Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines

Aditya Chilukuri
aditya.chilukuri@anu.edu.au
Australian National University
Canberra, ACT, Australia

Shoaib Akram
shoaib.akram@anu.edu.au
Australian National University
Canberra, ACT, Australia

Abstract

Managed search engines, such as Apache Solr and Elasticsearch, host huge inverted indices in main memory to offer fast response times. This practice faces two challenges. First, limited DRAM capacity necessitates search engines aggressively compress indices to reduce their storage footprint. Unfortunately, our analysis with a popular search library shows that compression slows down queries (on average) by up to 1.7× due to high decompression latency. Despite their performance advantage, uncompressed indices require 10× more memory capacity, making them impractical. Second, indices today reside off-heap, encouraging unsafe memory accesses and risking eviction from the page cache.

Emerging byte-addressable and scalable non-volatile memory (NVM) offers a good fit for storing uncompressed indices. Unfortunately, NVM exhibits high latency. We rigorously evaluate the performance of DRAM and NVM-backed compressed/uncompressed indices to find that an uncompressed index in a high-capacity managed heap memory-mapped over NVM provides a 36% reduction in query response times compared to a DRAM-backed compressed index in off-heap memory. Also, it is only 11% slower than the uncompressed index in a DRAM heap (fastest approach). DRAM and NVM-backed compressed (off-heap) indices behave similarly.

We analyze the narrow response time gap between DRAM and NVM-backed indices. We conclude that inverted indices demand massive memory capacity, but search algorithms exhibit a high spatial locality that modern cache hierarchies exploit to hide NVM latency. We show the scalability of uncompressed indices on the NVM-backed heap with large core counts and index sizes. This work uncovers new space-time tradeoffs in storing in-memory inverted indices.

ISMM '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595272

CCS Concepts: • **Information systems** → **Search index compression**; Search engine indexing; • **Hardware** → **Memory and dense storage**; • **Software and its engineering** → *Garbage collection.*

Keywords: Text search, inverted index, persistent memory, compression, managed heap, garbage collection

ACM Reference Format:
Aditya Chilukuri and Shoaib Akram. 2023. Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines . In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3591195.3595272

1 Introduction

Search engines enable locating web pages on the internet and are a critical component of social media, professional networking, and e-commerce platforms. The key to retaining satisfied users is to offer low query response times. Amazon reports that even a 100 ms delay results in revenue drops [36]. Similar observations guide Google's search infrastructure.

The critical data structure search engines use for locating documents (web pages or social media posts) matching a word (term) is an inverted index. An inverted index maps unique terms to posting lists, where each posting stores an integer document identifier (ID) and meta-data (term frequency and position). Associating terms to posting lists using an inverted index speeds up query evaluation dramatically.

Today's standard practice is to host the inverted index in off-heap main memory. Recent work shows that even PCIe NVMe SSDs with byte-addressable 3D XPoint memory cannot deliver real-time response times [2]. Therefore, service providers keep indices in memory [60]. Unfortunately, as datasets grow, the inverted index grows proportionally, and large indices put increased pressure on DRAM. On the other hand, DRAM scaling cannot cope with the growth in datasets [20, 42]. Specifically, as data volume doubles yearly, the DRAM capacity only scales by 10% [24, 28]. The result is either the poor quality of service due to index lookups from storage or exorbitant memory-related expenditures.

Problem # 1 (High Decompression Latency): Compression is a crucial technique search engines use to store large indices in limited DRAM. For example, Apache Lucene uses a compression scheme that reduces index size by 85–90%,

allowing efficient storage in memory and reducing disk-to-memory transfers in the warmup phase [69]. Although compression reduces the memory footprint, its major downside is the extra computation required for decompression. Large indices require decompressing an increasingly large number of postings, increasing the time to resolve search queries.

Problem # 2 (Off-Heap Index Placement): The second problem with current practice is placing the index in off-heap memory. Popular search engines, such as Apache Solr [14], and Elasticsearch [13], are written in Java, a managed language, with support for automatic memory management or garbage collection (GC). In a managed environment, such as the Java Virtual Machine (JVM), accesses to the heap are direct, fast, and safe. Off-heap accesses, although allowed, are unsafe (so-called "backdoor") [37], impose restrictions on index formats, and make it challenging to use arbitrary objects, the very reason programmers prefer managed languages.

Off-heap memory is motivated by three reasons. ❶ It enables transferring index segments from storage via demand paging. When the index exhausts memory capacity, OS transparently uses storage as a memory extension. For user-facing workloads, operators typically distribute indices horizontally in memory [60]. Long queries generating I/O violate response time constraints and are killed or restarted [25, 41]. Storing index pages on anonymous heap memory is desirable, as it prevents their eviction due to conflicts and eviction algorithms skewed heavily towards page cache [56]. ❷ With off-heap placement, updates transparently move to storage for long-term preservation. However, inverted indices are immutable. New insertions happen in a separate ingestion pipeline [35, 52], and tombstones filter deletions [39]. ❸ Finally, it is possible to store indices in large heaps memory-mapped over storage [29, 31] but requires extra effort to prevent device and GC-related performance anomalies [31, 64]. Prior art reports high GC overhead for large heaps [31, 44, 45], on top of typical tuning effort [10, 33, 50, 57]. Avoiding off-heap indices requires effort but brings advantages for managed search engines.

To cope with growing index sizes and provide fast, safe, and direct access to inverted indices while allowing programmers to use rich object formats to compute efficiently over the index, we need to find ❶ scalable storage media for hosting uncompressed indices and ❷ mechanisms for storing the indices on the managed heap. This work aims to tackle the two challenges without degrading search performance.

Non-volatile main memory (NVM) technologies, such as 3D XPoint (Intel) [3, 62], carbon nanotubes (Nantero) [16], and Spin-transfer Torque MRAM (STT-MRAM) devices (Everspin) [1], offer byte-addressability and high capacity. They offer up to 10× lower latency than NVMe SSDs, but their read and write latency is higher than DRAM. Recent work shows that Intel's Optane NVDIMMs are 2–3× slower than DRAM [62]. NVM's high density can mitigate DRAM pressure, enable large (uncompressed) in-memory indices, reduce total memory expenditure, and improve service quality.

We open up and rigorously evaluate a novel space-time tradeoff in this work. Specifically, on the one hand, capacity-limited DRAM-backed compressed indices in off-heap memory suffer from high decompression latency. On the other hand, scalable NVM-backed uncompressed indices suffer from high memory access latency. We use Apache Lucene, a widely used Java search library, including Twitter [53] and Linkedin [49], to explore the query response times with DRAM/NVM-backed compressed/uncompressed indices. Our work involves extensive experimentation to set up Lucene's index on a datacentric server and ensure that our experiments represent real-world behavior and not an odd configuration. We report statistically significant query execution times and account for non-determinism, including hyperthreading, NUMA, JIT compilation, OS, and managed heap settings. We use Intel Optane Persistent Memory (PM) in direct access mode without a DRAM (page) cache. We carefully tune and modify Lucene's indexing formats for a fair comparison of compressed and uncompressed indices. We also fix a performance bug.

Key Finding: We discover that decompression latency hurts query response times much more significantly than high NVM latency. Across compressed and uncompressed postings formats, different query types, and varying core counts, changing the memory technology from NVM to DRAM reduces query response times by up to 11%. On the other hand, using uncompressed instead of compressed indices results in (up to) a 40% reduction in query execution times. Our key result is that, compared to the state-of-the-art (compressed index in off-heap memory), a hybrid DRAM-NVM system, where the DRAM-backed young generation (two gigabytes) is 64× the LLC size, and a separate NVM heap (few terabytes) backs the index, we can reduce query response times by up to 36%. This proposed system delivers safe access to large uncompressed indices in scalable main memory. Our findings are somewhat surprising because Optane PM is up to 3× slower than DRAM.

To explain the surprising result, we conduct a thorough performance counter analysis using Intel's recommended and rigorous top-down methodology [66]. We bring the insight that although search engines demand massive memory capacity to store indices and are backend (memory) bound, their query evaluation algorithms exhibit high spatial locality. Backend-related stalls represent more than 50% of the total issue slots, and almost 50% of those are due to hits/misses in the cache hierarchy. On the other hand, LLC misses per kilo instructions are low. Our results confirm that modern prefetchers aid search by learning access patterns and bringing postings in on-chip caches ahead of time [19]. Our analysis and related findings have broader implications for search infrastructure provisioning.

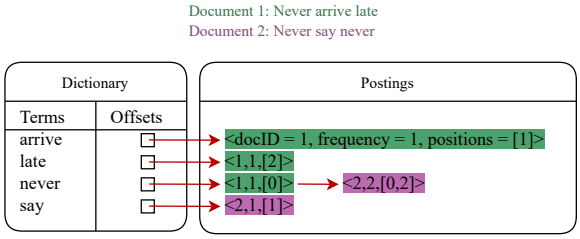


Figure 1. Structure of an Inverted Index.

Our other findings are: ❶ with properly sized heaps, the overhead of JVM's production Garbage-First (G1) collector [12] for hybrid DRAM-NVM systems is up to 6%, ❷ hardware prefetchers are more effective when the postings volume per query is relatively large. Finally, our work fixes a performance bug in Lucene's uncompressed postings format.

This paper opens up new and appealing tradeoffs at the intersection of established practices, i.e., compression and the use of off-heap memory, and emerging trends, i.e., improved software safety, huge managed heaps, and scalable memory technologies. Next, we provide background on Lucene's inverted index and other structures and algorithms it uses.

2 Background

We provide background on Apache Lucene, a high-performance, full-text search engine (Java) library. Lucene is industrial-strength and the backbone of Apache Solr [14] and Elasticsearch [13]. We discuss Lucene's inverted index, the compression algorithms it uses, and its search algorithms.

2.1 Lucene Inverted Index

An inverted index maps words (terms) to their location in a set of documents. (See Figure 1.) Its two main components are 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 that identify documents containing terms, 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.

Lucene sorts posting lists by document ID. It is possible to control the meta-data stored in each posting. We store term frequency and position. Lucene stores postings in multiple blocks, and each block contains postings for 128 documents.

2.2 Index Compression in Lucene

We now discuss techniques Lucene employs for compressing the index. (See prior work for a full survey [47, 69].) Lucene offers the `PostingsFormat` interface to implement custom compression techniques. It also provides a default implementation called `Lucene84PostingsFormat` in our version. Applications typically use the default implementation.

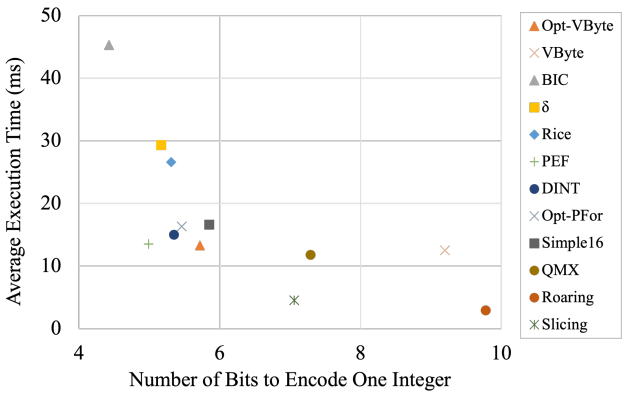


Figure 2. Compression efficiency versus *AND* query response time for different compression schemes. Adapted from [47].

Lucene uses three compression techniques, FOR-delta, PFOR, and VLB. Document IDs are stored in sorted lists using FOR-delta encoding. Term frequencies use PFOR encoding, and term positions use VLB encoding. Lucene stores the terms in the term dictionary using a custom ASCII compression scheme, and the posting list offsets as VLB integers.

Delta (Δ) encoding stores the difference between sequential values instead of the values themselves. The intuition for delta encoding is that document IDs are sorted numerically and ascend uniformly. Computing their delta encodings reduces the bit count to represent integers.

Frame of reference (FOR) encoding is a popular block-based compression scheme [17]. It breaks the posting (integer) list into blocks and calculates the minimum number of bits needed to represent the largest integer. It then stores each integer in the block with a minimum bit count. The bit count is kept in a block header. Typically, Δ precedes FOR, and the combo is called FOR-delta encoding. A huge integer in a block reduces FOR's compression efficiency. The *patched frame of reference (PFOR)* encoding stores large values separately at the end of the list [70]. PFOR is more aggressive than FOR, but the resulting decompression contains hard-to-predict branches.

Variable length byte (VLB) encoding uses the minimum number of bytes needed to store each integer. Each byte uses seven bits to store the integer's value, with the eighth bit reserved as a continuation bit. The continuation bit marks either the end of the integer or the start of a new integer.

There is a fundamental space-time trade-off in the design of compressed index storage formats. Figure 2 uses compression schemes from the prior art to compare the space efficiency of a compression algorithm versus query times. Algorithms that aggressively compress posting lists typically require more computation for decompression, resulting in worse query execution performance.

2.3 Search Query Evaluation in Lucene

Evaluating a search query requires finding document IDs matching query terms. The first step is to parse the query to identify 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. Lucene stores terms and offsets together in the same file sorted alphabetically [15]. Since the sorting is alphabetical, terms are recursively split into prefixes and suffixes, and common prefixes are only stored once. Splitting terms in the dictionary generates a sorted tree of terms. The term dictionary stores suffixes and matching term offsets in 24–48 length blocks for each prefix.

Lucene's default term dictionary is optimized to reduce storage I/O. A DRAM-backed index maps terms to block addresses containing offset data for the term. The block address points to a block of the term dictionary containing the term's suffixes and corresponding posting list offsets. This way, a dictionary search completes in one I/O access to the suffix-offset table. Lucene uses a finite state transducer (FST), a space-efficient but computationally intensive method to map input strings to offsets in the dictionary [38].

Once the offsets into the postings file are available, the next step is decompressing posting lists (single-term and multi-term queries) and, in addition, performing set operations (multi-term queries) to find matching document IDs. Lucene's query evaluator lazily decompresses posting list blocks and only decompresses selected blocks for multi-term queries. Posting list traversal is a linear function of the posting list size and, therefore, the index size.

Search engines use a scoring function to compute a numerical score of each document's *relevance* to the query. Lucene uses the Okapi BM25 scoring function [48]. BM25 uses term frequency normalized to document length [22]. Lucene uses a heap to rank the top-scoring documents, scoring each document in the posting list, adding it to the heap, and reporting the top N documents at the end of the traversal.

2.4 AND Query Evaluation

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 [69]. However, using binary search over compressed indices introduces a serious inefficiency. Specifically, it requires decompressing all posting list blocks in the index before the set intersection.

Lucene uses skip lists to avoid decompressing all blocks. A single-level skip list (see Figure 3 (a)) is a second list containing the first document ID in each posting list block. The skip list can be used to decide whether to decompress a posting list block. Posting list blocks are fetched from off-heap memory into the managed heap and decompressed if they could contain the document ID, significantly reducing posting list traversal time.

Lucene uses a multi-level (recursive) skip list [40], forming a tree-like structure. The query evaluator caches higher levels of the multi-level skip list in the main memory. A multi-level skip list (see Figure 3 (b)) has logarithmic time complexity (same as binary search). Multi-level skip lists massively improve performance of Lucene by preventing unnecessary decompression.

3 Latency Impacts of Index Compression

We now discuss the baseline performance of Lucene with compressed and uncompressed indices. We explain our indexing and performance measurement methodology and experimental setup. We then discuss evaluation results.

3.1 Methodology

3.1.1 Non Volatile Main Memory. We use Intel Optane persistent memory (PM) as we do not have access to any other NVM. Optane PM prefers sequential accesses [62]. The smallest transfer unit is 256B (4× the cache line), and the controller coalesces sequential writes into a larger 256 B write. We set up PM in *app direct mode*. (The other mode is *memory mode*, where DRAM acts as a cache for NVM.) We use a PM-aware filesystem, `ext4`, with extended support for direct access (DAX) using regular load/store instructions [58]. A DAX filesystem enables applications to use the OS page fault mechanism to access PM.

3.1.2 Software Framework. We use *luceneutil*, the defacto benchmark suite for evaluating Lucene. Luceneutil provides the framework to construct an index. We use the full text of Wikipedia (as of 20 March 2021) to build a 5.3 GB index on storage in the default (compressed) postings format. Query sets provided in the luceneutil project perform single term queries, *AND* and *OR* queries, and more complex phrase and fuzzy match queries. We use the luceneutil version from 12 April 2021 and Lucene version 8.9.0.[1] We use OpenJDK Java 13 and the production G1 garbage collector. We use modified `Indexer.java` and `SearchPerfTest.java` available online [2]. We use two query sets for evaluating queries: (1) a set of single-term queries of 45786 English words, and (2) a set of 5921 two-term *AND* queries.

3.1.3 Hardware Platform. We conduct experiments on a dual-socket Dell PowerEdge R740 server running Ubuntu Linux 18.04 LTS. Each socket contains Intel Xeon Gold 6252N running at 2.3 GHz, each with 24 physical cores (48 logical cores) and 96 cores in the system. Each core has 35.75 MB

[1] `https://archive.apache.org/dist/lucene/java/8.9.0/`

[2] The source code for all experiments is found in `anonymous-link-for-peer-review`

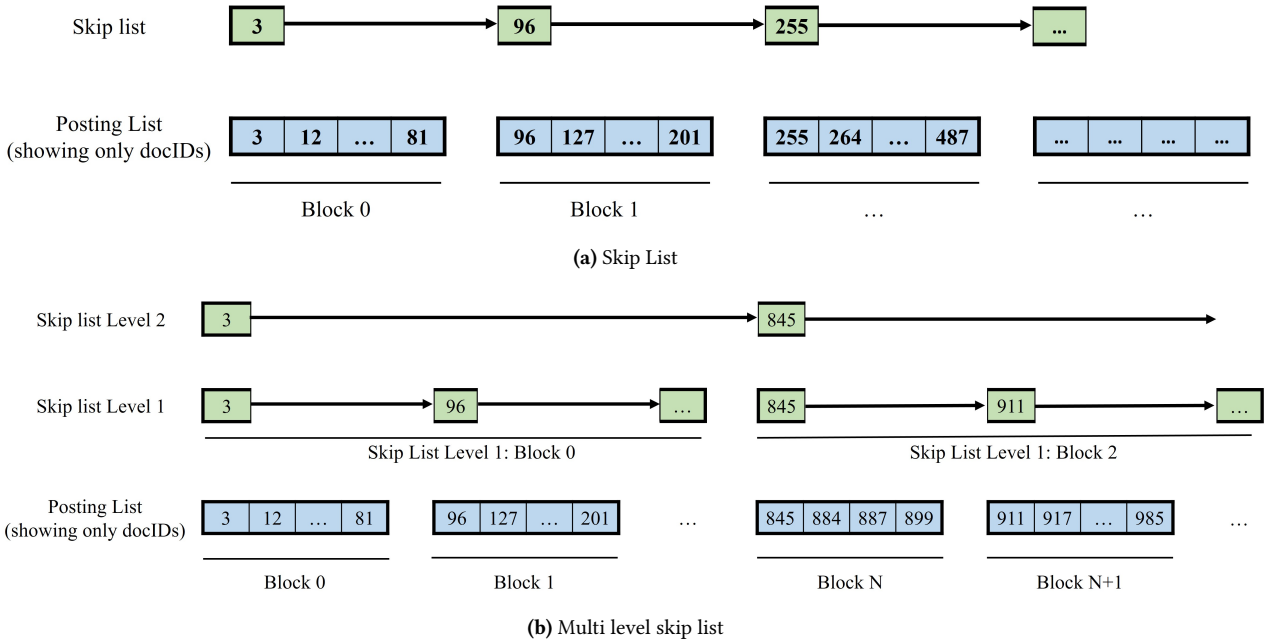


Figure 3. A single-level (a) and multi-level (b) skip list implemented for a block-structured posting list. In (a), if the intersection algorithm is searching for document 259, it could safely skip decompressing Block 0 and Block 1. In (b), the second level of the skip list enables skipping even more blocks than with a single-level skip list by performing only a single lookup.

of shared L3 cache and an integrated memory controller supporting six memory channels. Each memory channel connects to a 32 GB Micron DDR4 DIMM and a 128 GB Intel Optane NVDIMM. With 12× 32 GB DIMMs and 12× 128 GB Intel Optane DIMMs, the total memory capacity of the system is 384 GB of DRAM and 1.5 TB of Optane PM. The system has a 1.5 TB Intel Optane PCI Express NVMe SSD (DC P4800X). We disable non-uniform memory access (NUMA) accesses in all our experiments.

3.1.4 Indexing Formats.

We use two `PostingsFormat` implementations in Lucene, a widely used compressed format and an uncompressed (so-called Direct) postings format. Table 1 presents the data structures each format uses to store the index and traverse the posting lists.

Compressed Index (LPF): Our baseline compressed index uses the default index implementation of Lucene 8.9.0.

Uncompressed Index (DPF): The uncompressed postings format is implemented by `DirectPostingsFormat`. The index is stored on the filesystem in LPF format. In a warmup phase, the engine decompresses postings and other data structures into Java `byte` arrays and `int` arrays and stores them on the heap. The query evaluator resolves queries by directly accessing postings on the heap. All decompression happens ahead of time. DPF uses binary search over an uncompressed index on the managed heap to perform a set intersection for evaluating AND queries.

3.1.5 Performance measurement methodology.

We use a thread pool for running query workloads. To measure the scalability of our experimental setups with increasing thread (core) counts, we vary the thread count from 1 to 48. We pin query processing (worker) threads to logical SMT cores using the *Affinity* library [63]. Pinning threads to cores mitigates non-determinism due to scheduling and eases sound analysis of measurements from performance counter hardware.

We initially store the compressed index on an `ext4` filesystem over an Optane SSD. We warm up the index and bring it into the OS cache. Specifically, we run each query workload five times and measure the fifth run. All our indices fit in memory. Unless otherwise stated, we use the average query execution times as our evaluation metric.

We account for non-determinism due to just-in-time (JIT) compilation. We invoke the JVM five times for each query workload iteration. We time the last invocation. We present the mean and the 95% confidence interval in all performance data. 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.

We take a snapshot of counters at the start and end of every evaluated query for measurements with performance counter hardware. This methodology requires the `libpfm` library [34], and prior work provides more details [63].

Table 1. Key Features of Compressed and Uncompressed Inverted Index Formats

	Compressed Index (LPF)	Uncompressed Index (DPF)
Skip List Data Structure	Multi Level Skip List (on filesystem, cached on heap)	None
Posting List Compression	The document IDs are FOR-delta encoded, the frequencies are PFOR encoded, and the positions are stored in VLB integers	None
Term Dictionary	Terms are compressed using an ASCII compression scheme. Offsets are VLB-delta encoded and stored in variable length blocks	Stored as an uncompressed, sorted array of strings
Term Index	A finite state transducer (on heap) maps each query term to an address pointing to a block of the term dictionary	None; performs binary search directly on the term dictionary
Index Storage Location	Filesystem	Managed (JVM) Heap
Size of Wikipedia Index	5.38GB	52.78GB

We configure Lucene to report all results to fairly compare LPF and DPF across all query workloads. Specifically, we turn off the maximum impact indexing optimization in LPF, which appears in Lucene in version 8.0 in 2019 [59]. With the optimization, before decompressing and scoring documents in a block, the impact score of the block is checked to determine whether the block contains any documents that are competitive with the current candidates of most relevant documents to the query. The query evaluator only decompresses and traverses a block if it contains documents more relevant to the search query than the current candidate results. (*Impacts* are an upper bound on maximum relevance scores possible for a set of documents, and Lucene computes this score during indexing by exploiting specific mathematical properties of its scoring function [18].)

3.1.6 Heap Size. Controlling for heap Size is vital for managed benchmarking [8]. We aim to compare compressed and uncompressed posting formats without conflating GC costs. The compressed index resides off-heap in the page cache. Thus, for the LPF experiments, the heap contains only objects generated during query evaluation. These objects are short-lived and are quickly collected upon query completion (tens of milliseconds) during young generation collections.

On the other hand, the uncompressed index is stored as integer and byte arrays on the managed Java heap. The space for the index is allocated on the heap before the workload starts executing queries. The uncompressed index objects are immortal and read-only and remain that way throughout execution. For this reason, the heap settings differ for compressed and uncompressed index setups. We choose an initial heap size of 2 GB for experiments with the compressed index and a maximum heap size of 32 GB. For experiments with the uncompressed index, we set both sizes to 128 GB.

3.2 Evaluation Results

We first establish a robust baseline for comparing Lucene's compressed and uncompressed index formats in DRAM and NVM. Running experiments using the codebase from stock Lucene leads to a surprising result. Specifically, our initial measurements with DRAM-Only and single-term queries show that DPF behaves worse than LPF for long posting lists. This result is counter-intuitive as DPF does not incur the decompression cost and must resolve queries faster than LPF. We first investigate this result further.

The execution time of a single-term query is proportional to the length of posting lists across index segments. We use performance counter data and find that, counter-intuitively, for posting lists containing more than 10^5 postings, DPF executes more instructions than LPF. We investigate the reason and discover a performance bug in the posting list traversal of the DPF index.

Recall that the uncompressed index stores document IDs as Java integer arrays. Iterating to the next element of the posting list should increment an index variable. However, Lucene's default behavior (with DPF) is to perform a binary search to access the next element in the array, leading to unnecessary computation. Fixing this performance bug improves DPF's performance across all posting list sizes. This performance bug fix applies to single-term queries only. For multi-term queries, any search intersection algorithm uses binary or finger search to find a specific posting in a posting list and advance search across the list.

Figure 4 and Figure 5 compare final query execution times with LPF and DPF indices for single-term and two-term *AND* queries, respectively. DPF consistently outperforms LPF across different core counts. With 48 cores, both query workloads show a 37% performance improvement. We observe that single-term queries benefit more from DPF than AND queries. Decompression latency dominates the execution times of single-term queries, whereas conjunctive

queries incur additional overheads due to set intersection. **The bottom line is that queries resolve significantly faster with DPF than LPF, but it demands 9.8× more memory capacity.**

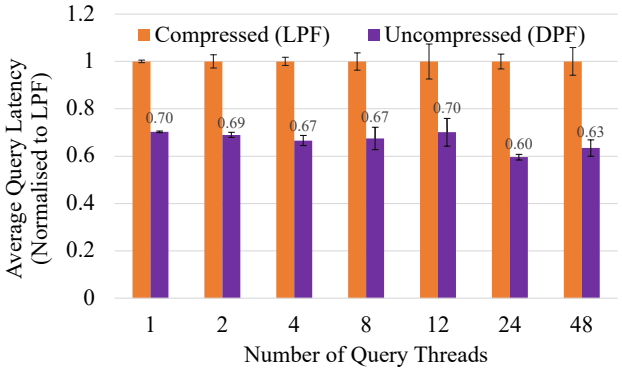


Figure 4. Showing the average query latency for single-term queries with compressed (LPF) and uncompressed (DPF) indexing formats.

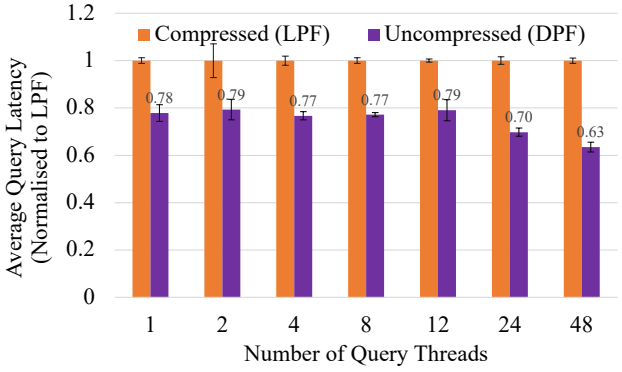


Figure 5. Showing the average query latency for two-Term *AND* Queries with compressed (LPF) and uncompressed (DPF) indexing formats.

4 Index Placement in Hybrid Memory

This section explores various DRAM-NVM systems with the index placed in the OS cache or the managed heap. We discuss different options to isolate the inverted index on the NVM-backed managed heap. We then present results evaluating the entire design space for the first time in literature.

4.1 DRAM and NVM-Backed LPF Index

Figure 6 (a) and (b) shows how the index is stored in DRAM and NVM-backed memories. To store the compressed LPF index in DRAM and NVM, we use a warmup-based approach (a common practice). The index initially resides on SSD. For LPF-DRAM, we run several query workloads to bring the index into the page cache, which is several times larger in capacity than our index. Similar demand paging-based warmup of LPF-NVM is unavailable with the stock kernel and NVM filesystem. Therefore, we copy the index in an NVM-backed file as shown in Figure 6 (b), mimicking an NVM page cache. We then directly perform search queries over the LPF index in NVM. The memory capacity consumed by LPF-based systems is proportional to the index size.

When the search engine receives a query, it decompresses the relevant postings from the page cache and copies them into the managed heap. Doing so requires the search engine to allocate temporary objects. The initial allocation of such objects happens in the young generation of the managed heap. Most objects die young [54] (e.g., after the query terminates, a few milliseconds for most queries), and the collector copies surviving long-lived objects into the old generation. The index stays in the OS cache from where managed code accesses it (unsafely) for query evaluation.

4.2 DRAM and NVM-backed DPF Index

The goal of DPF-based systems is to store the (immutable) uncompressed index on the managed heap. Search queries then compute over the on-heap index, possibly using rich types for manipulating postings. We first discuss DPF-DRAM (shown in Figure 6 (c)). The index initially resides on the SSD in LPF format. We use an extended warmup phase to ❶ read the compressed index into the page cache, ❷ decompress the index, and store all terms and postings on the managed heap as `byte[]` and `int[]` arrays. The arrays are first allocated to the young generation. When the young space is exhausted, a minor GC copies the live arrays into the old generation. As a result, all accesses to the index in production are safe.

There are two approaches for storing the uncompressed index on the managed heap. The first is to extend the JVM with a second heap for immortal and immutable objects. Programmers can annotate objects, such as the inverted index, for placement on the second high-capacity heap. This approach requires complex JVM and GC changes and leads to custom JVMs, limiting applicability. It also requires extra effort from the programmers who need to annotate candidate objects for placement on the second (specialized) heap. In this work, we exploit the GC mechanism for moving the immutable index to the old generation. We use the insight that each minor GC automatically moves live index data to the old generation.

One drawback of our approach is that index data is intermixed with long-lived (mortal) objects, and thus collector must scan the old generation to collect garbage. This scanning over NVM is slow and does not scale to high-capacity heaps [64]. However, for our heap settings, the GC overhead is low. The reasons include: ❶ the volume of live long-lived objects is tiny relative to the index size, and ❷ the collector does not trace primitive arrays, avoiding scanning costs.

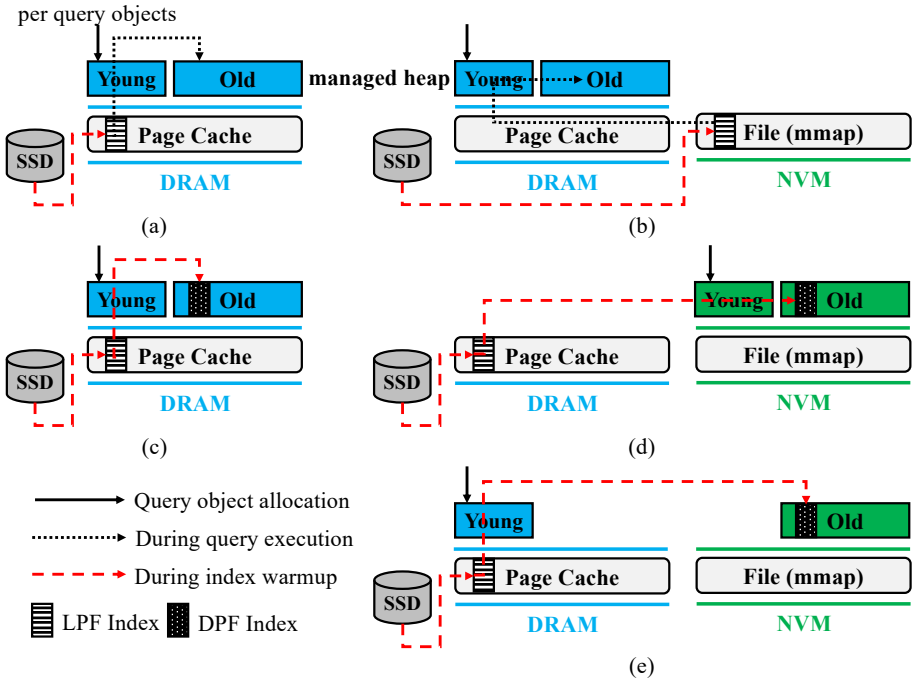


Figure 6. Showing five different memory systems for index placement: (a) LPF-DRAM, (b) LPF-NVM, (c) DPF-DRAM, (d) DPF-NVM, (e) DPF-HYB.

In Figure 6 (d), we show a DPF-based NVM system, namely DPF-NVM. We use the JVM feature for placing the entire managed heap on NVM. Next, in Figure 6 (e), we show a system that uses hybrid DRAM-NVM memory and splits the managed heap to place the young generation in DRAM and the old generation in NVM. We call this hybrid system DPF-HYB. The intuition for the hybrid heap is that, by allocating per query objects in a fats DRAM nursery, we can reduce the query response times compared to a slow NVM nursery.

The default G1 nursery is variable-sized. However, JVM allows configuring the nursery to a fixed size and placing it in DRAM, whereas the old generation resides in NVM. We use a fixed nursery for evaluating DPF-HYB. We measure the sensitivity of nursery size to performance and use prior advice to size the nursery as a multiple of the LLC size [4–6]. Our LLC size is 35.75 MB, and we experiment with nurseries between 8.94 MB (0.25× LLC) and 2.288 GB (64× LLC).

A final problem with our DPF-based systems is the possibility that index data remain in the nursery beyond warmup. Therefore, we trigger a full-heap GC after warmup. We modify Lucene to call `System.gc()` before starting our measurements. Doing so also ensures that all garbage is collected and the heap is compacted for the maximum locality.

DPF consumes memory capacity proportional to the heap size, and the heap grows to 10× the LPF index size. After the warmup is complete and the uncompressed index resides on the managed heap, OS can drop the original index from its cache. DPF's capacity requirements and NVM's density advantage complement each other, a tradeoff we explore next.

4.3 Evaluation Results

We now discuss the results of our evaluation with different placement settings. We also show microarchitectural analysis to explain our key observations better.

4.3.1 Overall Performance. We show the results of our evaluation in Figure 7 (single-term) and Figure 8 (two-term AND). We evaluate DRAM/NVM-backed LPF/DPF indices (LPF-DRAM, DPF-DRAM, LPF-NVM, DPF-NVM) and hybrid approaches that are similar to DPF-NVM, except the nursery resides in DRAM (DPF-HYB:NurserySize). We normalize the average query execution times to the state-of-the-art compressed postings format. For single-term queries, we first note that placing the compressed index in NVM increases the average query execution time by 2% for 48 cores and up to 10% (4 cores). NVM is thus suitable for hosting in-memory compressed indices when a modest reduction in query response times is tolerable. Next, we observe that DPF-DRAM is the best-performing approach, which reduces query execution times by 37% (48 cores). Unfortunately, DPF-DRAM demands massive (10× more than LPF-DRAM) DRAM capacity, which makes it an expensive and impractical system.

Next, we observe that DPF-NVM delivers a 30% (48 cores) boost in query performance compared to LPF-DRAM. Fortunately, owing to NVM's high density, DPF-NVM is a practical system and enables storing uncompressed postings in

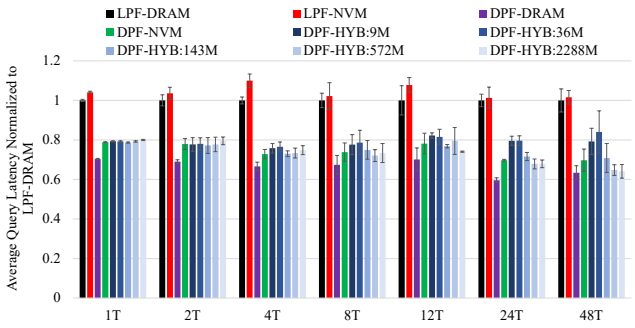


Figure 7. Showing average query latency normalized to the LPF-DRAM baseline for different systems with single-term queries. (T stands for thread count.)

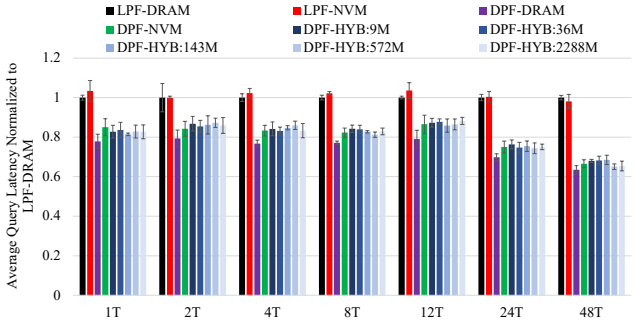


Figure 8. Showing average query latency normalized to the LPF-DRAM baseline for different systems with 2-term AND queries. (T stands for thread count.)

memory. On the other hand, DPF-NVM is slower (7%) than DPF-DRAM, the fastest approach today, i.e., hosting uncompressed indices in DRAM. We regain the lost performance with the intuition that most queries last only a few tens of milliseconds, but they allocate temporary, short-lived objects in the nursery. Queries incur a high latency if new object allocation happens in a slow media (NVM). Figure 7 shows that a hybrid system with a well-tuned nursery size (4× to 64×) is 6% faster than DPF-NVM (36% compared to LPF-DRAM), which places the entire heap on slow NVM. Our proposed hybrid system offers an attractive solution: ❶ it consumes between 0.5 GB to 2 GB DRAM, ❷ exploits the NVM capacity for hosting the uncompressed index, ❸ and is only 1% (48 cores) slower than DPF-DRAM. We observe that the best-performing nursery size varies across thread counts, and a hybrid system's performance benefits are more significant at high concurrency levels. This observation is in line with prior work that reports NVM bandwidth saturates rapidly beyond eight threads [3, 62], and hence the benefits of a hybrid approach that mitigates pressure on NVM by isolating new allocations in DRAM are higher. On average, small nurseries increase the GC overhead. Automatically tuning the DRAM nursery size is future work.

We observe similar behaviors for two-term conjunctive queries. One notable difference is that the gap between LPF-DRAM and DPF-DRAM is generally less wide, especially at low thread count. In general, conjunctive queries are more compute-intensive than single-term queries because of the large number of comparisons across multiple posting lists. At 48 threads, conjunctive queries with DPF-NVM, on average, suffer a 4% slowdown compared to the fastest system (DPF-DRAM). A hybrid system with a well-tuned nursery (15× the LLC size) bridges the gap to only 2% of DPF-DRAM.

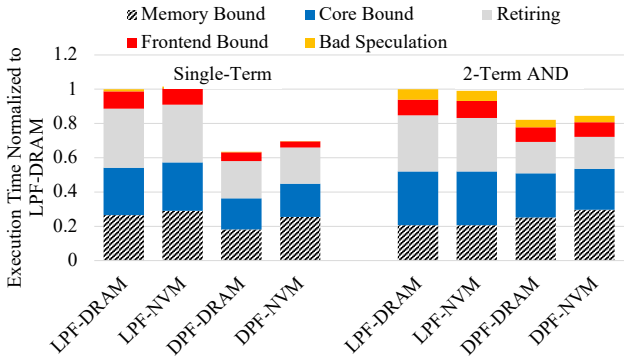


Figure 9. Showing the breakdown of per-query execution time into various components representing microarchitectural bottlenecks.

4.3.2 Microarchitectural Analysis. We now report observations from our detailed microarchitectural analysis of query workloads. We aim to understand how search queries interact with a server's cache and memory hierarchy. We use the top-down methodology [66] that systematically identifies *true bottlenecks* in an out-of-order processor. It identifies bottlenecks by rigorous performance counter measurements. Figure 9 shows the results of our analysis for two query workloads and different memory systems (48 threads). We breakdown query execution times (normalized to LPF-DRAM) into five components: ❶ backend memory-bound due to long-latency memory operations (cache hits or misses), ❷ backend core-bound due to lack of core resources, such as functional unit or reservation station, ❸ smoothly retiring instructions, ❹ frontend bound due to, e.g., lack of decoded microps, and ❺ recovering from misspeculation. Unfortunately, we observe that queries spend a significant portion of the execution time resolving memory loads (high memory-bound component) due to the data-intensive nature of search workloads. ILP is low due to the dependencies between instructions performing binary searches and skip-list traversals. Mispredicted branches cost very few cycles.

We observe that the memory-bound portion of the execution time is the highest (up to 36%) for DPF-NVM, while it is 30% for other systems. Reading uncompressed indices stresses NVM's bandwidth, especially at high thread count.

Typically, uncompressed indices stress the memory system more than compressed ones. The memory-bound components in Figure 9 do not convey the complete picture. Memory-bound nature is either due to cache or memory accesses.

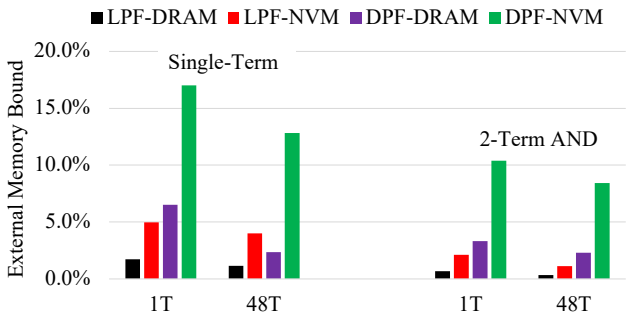


Figure 10. Showing the percentage of execution time spent resolving main memory accesses for two query workloads and different thread counts. (T stands for thread count.)

To further investigate which of the two (cache or memory) is responsible for memory-bound, we show the *external memory bound* component of the execution time in Figure 10. The time spent waiting for memory constitutes a small component of execution time (and memory bound). With 48 threads, less than 5% of the time is spent waiting for memory. We observe that in the worst-case (DPF-NVM), time spent waiting for memory is up to 17% of the execution time. Our index size is roughly 200× the LLC size, and the postings working set cannot fit in the on-chip caches. Therefore, the narrow gap in query response times between DRAM and NVM-backed indices is explained by high cache locality. (We elaborate on locality and prefetching impacts later.)

We measure LLC misses per kilo instructions (mpki) and observe very low mpki for our query workloads. Specifically, the LLC mpki is less than one for experiments with the compressed postings format, and it is 1.7 with the uncompressed postings format. Our overall conclusion from this analysis is that although search demands a large memory capacity and is backend memory bound, it exhibits high cache locality, and caches hide the high NVM latency, making it a strong candidate for hosting massive indices.

4.3.3 GC Overhead. We now discuss GC overhead for our workloads. We aim to observe if GC overhead is a significant fraction of total time and if GC over the NVM heap is slower than the DRAM heap. We use the Java *GarbageCollectorMXBean* interface, which monitors G1's internal data structures to monitor stop-the-world GC pauses. Figure 11 shows the time the application stops due to GC pauses in the measured (fifth) run of query workloads. We see the GC overhead is less than 6%. We only show the GC overheads for the compressed index setups, as the young and old garbage collections never occur for the uncompressed setups due to

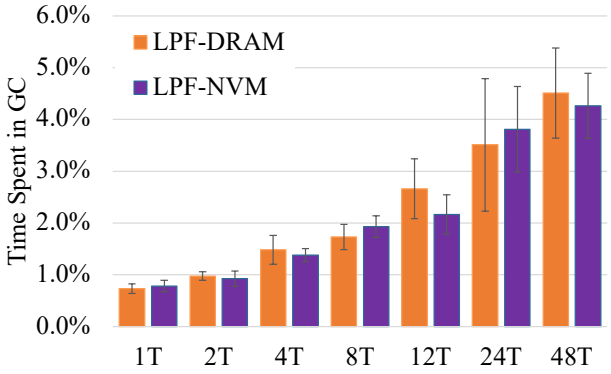


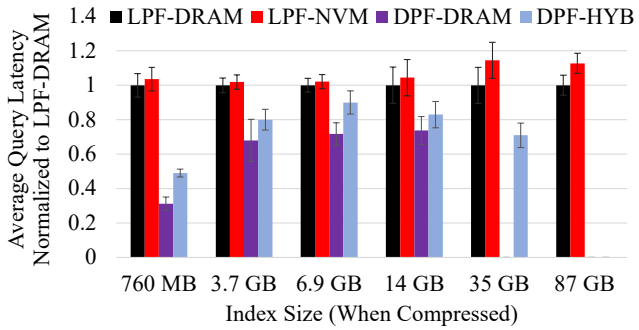
Figure 11. Showing the GC overhead as a percentage of the total workload execution time for single-term query workload with varying thread counts. (T stands for thread count.)

their large heap sizes. The GC overhead increases with rising thread count as allocation rates increase, stressing the collector. This result further validates that the ≈ 30% difference between the uncompressed and compressed postings format and the little performance difference between DRAM and NVM-backed indices despite NVM's high latency is not due to GC effects.

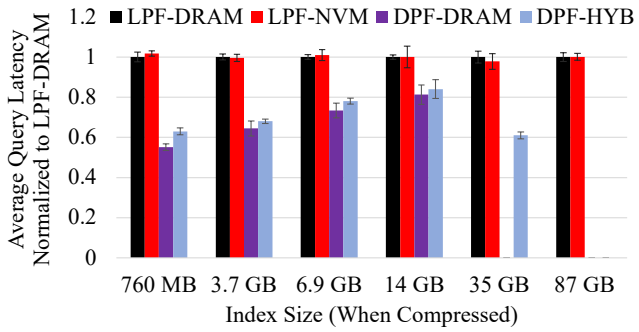
4.3.4 Sensitivity to Index Size. We validate that our findings are robust to index size. We also observe caching effects with increasing index size. We create indices of various sizes using the January 2022 CommonCrawl data set [43], a large repository of web crawl data. Index size in Lucene is difficult to control precisely. We, therefore, index between 250 K and 33 M documents. The DPF to LPF index ratio is 8×. The heap setting is unchanged for LPF-based systems, while DPF systems use the entire address space, 180 GB DRAM or 585 GB NVM. Figure 12 (a) and (b) show average query execution times (normalized to LPF-DRAM) for single-term and two-term queries (48 cores), respectively. Missing bars indicate an out-of-memory error because the index outgrows heap capacity. DPF-HYB uses a 2 GB nursery.

For the largest index in Figure 12 (a), LPF-NVM is 13% slower than LPF-DRAM. Thus for large indices, single-term queries over an NVM-backed compressed index incur a higher latency than indices of modest size. Next, DPF-HYB outperforms the state-of-the-art practice (LPF-DRAM) for modest (51%) and large (29%) index sizes. We observe DPF-DRAM consistently outperforms all other configurations.

We study the behavior of DPF-HYB with increasing index size at the microarchitectural level. Figure 13 (a) shows the LLC mpki for single-term queries with increasing index size. The LLC mpki of single-term queries decreases as the index size increases. Single-term queries involve accessing the posting list sequentially to score and rank document IDs. Bigger indices contain longer posting lists, and when these lists are sequentially accessed, the hardware prefetcher is

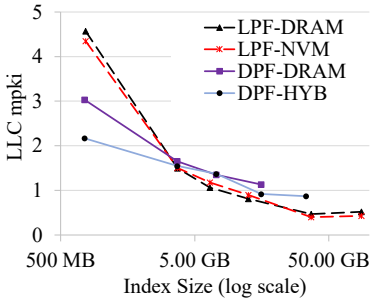


(a) Single-Term Queries

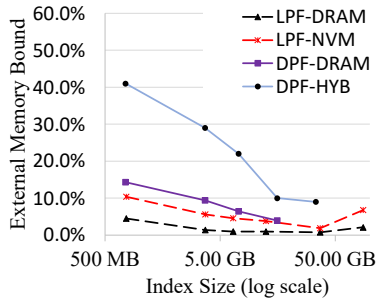


(b) 2-Term AND Queries

Figure 12. Showing the average query execution times with different index sizes for four memory systems.



(a) LLC mpki



(b) External Memory Bound

Figure 13. Showing the LLC mpki and external memory-bound component of the execution time for single-term queries and increasing index sizes.

more effective at predicting postings (cache lines) needed for resolving the query.

Figure 13 (b) shows the external memory-bound component of the execution time and represents the proportion of execution time the core is stalled waiting for memory. Since cache misses occur less frequently as the index size increases, we observe a reduction in the external memory-bound component. The prefetcher reduces the external memory-bound execution time for LPF-NVM from 45% to 8% as we increase the index size from 760 MB to 35 GB. Therefore, prefetching helps text search scale performance-wise to large, NVM-backed, uncompressed indices in memory.

In Figure 12 (b), we show the average normalized query execution times for two-term AND queries. These queries require more computation than single-term queries, and we observe slightly different trends for AND queries. For all the index sizes, the performance of LPF-NVM matches LPF-DRAM. DPF-HYB is up to 39% better than LPF-DRAM and consistently outperforms the state-of-the-art LPF-DRAM.

4.3.5 Tail Latency. Tail latency is a crucial metric for search operators to target a broader customer base. Table 2 shows the 99^{th} percentile latency over three setups, LPF-DRAM, LPF-NVM, and DPF-HYB with a 35 GB index. For two-term *AND* queries, we see a 322 ms (36%) improvement in 99^{th} percentile latency of the slowest queries with DPF-HYB (relative to LPF-DRAM). LPF-NVM provides a 99^{th} percentile tail latency comparable to state of art.

Table 2. 99^{th} percentile latency (+/- confidence interval) for LPF-DRAM, LPF-NVM, and DPF-HYB.

	LPF-DRAM	LPF-NVM	DPF-HYB
Single Term	65 (±8)	76 (±7)	61 (± 7)
2-Term *AND*	904 (±38)	873 (±53)	582 (± 51)

5 Related Work

Our work on scalable managed indices is related to large managed heaps, NVM exploitation for big data frameworks, and accelerating search with custom hardware. Recent work exploits NVM and PCIe NVMe SSDs for key-value stores, databases, and filesystems [7, 11, 26, 27, 32, 61, 62, 65, 67, 68]. However, despite its ubiquity, little prior works focus on exploiting emerging fast storage for managed search engines.

Akram [2] uses the Psearchy search engine provided in the MOSBENCH benchmark suite [9] to explore indexing and query evaluation over Optane Persistent Memory. They

explore search indexing performance, crash consistency, and query evaluation on various hardware/software system designs incorporating PM into the existing search application. They find that single-term queries perform similarly on a Wikipedia search index placed on Optane DIMMs compared to DRAM. Also, they conclude that the 2-term *AND* queries do not perform as well on NVM as on DRAM. The Psearchy engine does not use index compression [9, 51]. The Psearchy scoring function is simplistic, and to the best of our knowledge, the scoring function is not referenced in prior literature. Our work uses Lucene, a more realistic industry search engine library. We use Lucene and compare search evaluation on a state-of-the-art search engine using index compression on DRAM and NVM. We also investigate how on-demand decompression affects a modern search engine's performance. To our knowledge, no prior work characterizes the performance of an on-heap uncompressed search index in Lucene.

The basic idea of an inverted index is similar to a log-structured merge (LSM) tree. Prior work extends LSM-tree-based key-value stores [11, 26, 27, 65] to exploit heterogeneous storage. Existing efforts try to establish NVM as a new tier in the storage hierarchy, which does not fully exploit NVM's potential as byte-addressable memory. NoveLSM [27] uses an NVM-backed Memtable, while SLM-DB [26] places a global index in NVM and maintains single-level SSTables, unlike traditional LSM. MatrixKV [65] exploits NVM for fine-grained column compaction. SpanDB [11] exploits different types of SSDs to offer cost efficiency. Our work uses NVM as a first-class citizen in the address space, placing the entire inverted index directly in a managed NVM heap.

Prior work characterizes Intel Optane PM for native and managed workloads [3, 62]. Both focus on uncovering Optane's behavior but for different workloads, and they report Optane's limited scaling with increasing thread count. Specifically, Akram [3] finds that traversal algorithms (e.g., heap traversals in garbage collection) incur high latency in Optane-backed data structures compared to DRAM-backed structures. For example, they report that copying from nursery to mature space happens almost as fast with Optane as with DRAM. On the other hand, scanning objects to perform the transitive closure is slowed down significantly. In contrast, we focus on Optane as a high-capacity medium for storing uncompressed indices. Other prior efforts propose partitioning managed heaps into NVM and DRAM heaps and placing special objects in the NVM heap to improve write endurance, performance, and scalability [3, 46, 55].

Finally, recent work explores growing terabyte-scale managed heaps in big data frameworks over NVM and NVMe SSDs [30]. Their system, namely TeraHeap, eliminates the serialization cost by memory-mapping a second high-capacity heap over a storage device. They use a custom memory allocator for the second high-capacity heap. Their work fences the garbage collector from scanning the second heap. We leave integrating Lucene with TeraHeap to future work.

Finally, efforts that accelerate text search, e.g., by reducing decompression latency, using customized hardware [21, 23], are complementary to enabling huge in-memory indices.

6 Outlook and Conclusion

We now reflect upon potential impacts and draw conclusions.

6.1 Outlook

In our view, this paper and rigorous analysis with Lucene could have two broad impacts.

Potential Impact # 1: This paper asks if spending CPU cycles in decompressing the compressed index is always necessary. We instead show the feasibility of storing uncompressed indices directly on fast storage. Both options preserve DRAM capacity, and the former is predominant today. However, with emerging fast storage, we can avoid the decompression penalty. Our results can benefit K-V and document stores, and analytic engines. Keeping the entire index in the uncompressed form will not be feasible. However, frequently accessed index segments can be stored in uncompressed form without wasting DRAM capacity. We use NVM in this work as a fast storage device. Future work will evaluate NVMe SSDs, and remote memory, for storing the uncompressed index.

Potential Impact # 2: Our paper motivates growing managed heaps over fast storage and bringing large structures, such as inverted indices, compute caches, and graph partitions on the heap, to avoid unsafe accesses. The current approach uses off-heap accesses as index sizes quickly outgrow available DRAM capacity. We demonstrate our ideas using an inverted index and believe others will use similar techniques for more frameworks. Growing the heap over storage mitigates DRAM pressure, and the required machinery is non-obvious and should inspire future work.

6.2 Conclusion

To conclude, we have evaluated text search with compressed and uncompressed indices over DRAM/NVM off-heap and on-heap memory. We conclude that an NVM-backed uncompressed index on the managed heap delivers lower query latency than a DRAM-backed compressed index in off-heap memory. NVM is competitive to DRAM for search queries because they expose a high locality that modern cache hierarchies exploit. Our proposed hybrid system with a DRAM nursery and an uncompressed index in scalable NVM provides safe access to the index in production. It enables programmers to store index data in formats other than primitive types. Rigorously tuning the nursery size, pinpointing its placement in hybrid memory, and exploiting GC to move the long-lived uncompressed index in an NVM-hosted old generation delivers a 36% gain, and the machinery/result is non-obvious. Future work will investigate specialized high-capacity heaps for diverse managed frameworks.

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 *2019 IEEE International Electron Devices Meeting (IEDM)*. 2.1.1–2.1.4. https://doi.org/10.1109/IEDM19573.2019.8993516

[2] Shoaib Akram. 2021. *Exploiting Intel Optane Persistent Memory for Full Text Search*. Association for Computing Machinery, New York, NY, USA, 80–93. 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] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2016. Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors. *ACM Trans. Archit. Code Optim.* 13, 1, Article 4 (mar 2016), 25 pages. https://doi.org/10.1145/2875424

[5] Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout. 2017. DEP+BURST: Online DVFS Performance Prediction for Energy-Efficient Managed Language Execution. *IEEE Trans. Comput.* 66, 4 (2017), 601–615. https://doi.org/10.1109/TC.2016.2609903

[6] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392

[7] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. In *VLDB*. ACM, XXX–XXX. https://doi.org/10.14778/3461535.3461543

[8] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM* 51, 8 (aug 2008), 83–89. https://doi.org/10.1145/1378704.1378723

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. MOSBENCH. https://pdos.csail.mit.edu/archive/mosbench/

[10] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2006. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. *ACM Trans. Program. Lang. Syst.* 28, 5 (sep 2006), 908–941. https://doi.org/10.1145/1152649.1152652

[11] 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 21)*. USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[12] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

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

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

[15] The Apache Software Foundation. 2021. Lucene™ Release Docs. https://lucene.apache.org/core/documentation.html

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

[17] J. Goldstein, R. Ramakrishnan, and U. Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*. 370–379. https://doi.org/10.1109/ICDE.1998.655800

[18] Adrien Grand, Robert Muir, Jim Ferenczi, and Jimmy Lin. 2020. From MAXSCORE to Block-Max Wand: The Story of How Lucene Significantly Improved Query Evaluation Performance. In *Advances in Information Retrieval - 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14-17, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12036)*, Joemon M. Jose, Emine Yilmaz, João Magalhães, Pablo Castells, Nicola Ferro, Mário J. Silva, and Flávio Martins (Eds.). Springer, 20–27. https://doi.org/10.1007/978-3-030-45442-5_3

[19] Zacharias Hadjilambrou, Marios Kleanthous, Georgia Antoniou, Antoni Portero, and Yiannakis Sazeides. 2019. Comprehensive Characterization of an Open Source Document Search Engine. *ACM Trans. Archit. Code Optim.* 16, 2, Article 19 (may 2019), 21 pages. https://doi.org/10.1145/3320346

[20] Jim Handy. 2018. Emerging Memories Today: Why Emerging Memories are Necessary. https://thememoryguy.com/

[21] Jun Heo, Seung Yul Lee, Sunhong Min, Yeonhong Park, Sung Jun Jung, Tae Jun Ham, and Jae W. Lee. 2021. BOSS: Bandwidth-Optimized Search Accelerator for Storage-Class Memory. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 279–291. https://doi.org/10.1109/ISCA52012.2021.00030

[22] Jun Heo, Jaeyeon Won, Yejin Lee, Shivam Bharuka, Jaeyoung Jang, Tae Jun Ham, and Jae W. Lee. 2020. IIU: Specialized Architecture for Inverted Index Search. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1233–1245. https://doi.org/10.1145/3373376.3378521

[23] Jun Heo, Jaeyeon Won, Yejin Lee, Shivam Bharuka, Jaeyoung Jang, Tae Jun Ham, and Jae W. Lee. 2020. IIU: Specialized Architecture for Inverted Index Search. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378521

[24] Joel Hruska. 2018. Why RAM Prices Are Through the Roof. https://www.extremetech.com/computing/263031-ram-prices-roof-stuck-way

[25] Jon. 2017. SNIA: Avoiding tail latency by failing IO operations on purpose. https://faststorage.eu/

[26] 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 *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[28] Patrick Kennedy. 2018. Why Server ASPs Are Rising the 2017-2018 DDR4 DRAM Shortage. https://www.servethehome.com/why-server-asps-are-rising-the-2017-2018-ddr4-dram-shortage/

[29] Iacovos G Kolokasis. 2020. TeraCache: Efficient Spark Caching Over Fast Storage Devices.

[30] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Anastasios Papagiannis, Foivos Zakkak, Christos Kozanitis, Polyvios Pratikakis, and Angelos Bilas. 2023. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *ASPLOS '23: Architectural Support for Programming Languages and Operating Systems*. ACM.

[31] Iacovos G. Kolokasis, Anastasios Papagiannis, Polyvios Pratikakis, Angelos Bilas, and Foivos Zakkak. 2020. Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O. In *12th*

USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/kolokasis

[32] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[33] Philipp Lengauer and Hanspeter Mössenböck. 2014. The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (Dublin, Ireland) *(ICPE '14)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/2568088.2568091

[34] Libpfm. 2008. Libpfm4: a helper library for performance tools using hardware counters. http://perfmon2.sourceforge.net/

[35] J. Lin, P. Lok, B. Larson, K. Gade, S. Luckenbill, and M. Busch. 2012. Earlybird: Real-Time Search at Twitter. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/ICDE.2012.149

[36] Greg Linden. 2006. Marissa Mayer at Web 2.0. https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

[37] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 695–710. https://doi.org/10.1145/2814270.2814313

[38] Michael McCandless. 2019. Using Finite State Transducers in Lucene. https://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html

[39] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., USA.

[40] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., USA.

[41] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 17, 15 pages. https://doi.org/10.1145/3302424.3303973

[42] Onur Mutlu and Lavanya Subramanian. 2014. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations* 1, 3 (Oct 2014).

[43] Sebastian Nagel. 2022. January 2022 crawl archive now available. https://commoncrawl.org/2022/06/may-2022-crawl-archive-now-available/

[44] Khanh Nguyen, Lu Fang, Guoqing Xu, and Brian Demsky. 2015. Speculative Region-Based Memory Management for Big Data Systems. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems* (Monterey, California) *(PLOS '15)*. Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/2818302.2818308

[45] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 349–365.

[46] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, USA, 349–365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen

[47] Giulio Ermanno Pibiri and Rossano Venturini. 2020. Techniques for Inverted Index Compression. *ACM Comput. Surv.* 53, 6, Article 125 (dec 2020), 36 pages. https://doi.org/10.1145/3415148

[48] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. 1994. Okapi at TREC-3. In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994 (NIST Special Publication, Vol. 500-225)*, Donna K. Harman (Ed.). National Institute of Standards and Technology (NIST), 109–126. http://trec.nist.gov/pubs/trec3/papers/city.ps.gz

[49] Sriram Sankar. 2015. Did you mean Galene. https://engineering.linkedin.com/search/did-you-mean-galene

[50] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. 2011. Garbage Collection Auto-Tuning for Java Mapreduce on Multi-Cores. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. Association for Computing Machinery, New York, NY, USA, 109–118. https://doi.org/10.1145/1993478.1993495

[51] Jeremy Stribling, Jinyang Li, Isaac G. Councill, M. Frans Kaashoek, and Robert Tappan Morris. 2006. OverCite: A Distributed, Cooperative CiteSeer. In *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*, Larry L. Peterson and Timothy Roscoe (Eds.). USENIX. http://www.usenix.org/events/nsdi06/tech/stribling.html

[52] Nico Tonozzi and Dumitr Daniliuc. 2020. Reducing search indexing latency to one second. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/\reducing-search-indexing-latency-to-one-second.html

[53] Nico Tonozzi and Dumitru Daniliuc. 2020. Reducing search indexing latency to one second. https://blog.twitter.com/

[54] David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*.

[55] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, 347–362. https://doi.org/10.1145/3314221.3314650

[56] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[57] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. 2013. Control Theory for Principled Heap Sizing. In *Proceedings of the 2013 International Symposium on Memory Management* (Seattle, Washington, USA) *(ISMM '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2491894.2466481

[58] Matthew Wilcox. 2014. Add Support for NV-DIMMS to Ext4. https://lwn.net/Articles/613384/

[59] Alan Woodward. 2019. What's New in Lucene 8. https://www.elastic.co/blog/whats-new-in-lucene-8

[60] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 641–652. https://doi.org/10.1109/ICDE.2018.00064

[61] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 427–439. https://doi.org/10.1145/3297858.3304077

[62] 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

[63] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 309–322. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang

[64] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. 2021. Bridging the Performance Gap for Copy-Based Garbage Collectors atop Non-Volatile Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 343–358. https://doi.org/10.1145/3447786.3456246

[65] 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[66] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[67] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 194–209. https://doi.org/10.1145/3447786.3456237

[68] 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

[69] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. *ACM Comput. Surv.* 38, 2 (jul 2006), 6–es. https://doi.org/10.1145/1132956.1132959

[70] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. 59–59. https://doi.org/10.1109/ICDE.2006.150

Received 2023-03-03; accepted 2023-04-24