Fast and Scalable Text Search using Non-Volatile Main Memory

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

24 pt Honours project, S2/S1 2021–2022

By:
Aditya Chilukuri

Supervisor:
Dr. Shoaib Akram

School of Computing
College of Engineering and Computer Science (CECS)
The Australian National University

July 2022

Declaration:

I declare that this work:

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

July, Aditya Chilukuri

Acknowledgements

Thank you Shoaib, my supervisor, for your invaluable guidance throughout my project. You taught me how to conduct research, how to think critically, and how to keep fighting a research problem, even when nothing makes sense yet. I've learnt so much from working with you. Thank you.

Thank you Ben and Zak, my roommates, for your kindness and understanding. Your motivating words and friendly checkins helped me immeasurably in completing this thesis.

Thank you Kunal and Zak for your time in reviewing my thesis. Your valuable insights helped me present all the hard work I put into this project in a clear and polished way to the interested reader.

Thank you all my friends. Your company, conversation and kindness mean so much to me. I'd particularly like to thank Ashleigh Johannes, Calum Snowdon, Declan Hunt, Erina Carmichael, Gabe Bolton, George Bellas, Grace Brown, Jackie Murtagh, James Taylor, Kiara Chen, Maja Wilbrink, Max Barnes and Zixian Cai. Many of you have patiently listened to far too many of my rants about this thesis. Finally, it is done, and it wouldn't be possible without your support and kindness.

To my mum and dad, Sudha and Ravi Chilukuri: no amount of thanking you will every be enough to express my gratitude. Thank you for Everything. Thank you Sumedha, my sister, for your love. Thank you to our puppy, Olive, for being so cute and the best dog we could have. I love you all very much. This thesis, my university degree, my job and everything else in my life would not be possible without all your support, understanding and encouragement.

Abstract

In our information-driven society text search is ubiquitous. Searching large document sets such as the world wide web or social media posts requires building a search index. Search indices consume large volumes of main memory (DRAM) capacity, as even the fastest storage devices cannot satisfy the latency requirements for today's user-facing applications. Unfortunately, main memory is a limited and expensive resource. Specifically, the volume of information our society produces on the web doubles every year, while DRAM capacity only scales by roughly 10%. On the other hand, as datasets grow, the size of the inverted index grows proportionally. The current approach which search engines take to conserve memory capacity is to compress the search index and store it on a fast storage device. Unfortunately, searching over a compressed index slows down queries significantly. Our findings show that using a compressed index slows down queries by 1.5× on average, compared to using an uncompressed index stored on the program heap. Despite their performance advantage, compressed indices consume a massive heap (DRAM) capacity. Our results indicate that the uncompressed index is 7× to 10× larger than the compressed index.

Contribution 1: On the hardware side, emerging non-volatile main memory (NVM) is similar to DRAM as it offers byte-addressability, but also provides high capacity and scalability. This work is the first to evaluate the performance of search queries over two different index formats backed by DRAM and NVM. Our results show that commercially available NVM is a viable option for storing a search index. More specifically, we observe a slowdown of 10% between queries on placing the uncompressed index on DRAM and our new method which places the uncompressed index on NVM. Our approach is 30% faster than the state-of-the-art.

Contribution 2: We do extensive performance analysis to understand the narrow difference in search speed between DRAM-backed and NVM-backed indices. Specifically, we use Intel's recommended top-down methodology for evaluating bottlenecks. Our key finding is that search engines demand massive memory capacity, but the algorithms used for query evaluation exhibit high spatial locality. Therefore, the higher latency of NVM does not hinder the performance of search queries, as modern cache hierarchies are able to exploit locality and hide NVM latency.

Contribution 3: We rigorously perform a range of sensitivity studies to evaluate the NVM-backed managed heap for inverted index storage. Our analysis also confirms that search over an NVM-backed index scales well over multiple cores and large index sizes.

In summary, we contribute a novel approach for searching over ever-growing search indices targeting contemporary memory and storage technology trends. Our approach outperforms the state-of-the-art approach by 30%.

We recommend a novel two-step approach for scalable in-memory indices for search engines:

- storing the inverted index in an uncompressed format on the managed heap, and
- backing the managed heap by scalable NVM.

Table of Contents

Chapter 1

Introduction

Search engines are ubiquitous in modern life. For example, Google and Bing inform people of recent and historical events, while social media platforms keep us updated on real-time news stories. For these internet search providers, the key to attracting and retaining users is to have low response times for user queries. Web search engines, social media platforms, and e-commerce websites invest enormously in optimising search response times. Amazon investigated how user experience is affected by search latency as long ago as 2006 (Linden, 2006) and found that even increments of 100ms in delays for search results resulted in "substantial and costly" drops in revenue. Similar observations have guided Google's search platform design for the last 15 years.

We focus on plain-text search: finding documents in a large data set that contain words in a search query. We can perform a simple text search using command-line tools such as grep to scan the entire document set for text matches. However, this approach does not scale to large document sets. Since the conception of the search engine, the data structure at its core has been the inverted index. The inverted index (or simply, the index) is essentially a dictionary mapping all unique words in the document set to lists of document IDs that contain each word. Associating words to documents that contain them speeds up search query evaluation drastically.

In the information age, data set sizes are evergrowing, and search engines must index large data sets in a space-efficient manner. To this end, the Information Retrieval (IR) research community has designed many specialised and space-efficient compression schemes. Apache Lucene, a popular Java-based search engine library, uses a compression scheme that reduces index size by 85–90%, allowing efficient storage on the file system. The downside of index compression is that the search engine must decompress parts of the index when evaluating a query. Doing so increases the computation required to serve user queries. Nevertheless, for an index stored on a hard disk or solid-state drive, a well-designed compression algorithm reduces memory transfer costs (since the document lists are shorter) and seek times (Zobel and Moffat, 2006). Secondary storage has been the bottleneck of search performance for many decades (Zobel and Moffat, 2006). Consequently, modern search engines use large capacities of Dynamic Random Access Memory (DRAM) to cache sections of the index directly on main memory using the operating system page cache. However, with DRAM capacity growth slowing and the DIMM interface's power limitations, search providers must find a more scalable and performant alternative to DRAM for index storage.

Non-volatile main memory (NVM) technologies, such as the recently introduced Intel Optane Persistent Memory (PM) offer large capacity memory over DRAM. PM is byte-addressable and more capacity scalable than DRAM. However, its read and write latency is higher than DRAM. PM can be used in two modes.

- In the *Memory Mode*, it assumes the role of the system's entire physical address space. DRAM is transparently managed as a cache by the memory controller.
- In the *App Direct Mode*, PM is exposed to the operating system as a light-weight filesystem. Memory-mapped files then provide direct access to the applications for storing data at a byte granularity. In this later mode, the physical address space consists of the combination of DRAM and PM (called hybrid DRAM-PM memory).

Optane PM has orders of magnitude faster latency than the best NVMe SSDs. However, it is 2× to 3× slower than DRAM (Yang et al., 2020). Optane PM opens up new opportunities for search engines to make the use of main memory more efficient and also reduce the total memory expenditure in data centers, while improving the quality of service.

In this thesis, we explore the following question:

Can we use scalable non-volatile memory (NVM) to host ever-growing inverted indices in main memory?

1.1 Findings and contributions

We explore and evaluate two approaches for placing an Apache Lucene index on NVM:

1. placing an uncompressed index on the Java heap, where the heap is memory-mapped on NVM
2. placing a compressed index on an NVM-backed file.

Since NVM is 2–3× slower than DRAM, we expected our two approaches using NVM for index storage to perform significantly worse than equivalent approaches for storing indices on DRAM. However, our experiments showed surprising positive results.

1. Our first approach provides a 30% improvement in search performance (i.e. average query latency) over the current approach of searching a compressed index on

DRAM. Our first approach is only 10% slower than an equivalent system using an uncompressed index on the DRAM-backed heap.

2. Our second approach of executing queries on a compressed index on NVM has minimal difference in search performance over the current approach of placing the compressed index on DRAM.

To explain these surprising results, we conduct a thorough performance counter analysis and find that search is a cache-friendly application. Search algorithms are not sensitive to the worse memory latency of NVM, making NVM a strong candidate for a scalable and performant storage platform for inverted indices.

We validate this result with a sensitivity study on various index sizes and core counts. We find that large inverted indices are more cache-friendly to search on, so our proposed approach of using an uncompressed index on NVM performs much better than the state-of-the-art for large inverted indices. All our findings are valid for both single threaded and multi-threaded search. Specifically, we find that search performance with NVM-backed indices is competitive or better than the DRAM-backed indices at all core counts.

We use our findings to justify two new approaches for inverted index storage on NVM:

1. the *fast* approach: placing the uncompressed index on the heap, where the heap is memory-mapped to NVM. This approach makes search queries 30% faster compared to the state-of-the-art. The *fast* approach has two major benefits:
   a) the reduction in average query latency allows each node in a search server farm to resolve more queries per second, reducing search infrastructure costs
   b) the reduction in 99^{th} percentile latency improves user experience drastically for high latency queries.
2. the *scalable* approach: placing the compressed index on a file in an NVM-backed file system. This approach provides search performance that is comparable to the state-of-the-art. The *scalable* approach makes best use of NVM's large capacity to store colossal indices entirely on NVM, where the processor can access these indices with no IO costs. This approach enables search over evergrowing datasets which are infeasibly expensive to place on DRAM.

1.2 Thesis Organization

In chapter 2 we introduce the algorithms used in a plain-text search engine and the new NVM technology. Chapter 3 describes previous works that explore NVM technology of text search and other similar applications. Chapter 4 presents our methodology for constructing an uncompressed index on a DRAM-mounted managed heap and compares against the state-of-the-art. In chapter 5, we describe how we place the compressed and uncompressed indices on NVM and present our work's key findings. Chapter 6 presents a sensitivity study on various index sizes and finds that our approach performs best in the critical case of for large indices. We note drastic performance gains using our approach over the state-of-the-art. Finally, chapter 7 summarises our findings and concludes with ideas for future work.

Chapter 2

Background

In this chapter, we provide an overview of text search engines with a special focus on Apache Lucene. We also provide a background on non-volatile main memory (NVM) technology.

2.1 The Text Search Engine

This section serves as a tutorial for constructing and using a search index. We keep the explanation broadly applicable to many search engine implementations. Where it helps to explain a concrete example, we focus specifically on Apache Lucene's implementation.

Apache Lucene, or simply Lucene, is an open-source search engine library written in Java. Lucene supports concurrent indexing and search and uses a highly expressive query language (Apache, 2022). Twitter (Tonozzi and Daniliuc, 2020) and LinkedIn (Sankar, 2015) employ Lucene in the back end of their search functionalities. Lucene provides the fundamental indexing and querying functionalities for the popular ElasticSearch and Apache Solr open-source search engines.

2.1.1 Constructing a Search Index

The *inverted index* data structure shown in figure 2.1 is a dictionary mapping words (also called *terms*) to *posting lists* representing documents containing those words. A *posting* is a tuple made from:

1. *Document ID*: a unique identifier for a document containing a specific term.
2. *Frequency*: the occurrence count of the term within this document.

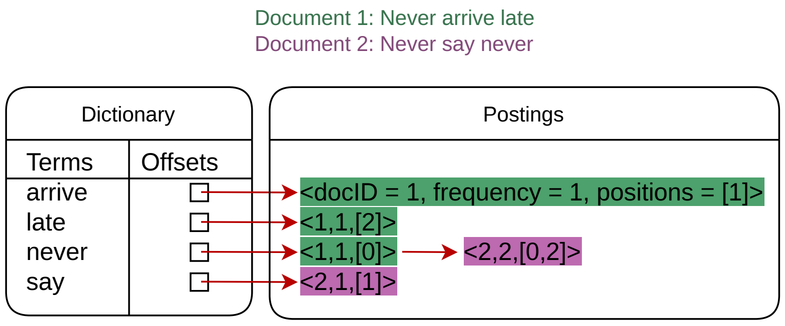

Document 1: Never arrive late
Document 2: Never say never
Dictionary
Terms
Offsets
arrive
late
never
say
Postings
<docID = 1, frequency = 1, positions = [1]>
<1,1,[2]>
<1,1,[0]>
<2,2,[0,2]>
<2,1,[1]>

Figure 2.1: Inverted Index Data Structure

3. *Positions*: a list of integers denoting the positions in which this term occurs in the document.

A *posting list* is a list of these postings that specifies which documents contain a specific term, how many times the term appears in each document, and where the term appears in each document. The inverted index is typically split into two files when stored on a file system. The posting lists are stored together in a postings file, and the dictionary is stored in a term dictionary file. As shown in figure 2.1, the term dictionary maps each term to an offset into the postings file where the posting list for that specific term starts.

Figure 2.2 shows each major aspect of index construction. The *indexer* creates an index from the document set. We explain figure 2.2 in detail.

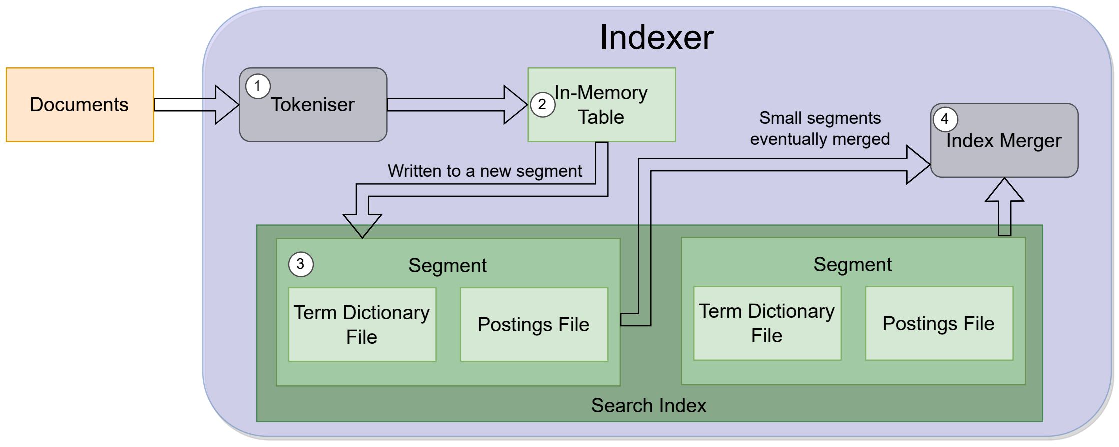


Figure 2.2: How search indices are constructed

Tokeniser ①

The *tokeniser* (figure 2.2 step ①) processes documents to collect words (also called terms) into a term list. Search engine implementations may preprocess the documents to improve search quality and reduce the index size. Implementations employ various techniques for tokenisation. As an example, Twitter performs the following changes during tokenisation (Singer and Wilcox, 2021):

1. *Stemming*: replacing words with their grammatical roots. For example, “eats”, “eating”, “ate” are all replaced with the root word “eat”. This improves search quality as queries will match documents even when the tense or the plurality of a word does not match exactly.
2. *Filtering articles*: Words such as “a”, “an” and “the” rarely add meaning to a document, and hence they can be removed to reduce the size of the index.
3. *Normalisation*: removing punctuation marks and accents.
4. *Synonym Addition*: to improve query results, common synonyms for various terms are added into the token list.

In-Memory Table ②

The in-memory table or memtable (figure 2.2 step ②) is the core of the indexer. The *memtable* is a small inverted index, typically stored on the indexer's heap. The indexer updates the memtable to include the term lists output by the tokeniser for new documents. The memtable contains the documents most recently added to the search index. Lucene does not support a searchable memtable. A searchable memtable is helpful for many applications where indexing occurs in real-time over a dynamic document set. For example, Twitter implements a real-time search engine that indexes new tweets within 1 second so that Twitter's search function returns new tweets instantly (Singer and Wilcox, 2021). The memtable is typically searchable for real-time applications, so the searcher threads can concurrently access the memtable the indexer is writing. As a contrasting example, an online store's search index is not real-time and may only be updated once a day with new items, so it would not need a searchable memtable.

Index Segments ③

When the memtable reaches a specified maximum size, the indexer stores the memtable's contents on the file system and clears the memtable. The indexer writes the memtable on the file system in a data structure called an index segment (figure 2.2 step ③).

An *index segment* is a serialised inverted index. An index segment typically consists of two files: a postings file and a term dictionary file, as shown in figure 2.1. The postings file contains all the posting lists in the memtable. The posting lists can be stored unsorted or sorted. By default, Lucene index segments store posting lists sorted by document ID. However, it is also possible to store posting lists sorted by the relevance of the document to user searches (Zobel and Moffat, 2006). We measure relevance by the document scoring function described in section 2.1.4. Each approach has various complex tradeoffs, which are discussed at length by Zobel and Moffat (2006). The term dictionary file contains all the terms in the memtable, mapping them to offsets into the postings file. Lucene's term dictionary structure is defined in detail in section 2.1.3.

Segment Merging ④

Storing a large number of index segments is detrimental to search performance, so segment merger threads merge multiple segments by merging their respective term dictionaries and postings files (figure 2.2 step ④). Lucene allows merging to be performed concurrently to indexing or after completing indexing. Lucene allows programmers to specify custom merge policies and merge scheduling policies. Merge policies define the criteria by which the indexer decides which index segments to merge, and merge scheduling policies define when the indexer should merge segments. We use the same default settings for segment merging for all our experiments.

2.1.2 Index Compression Algorithms

Text search presents two major challenges for practical systems.

1. Search engines must support querying billions of documents. As all index data structures grow linearly in size with the size of the document set (Zobel and Moffat, 2006), indices must be stored efficiently.

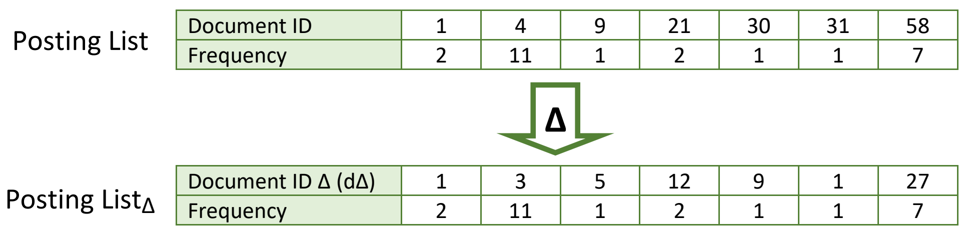
2. Search engines must complete queries within stringent performance requirements for the best user experience.

Compression of posting lists and term dictionaries has historically been an essential step of this process as compression addresses both challenges of text search. Compression reduces the index size, enabling the construction of indices over large corpora. Search over a compressed index also requires fewer memory transfers from slower storage devices (Pibiri and Venturini, 2020). This speeds up query processing significantly when accessing disk storage is the bottleneck for search.

Pibiri and Venturini (2020) survey the available index compression algorithms developed by the information retrieval research community. This section covers key ideas in compression and shows how popular search frameworks employ compression. Index compression techniques employed by Lucene are detailed in section 2.1.3. We defer to Pibiri and Venturini (2020) and Zobel and Moffat (2006) for a wide-ranging exploration of index compression techniques explored in research.

Delta Encoding

A key idea employed in many posting list compression algorithms is *delta (Δ) encoding*: to store the difference between subsequent values instead of the values themselves. This is shown in figure 2.3. Delta encoding works on the assumption that document IDs in posting lists are typically sorted numerically and ascend uniformly randomly. Computing the delta encodings of sorted lists reduces the number of bits needed to represent each integer in the list.



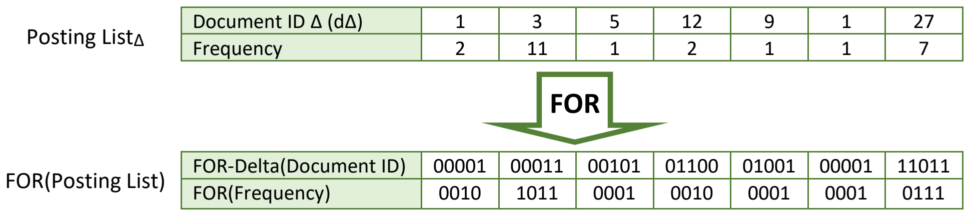
Posting List

Document ID	1	4	9	21	30	31	58
Frequency	2	11	1	2	1	1	7

Posting List$_\Delta$

Document ID Δ (dΔ)	1	3	5	12	9	1	27
Frequency	2	11	1	2	1	1	7

Figure 2.3: Delta encoding of document IDs in a posting list



Posting List$_\Delta$

Document ID Δ (dΔ)	1	3	5	12	9	1	27
Frequency	2	11	1	2	1	1	7

FOR(Posting List)

FOR-Delta(Document ID)	00001	00011	00101	01100	01001	00001	11011
FOR(Frequency)	0010	1011	0001	0010	0001	0001	0111

Figure 2.4: Frame of reference (*FOR*) encoding of a block of document ID deltas and frequencies

Variable Length Byte encoding

In *variable length byte (VLB) encoding*, we store each integer in the minimum number of bytes needed to represent it. Each byte uses seven bits to store seven bits of the integer's value, with the eighth bit reserved as a continuation bit. The continuation bit marks whether the following byte is a continuation of the current integer or if the following byte contains the value of the next integer in the list.

Frame of Reference (*FOR*) Encoding Family

Frame of reference (FOR) encoding, designed by Goldstein et al. (1998), is the most popular block-based compression scheme. *FOR* works by breaking the integer list into blocks and calculating the minimum number of bits needed to represent the largest integer. We then store each integer in the block in this minimum number of bits. Every block has a header field indicating the number of bits used to store each integer in the block. Figure 2.4 shows *FOR* encoding applied to a single block of the posting list.

It is common to perform *FOR* encoding after applying a delta encoding to the list. As discussed earlier, delta encoding typically reduces the bits needed to store each integer, which *FOR* exploits to reduce the storage space of each block. Combining *FOR* and delta encoding is called *FOR-delta* encoding. We demonstrate *FOR-delta* encoding to compress the document IDs (which were already delta-encoded) in figure 2.4.

An obvious problem with *FOR* is that the presence of exceptionally large integers in a block will increase the minimum bits per integer needed to store each value in the block, reducing compression efficiency. The *patched frame of reference (PFOR)* encoding modifies *FOR* by extracting "large" values in the list and storing them separately at the end of the list (Zukowski et al., 2006). *PFOR* compresses integer lists more aggressively than *FOR*. However, compared to *FOR*, the decompression algorithm for *PFOR* is slower as it contains hard-to-predict branches.

Sphinx Search Index Compression

Sphinx (Aksyonoff, 2022) is an open-source search server application[1]. Sphinx powers Craigslist and the Chinese video content provider Youku. For each document containing the search term, Sphinx stores the document ID, the number of hits within the document, and other information for scoring the document in a struct on contiguous memory. Sphinx first delta-encodes document IDs and then compresses the whole posting list using a *VLB* encoding.

2.1.3 Lucene Index Compression

We use Lucene Version 8.9 in this work[2]. Lucene provides the *PostingsFormat* interface to allow implementers to write custom compression and decompression algorithms. Lucene provides a default implementation of the *PostingsFormat* interface. For the version of Lucene we are using, this is called *'Lucene84PostingsFormat'*. This default implementation defines a compressed index data structure that search functionality implementers most commonly use. We now describe the default index implementation in Lucene 8.9.

The Lucene index stores posting lists sorted by document ID. We can control what the Lucene index stores in each posting. We configure the index to store term frequencies and term positions in our experiments.

Lucene stores postings in multiple blocks in a posting file. Each block contains the postings for 128 documents. Document IDs are stored in sorted lists using *FOR-delta* encoding. Term frequency storage uses *PFOR* encoding. Term positions are stored using *VLB* encoding. Lucene stores the terms in the term dictionary using a custom ASCII compression scheme, and the posting list offsets in *VLB* integers.

[1]Only versions up to 2.3.2 are open-source. The most recent version, Sphinx 3.4.1, is not open-source.
[2]See `https://archive.apache.org/dist/lucene/java/8.9.0/`

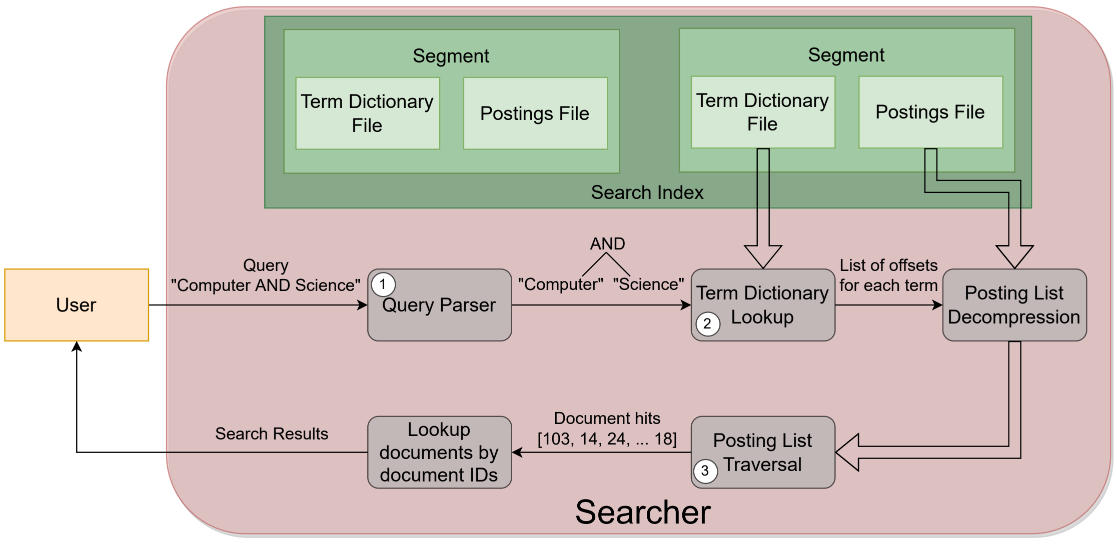


Figure 2.5: How a search query is executed

In the last decade, innovation in compression algorithms has focused on efforts to optimise encodings to use single instruction multiple data (SIMD) instructions for compression and decompression of the index (Wang et al., 2017; Lemire and Boytsov, 2015). As a result, optimised implementations of *VLB*, *FOR*, and *PFOR* use SIMD instructions. Unfortunately Lucene does not use SIMD optimisations for the compression and decompression of the posting lists since Java does not support SIMD optimisations natively.

2.1.4 Search Query Evaluation Algorithms

Query evaluation is finding documents in the search index that are most relevant to the query provided. As seen in figure 2.5, query evaluation splits into three steps:

1. *Query parsing*: to find the terms relevant to the search and the conditional operators applied to each term.
2. *Term dictionary lookup*: to find posting lists for terms in the search query.
3. *Posting list traversal*: to search the posting lists for documents satisfying the query.

Our work focuses on single term and 2-term *AND* queries; we focus on these query types in our explanations.

Query Parsing ①

Queries are constructed using terms and operators. The most straightforward queries are single-term queries that request documents most relevant to a single search term. Boolean queries use conjunctions (*AND*) and disjunctions (*OR*) to request documents containing any or all of the input search terms. An example of a boolean query is shown in the output of figure 2.5 step ①. In addition, we can nest boolean operations to create more complex search queries. Finally, we note that executing boolean queries is equivalent to recursively performing basic set operations (intersections and unions) on the posting lists.

The primitive operators *AND* and *OR* are the building blocks of more complex operators. For example, a phrase query asks for documents where terms appear in a particular order. A phrase query is essentially equivalent to an *AND* query over the terms in the phrase and uses the position data stored in the posting lists. Similarly, a prefix query, which asks for documents containing terms with a specific prefix, is evaluated using an *OR* query over all the terms in the term dictionary matching the prefix. Query parsing simplifies complex operators into *AND* and *OR* queries since boolean queries only require performing basic set operations.

Term Dictionary Lookup ②

The terms in the query are searched in the term dictionary to acquire offsets into the postings file as shown in figure 2.5 step ②. Lucene stores terms and offsets together in the same file, alphabetically sorted. 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 data structure of terms. The term dictionary stores suffixes and matching term offsets in 24-48 length blocks for each prefix.

Lucene's default term dictionary implementation uses a DRAM-backed term index that maps terms to blocks containing the offset data for the term. To search for a term, we first query the DRAM-backed index for a block address. The block address points to a block of the term dictionary containing the term's suffix and its posting list offset. Using a DRAM-backed index allows term dictionary search to complete in a single IO access to the suffix-offset table stored on the file system.

Lucene implements the term index as a finite state transducer (FST), described in McCandless (2019b). The FST is a space-efficient (but computationally intensive) method of mapping input strings to offsets in the term dictionary.

Term dictionary lookup generally has a complexity of $\mathcal{O}(\log N)$ where N is the number of terms in the dictionary. As the number of terms in a dictionary is typically a linear function of the document set size (Zobel and Moffat, 2006), dictionary lookup time is a logarithmic function of search index size.

Posting List Traversal ③

Step ③ of figure 2.5 shows posting list traversal. The workflow for this stage is in three stages:

1. decompress the necessary posting lists

2. perform the necessary set operations on the posting lists to find the documents matching the search query; and
3. score the documents by relevance and rank them to find the best results.

During posting list traversal, the searcher decompresses posting list blocks in a lazy fashion. The searcher decides which posting list blocks it needs during posting list traversal and only decompresses those blocks. While there are different algorithms for matching single term queries and boolean queries, posting list traversal is typically a linear function of the posting list size and hence a linear function of the index's size.

Scoring and Ranking

A scoring function uses the information stored within a document's posting to compute a numerical score of the *relevance* of the document to the query. The query can define which terms are more relevant to the user. For example, the user may send a weighted query (not tested in our workload) that requests specific terms in the query to be more prominent in the search results.

Lucene uses the Okapi BM25 scoring function for scoring queries, developed and described in detail by Robertson et al. (1994). BM25 scores documents by using the frequency of occurrence of each term in the query, normalised by the document's length (Heo et al., 2020). For example, consider the two-document index defined in figure 2.1. A query of "never" will result in a hit in document two: "never say never". We provide a sample calculation of document two's relevance score according to equation 2.1.

$$\begin{aligned}\text{Score}(doc) &= \sum_{term \in query} \text{IDF}(term) \cdot \frac{freq(term, doc) \cdot (k_1 + 1)}{freq(term, doc) + k_1 \cdot (1 - b + b \cdot \frac{length_d}{avgdl})} \qquad (2.1)\\ &= \text{IDF}(\text{“never”}) \cdot \frac{2 \cdot (1.2 + 1)}{2 + 1.2 \cdot (1 - 0.75 + 0.75 \cdot \frac{3}{3})}\\ &= \text{IDF}(\text{“never”}) \cdot 1.375\end{aligned}$$

Here, q is the query, and $t \in q$ are the query's terms. Each term's frequency controls a component of the final score. The frequency of each term positively affects the score. $freq(term, doc)$ is the frequency of a term in a document (doc). $k_1 = 1.2$ is a constant used to limit term frequency scaling of the score, and $b = 0.75$ is a constant used to control the effect of document length normalization (Heo et al., 2020; Lucene, 2021). The $\frac{length_d}{avgdl}$ is the length of the document normalised by the average document length in the index.

The inverse document frequency (IDF) defined in equation 2.2 is used to weight rarer terms in the search index more strongly. For example, an internet search of "Mount

Everest" should find documents more strongly relevant to the term "Everest". "Everest" is less likely to be in as many documents as "Mount" and is more likely to be the word the user finds most relevant.

$$
\begin{aligned}
\mathrm{IDF}(t) &= \log \frac{W - n(t) + 0.5}{n(t) + 0.5} \\
\mathrm{IDF}(\text{“never”}) &= \log \frac{6 - 3 + 0.5}{3 + 0.5} \\
&= 0
\end{aligned}
\tag{2.2}
$$

W is the total word count of the document set. $n(t)$ is the length of t's posting list. If a term accounts for half or more of the document set, as in our example, the IDF is 0.

As the goal of scoring is to report the top N most relevant results, Lucene uses a heap data structure (the heap algorithm, not to be confused with the heap memory of a program) to rank the top-scoring documents. Single term query evaluation is easy to implement. Lucene scores each document in the posting list for a single term query, adds the document to the heap, and reports the top N documents at the end of the traversal.

AND Query Evaluation

For *AND* queries, we must find the intersection of both posting lists. Since all Lucene posting lists are sorted by document ID, the posting lists are scanned from start to end to find document IDs in both lists. This algorithm yields an overall execution time proportional to the length of the longest posting list. Figure 2.6 shows an outline of the intersection algorithm for two posting lists. The arrows show the steps taken by the algorithm. The numbers labelling the arrows show that both posting lists are advanced in alternate steps. A *candidate* (circled red in the diagram) is a document ID last seen in a posting list. At each numbered step shown in the figure, we try to find a document ID in a posting list equal to the candidate from the other posting list. If we find the same document ID, there is a match. Steps 3 and 7 resulted in matches. If the candidate document ID does not exist in the current posting list, we set the following (higher) document ID seen in the posting list as the new candidate. The process continues until we find all documents in the intersection of the posting lists.

At every step in the algorithm (shown by arrows in figure 2.6), we must scan the posting list for a document ID. The naive implementation would scan the posting list sequentially from left to right. Since the posting lists are sorted, we can optimise the naive implementation by using a binary search starting from the previous candidate.

Skip Lists for Postings

Since the algorithm for posting list intersection involves binary searching over multiple posting lists, the memory access pattern for *AND* queries is more random than a single

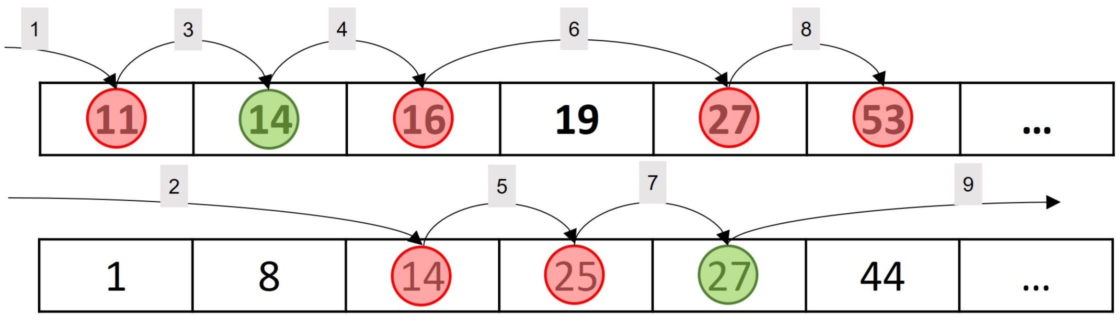
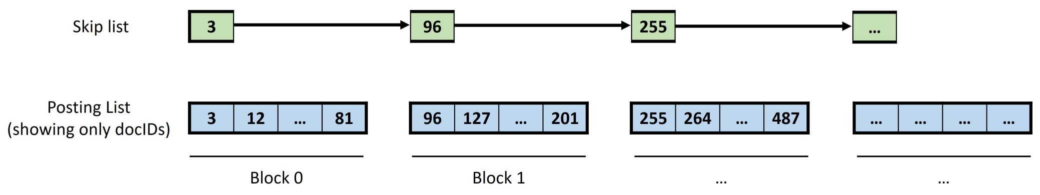
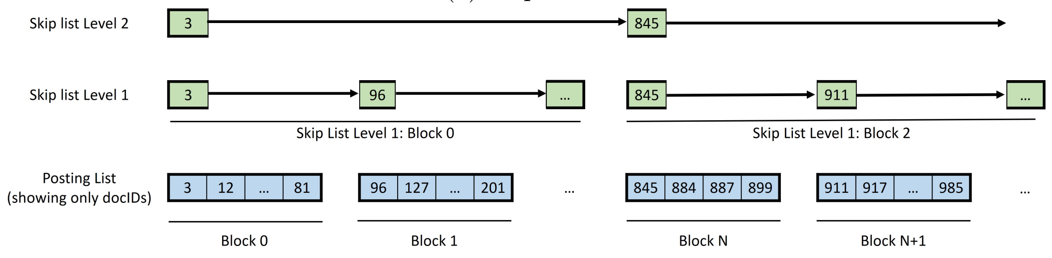


Figure 2.6: Posting List Intersection: numbered arrows show steps taken by algorithm



(a) Skip List



(b) Multi level skip list

Figure 2.7: A single-level and multi-level skip list implemented for a block structured posting list

term query where the algorithm is a linear scan through a single posting list. If the index is stored on a hard drive, the non-sequential memory access pattern of the intersection algorithm causes long disk seek times. A second problem is that binary search requires wasted computation in decompressing all posting list blocks in the index.

To alleviate both problems, modern search implementations use the skip list data structure shown in figure 2.7a. We still store posting lists in compressed blocks when using a skip list. The single-level skip list is a second list containing the first document ID in each block of the posting list. When searching for a candidate document ID, we can use the skip list associated with the posting list to decide whether to read and decompress a block of the posting list. Posting list blocks are only fetched from storage and decompressed if they could contain the candidate document ID, significantly reducing posting list traversal time.

The single-level skip list is large enough to be stored in compressed form in multiple blocks on the file system for a sufficiently large posting list. Searching for a single document ID in large posting lists would require a wasteful binary search on the skip list, facing the same performance issues outlined earlier. Lucene uses a multi-level skip list (McCandless et al., 2010). A multi-level skip list is essentially a recursive skip list, forming a tree-like structure shown in figure 2.7b. The searcher caches higher levels of the multi-level skip list in main memory at the start of the search. Search on a multi-level skip list has a logarithmic time complexity, the same algorithmic complexity of the binary search algorithm. However, using a multi-level skip list massively improves performance by reducing disk seeking and avoiding unnecessary decompression.

Parallelising Lucene Search

There are two methods for parallelising search queries. First, different threads can satisfy different queries in parallel. This approach is trivial to implement on any search engine as query search does not modify the index. This approach improves query evaluation bandwidth but does not improve query latency (in fact, multi-threading overheads may increase the latency of individual queries). We use this first option for parallelising search in our experiments throughout this work.

The second approach supported by Lucene is parallelising a single query across segments. Since a Lucene index segment is effectively a self-contained index, worker threads can execute a search query on individual segments, and the best results in each segment can be accumulated and ranked at the end. This second approach helps speed up individual queries on large indices, reducing the search latency. Unfortunately, this little known Lucene feature is not supported by ElasticSearch and Apache Solr (McCandless, 2019a).

Another possible benefit of the second approach is that it can alleviate performance degradation due to non-uniform memory access (NUMA) effects. We can ensure each thread accesses index segments corresponding to its NUMA node. Wwe can place index segments to memory mapped on different NUMA nodes. Worker threads can then perform search queries on index segments stored in corresponding NUMA nodes.

2.2 Design Challenges for Search Indices

We want to emphasise two fundamental tradeoffs to consider when designing index storage formats. As with most fields in computer science, there is a space-time tradeoff. Figure 2.8 uses compression schemes from industry and research benchmarks to compare the space efficiency of a compression algorithm with search performance. Algorithms that aggressively compress posting lists typically require more computation for decompression, resulting in worse query execution performance.

The second tradeoff is between the computational time of search and input-output (IO) time. Designs for index data structures have aimed to minimise IO to data stored on the file system. For example, key-value stores use computationally intensive filters to

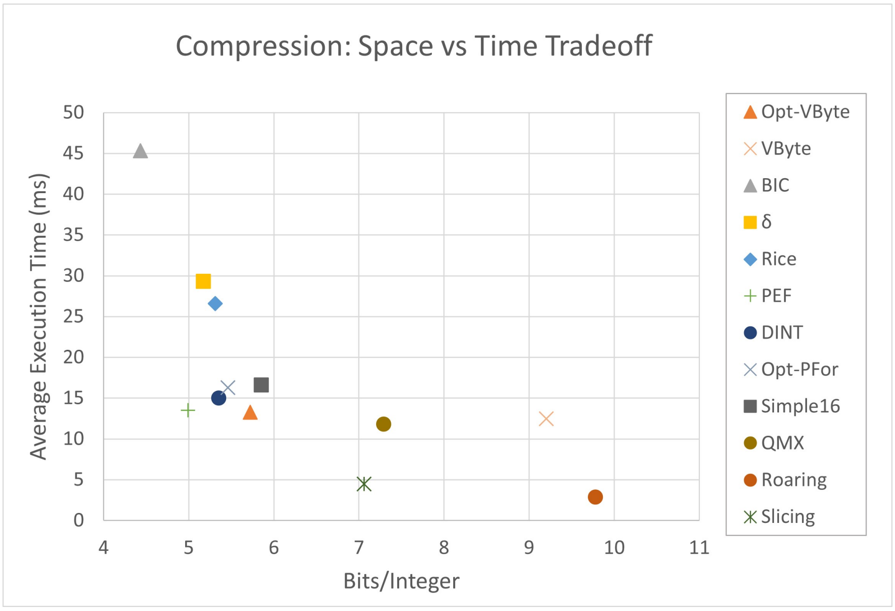


Figure 2.8: Compression efficiency vs *AND* query evaluation performance for various compression schemes. Adapted from Pibiri and Venturini (2020).

reduce the number of IO reads needed (Benson et al., 2021; Zhang et al., 2021). Similarly, Lucene's term dictionary uses an on-heap term index to reduce IO during term dictionary lookup and multi-level skip lists to reduce IO during posting list traversal.

The advent of commercially available NVM has changed these tradeoffs. With high capacity, low latency memory available on the main memory interface, the space-time tradeoff of compression algorithms has shifted. It is feasible to increase the size of search indices by making compression algorithms less aggressive. In doing so, we gain search performance improvements. Our work explores this tradeoff by experimenting with an entirely uncompressed index. The second tradeoff caused by block storage devices' high disk seek times no longer exists if the index is stored on NVM. Memory access latency of NVM is on the nanosecond-scale (Yang et al., 2020), unlike SSDs (microsecond-scale access latency) and hard drives (millisecond-scale access latency). Since memory accesses to NVM are not as expensive as secondary storage accesses, we must redesign caches and filters.

2.3 Non Volatile Main Memory

DRAM technology poses problems for system scalability due to its capacity limitations and high cost per gigabyte. DRAM capacity has doubled roughly every three years, and core counts for server-end processors have doubled every two years (Lim et al., 2009). We have seen that memory capacity per core is dropping 30% every two years (Lim et al., 2009). Document sets such as tweets, social media posts, blogs and websites continue to grow and must be indexed and stored efficiently to allow low-latency search.

Non-volatile main memory (NVM) technologies use non-volatile storage over the DIMM interface. NVM addresses DRAM's scalability concerns as non-volatile storage technologies scale better than DRAM in memory density. Further, NVM provides two significant benefits over DRAM due to its non-volatility. First, using NVM eliminates the system warmup cost of bringing application data stored on secondary storage devices to main memory. Second, while a power loss causes data loss on DRAM memory, NVM technologies can allow the intelligently designed software to recover data from mid-execution and even continue execution from a saved state with minimal performance overhead.

2.3.1 Intel Optane Persistent Memory

Intel Optane persistent memory (PM) is the most promising NVM technology for scalability and cost per gigabyte. The largest available Optane DIMM is 512GB, 4× larger than typical high capacity DRAM sticks. Optane PM has limited write endurance, similar to current SSDs employing flash technology. A controller on Optane PM handles the remapping of addresses to manage wear levelling. The write endurance of Optane PM is unclear due to the novelty of the technology. Intel rates Optane PM to perform reliably for five years under normal wear conditions Intel (2015).

Yang et al. (2020) characterize Optane PM using microbenchmarks. First byte read latency of Optane PM is 2× to 3× higher than DRAM. Interestingly, the gap between latency of sequential and random accesses is 20% for DRAM and 80% for Optane PM; Optane PM strongly prefers sequential memory access (Yang et al., 2020). Optane PM memory access throughput is heavily dependent on the application. Optane PM suffers from poor scaling of memory access throughput for multithreaded applications

The minimal unit of data that can be written on Optane PM is 256B, four times the cache line size of modern processors. Sequential writes to Optane PM are coalesced by the PM controller before making a single 256B write. However, random writes to Optane PM will not be coalesced, and consequently use four times the write bandwidth when compared to sequential writes.

Non-uniform memory access (NUMA) effects for Optane are much more severe for NVM than they are for DRAM, so programs must avoid cross-socket memory traffic Yang et al. (2020). Programs must especially avoid cross-socket memory accesses by multiple threads with a mix of loads and stores. In our experiments, we explore how search performance scales over multiple cores on both DRAM and NVM. We do not explore how NUMA affects search performance.

Optane PM can be used in two modes: *memory mode* and *app-direct mode*.

In our experiments, we set up PM as a file system in *app direct mode*. Figure 2.9 shows how an Optane persistent memory module set up in an app-direct file system is accessed

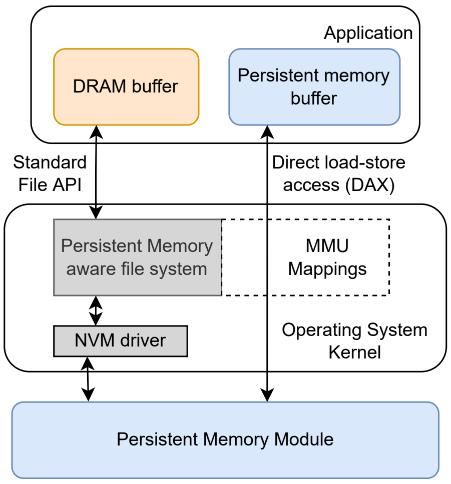


Figure 2.9: Optane PM as an *App-Direct* file system

from an application. Applications can access files on Optane PM using the file API's read and write system calls to copy files to and from a DRAM buffer. Alternatively, PM-aware file systems, such as ext4, support *direct load-store access* (DAX) (Wilcox, 2014). Using a DAX file system, applications can map persistent memory directly to their address space. Applications can then access files on Optane PM using simple load-store instructions. Using a DAX file system removes the need to first cache files on DRAM before accessing them.

In *memory mode*, Optane PM acts as main memory, massively increasing the main memory capacity available to programs. DRAM modules installed on the system alongside PM act as a cache for Optane PM main memory. Using Optane PM in memory mode is an interesting approach to potentially optimising search engine systems.

Prior NVM technology, such as AgigA Tech's AGIGARAM4, combined DRAM-flash hybrid DIMMs with supercapacitors to store DRAM state on flash memory in case of power loss (Sartore, 2011). While AGIGIARAM4 is as performant as regular DRAM, this technology does not scale in storage capacity to match growing index sizes. Another technology, IBM's eXFlash memory, was flash memory attached to a DRAM cache, connected over the DIMM interface (IBM, 2014). Flash memory's storage capacity scales far better than DRAM in absolute and per unit-cost terms. However, flash memory has a limited number of writes. In addition, the high-latency block-based interface used by flash memory is not suitable for latency-critical applications such as text search.

Chapter 3

Related Work

3.1 Text Search over Hybrid DRAM and Optane Persistent Memory

The closest related work to this work is a study by Akram (2021) using the C++ Psearchy search engine provided in the MOSBENCH benchmark suite (Boyd-Wickizer et al., 2010) on Optane Persistent Memory. Akram studies search indexing performance, crash consistency and query evaluation on a wide range of hardware-software system designs incorporating NVM into the existing search application. Akram finds that single term queries perform similarly on a Wikipedia search index placed on Optane DIMMs compared to DRAM. However, Akram finds that the 2-term *AND* queries do not perform as well on PMEM as on DRAM. The Psearchy engine does not use any index compression algorithms Boyd-Wickizer et al. (2010); Stribling et al. (2006). 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 the best of our knowledge, no prior work has considered using an entirely decompressed search index.

3.2 Optimising Key-Value Stores for Optane Persistent Memory

Prior works by Zhang et al. (2021) and Benson et al. (2021) have focused on designing and evaluating key-value stores (KV stores) for persistent memory. A KV Store is an application that is effectively a dictionary mapping keys to arbitrarily structured values.

A search engine's term dictionary is a KV store, except the values are posting lists. We present a review of these two works.

Zhang et al. design, create and evaluate ChameleonDB, an optimized KV store written in C/C++ for Optane persistent memory. Zhang et al. focus on reducing write amplification on Optane memory by aggregating writes to persistent memory when the KV store is updated. On the search side, Zhang et al. note that the log-structured merge tree (LSM tree) data structure used commonly in key-value stores on block devices such as SSDs does not work effectively on Optane DIMMs. We must not simply think of Optane DIMMs as fast SSDs. LSM tree data structures are designed to minimise disk reads needed to service a single search. Applications using LSM trees keep in-DRAM data structures for each block of values. The in-DRAM data structures are queried before making an expensive request to secondary storage. The main idea is that the extra computation cost on the nanosecond level is compensated many times over by saved microsecond or millisecond-level random accesses to block devices. Zhang et al. note this approach does not make the best use of Optane memory, which is only about 2 to 3× slower than DRAM. Using this observation, Zhang et al. designed a PMEM-backed hash table for mapping terms to their values. Zhang et al. find that their approach outperforms traditional LSM tree implementations targetted at hard disks and SSDs.

Benson et al. (2021) also construct a KV store for persistent memory. They optimise operations on KV stores using DIMM-aligned storage segments. We can connect one CPU socket to up to 6 Optane non-volatile DIMMs and map these DIMMs into program memory. The operating system interleaves memory maps so that different non-volatile DIMMs contain subsequent 4KB pages. Benson et al. reason that by sizing storage segments of their KV store to 4KB chunks, they can minimise DIMM contention on writes. Benson et al. also align threads uniformly to DIMMs by assigning threads to work on different memory regions. These load balancing and contention minimising optimisations cause a 4–18× improvement to various disk-based KV store algorithms.

These prior works in KV store optimisation highlight the point that algorithms initially designed for block devices must be modified to reflect the unique tradeoffs offered by NVM. To the best of our knowledge, we are the first to consider how search index algorithms, in particular search index compression, interact with NVM technology. We tackle the challenges posed by this novel memory technology by optimising a widely adopted, managed search engine framework.

3.3 Software Emulation

Many prior works have explored software emulation of NVM technology (Coburn et al., 2011; Kwon et al., 2017; Moraru et al., 2013). In contrast, our work presents data on a system with real persistent memory. Yang et al. (2020) used Intel Optane DC Persistent Memory to provide a detailed review of a prior work by Xu et al. (2019). Xu et al. (2019) used NVM emulation to evaluate two NVM-specific optimisations for

RocksDB. Yang et al. found that hardware tests reached the opposite conclusion to emulation, showing that emulation is insufficient for modelling the performance impact of persistent memory.

Chapter 4

Latency Impacts of Index Compression on Search

We developed a methodology to create compressed and uncompressed inverted indices. We then designed an experiment to measure the performance impact of compression for search queries executed on a DRAM-backed search index. The methodology we present in this chapter is vital to our work exploring NVM in chapters 5 and 6.

4.1 Methodology

This section details the hardware and software platforms we used for our experiment. To measure the impact of compression on search performance (i.e. average query latency), we compare the performance of search on a DRAM-backed compressed index and an uncompressed index on the DRAM-backed managed heap. Our implementations of these two systems account for a wide range of possible sources of unpredictability to perform a fair test to evaluate the impact of compression on search performance.

4.1.1 Software Platform

We use the *luceneutil* project: the de-facto benchmark suite for the Lucene project. Integration tests developed by the Lucene core developer team track the change in indexing and query evaluation performance using a range of diverse query sets and indices. Luceneutil provides the infrastructure to construct an index built from the full text of English Wikipedia. We use the full text of Wikipedia as of 20 March 2021 to build a 5.3GB on disk in the default Lucene compressed index format. Query sets provided in the luceneutil project perform single term queries, *AND* and *OR* queries, and more complex phrase and fuzzy match queries. Due to the rapid release cycles of both Lucene and luceneutil, we froze the luceneutil version current at 12 April 2021 and Lucene version

8.9.0 for this project. We use OpenJDK Java 13 for this project, and all experiments in this project use the G1 garbage collector. For all experiments we used the `Indexer.java` and `SearchPerfTest.java` classes from luceneutil, with some modifications [1] described in section 4.1.3.

We briefly considered using the Dacapo benchmark suite's *lusearch* benchmark as the starting point of our experiments (Blackburn et al., 2006). Lusearch is a Lucene text search benchmark in the Dacapo benchmark suite, an actively maintained Java benchmark suite with extensive support for performance analysis. We chose luceneutil as the Lucene's maintainer (Mike McCandless) manages it. Further, luceneutil provided support for quickly changing the Lucene indexing and search configurations.

4.1.2 Hardware Platform

We conducted all experiments on a dual-socket Dell PowerEdge R740 at ANU running Ubuntu Linux Version 18.04 LTS. The processors on each socket are both Intel Xeon Gold 6252N at 2.3Ghz, each with 24 physical cores (48 logical cores), making up 96 cores in the system. Each core has 35.75MB of shared L3 cache. Each core has an integrated memory controller supporting six memory channels. Each memory channel connects to a 32GB Micron DDR4 DIMM and a 128GB Intel Optane DIMM. With 12× 32 GB DIMMs and 12× 128GB Intel Optane DIMMs, the total memory capacity of the system is 384GB of DRAM and 1.5TB of Optane PM. The system has a 1.5 TB Intel Optane PCI Express NVMe SSD (DC P4800X). Interestingly, both the Optane SSD and Optane DIMMs use the same storage medium but sit behind different interfaces. We do not investigate NUMA effects, so we perform our experiments on only one of the CPU sockets.

In this chapter, we do not use the Optane PM hardware available on this system. All experiments use DRAM. Chapters 5 and 6 use the Optane PM for experiments and refer back to this section which describes the hardware platform used for the experiments in those chapters.

4.1.3 Experimental Setup

To test the impact of on-demand index decompression, we use two Lucene *PostingsFormat* implementations, each defining the data structures and algorithms used for term dictionaries and document lists. We visually summarise the two index implementations in figure 4.1.

1. *Compressed index*: We build our baseline compressed index using the default index implementation of Lucene 8.9.0 that we described in section 2.1.3. The default index implementation is provided in the `Lucene84PostingsFormat` class. In this

[1] The source code for all experiments is found in `gitlab.anu.edu.au/u6679031/luceneutil`

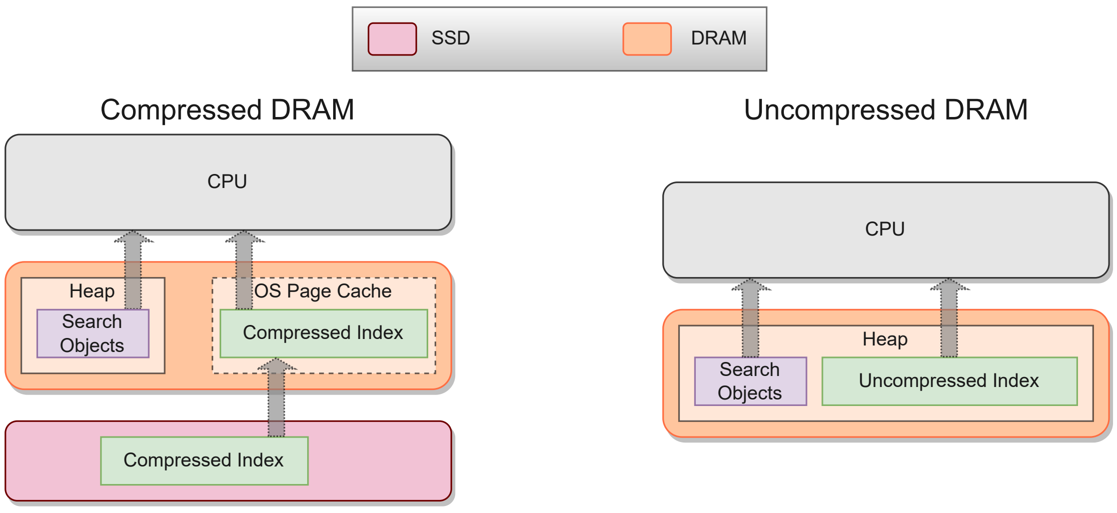


Figure 4.1: Showing how the CPU accesses the *compressed* index (left) and the *uncompressed* index (right)

implementation, the index is in compressed form on the file system, and when the searcher services a query for a specific term, it decompresses the posting list associated with that term. We described the data structures and algorithms used in the compressed index in full detail in section 2.1, but replicate the most relevant information in table 4.1.

2. *Uncompressed index*: The uncompressed search index format is provided by the `DirectPostingsFormat` class in Lucene 8.9.0. `DirectPostingsFormat` is essentially a wrapper on the baseline `Compressed` format. The index is created and stored on the file system in the same data structures using the same compression algorithm used by the `Lucene84PostingsFormat`. However, before servicing search queries, we first decompress index data structures stored on the file system into Java `byte` arrays and `int` arrays stored on the Java heap. The searcher services queries by accessing the necessary data on the heap in decompressed form. Importantly, we perform all decompression ahead of time. We do not spend computational resources decompressing any index data structures during query evaluation.

We present the relevant differences between the two configurations we used in table 4.1.

	Compressed Index	Uncompressed Index
Skip List Data Structure	Multi Level Skip List (on file system, 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 a custom ASCII compression scheme. Postings offsets are VLB-delta encoded and stored in variable length blocks	Stored in 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	File system	JVM Heap
Size of Wikipedia Index	5.38GiB	52.78GiB

Table 4.1: Configuration differences between the compressed vs uncompressed search index

4.1.4 Query Set

In all the experiments in this thesis, we use two query sets to evaluated query latency:

- A set of single term queries of 45786 English words.
- A set of 5921 2-term *AND* queries querying the index for documents containing two English words.

4.1.5 Controlling Jitter in Experiments

We designed experiments to remove possible sources of unpredictability or jitter from our results. We first describe the sources of unpredictability and then present the steps we took to account for them.

NUMA Effects for Remote memory accesses

On the dual-socket machine we used for testing, non-uniform memory access slows down memory accesses significantly. If we do not account for the difference between memory accesses on local and remote NUMA nodes, we expect to find more variability in our performance results, making it more difficult to conclude differences between the different setups we test. In all our experiments, we did not allow non-uniform memory access (NUMA) to affect performance results. We used *numactl*, a command-line utility, to allocate all memory to our benchmark program on the same NUMA node as the socket on which we conducted experiments.

Controlling SMT and Operating System Scheduling Jitter

To measure the scalability of our results with increasing core counts, we ran experiments running search queries using thread pools containing 1 to 48 threads. We used the Affinity library developed by Yang et al. (2016) to pin each query processing thread (worker thread) to a specific logical SMT core. Pinning threads to specific cores guarantees that no two worker threads are executing on the same physical core unless there are more worker threads than available physical cores.

We performed extensive performance counter analysis later in this work. Performance counter analysis is also more straightforward and accurate when worker threads are pinned to specific logical cores as we avoid all non-determinisic thread migrations initiated by the operating system scheduler.

Operating System Paging Effects

We store the compressed index on an EXT4 file system on an SSD. When the searcher accesses index data structures for the first time, the operating system caches the operating system (OS) pages accessed in the OS page cache on DRAM. The OS page cache services the request the next time the same page is accessed. We run our query set five times and take performance measurements on the fifth run. Since the index size is far smaller than the available DRAM capacity, we expect to always to hit the OS page cache when accessing the index on the fifth run of the query set. We considered using a Temporary File System (TMPFS) to place the compressed index directly on DRAM. However, TMPFS scales poorly on multithreaded benchmarks (Akram, 2021).

Blackburn et al. (2008) identify three key runtime variables that affect a benchmark executing on a managed language runtime: heap size, non-determinism and warm-up. We describe how we control these variables in our experiments.

Heap Size

Controlling the Heap Size is vital when benchmarking Java programs. We do not perform a sensitivity analysis of the effect of the Java heap size on search performance. Measuring the effect of garbage collection (GC) on search performance is not the goal of our work. In our work, we are comparing how the algorithms for search on compressed and uncompressed indices interact with the memory technology used to store the index. We took steps to control the heap size so that we can compare performance across algorithms and hardware without conflating the costs of garbage collection.

Figure 4.1 shows that the compressed index is on the file system in SSD. It is copied to the OS page cache during search. The index data structures are almost entirely off-heap. Instead, the heap contains objects used when computing individual queries: we call these objects *search objects*. Search objects are produced during each query, and they become garbage upon query completion.

The uncompressed form is stored as integer and byte arrays on the Java heap. The index is allocated on the heap before the program starts processing queries. The objects comprising the uncompressed index are immortal, and are read-only during query evaluation. Thus, the on-heap uncompressed index is a large set of immortal objects, while Lucene search objects have a small lifespan.

We show the garbage collector settings for both the compressed and the uncompressed index in table 4.2. The garbage collector settings are necessarily different between the uncompressed and compressed index setups. The large heap size necessary for uncompressed indices will result in different garbage collector behaviours between the two experiments. We use a relatively large heap size for the compressed index experiments as well, so GC is not a significant proportion of execution time. We find this approach to be quite successful: in the experimentation for chapter 5, we verify that GC pause times are below 6% of the elapsed time for both the uncompressed and compressed index setups.

	Compressed	Uncompressed
Initial Heap Size	2GB	128GB
Maximum Heap Size	32GB	128GB

Table 4.2: Uncompressed vs compressed search index experiment: Heap size settings

Non-determinism

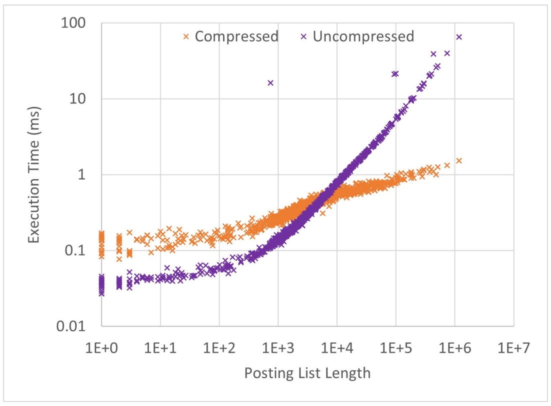
Java uses just-in-time compilation. This means the Java virtual machine (JVM) generates optimised machine code from an intermediate representation called Java bytecode at runtime. JVM optimisations are guided by metrics collected during runtime. The optimisation is thus non-deterministic across different invocations of the Java program (Blackburn et al., 2008). Blackburn et al. suggest three methods to account for this non-determinism. We chose the most accurate method: statistical analysis on sufficient data points. The benefit of this approach is that it accounts for the broadest range of sources of non-determinism on the machine tested. For every configuration for which we report performance measurements, we run at least five invocations of the JVM. We present the mean and the 95% confidence interval in all performance data.

Warm-up

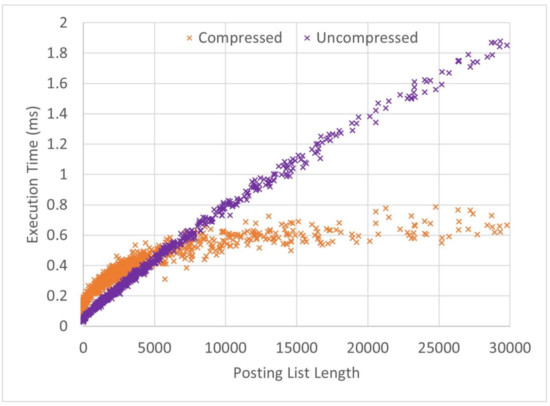
As the program executes code in classes more frequently, the JVM uses increasingly aggressively optimising compilers to optimise the native code. This process is called tiered compilation. A single invocation of our program executes the same code for query evaluation repeatedly until the query set is exhausted. The first few queries executed by the benchmark application are the slowest as the JVM must first compile the class bytecode into machine code. Later iterations of the query set will be less affected by compilation overhead, and will be executing machine code of a higher quality when compared to earlier iterations. This effect is called *warm-up*. We must collect performance data only after the code has warmed-up. To reduce this warm-up 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.

To account for warm-up time in our experiments, we run the query set five times within each invocation and collect performance measurements for all queries in the last run of the query set.

4.2 Preliminary Results



(a) Entire query set on logarithmic axes



(b) Showing posting lists of length ≤ 30000

Figure 4.2: Single term query execution time plotted against total posting list length for the query term (lower is better)

Single term queries require scanning through all postings in the index to find the most relevant documents. Therefore, we expect a linear relationship between the execution time of a query and the total length of posting lists across index segments.

Our experiments found that query evaluation the uncompressed index has $3\times$ the average latency as the compressed index. It is counterintuitive for the uncompressed index to perform worse than the compressed index as search on the uncompressed index need not perform posting list decompression, saving computation time.

We explore these trends. Figure 4.2 plots total posting list length against single term query performance for the uncompressed and compressed index formats on DRAM. We see that search performance is linear with posting list length on the uncompressed index. However, search performance is logarithmic for the compressed index format. This suggests that searching on the compressed index search uses a fundamentally different algorithm to searching on the uncompressed index.

4.2.1 Changes to Search on Compressed Indices in Lucene

The data in figure 4.2 suggested that the algorithm used for searching compressed indices is of a different complexity class than the uncompressed index search algorithm. After code analysis, we found that Lucene implements "maximum impact indexing" for its compressed index format.

Maximum impact indexing is an optimisation applied in Lucene to reduce posting list traversal time significantly. *Impacts* are an upper bound maximum relevance scores that are possible for a given document or set of documents. Lucene can calculate this upper bound for the relevance score of a document during indexing, before queries are executed. To achieve this, Lucene exploits specific mathematical properties of its scoring function detailed by Grand et al. (2020). The multi-level skip list stores impact scores that represent the maximum relevance scores possible for any document in a large chunk of the posting list.

Without the maximum impact indexing optimisation, the posting list is traversed from left to right, scoring every document in every block of the posting list. With the optimisation, 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 for the query. The searcher only decompresses and traverses a block if it contains documents more relevant to the search query than the current candidate results. For longer posting lists, maximum impact indexing speeds up query evaluation drastically. Much of the posting list is not decompressed and scored at all.

To compare the uncompressed and compressed indices fairly, we turn off the maximum impact indexing optimisation when searching over the compressed index. We justify this change to the state-of-the-art system for three reasons. First, the optimisation was added to Lucene in version 8.0 in 2019 (Woodward, 2019). Indices built with older versions of Lucene will not have this optimisation enabled as the optimisation requires significant changes to the index structure (typically done by re-indexing) (Grand et al., 2020). Second, many search index implementations do not implement this optimisation. The maximum impact indexing optimisation is possible for Lucene's index due to specific mathematical properties arising from the design of Lucene's scoring function (Grand et al., 2020). On the contrary, Bing, for example, uses machine learning models for relevance scoring (Janapa Reddi et al., 2010) and hence can not implement maximum impact indexing. Thirdly, the total hit count for queries reported by the optimised implementation is inaccurate since not all matches are visited. Users may wish to know how many results exist for a specific query, which is no longer possible. While the maximum impact indexing optimisation is a critical performance optimisation to Lucene, it is not characteristic of what many search providers implement. We turn off this optimisation.

4.2.2 Hardware Performance Counter Analysis of Lucene Search

After removing the maximum indexing optimisation, we found that search performance on the uncompressed index was still worse than on the compressed index for long posting lists. However, it now performed much better for short posting lists.

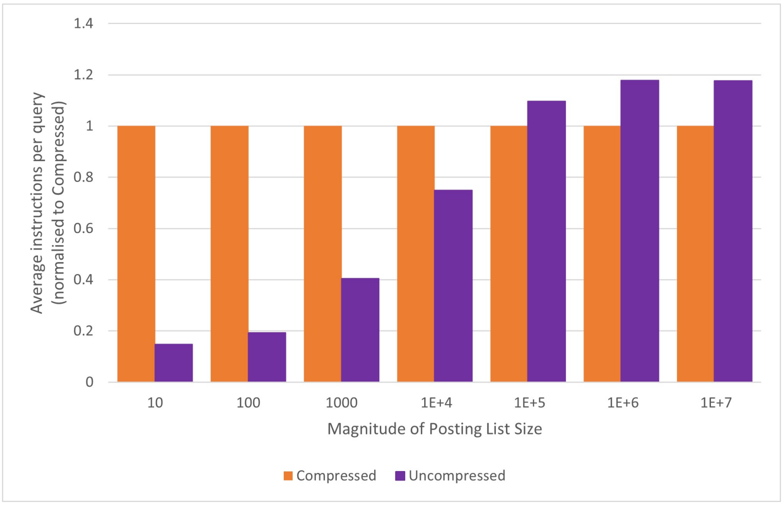


Figure 4.3: Average instructions per query, bucketed by the magnitude of their posting list sizes and normalised to the compressed index (lower is better)

We conducted some performance analysis using Intel hardware performance counters to analyse why the decompressed index performs worse than the compressed index for long posting lists. Performance counters are special registers on Intel CPUs which can be programmed to count hardware events such as the number of CPU instructions completed or the number of CPU cycles passed. With the support of the operating system scheduler, we can measure these performance counters at the CPU hardware core level or the software thread level.

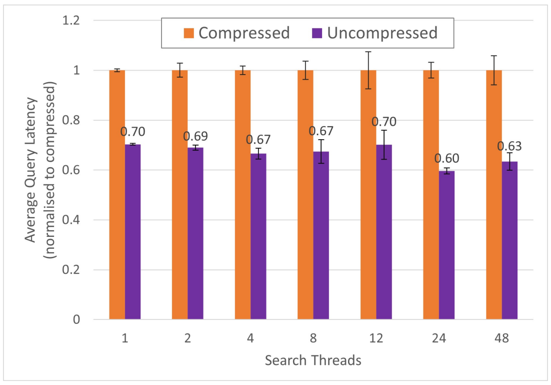
We measure performance counters at the start and end of every query evaluated using the methodology developed by Yang et al. (2016) using the Libpfm (2008) library. We report the average instructions executed per query for both index setups in figure 4.3, bucketed by the magnitude of the posting list size. We find that the searcher executed fewer instructions on an uncompressed index for short posting lists. However, for posting lists containing more than 10^5 postings, a query on the uncompressed index executes more instructions than the compressed index. It is counter-intuitive that a query on the uncompressed index requires more instructions since it does not require posting list decompression. Our observation hinted at the possibility of a performance bug in the uncompressed index implementation.

Bug Fix to Lucene Uncompressed Index Search

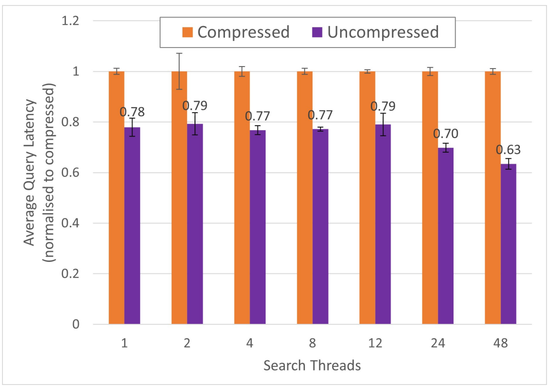
We searched the code base and found a performance bug in Lucene's algorithm for search on an uncompressed index. We found the bug in the code that iterates the posting list. Recall that the uncompressed index stores document IDs in java integer arrays. Iterating to the next element of the posting list should be as simple as incrementing an index variable. However, Lucene's search implementation on uncompressed indices performs a binary search every time it needs the next element in the array, which is unnecessary computation. We fixed this performance bug and saw a significant improvement in search performance for all values of posting list length. We report these results in section 4.3.

4.3 Results

We present a comparison of search performance over the compressed and uncompressed index setups for single term and 2-term *AND* queries in figure 4.4. We find a consistent performance improvement for both query sets using the uncompressed search index, which scales with increasing core counts. For search using 48 logical cores, both query-sets measure a 37% performance improvement.



(a) Single Term Queries



(b) 2-Term *AND* Queries

Figure 4.4: Query execution time for search over scaling core counts on Compressed and Uncompressed index setups (lower is better)

4.4 Summary

This chapter measures the performance improvement gained from searching on an uncompressed index over a compressed index. We store both indices on DRAM. We design a methodology to measure Lucene search performance using the luceneutil benchmarking project to make an index from the full English text of Wikipedia and perform single term and 2-term *AND* queries on it. We modify the implementations of both setups to ensure a fair comparison. We find that average search latency on an uncompressed index is 37% faster than search on a compressed index; however, the uncompressed index is 9.8× larger than the compressed index.

Chapter 5

Search Indices on Non-Volatile Memory

In the previous chapter, we focused on DRAM and designed a methodology to evaluate the effect of on-demand decompression on search performance. We found that search on an uncompressed index is 37% faster than a compressed index. This chapter considers methods to place the uncompressed index on non-volatile memory (NVM). We then compare the search performance of both the compressed and uncompressed indices on DRAM and NVM.

5.1 Methodology

We modify the methodology developed in chapter 4 to store the compressed and decompressed search indices on NVM.

5.1.1 Placing the Compressed Index on NVM

Storing the compressed index on NVM is easy. In section 2.3.1 we discuss using the persistent memory DIMMs as a direct access (DAX) file system in App-Direct mode. We use the DAX feature to store the compressed index on an NVM file system and perform search queries directly on it.

5.1.2 Placing the Uncompressed Index on NVM

We encountered a significant challenge in placing the uncompressed index on NVM. Since the uncompressed index sits in the JVM heap and not in files like the compressed index, we must use a different method than the simple method described for the compressed index. We considered four approaches to placing the uncompressed index on NVM.

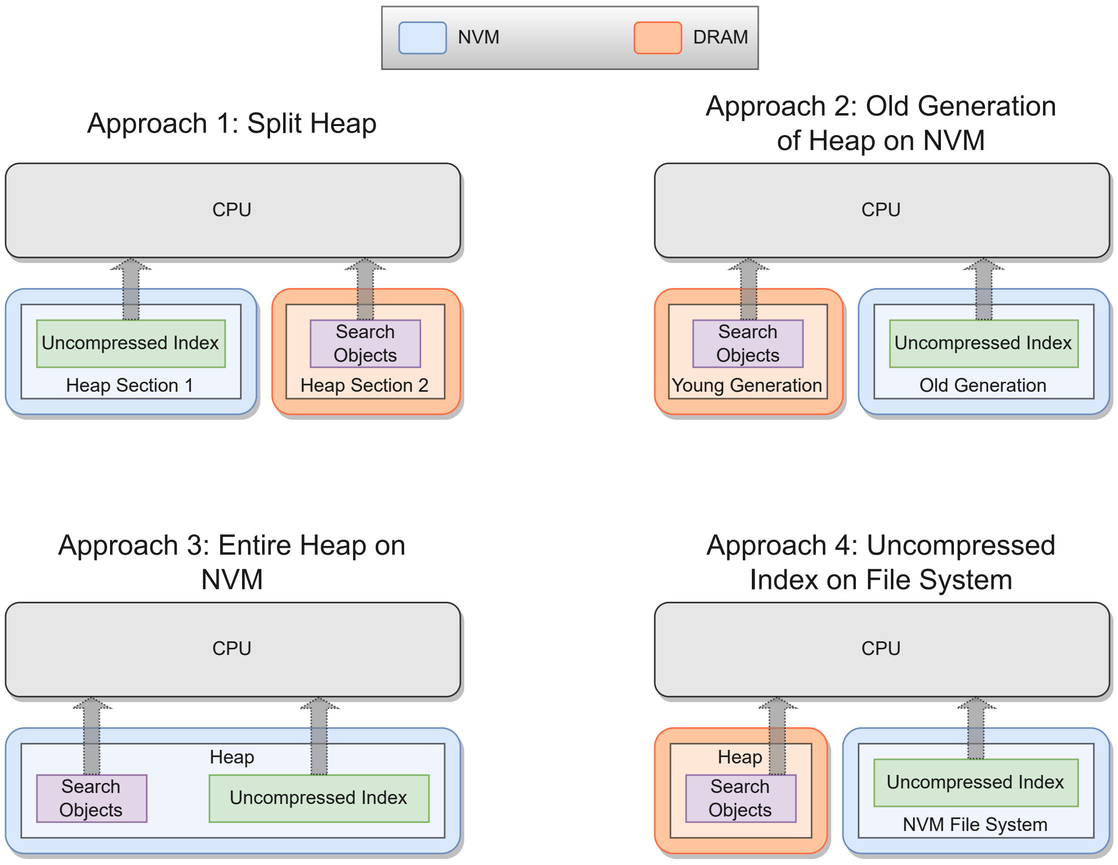


Figure 5.1: Proposed approaches to placing the uncompressed index on NVM

Approach 1: Split Heap

Approach 1 splits the JVM heap into two segments. We put the first segment to NVM-backed file. Before servicing queries, we decompress the index and place it in the NVM-backed heap segment. Next, we put the second segment of the heap to DRAM. When the searcher services a query, the temporary objects it creates for query resolution are placed on the DRAM-backed heap segment.

Approach 1 has the benefit that once we implement the necessary changes to the JVM, it is straightforward to programmatically split the heap. In the codebase for creating the uncompressed index, we can annotate all code that allocate objects allocate them to the heap segment on NVM. All the search objects will be allocated to DRAM as usual.

However, we need to make complex changes to the JVM to implement this approach. We wanted to develop an approach using an industry-standard JVM implementation. We can make a stronger argument for why commercial search implementers should use our implementation. Kolokasis et al. (2021) implement *Teraheap*, an OpenJDK extension that splits the heap into two segments precisely as we need for approach 1. However, the code base for this research JVM is still experimental. We decided to to explore this approach to avoid stability issues arising from using an experimental JVM.

Approach 2: Old Generation Heap on NVM

Approach 2 attempts to approximate approach 1 by using the default Java heap layout. The JVM heap is split into the young and old generations. The *young generation* is a region of the heap that contains newly allocated objects. Objects in the *young generation* that are still in the program's scope after garbage collection are moved to the *old generation* region of the heap. Approach 2 places the heap's old generation on NVM and the young generation onto DRAM.

1. We place the uncompressed index on the heap at the program's start.
2. We prompt a full garbage collection. The garbage collector will shift the index objects into the NVM-backed old generation.
3. The searcher then starts executing queries.

Search objects will be allocated to the young generation heap on DRAM, and young generation garbage collection will clear garbage created from query resolution.

OpenJDK versions 12 to 15 provide a Java command-line option called `AllocateOldGenAt` (Kharbas, 2018; Schatzl, 2020) which allows us to place the old generation heap on NVM.

Approach 3: Entire Heap on NVM

Approach 3 (A3) places the entire heap on NVM using a Java command-line option called `AllocateHeapAt` (Kharbas, 2016). Since both the search objects and the index are stored on NVM, we expect this approach to perform worse than A2 since memory accesses to search objects will be slower.

Approach 4: Uncompressed Index on File System

Approach 4 (A4) is to design a new file-based uncompressed index format in Lucene and place the index files on an NVM-backed DAX file system. A4 uses the file system for storage, which provides the practical benefits of permanence and ease of creating backups.

Designing a file-based index format requires significant software engineering effort of thousands of lines of code as we must implement all the methods of a Lucene *postingsFormat*. We decided to implement this approach if time permitted after experimenting with A2 and A3. Our experiments with A2 and A3 found exciting results, so we performed extensive performance analysis to explain these results. After this analysis, we did not have time to implement A4 and leave it to future work to investigate.

5.1.3 Evaluation of NVM Placement Approaches

Approaches 1,2 and 3 place the index on the JVM heap. However, popular garbage collectors use tracing algorithms that scale in overhead with the number of live objects on the heap. Kolokasis et al. (2021) use Apache Spark to show that storing large datasets on the JVM heap causes garbage collection time to exceed half of the total execution time of Spark benchmarks. The poor scalability of garbage collectors to large heaps is greatly exacerbated with NVM-backed heaps since traversals over heap objects in NVM are much slower than for DRAM-backed heaps. Since data caches are typically allocated and freed in bulk, frequent garbage collection over data heaps is unlikely to free much memory. To this end, Teraheap's garbage collector (Kolokasis et al., 2021) improves the performance of collections and reduces the frequency of collections over the large data heap.

We expect that approach 1 is likely to be the most performant implementation of approaches 1,2 and 3; since a split heap designed for storing large data sets will incur lower garbage collection costs. However, we decided not to implement approach 1 as it is still an experimental codebase and requires a non-standard JVM. Furthermore, our results in section 5.2 show that garbage collection did not have a major impact on performance.

We expect approach 2 to perform better than approach 3, since it places search objects on the DRAM-backed young generation heap. Approach 3 provides a lower bound for the performance of approach 1 and approach 2: since both the index and the search objects are on NVM, more memory accesses to NVM will occur using A3 compared to A1 and A2.

We continue experimentation using approaches 2 and 3. We choose these approaches as the infrastructure for NVM heap placement is readily available in OpenJDK, allowing us to generate some indicative results and conduct performance analysis.

5.1.4 Experiment Design

Proportion of LLC	Size
0.25× LLC	8.94MB
1× LLC	35.75MB
4× LLC	143MB
16× LLC	572MB
64× LLC	2288MB

Table 5.1: Young generation sizes tested for *Uncompressed NVM 1*

We compare five distinct configurations in our experiment.

Compressed DRAM and *Uncompressed DRAM*: These two systems are the same as from section 4.1.3.

Compressed NVM: The compressed index is placed on an NVM-backed DAX file system.

Uncompressed NVM 1: This is approach 2 described in 5.1.2. The old generation containing the index is sized at 128GB (fitting the uncompressed index, 53GB) and placed on NVM. The young generation is on DRAM, and we must configure it separately to limit DRAM usage. We used the default garbage collector for our JVM, the garbage first garbage collector (G1GC) for all experiments. G1GC uses a variable-size young generation, but we can configure the maximum size for the young generation. We measure the effect of the maximum young generation size by testing five different young generation sizes set at multiples of the last-level cache (LLC) size (35.75MB), summarised in table 5.1. We also test this setup without restricting the young generation size.

Uncompressed NVM 2: This is A3 described in 5.1.2. The entire heap is sized at 128GB and placed on NVM.

For *Uncompressed NVM 1*, we must ensure the index is stored in the old generation heap on NVM. We call a full garbage collection using `System.gc()` after loading in the decompressed index onto the heap. A full garbage collection serves two purposes. First, since the decompressed index survives garbage collection, it will be placed on NVM. Secondly, decompressing the index onto the heap creates huge amounts of garbage. If this fills up the old generation, the old generation garbage collection on NVM can seriously impact search performance on the *uncompressed NVM 1* and *uncompressed NVM 2* and UN2, making it difficult to compare memory technologies and search algorithms. For consistent methodology, we call `System.gc()` right before executing queries on all five configurations.

We also note that the heap sizes are consistently 128GB across the uncompressed setups, except that *Uncompressed NVM 1* also has a young generation heap on DRAM. Using a large heap avoids garbage collection affecting performance results, making it easier to compare memory technologies and index setups.

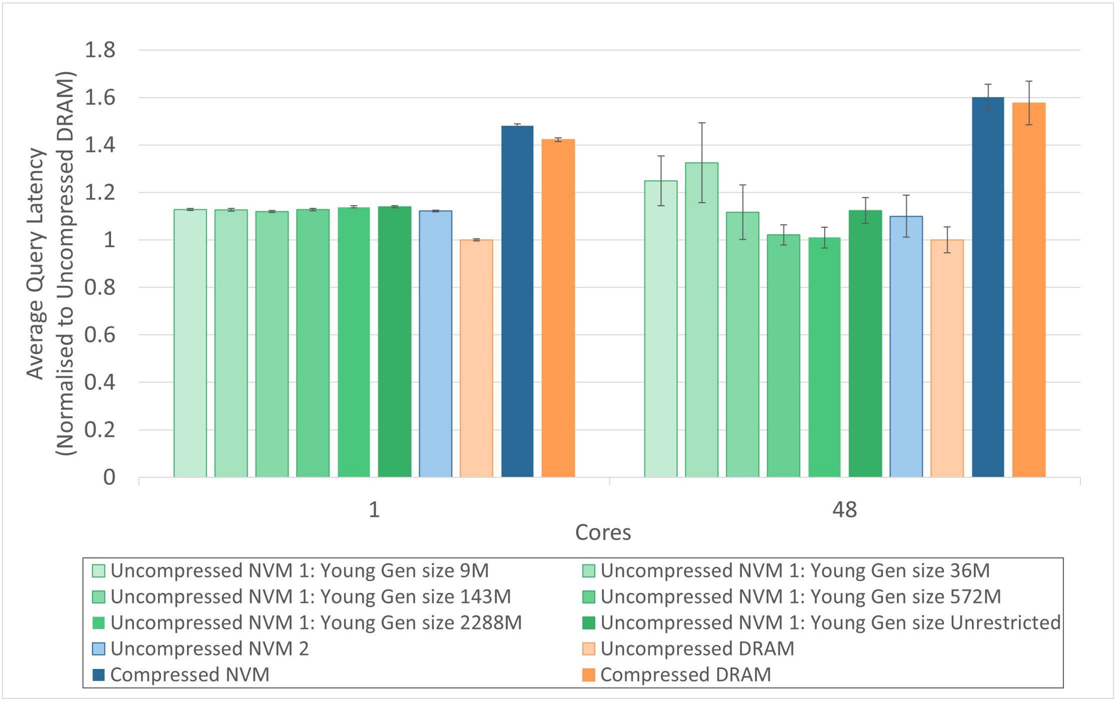


Figure 5.2: Comparing average latency of single term queries between NVM-backed indices and DRAM-backed indices (lower is better)

5.2 Results

In figure 5.2, we present the comparison of average latency for single term queries executing inverted indices stored in the five configurations we tested. We present data for single threaded and fully multithreaded (48 core) workloads. We first observe that *Uncompressed NVM 1* and *Uncompressed NVM 2* both perform better than *Compressed NVM*. This observation matches our expectations as we observed the same trend between *Uncompressed DRAM* and *Compressed DRAM* in the previous chapter.

For a 48 core configuration, *Uncompressed NVM 1* marginally outperforms *Uncompressed NVM 2* when we provide it with a sufficiently large young generation heap size on DRAM. However, on 48 cores, *Uncompressed NVM 1* is worse than *Uncompressed NVM 2* for small young generations. We see that search performance of *Uncompressed NVM 1* is sensitive to the number of search threads, since multithreaded search produces garbage at a faster rate, causing more garbage collection overhead for *Uncompressed NVM 1* with small young generation heaps. The other four configurations show no difference in their relative performance for different core counts.

The exciting result is that *Uncompressed NVM 2* outperforms the state-of-the-art *Com-*

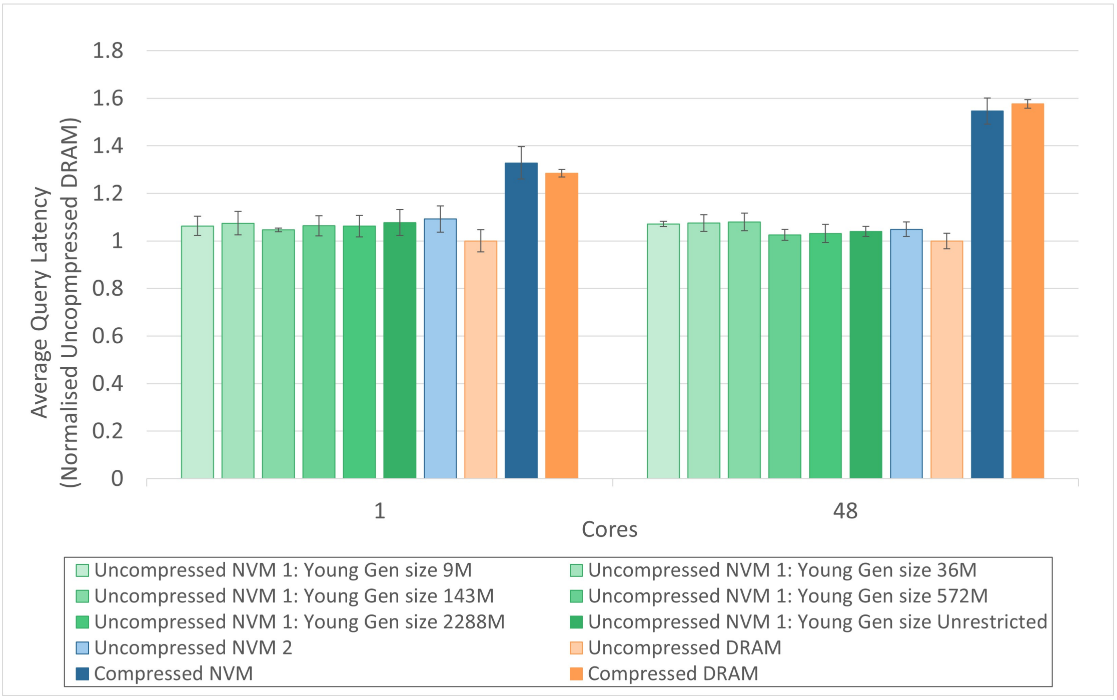


Figure 5.3: Comparing average latency of 2-term AND queries between NVM-backed indices and DRAM-backed indices (lower is better)

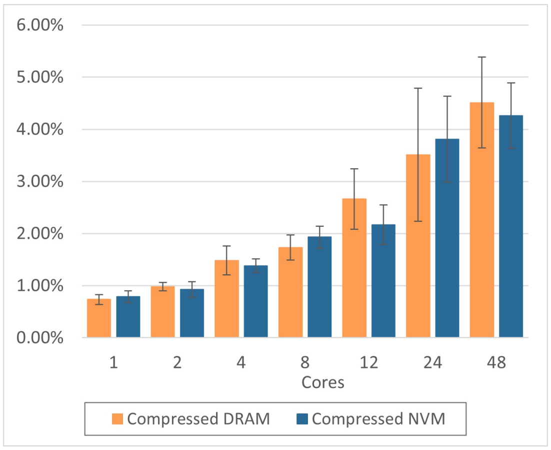
	Compressed		Uncompressed	
	DRAM (SoA)	NVM	*DRAM*	NVM 2
Single Term	15 (± 1)	15 (±1)	11 (±1)	11 (± 1)
2-Term AND	301 (±10)	293 (±8)	194 (±11)	194 (± 5)

Table 5.2: 99th percentile latency for 48-core search

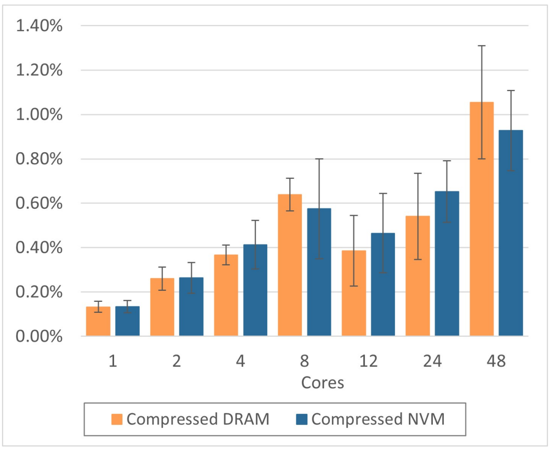
pressed DRAM setup by 30% for single term queries. Figure 5.3 shows that 2-term *AND* query performance is also 33% better for *Uncompressed NVM 2* compared to *Compressed DRAM*. Table 5.2 shows the latency of the 99th percentile longest executing queries for all setups. We note that *Uncompressed NVM* 35% (100ms) improvement in the latency of the slowest 1% of 2-term *AND* queries.

There is only a little performance benefit from placing search objects on DRAM instead of NVM. Placing the young generation heap on DRAM does not significantly improve performance. so we focus on *Uncompressed NVM 2* as the *uncompressed NVM* system for our discussion herein since it is simple and performant. While the uncompressed index is 9.8× larger than the compressed index, *Uncompressed NVM 2* is logistically feasible due to the scalability of NVM technology.

Perhaps the most puzzling observation from this data is that the performance difference between NVM and DRAM is little for both the compressed and uncompressed indices. For the compressed index, placement on NVM had a statistically insignificant effect on search performance. Focusing on the uncompressed index, *Uncompressed NVM 2* is only 10% slower than *Uncompressed DRAM* for single term queries and only 5% slower for 2-term AND queries. Yang et al. found that memory access latency of Intel persistent memory (NVM) is 2× to 3× higher than DRAM for microbenchmarks. Further, the memory bandwidth of NVM is at least 65% lower than for DRAM for a multi-threaded microbenchmark. Given the considerable performance gap between the two memory technologies, observing only a slight difference in search performance between NVM and DRAM is puzzling.



(a) Single Term Queries



(b) 2-Term AND Queries

Figure 5.4: Time elapsed in garbage collector as a proportion of elapsed time

We validate that garbage collection overhead is not a significant proportion of execution time. We use the Java *GarbageCollectorMXBean* interface, which accesses the G1GC's internal data structures containing monitoring information, including the young and old generation counts and elapsed times. Figure 5.4 shows that time elapsed in garbage collection as a proportion of elapsed wall-clock time in the fifth run of the query sets when performance data was collected. The garbage collection overhead is less than 6% of the elapsed time for all experiments. We only show the garbage collection overheads for the compressed index setups, as the young and old garbage collections never occurred for the uncompressed index setups *Uncompressed NVM 1* and *Uncompressed DRAM*, due to their large heap sizes. This measurement validates that the $\approx$ 30% difference between the uncompressed and compressed index setups is not due to garbage collection artefacts. We have also ruled out garbage collection effects as a cause for the puzzling observation that NVM is only slightly worse than DRAM for search index storage.

We argue that search algorithms are memory capacity intensive but not sensitive to memory latency. The computations for intersection, scoring and ranking help hide the higher memory latency of NVM. The bandwidth of NVM is still sufficient to satisfy search queries. To justify this claim, we conduct an extensive microarchitectural performance analysis in section 5.3.

5.3 Microarchitectural Performance Analysis

We make use of the *Top-Down method for performance analysis and counters architecture*, a paper of the same name by Yasin (2014). We use the top-down methodology to deepen our understanding of how the algorithms interact with memory devices and explain our results from a microarchitectural perspective. The top-down methodology hierarchically finds *true bottlenecks* on modern out-of-order processors by measuring heuristics at different levels of the analysis hierarchy shown in figure 5.5. By recursively narrowing down on components of the processor causing bottlenecks, we can find and measure specific performance pathologies of both the compressed and uncompressed index search algorithms on DRAM and NVM.

5.3.1 Top level Breakdown

The modern out-of-order CPU has two parts: a frontend and a backend (Yasin, 2014). The frontend fetches (CISC) instructions from memory and decodes them into micro-operations (μops), and feeds the μops to the backend portion. The backend schedules the μops, executes (performs) the μops out of program order, and commits (retires) these μops in program order.

An ideally optimised program on a modern superscalar Intel processor executes four μops every clock cycle. A pipeline slot refers to the behaviour of one of the four superscalar units on a particular clock cycle. The top level metrics classify each pipeline slot on the processor into one of four categories shown in figure 5.6. If a μop is issued at a particular slot, it is either completed to retirement (*retire*) or cancelled and rolled-back due to a *bad speculation*. If a μop is not issued, the processor is not executing instructions at maximal throughput. μop issue may stall because of either the unavailability of resources

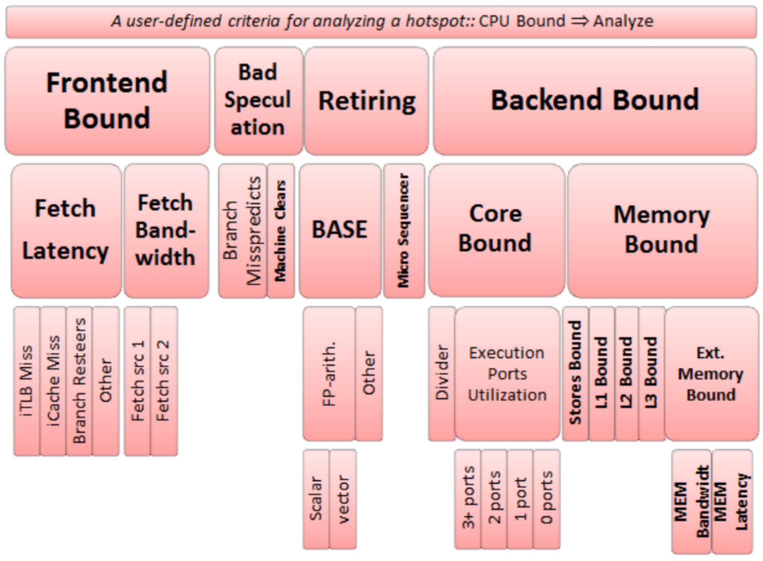


Figure 5.5: The Top Down Performance Analysis Hierarchy

in the *backend* (e.g. fully occupied ALUs or load-buffer entries); or the unavailability of decoded μops from the *frontend* (e.g. due to an instruction cache miss). The four top-level metrics are reported as proportions of all slots (4× CPU cycles elapsed).

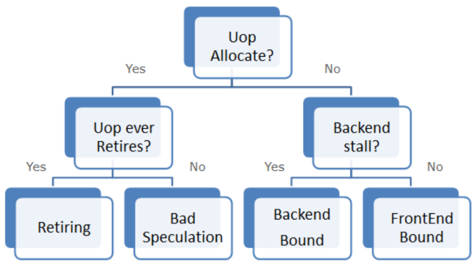


Figure 5.6: Top level metric definition

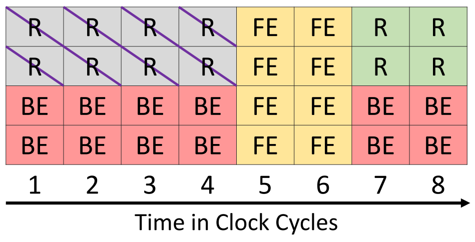
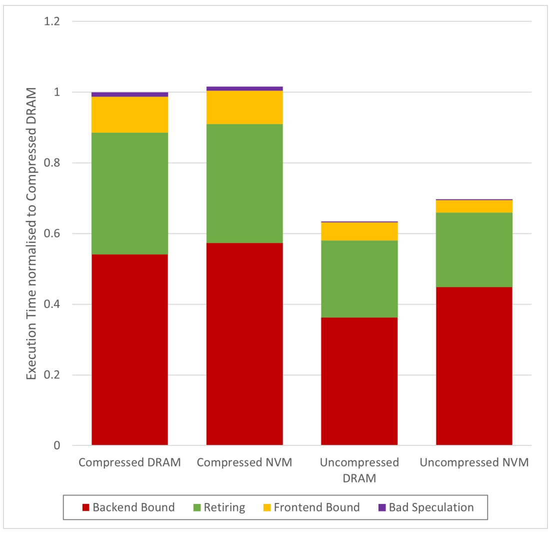
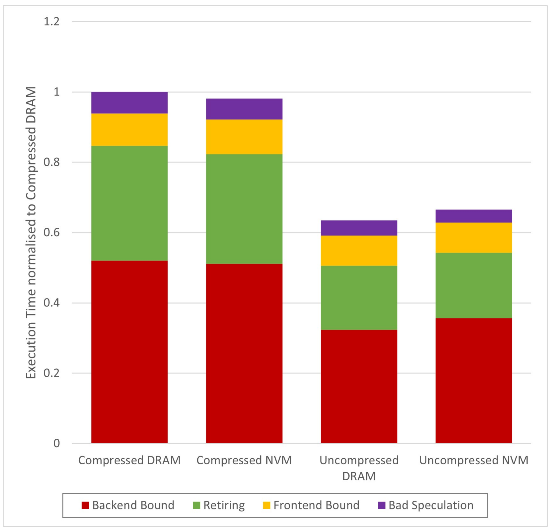


Figure 5.7: Example processor trace showing pipeline slots

Figure 5.7 is a pedagogical example of a single Intel processor's pipeline to illustrate the top level breakdown. There are four pipeline slots in every clock cycle, and each slot counts towards retiring (R), frontend bound (FE), or backend bound (BE). The core is shown to execute a memory-intensive workload, so only two slots are utilised every clock cycle between cycles 1 to 4 and 7 to 8. The remaining two slots per clock cycle are unused due to a lack of execution units available in the backend (BE). At clock cycle 5, the core finds out it has incorrectly speculated on a branch. Rollback causes execution stalls in cycles 5 and 6 as adequate instructions from the correct execution path are fetched and decoded by the frontend (FE). Since they were from the wrong code path, the core cancels the μops issued for execution between cycles 1 and 4. The cancelled μops are accounted in *bad speculation* and not retiring (R).



(a) Single Term Queries



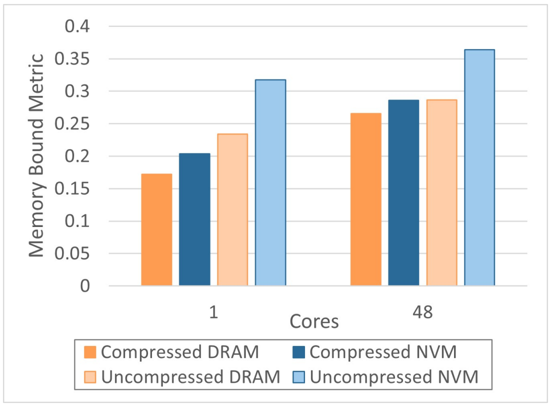
(b) 2-Term AND Queries

Figure 5.8: Top level breakdown of Top Down Hierarchy for 4 index setups

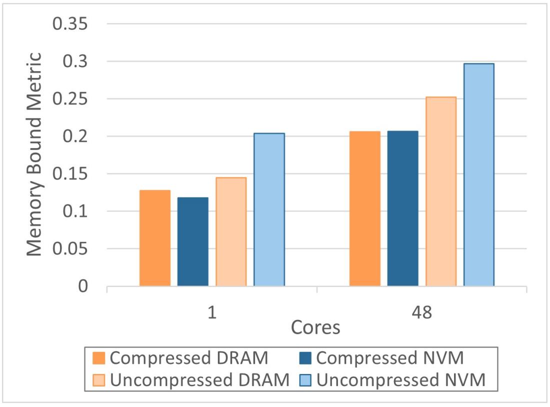
Yasin presents the definitions of the top-down metrics for Intel's Ivy Bridge microarchitecture. In this project, we found the updated performance counter events for Intel Cascade Lake CPUs by exploring available counters using Libpfm (Libpfm, 2008). We found later that Kleen (2022) keeps an up-to-date repository of the performance counter events needed for measuring top-down metrics on various microarchitectures. We present the definitions of each of the top-down metrics for the Cascade Lake processor in table 5.3.

We present the top-level metrics, scaled to the average execution time for the four index setups in figure 5.8. For both the query sets, we find that increases in the backend bound component of the top level breakdown explain the differences in performance for NVM and DRAM. We present the *memory bound* component of backend bound in figure 5.9. The memory bound metric measures the proportion of pipeline slots where the CPU is not issuing μops due to unavailable memory resources (e.g. fully occupied load buffer entries for the various levels of cache).

The uncompressed index is more memory bound than the compressed index. We explain this by noting that search on the uncompressed index requires reading a larger volume of memory from main memory into the CPU since the posting lists needed for search are much longer when uncompressed. For the uncompressed index, NVM has a significantly higher memory bound metric compared to DRAM, explaining the performance difference between NVM and DRAM for the uncompressed index. On the other hand, the memory bound metric differs minutely from DRAM and NVM for the compressed index. This observation is in line with the performance results in figures 5.2 and s 5.3; however, we must explore deeper to explain these results.



(a) Single Term Queries



(b) 2-Term AND Queries

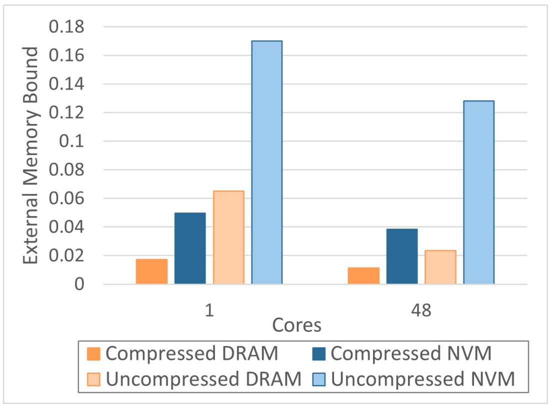
Figure 5.9: Memory Bound Metric for 4 index setups (lower is better)

5.3.2 Memory Bound Breakdown

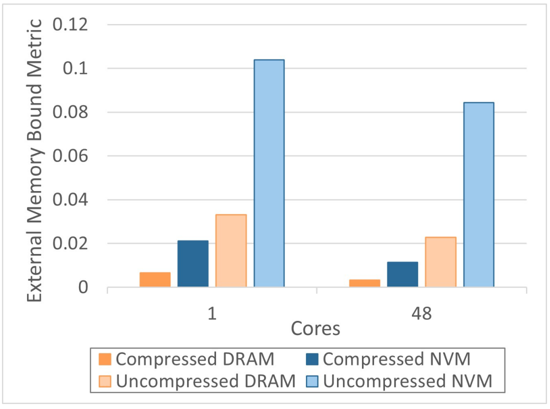
Having verified that the critical bottleneck for search on NVM is *memory bound*, we explore further down the hierarchy to the component of memory that we expect to affect execution the most: *external memory bound*. The external memory bound metric measures the proportion of pipeline slots where the CPU is not executing μops because it has stalled accessing main memory (DRAM or NVM). The external memory bound metric can measure how much the memory latency and bandwidth are true bottlenecks to performance. Figure 5.10 shows the external memory bound metric for all 4 index setups. The external memory bound metric is surprisingly low across all the index setups for both query types. In the worst case (single term queries on uncompressed NVM index), only 13% of CPU pipeline slots are stalls caused by a main memory access.

To explain these results we look at the *last level cache misses per 1000 instructions* (LLCMPKI). In figure 5.11, we find that for all setups, the LLCMPKI is very low at only one miss per kilo instruction. With a low cache miss rate, the CPU can hide the latency of these infrequent cache misses by executing other instructions out of order. The low LLCMPKI we observed explains why the external memory bound was low for all setups. Text search as a benchmark has cache-friendly memory access behaviour.

Cache-friendly behaviour could be a result of one of two phenomena. The first possibility is that the query sets' relevant posting lists fit in the last level cache. The second possibility is that the latency of memory accesses is being effectively hidden by the hardware cache-line prefetchers, reducing the average time spent on a cache miss by prefetching the necessary chunks of posting lists. Cache-line prefetchers could be reducing the performance impact of storing indices on slower NVM technology.



(a) Single Term Queries

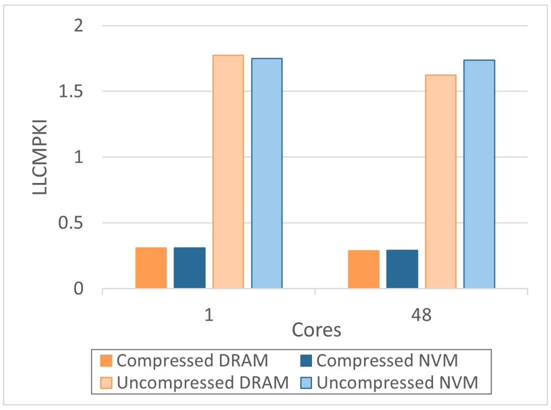


(b) 2-Term AND Queries

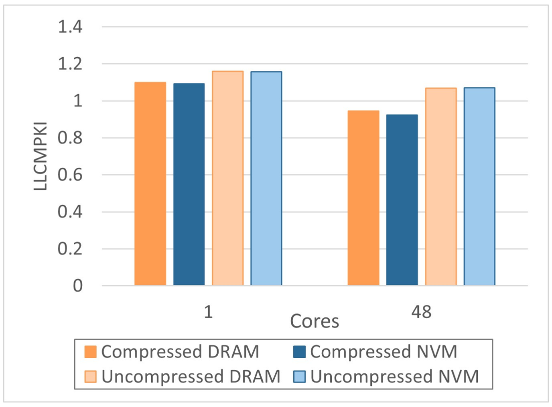
Figure 5.10: External Memory Bound Metric for 4 index setups (lower is better)

The first possibility is unlikely as the index sizes (5.4GB for compressed; 53GB for uncompressed) are far larger than the CPU cache size of 35.75MB. The second possibility is more likely than the first, based on results reported in previous work by Hadjilambrou et al. (2019). Hadjilambrou et al. characterised Lucene search from a microarchitectural perspective, and their results agreed with ours: LLCMPKI was low for Lucene search on the index created in the default compressed format. Hadjilambrou et al. also experimented searching on a processor with hardware prefetching disabled. They found a 30% increase in LLC miss rates when hardware prefetching was disabled. On the DRAM-backed compressed index Hadjilambrou et al. experimented on, the effect of prefetching on practical search performance was minimal. However, prefetching is crucial for the performance of an NVM-backed index since the penalty per LLC miss is much higher (as observed in the external memory bound metric shown in figure 5.10).

In chapter 6, we perform our investigation into which of these two phenomena is causing cache-friendly behaviour for search. We measure how each of the four setups scales in performance by increasing the index size. We find that the second phenomenon (effective prefetching) ensures that NVM performs comparably to DRAM for search.



(a) Single Term Queries



(b) 2-Term AND Queries

Figure 5.11: Last level cache misses per kilo-instruction (LLCMPKI) for 4 index setups (lower is better)

Metric	Explanation	Definition
Clocks	Number of clock cycles executed	`CPU_CLK_UNHALTED`
Slots	The number of pipeline slots that can be filled with μops. The Intel Cascade Lake processor used for all experiments can issue 4 μops per clock cycle per core.	4$\times$ Clocks
Retiring	Proportion of slots where a μop is issued and completes to retirement	`UOPS_RETIRED.RETIRE_SLOTS` / Slots
Bad Speculation	Proportion of slots where a μop is issued but is later cancelled due to a bad speculation	(`UOPS_ISSUED.ANY` - `UOPS_RETIRED.RETIRE_SLOTS` + 4$\times$ `INT_MISC.RECOVERY_CYCLES`) / Slots
Frontend Bound	Proportion of slots that are unutilised due to unavailablility of decoded μops from the frontend	`IDQ_UOPS_NOT_DELIVERED:CORE` / Slots
Backend Bound	Proportion of slots that are unutilised due to unavailability of required backend resources (e.g. data cache misses, overloaded divider unit)	1 - Retiring - Bad Speculation - Frontend Bound
Memory Bound	Proportion of cycles which are unutilised due to unavailablility of memory resources (full load-store buffers, cache misses for μop operands, etc.)	`CYCLE_ACTIVITY.STALLS_MEM_ANY` + `RESOURCE_STALLS.SB`) / Clocks
External Memory Bound	Proportion of cycles where the CPU is stalled while at least one demand load to main memory is outstanding	`CYCLE_ACTIVITY:STALLS_L3_MISS` / `CPU_CLK_UNHALTED`
LLCMPKI	The number of last level cache misses per 1000 instructions executed	`PERF_COUNT_HW_CACHE_MISSES` $\times$1000/ `PERF_COUNT_HW_INSTRUCTIONS`

Table 5.3: Defining the metrics used for the top-down approach to performance analysis (Yasin, 2014; Kleen, 2022)

5.4 Summary

In this chapter, we benchmark the performance of the uncompressed and compressed indices on non-volatile main memory. We developed a methodology and found that our new approach of placing an uncompressed index on NVM outperforms the state-of-the-art compressed index on DRAM by 30% for single term queries and 33% for 2-term AND queries. We found only a 10% degradation for the uncompressed index when using NVM in place of DRAM. Further, we found that NVM shows no performance difference from DRAM as a compressed index storage medium. All our findings are validated over multiple core-counts: multi-core search scales equally well on DRAM and NVM. Based on these findings we recommend two approaches for index storage on NVM:

- The *fast approach*: placing the uncompressed index in the NVM-backed heap. This approach provides faster average latency for single term queries and 2-term *AND* queries, reducing the cost of search engine infrastructure needed to service the same customer base. This approach also provides a faster 35% faster 99^{th} percentile search latency for 2-term *AND* queries, improving user experience.

- The *scalable approach*: placing the compressed index as a file on an NVM-backed filesystem. This approach allows search implementers to store larger indices entirely on main memory by exploiting NVM's massive capacity, while providing a performance is comparable to the performance of the current approach taken in industry.

The top-down methodology to performance analysis designed by Yasin that we followed in this chapter is superior to the ad-hoc bottom-up performance counter analysis that is common in systems research. Using the top-down methodology, we systematically verified our results and explained on a conceptual level why the 2–3× difference in microbenchmark performance between NVM and DRAM only translates to a 10% real performance difference for practical text-search systems. We conclude that search is a cache-friendly application. We observe this because text search interleaves memory-intensive posting list traversal with computationally intensive scoring and ranking, reducing cache miss rates and allowing the out-of-order processor to hide memory access latency.

Chapter 6

Scaling Text Search on Non-Volatile Memory

In chapter 5, we found that NVM is a scalable, performant alternative to DRAM for placement of search indices. Our approach of searching on an uncompressed index stored on NVM gave at least 30% search performance improvement over the state-of-the-art approach. In addition, we also found that searching the compressed index on NVM gave equivalent search performance to searching the compressed index on DRAM.

In this chapter, we validate these surprising findings on large search indices. We create indices of various sizes from a web-crawl data set and perform search on them. Our results show that for a 35GB compressed search index, by storing it in uncompressed form on NVM, we achieve performance improvements of 32% for single term and 38% for 2-term *AND* query over the state-of-the-art approach.

6.1 Methodology

We design the methodology to create a set of Lucene indices of different sizes to test search performance as index sizes scale. We obtained the January 2022 CommonCrawl data set (Nagel, 2022), a large open repository of web crawl data, and constructed search indices of various sizes using subsets of the data set.

6.1.1 Experimental Setup

We test the same four setups from chapter 5 on the hardware platform defined in section 4.1.2:

1. *Compressed DRAM*: We place the compressed index on an SSD-backed file system, and warm it up into the DRAM-backed OS page cache. This is the state-of-the-art

approach for search.

2. *Compressed NVM*: We place the compressed index on an NVM-backed EXT4 direct access (DAX) file system. This is our novel *scalable* approach.
3. *Uncompressed DRAM*: We place the uncompressed index to the DRAM-backed Java heap.
4. *Uncompressed NVM*: We place the uncompressed index to the NVM-backed Java heap. This is our novel *fast* approach.

In this chapter, we measure performance over different index sizes by creating two sets of indices, uncompressed and compressed, over six index sizes ranging from $\approx$ 760MB to $\approx$ 87GB. The code Lucene provides for decompressing an index to the program heap requires the index to have been created using a specific format called *DirectPostingsFormat*. The *DirectPostingsFormat* is identical to Lucene's default index format called *Lucene84PostingsFormat* when the index is stored compressed on the file system. However, we must create one index in each format for each index size we test.

Since Lucene does not offer a reliable way to measure index sizes while indexing is occurring, we constructed the indices by adding documents until the index size on the file system reaches a limit. We then stop indexing new documents at this point and run a final segment merge. As a result of this methodology, there are some slight differences in index size and number of segments of the compressed and uncompressed indices. A description of all the indices we created is provided in Appendix A.

As the uncompressed indices are placed on heap, we size the heaps to use the entire available memory for the uncompressed format tests. We summarise the heap sizing parameters we used for this experiment in table 6.1.

	Compressed		Uncompressed	
	DRAM	NVM	DRAM	NVM
Initial Index Size	2GB	2GB	2GB	2GB
Maximum Index Size	32GB	32GB	180GB	585GB

Table 6.1: Uncompressed vs compressed search index experiment: Heap size settings

We use all 48 logical cores for multi-threaded search using the single term and 2-term *AND* query sets we defined in section 4.1.4. All experiments are run on the hardware platform defined in section 4.1.2.

6.2 Results

We graph the search performance (i.e. average query latency) of our different setups of the single term query set in figure 6.1 and the *AND* query set in figure 6.3. We note that the missing results for large indices using the uncompressed setups are caused by the index not fitting in the maximum heap space allocated.

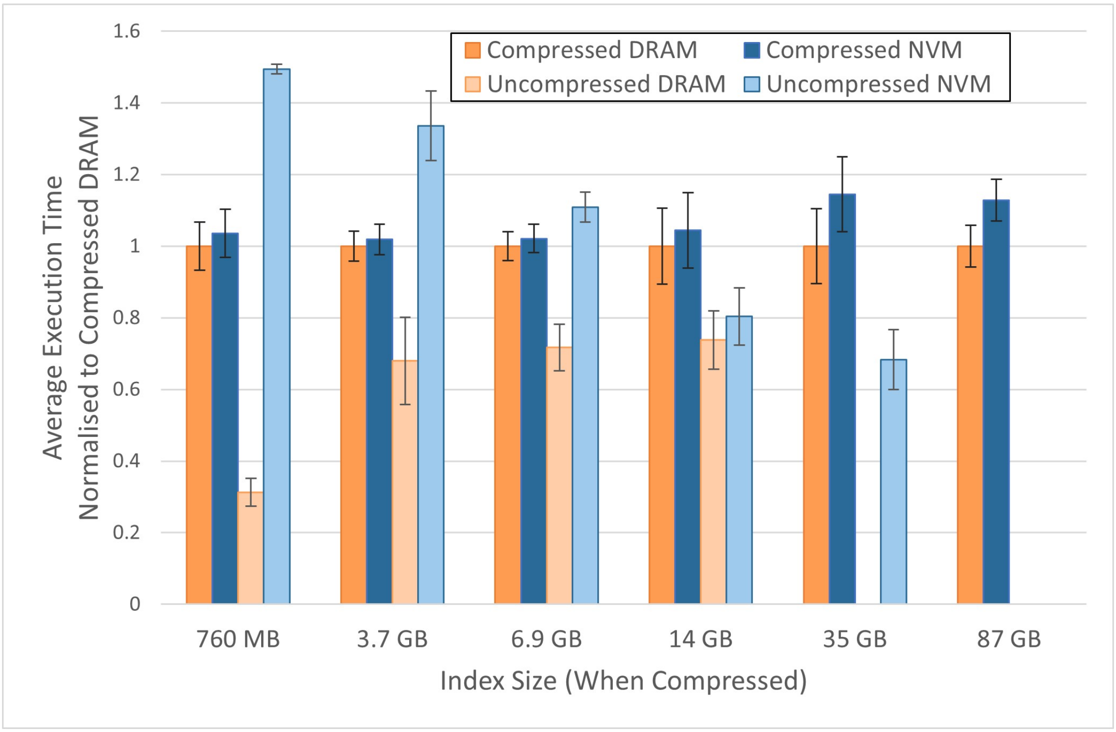
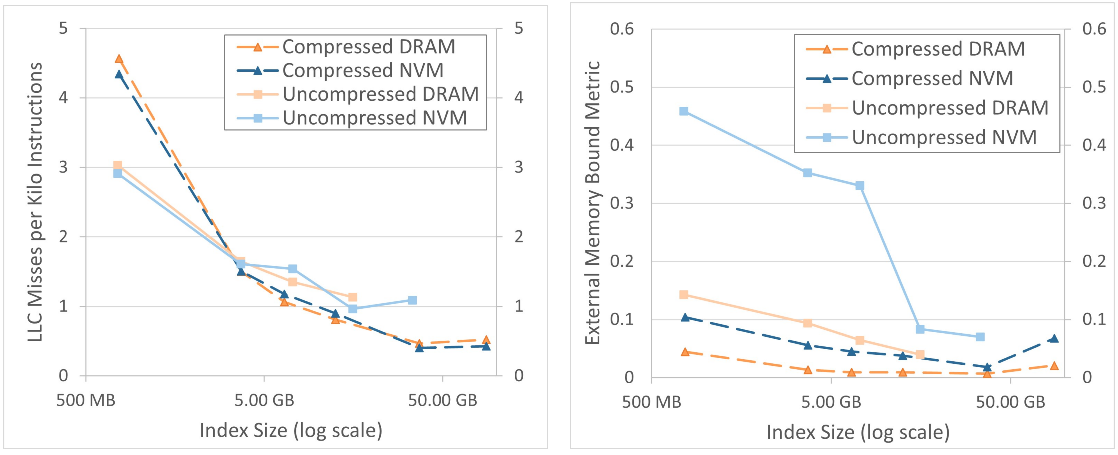


Figure 6.1: Single term query performance on indices of varying sizes (lower is better)

We focus first on the single-term query set (figure 6.1). For the largest index size of 87GB, *Compressed NVM* is 13% slower than *Compressed DRAM*. Nevertheless, for this small performance hit compared to the state of the art, using NVM allows the programmer to store much larger indices entirely on main memory. Our approach is much more scalable to large indices. To show this, we measured the performance of the compressed index on a warm-up run (when it is being read from SSD into the filesystem cache), and find a 16% performance improvement using NVM. For extremely large indices that do not fit on the available DRAM, the cost of reading the parts of the index needed for a query from the SSD must be factored into the performance of the state-of-the-art approach.

We observe that the *Uncompressed NVM* performs poorly for small index sizes but outperforms the state-of-the-art as we increase the index size. For the largest index we can fit in uncompressed format on NVM (35GB), we observe 32% improvement over *Compressed DRAM* with *Uncompressed NVM* our *fast* approach.

In figure 6.2a, we present the rate of last level cache misses per 1000 instructions (LLCMPKI) for single term queries. LLCMPKI of single term search decreases as the index size increases. Single term search on both the compressed and uncompressed indices involves sequentially accessing every posting in a posting list to score and rank all the documents containing that term. Larger indices contain longer posting lists. When longer posting lists are sequentially accessed, we expect the hardware cache-line prefetcher to



(a) Last level cache misses per 1000 instructions (LLCMPKI; lower is better)

(b) External Memory Bound Metric (lower is better)

Figure 6.2: Performance analysis of single term queries over indices of varying size

predict which cache lines are needed for search with greater accuracy. Figure 6.2b shows the external memory bound metric for the single term queries which and represents the proportion of execution time the core is stalled waiting for a main memory access to complete. Since cache misses occur less frequently as the index size increases, we observe a reduction in the external memory bound metric. In particular, the prefetchers reduce the external memory bound metric of search on *Uncompressed NVM* from 45% to 8% as we increase the index size from 760MB to 35GB. Therefore, our performance analysis shows that because of prefetching, single term search scales well on large, NVM-backed uncompressed indices.

We turn our focus to the 2-term *AND* query set (figure 6.3). For all the index sizes, there is no difference in performance between the *Compressed NVM* and the *Compressed DRAM*. The performance of the *Uncompressed NVM* is similar to the performance of *Uncompressed DRAM* for indices larger than 760MB. *Uncompressed NVM* consistently outperforms the state-of-the-art *Compressed DRAM*.

In figure 6.4a, we observe that the LLCMPKI of 2-term *AND* query search stays constant as the index size increases for the uncompressed indices. However, the LLCMPKI decreases as index size increases for the compressed indices. This difference we observe between search on the compressed indices and uncompressed indices is because both setups necessarily use different algorithms fo *AND* query evaluation. To compute *AND* queries on either index format, the postings must be repeatedly searched for candidate document IDs (refer back to figure 2.6 for an example). Searching for candidate document IDs on the uncompressed index requires a binary search on the posting list, an algorithm which has a random memory access pattern. As the index size increases, search on the uncompressed index performs the binary search over increasingly longer

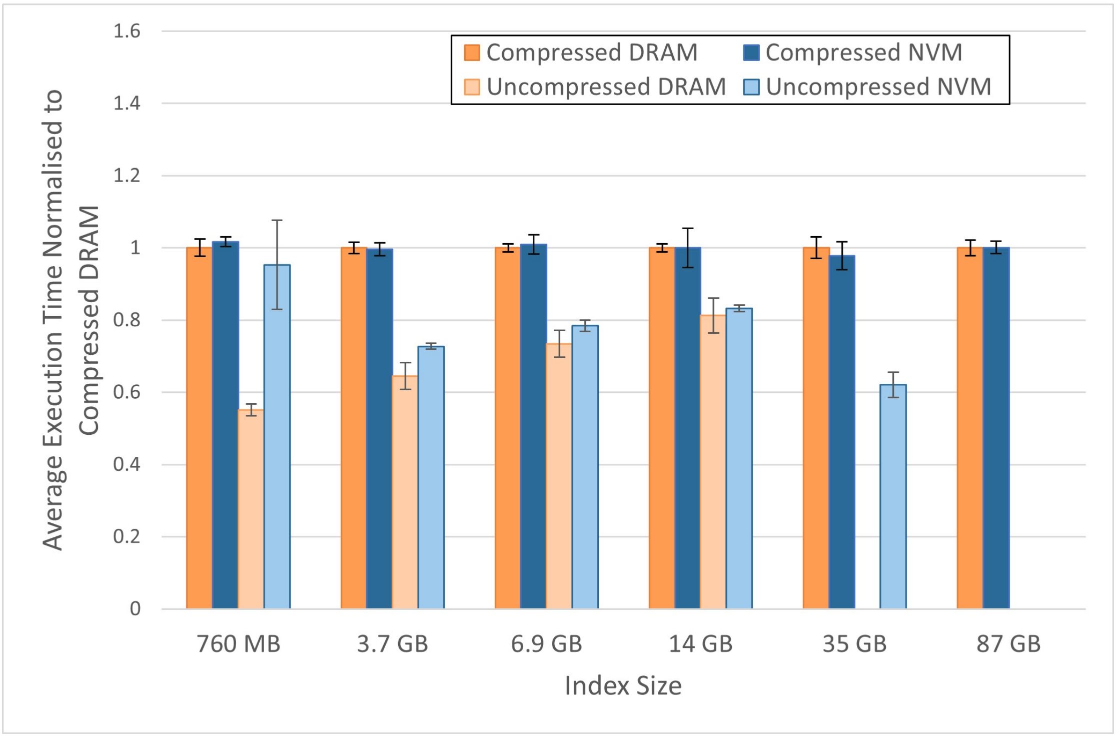
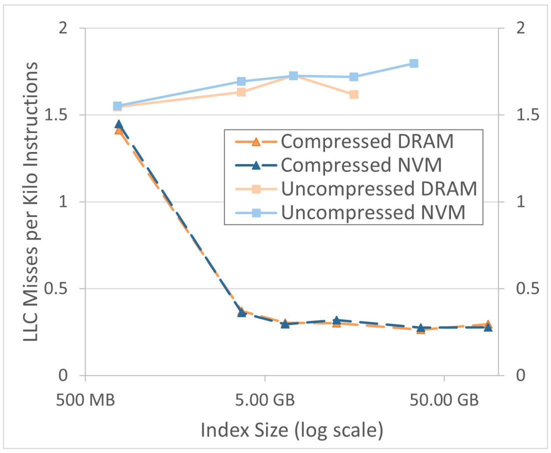


Figure 6.3: 2-term *AND* query performance on indices of varying sizes (lower is better)

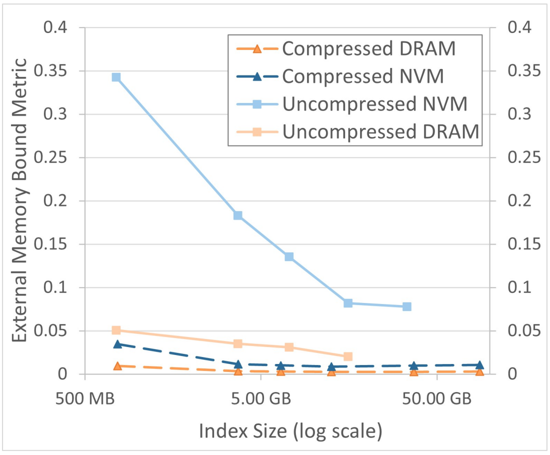
posting lists, which is a cache-unfriendly operation.

However, figure 6.4b paints a contradictory picture: the external memory bound metric goes down significantly, particularly for the NVM-backed uncompressed index. While the last-level cache miss rate stays the same, the proportion of clock cycles stalled on a last-level cache miss decreases as we increase the index sizes. This is an interesting observation that we want to examine in future work. More specifically, we want to answer the question: using the same memory hardware (NVM) and the same algorithm (posting list intersection on uncompressed indices), when the index size changes, why do we see a constant rate of last-level cache misses (LLCMPKI) but see a reduction in the proportion of clock cycles the core is stalled on a last-level cache miss? If the cache-line prefetchers are improving performance, why do we not see a matching reduction in the LLCMPKI as the index size grows? Nevertheless, the bottomline is that *Uncompressed NVM* outperforms the state-of-the-art for 2-term *AND* query execution on large indices.

Focusing on the compressed index setups in figure 6.4a, we observe that the LLCMPKI of 2-term *AND* query search decreases as the index size increases. This observation can be explained by understanding the *AND* query evaluation algorithm on compressed indices. The compressed index format stores the posting list in blocks and uses a multi-level skip list to find which block contains a posting matching a candidate document ID. The multi-level skip list was designed specifically to avoid random memory accesses



(a) Last level cache misses per 1000 instructions (LLCMPKI; lower is better)



(b) External Memory Bound Metric (lower is better)

Figure 6.4: Performance analysis of 2-term *AND* queries over indices of varying size

	Compressed DRAM (SoA)	Compressed NVM (scalable)	Uncompressed NVM (fast)
Single Term	65 (±8)	76 (±7)	42 (±6)
2-Term *AND*	904 (±38)	873 (±53)	548 (±37)

Table 6.2: 99th percentile latency (+/- confidence interval) for the state-of-the-art, *scalable* and *fast* approaches

inherent to the binary search algorithm. Using a multi-level skip list shows better cache locality than binary search by ensuring only one access to the posting list is needed to find a candidate ID. The high levels of the multi-level posting list are likely to stay in the CPU cache as they are frequently accessed during posting list traversal.

In table 6.2, we explore the 99th percentile latency of search queries over the three setups: the state-of-the-art DRAM-backed compressed index, the *scalable* NVM-backed compressed index, and the *fast* NVM-backed uncompressed index. We present data for the largest index (35GB) on which we tested all three setups. For 2-term *AND* queries, we see a 325ms (37%) improvement in latency of the 99th percentile slowest queries on our *fast* approach. The *scalable* approach provides a comparable 99th percentile latency to the state-of-the-art.

6.3 Summary

In this chapter, we verified our surprising findings in chapter 5 by replicating our experiments on search indices of varying size. We found that our proposed *fast approach* of placing the search index in uncompressed format on the NVM-backed heap scales well in performance as index size increases. For the largest index (35GB) we tested on the *fast approach*, we report a 32% improvement for single term and 38% improvement for 2-term *AND* queries over the state-of-the-art DRAM-backed compressed index. This performance improvement enables the industry to downsize the search engine infrastructure needed to service the same customer base. Our *fast approach* also improves 99*th* percentile latency of search by 325ms (37%) over the state-of-the-art, improving the user experience significantly. The *scalable approach* of placing the search index in compressed format on the NVM file system shows minimal performance difference to the state-of-the-art across all index sizes and query sets. Our *scalable approach* has similar 99*th* percentile latency to the state-of-the-art.

Chapter 7

Concluding Remarks

7.1 Conclusion

In this thesis, we used the new non-volatile memory (NVM) to optimise a state-of-the-art search engine library. NVM is a high capacity, byte addressable memory technology that can be used as main memory, however, it is 2×–3× slower than Dynamic Random Access Memory (DRAM). Our experimentation found that searching on an uncompressed index stored on NVM provides a 30% performance improvement over the state-of-the-art. We also found that searching on a compressed index stored on NVM had comparable performance to the state-of-the-art. We explain these surprising findings with a thorough performance analysis using hardware performance counters. We find that search algorithms are cache-friendly, meaning the longer latency of memory accesses on NVM only slightly affects the practical performance of a search engine. We validate our findings across different search index sizes and over multiple cores. We find that both our approaches scale well to large index sizes, and for multi-threaded search. Using our observations, we present the following approaches to implementing search indices on NVM:

1. the *fast* approach: placing the uncompressed index on the heap, where the heap is memory-mapped to NVM. This approach makes search queries 30% faster compared to the state-of-the-art. The *fast* approach has two major benefits:

    a) it provides a reduction in average query latency that allows each node in a search server farm to resolve more queries per second, reducing search infrastructure costs

    b) it provides a reduction in 99^{th} percentile latency that improves user experience drastically for long running queries.

2. the *scalable* approach: placing the compressed index on a file in an NVM-backed

file system. This approach provides search performance that is comparable to the state-of-the-art. The *scalable* approach makes best use of NVM's large capacity to store colossal indices entirely on NVM, where the processor can access these indices with no IO costs. This approach enables search over evergrowing datasets which are infeasibly expensive to place on DRAM.

7.2 Future Work

We provide avenues for future work that arose from our background readings and experiments.

7.2.1 Exploring the Effect of Non-Uniform Memory Access

Microbenchmark results by Yang et al. (2020) show that NVM's memory access performance is poor for a system with non-uniform memory accesses (NUMA). Large search indices will necessarily have to be placed on computers with multiple processor sockets that exhibit NUMA. We would like extend our work in this thesis to investigate how search algorithms are effected by NUMA on the multi-socket systems commonly employed in search infrastructure.

7.2.2 Designing New Compression Schemes for Non-Volatile Memory

An interesting challenge would be to design compression algorithms for search indices that make use of NVM's unique memory tradeoffs. On the search side, decompression should be optimised for the read performance of NVM. The compression algorithm also affects the memory access patterns of indexing process. Search index storage formats must be designed to avoid NVM's unique performance pathologies. The indexer must make best use of NVM's low write latency, without causing write amplification or DIMM contention for writes. We would like to explore these tradeoffs by analysing the performance of compression schemes employed by search engines in greater depth on NVM.

Appendix A

Appendix: Description of Search Indices

We describe the search indices we used for all our experiment in chapter 6. We report the characteristics of the compressed indices in table A.1. We report the characteristics of the uncompressed indices in table A.2. Since the uncompressed index is also stored on the file system in compressed form (before being decompressed into the heap), we report the size of the index in compressed form on the file system, along with the size of the uncompressed index on the heap. The largest index was too large to fit on the heap in uncompressed form since we had constrained ourselves to only using half the available memory on the system (to discard any NUMA effects on performance results).

Index Size on Disk	Number of Documents Indexed	Total Size of Web Documents Indexed	Number of Index Segments
766 MB	255 K	1.86 GB	33
3.71 GB	1.34 M	9.80 GB	17
6.49 GB	2.36 M	17.28 GB	16
12.54 GB	4.63 M	33.78 GB	26
36.94 GB	13.86 M	101.11 GB	23
87.60 GB	33.12 M	241.28 GB	35

Table A.1: Describing the compressed indices tested in the scalability study

Index Size on Disk (Compressed)	Number of Documents Indexed	Total Size of Web Documents Indexed	Number of Index Segments	Decompressed Index Size (on heap)	Memory Cost of Decompression
753 MB	249 K	1.82 GB	30	5.60 GB	7.44 ×
3.69 GB	1.31 M	9.55 GB	36	28.67 GB	7.76 ×
7.22 GB	2.61 M	19.05 GB	36	56.49 GB	7.82 ×
15.63 GB	5.82 M	42.43 GB	27	123.62 GB	7.91 ×
33.77 GB	12.67 M	92.42 GB	30	268.56 GB	7.95 ×
87.02 GB	32.89 M	239.58 GB	29		

Table A.2: Describing the uncompressed indices tested in the scalability study

Bibliography

Akram, S., 2021. *Exploiting Intel Optane Persistent Memory for Full Text Search*, 80–93. Association for Computing Machinery, New York, NY, USA. ISBN 9781450384483. https://doi.org/10.1145/3459898.3463906. [Cited on pages 21 and 30.]

Aksyonoff, A., 2022. Sphinx: Open source search server. http://sphinxsearch.com/. [Cited on page 10.]

Apache, 2022. Apache lucene. https://lucene.apache.org/. [Cited on page 5.]

Benson, L.; Makait, H.; and Rabl, T., 2021. Viper: An efficient hybrid pmem-dram key-value store. In *VLDB*, XXX–XXX. ACM. doi:10.14778/3461535.3461543. https://doi.org/10.14778/3461535.3461543. [Cited on pages 17, 21, and 22.]

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khang, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06 (Portland, Oregon, USA, 2006), 169–190. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1167473.1167488. https://doi.org/10.1145/1167473.1167488. [Cited on page 26.]

Blackburn, S. M.; McKinley, K. S.; Garner, R.; Hoffmann, C.; Khan, A. M.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanovik, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2008. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51, 8 (aug 2008), 83–89. doi:10.1145/1378704.1378723. https://doi.org/10.1145/1378704.1378723. [Cited on pages 30 and 31.]

Boyd-Wickizer, S.; Clements, A. T.; Mao, Y.; Pesterev, A.; Kaashoek, M. F.; Morris, R.; and Zeldovich, N., 2010. Mosbench. https://pdos.csail.mit.edu/archive/mosbench/. [Cited on page 21.]

Coburn, J.; Caulfield, A. M.; Akel, A.; Grupp, L. M.; Gupta, R. K.; Jhala, R.; and Swanson, S., 2011. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI (Newport Beach, California, USA, 2011), 105–118. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1950365.1950380. https://doi.org/10.1145/1950365.1950380. [Cited on page 22.]

Goldstein, J.; Ramakrishnan, R.; and Shaft, U., 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, 370–379. doi:10.1109/ICDE.1998.655800. [Cited on page 9.]

Grand, A.; Muir, R.; Ferenczi, J.; and Lin, J., 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*, vol. 12036 of *Lecture Notes in Computer Science*, 20–27. Springer. doi:10.1007/978-3-030-45442-5_3. https://doi.org/10.1007/978-3-030-45442-5_3. [Cited on page 33.]

Hadjilambrou, Z.; Kleanthous, M.; Antoniou, G.; Portero, A.; and Sazeides, Y., 2019. Comprehensive characterization of an open source document search engine. *ACM Trans. Archit. Code Optim.*, 16, 2 (may 2019). doi:10.1145/3320346. https://doi.org/10.1145/3320346. [Cited on page 49.]

Heo, J.; Won, J.; Lee, Y.; Bharuka, S.; Jang, J.; Ham, T. J.; and Lee, J. W., 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*, 1233–1245. ACM. doi:10.1145/3373376.3378521. https://doi.org/10.1145/3373376.3378521. [Cited on page 13.]

IBM, 2014. IBM eXflash storage DIMM User Guide. https://download.lenovo.com/servers_pdf/ibm_doc_exflash_1.5.2_user_guide.pdf. [Cited on page 19.]

Intel, 2015. Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/. [Cited on page 18.]

Janapa Reddi, V.; Lee, B. C.; Chilimbi, T.; and Vaid, K., 2010. Web search using mobile cores: Quantifying and mitigating the price of efficiency. *SIGARCH Comput. Archit. News*, 38, 3 (jun 2010), 314–325. doi:10.1145/1816038.1816002. https://doi.org/10.1145/1816038.1816002. [Cited on page 33.]

Kharbas, K., 2016. Jep 316: Heap allocation on alternative memory devices. https://bugs.openjdk.java.net/browse/JDK-8171181. [Cited on page 39.]

Kharbas, K., 2018. Allocation of old generation of java heap on alternate memory devices. https://bugs.openjdk.java.net/browse/JDK-8202286. [Cited on page 39.]

Kleen, A., 2022. Pmu-tools. https://github.com/andikleen/pmu-tools. [Cited on pages 47 and 51.]

Kolokasis, I. G.; Evdorou, G.; Papagiannis, A.; Zakkak, F. S.; Kozanitis, C.; Akram, S.; Pratikakis, P.; and Bilas, A., 2021. Freeing compute caches from serialization and garbage collection in managed big data analytics. *CoRR*, abs/2111.10589 (2021). https://arxiv.org/abs/2111.10589. [Cited on pages 39 and 40.]

Kwon, Y.; Fingler, H.; Hunt, T.; Peter, S.; Witchel, E.; and Anderson, T., 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17 (Shanghai, China, 2017), 460–477. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3132747.3132770. https://doi.org/10.1145/3132747.3132770. [Cited on page 22.]

Lemire, D. and Boytsov, L., 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45, 1 (2015), 1–29. doi:https://doi.org/10.1002/spe.2203. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2203. [Cited on page 11.]

Libpfm, 2008. Libpfm4: a helper library for performance tools using hardware counters. http://perfmon2.sourceforge.net/. [Cited on pages 34 and 47.]

Lim, K. T.; Chang, J.; Mudge, T. N.; Ranganathan, P.; Reinhardt, S. K.; and Wenisch, T. F., 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, 267–278. ACM. doi:10.1145/1555754.1555789. https://doi.org/10.1145/1555754.1555789. [Cited on page 17.]

Linden, G., 2006. Marissa mayer at web 2.0. https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. [Cited on page 1.]

Lucene, 2021. Class BM25Similarity Java Documentation. https://lucene.apache.org/core/8_9_0/core/org/apache/lucene/search/similarities/BM25Similarity.html. [Cited on page 13.]

McCandless, M., 2019a. Concurrent query execution in apache lucene. https://blog.mikemccandless.com/2019/10/concurrent-query-execution-in-apache.html. [Cited on page 16.]

McCandless, M., 2019b. Using finite state transducers in lucene. https://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html. [Cited on page 12.]

McCandless, M.; Hatcher, E.; and Gospodnetic, O., 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., USA. ISBN 1933988177. [Cited on page 16.]

Moraru, I.; Andersen, D. G.; Kaminsky, M.; Tolia, N.; Ranganathan, P.; and Binkert, N., 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13 (Farmington, Pennsylvania, 2013). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2524211.2524216. https://doi.org/10.1145/2524211.2524216. [Cited on page 22.]

Nagel, S., 2022. January 2022 crawl archive now available. https://commoncrawl.org/2022/06/may-2022-crawl-archive-now-available/. [Cited on page 55.]

Pibiri, G. E. and Venturini, R., 2020. Techniques for inverted index compression. *ACM Comput. Surv.*, 53, 6 (dec 2020). doi:10.1145/3415148. https://doi.org/10.1145/3415148. [Cited on pages 8 and 17.]

Robertson, S. E.; Walker, S.; Jones, S.; Hancock-Beaulieu, M.; and Gatford, M., 1994. Okapi at TREC-3. In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994*, vol. 500-225 of *NIST Special Publication*, 109–126. National Institute of Standards and Technology (NIST). http://trec.nist.gov/pubs/trec3/papers/city.ps.gz. [Cited on page 13.]

Sankar, S., 2015. Did you mean galene. https://engineering.linkedin.com/search/did-you-mean-galene. [Cited on page 5.]

Sartore, R. H., 2011. Hybrid nonvolatile ram. https://www.freepatentsonline.com/8074034.html. [Cited on page 19.]

Schatzl, T., 2020. Remove allocation of old generation on alternate memory devices functionality. https://bugs.openjdk.java.net/browse/JDK-8256181. [Cited on page 39.]

Singer, M. and Wilcox, A., 2021. Using persistent memory to accelerate tweet search. https://www.techtarget.com/searchstorage/post/Using-Persistent-Memory-to-Accelerate-Tweet-Search. [Cited on page 7.]

Stribling, J.; Li, J.; Councill, I. G.; Kaashoek, M. F.; and Morris, R. T., 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. USENIX. http://www.usenix.org/events/nsdi06/tech/stribling.html. [Cited on page 21.]

Tonozzi, N. and Daniliuc, D., 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. [Cited on page 5.]

Wang, J.; Lin, C.; He, R.; Chae, M.; Papakonstantinou, Y.; and Swanson, S., 2017. MILC: inverted list compression in memory. *Proc. VLDB Endow.*, 10, 8 (2017), 853–864. doi:10.14778/3090163.3090164. http://www.vldb.org/pvldb/vol10/p853-wang.pdf. [Cited on page 11.]

Wilcox, M., 2014. Add support for nv-dimms to ext4. https://lwn.net/Articles/613384/. [Cited on page 19.]

Woodward, A., 2019. What's new in lucene 8. https://www.elastic.co/blog/whats-new-in-lucene-8. [Cited on page 33.]

Xu, J.; Kim, J.; Memaripour, A.; and Swanson, S., 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*, ASPLOS '19 (Providence, RI, USA, 2019), 427–439. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3297858.3304077. https://doi.org/10.1145/3297858.3304077. [Cited on page 22.]

Yang, J.; Kim, J.; Hoseinzadeh, M.; Izraelevitz, J.; and Swanson, S., 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang. [Cited on pages 2, 17, 18, 22, 23, 44, and 64.]

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 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*, 309–322. USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang. [Cited on pages 30 and 34.]

Yasin, A., 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. doi:10.1109/ISPASS.2014.6844459. [Cited on pages 45, 47, 51, and 52.]

Zhang, W.; Zhao, X.; Jiang, S.; and Jiang, H., 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*, 194–209.

ACM. doi:10.1145/3447786.3456237. https://doi.org/10.1145/3447786.3456237. [Cited on pages 17, 21, and 22.]

Zobel, J. and Moffat, A., 2006. Inverted files for text search engines. *ACM Comput. Surv.*, 38, 2 (jul 2006), 6–es. doi:10.1145/1132956.1132959. https://doi.org/10.1145/1132956.1132959. [Cited on pages 2, 7, 8, and 12.]

Zukowski, M.; Heman, S.; Nes, N.; and Boncz, P., 2006. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, 59–59. doi:10.1109/ICDE.2006.150. [Cited on page 10.]