRAM dictionary. The total segments we observe (1 M documents) are 193, 80, and 34 for 64 MB, 128 MB, and 256 MB segments, respectively. Figure 7(a) shows the total execution time and the busy time for

different indexing stages and segment sizes. The total time is reduced by 4% and 5% with 128 MB and 256 MB segment sizes, respectively. Flushing, engraving, and merging all happen faster with larger segment sizes. More specifically, flushing is 19% faster with a 256 MB segment size, and merging is 25% faster. We observe that ingestion remains a bottleneck, and it is only 2% faster with a larger segment size, as the overhead of allocating new segments is amortized with larger segments. Flushing and engraving take less time because both require copying large buffers in DRAM or from DRAM to NVM, and this cost is amortized with larger segment sizes.

We show the impact of the segment size (normalized to 64 MB segments) on QPS (WRT mode) in Figure 7(b) with merging disabled (M-) and enabled (M+). QPS increases with segment size. With 256 MB segments, L queries, in particular, benefit from large segments. With fewer segments, L queries perform fewer dictionary lookups, and dictionary lookup is the main bottleneck for L queries. The QPS increases by 4.74× (M-) from 64 MB to 256 MB segments. With merging enabled (M+) results in a smaller increase in QPS (3.48×) because merging reduces the total number of segments L queries need to look up. We note that M and H queries benefit less from large segment sizes. From 64 MB to 256 MB segments, H queries observe a 1.14× increase in QPS (M+). This smaller increase in QPS is because H queries spend more time traversing posting lists than looking up the dictionary. The work required in posting list traversals does not change drastically with larger segment sizes. As before, disabling merging results in a larger number of segments, and hence, the benefit of larger segment sizes is more pronounced (1.39×). Multi-term queries follow a similar trend. The LL queries observe a 2.5× increase in QPS (M+), whereas HH queries observe only a 3% increase in QPS. Overall, we conclude that dictionary lookups observe a boost with larger segment sizes, but the cost of posting list traversal does not change much with larger segments.



(a) Impact on Indexing (b) Impact on QPS

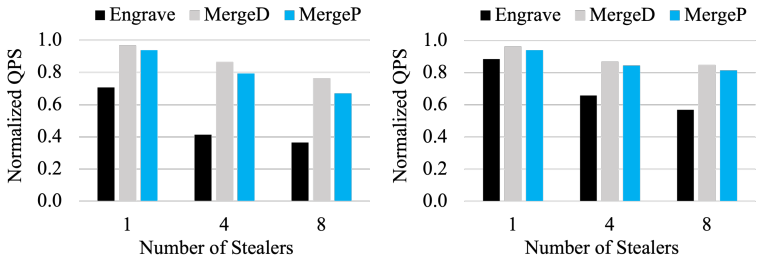
Fig. 7. Showing the impact of segment size on total indexing time, and time each stage is busy (a), and QPS with different segment sizes normalized to QPS with 64 MB segments (b).

6.5 QPS Impact of Limited NVM Bandwidth

NVM has limited bandwidth, and any future persistent storage with direct access is unlikely to match DRAM bandwidth. We analyze the impact of limited NVM bandwidth on QPS in two ways: ❶ impact of engraving on QPS, and ❷ impact of merging on QPS. In SPIRIT, engraving, merging, and query evaluation happen concurrently. Engraving is DRAM-NVM data movement, while merging is in-NVM movement. To isolate the impact of engraving/merging, we run two SPIRIT instances concurrently: ❶ *victim* process runs queries in NC mode, and ❷ *stealing* process builds a separate index and performs either engraving or merging concurrently with the victim. The stealer steals NVM bandwidth from the victim. We time the QPS for the victim instance. Figure 8 shows the results of the L and HH query workloads. We normalize QPS to a NC baseline. We synchronize the victim and stealing instances so that when the victim executes queries, the stealing process either engraves or merges segments. We observe in Figure 8 that for L queries that are usually short and

more sensitive to NVM bandwidth, engraving during real-time operation hurts QPS significantly (29% slowdown). As the number of stealing instances increases, engraving hurts QPS even more (up to 63% slowdown). Concurrent merging also induces a slowdown of up to 24% (eight stealing instances). Placing the dictionary in NVM further slows query evaluation by another 9%.

The HH queries are compute-intensive and less sensitive to NVM bandwidth, spending significant time comparing IDs across posting lists. Still, with eight stealers, engraving lowers QPS by 43% and merging (NVM-backed dictionary) by 19%. These results motivate opportunistic policies for engraving and merging to maximize QPS.



(a) L Queries (b) HH Queries

Fig. 8. Showing QPS (normalized to NC) with concurrent stealing processes.

7 CONCLUSION

Non-volatile memory (NVM) unlocks the potential of hosting large and persistent in-memory indices for data-intensive applications. We have presented SPIRIT, a real-time search engine that leverages NVM for mitigating DRAM pressure and simplifying the storage stack. The concepts SPIRIT relies on can be generalized to more applications: hosting persistent indices in on-heap memory and controlling the visibility of index segments to query evaluators using simple pointer-based structures. These features deliver several orders of magnitude faster tail latency and query throughput than existing baselines. We conclude that NVM saves DRAM capacity in a hybrid setting. In addition, moving index segments from DRAM to NVM via direct access helps SPIRIT offer stronger crash consistency guarantees than existing frameworks.

ACKNOWLEDGEMENT

We thank reviewers for their detailed feedback, Angus Atkinson for contributing towards the early prototype of SPIRIT, and Chethin Weerakkody for the query cache in SPIRIT.

REFERENCES

[1] S. Aggarwal, H. Almasi, M. DeHerrera, B. Hughes, S. Ikegawa, J. Janesky, H. K. Lee, H. Lu, F. B. Mancoff, K. Nagel, G. Shimon, J. J. Sun, T. Andre, and S. M. Alam. 2019. Demonstration of a Reliable 1 Gb Standalone Spin-Transfer Torque MRAM For Industrial Applications. In *IEEE International Electron Devices Meeting (IEDM)*. https://doi.org/10.1109/IEDM19573.2019.8993516

[2] Shoaib Akram. 2021. Exploiting Intel Optane Persistent Memory for Full Text Search. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. https://doi.org/10.1145/3459898.3463906

[3] Shoaib Akram. 2021. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Trans. Archit. Code Optim.* 18, 3, Article 29 (Apr 2021). https://doi.org/10.1145/3451342

[4] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast21/presentation/chen-hao

[5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378515

[6] Aditya Chilukuri and Shoaib Akram. 2023. Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. https://doi.org/10.1145/3591195.3595272

[7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/1629575.1629589

[8] Intel Corporation. 2021. pmemkv. https://github.com/pmem/pmemkv

[9] Elastic. 2021. Elastic Enterprise Search. https://www.elastic.co/elasticsearch/

[10] The Apache Software Foundation. 2021. Apache Solr 8.8.2. https://solr.apache.org/

[11] The Apache Software Foundation. 2021. Welcome to Apache Lucene. https://lucene.apache.org/

[12] Bill Gervasi. 2019. Will Carbon Nanotube Memory Replace DRAM? *IEEE Micro* 39, 2 (2019), 45–51. https://doi.org/10.1109/MM.2019.2897560

[13] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, and Feifei Li. 2022. Tair-PMem: A Fully Durable Non-Volatile Memory Database. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3346–3358. https://doi.org/10.14778/3554821.3554827

[14] Siddharth Gupta, Yunho Oh, Lei Yan, Mark Sutherland, Abhishek Bhattacharjee, Babak Falsafi, and Peter Hsu. 2023. AstriFlash A Flash-Based System for Online Services. In *IEEE International Symposium on Inverted Files Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA56546.2023.10070955

[15] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent memory hash indexes: an experimental evaluation. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 785–798. https://doi.org/10.14778/3446095.3446101

[16] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc18/presentation/huang

[17] Yichen Jia and Feng Chen. 2020. From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS48437.2020.00034

[18] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc18/presentation/kannan

[20] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc21/presentation/kassa

[21] Kinam Kim. 2008. Future memory technology: challenges and opportunities. In *2008 International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*. 5–9. https://doi.org/10.1109/VTSA.2008.4530774

[22] Martin Kleppmann. 2017. *Designing Data-Intensive Applications*. O'Reilly, Beijing. https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/

[23] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132770

[24] Stefan Lai. 2008. Non-volatile memory technologies: The quest for ever lower cost. In *2008 IEEE International Electron Devices Meeting*. 1–6. https://doi.org/10.1109/IEDM.2008.4796601

[25] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proc. VLDB Endow.* 15, 13 (2022), 4023 – 4037. https://doi.org/10.14778/3565838.3565854

[26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. 447–461. https://doi.org/10.1145/3341301.3359628

[27] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/1555754.1555789

[28] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on Inverted Files Comp Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2012.6168955

[29] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134
[30] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2021. Scaling Dynamic Hash Tables on Real Persistent Memory. *SIGMOD Rec.* 50, 1 (jun 2021), 87–94. https://doi.org/10.1145/3471485.3471506
[31] Michael McCandless. 201. Visualizing Lucene's segment merges. https://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html
[32] Michael McCandless. 2021. Luceneutil: Lucene benchmarking utilities. http://blog.mikemccandless.com
[33] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast19/presentation/nam
[34] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. 371–386. https://doi.org/10.1145/2882903.2915251
[35] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (jun 1996), 351–385. https://doi.org/10.1007/s002360050048
[36] Twitter. 2023. Tweet Search System (Earlybird). https://github.com/twitter/the-algorithm/blob/main/src/java/com/twitter/search/README.md
[37] Twitter. 2023. Twitter's Recommendation Algorithm. https://blog.twitter.com/engineering/en_us/topics/open-source/2023/twitter-recommendation-algorithm
[38] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (2022).
[39] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc22/presentation/wang-jing
[40] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 609–621. https://doi.org/10.1145/3503222.3507731
[41] Matthew Wilcox. 2014. Add Support for NV-DIMMs to Ext4. https://lwn.net/Articles/613384/
[42] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/ICDE.2018.00064
[43] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132761
[44] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang
[45] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang
[46] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc20/presentation/yao
[47] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3447786.3456237
[48] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 207–219. https://www.usenix.org/conference/fast19/presentation/zheng
[49] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. *ACM Comput. Surv.* 38, 2 (jul 2006), 56 pages. https://doi.org/10.1145/1132956.1132959
[50] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi18/presentation/zuo

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009