

Exploiting Off-Heap Persistent Memory for Java Programs

A report submitted for
SCNC3101 Advanced Studies Course

Zack Noyes

October 30, 2023

Abstract

As the amount of data used by large-scale applications increases, data storage and access mechanisms have largely stayed the same. This has created a need for traditionally DRAM-based applications to find new storage solutions.

Solid State Drives (SSDs) present one potential solution to larger memory demands, but have inherent issues when used in contexts where DRAM is usually used; they are block-addressable rather than byte-addressable and their read/write times are orders of magnitude slower than DRAM.

Emerging persistent memory devices present a potential solution to this. Like SSDs, they can have large capacity and are non-volatile, but they are byte-addressable and have similar performance characteristics to DRAM.

This project is a case study of integrating Intel's Optane persistent memory into Apache Lucene, a real-time search library used by many prominent applications. Lucene's combination of large memory demands and reliance on fast random-access data structures make it an appropriate case study.

Contents

Abstract	1
Overview	3
Related work	4
Characteristics of LLPL	5
Overview of LLPL functionality	5
Methodology	5
Timing behaviour	6
Heap creation	6
Memory allocation	7
Writing to a memory block	8
Reading from a memory block	9
Building a persistent memory index for Lucene	10
The interface of a Lucene Directory	10
Implementing a Directory using persistent memory	11
Opportunities for improvement in PMemDirectory	12
Performance analysis	13
Comparison to direct postings format	13
Comparison to FSDirectory	15
Conclusion	17

Overview

Lucene ([Apache, 2023](#)) is a Java library used for building real-time search applications. At the core of Lucene’s functionality is an index. The storage of this index is implemented by an instance of a `Directory` object.

Currently, the most common storage method is the `FSDirectory` subclass, which stores the index in a filesystem directory as multiple files. Figure 1 shows an overview of the structure of a Lucene index. In an `FSDirectory`, each of these boxes is stored as a separate file.

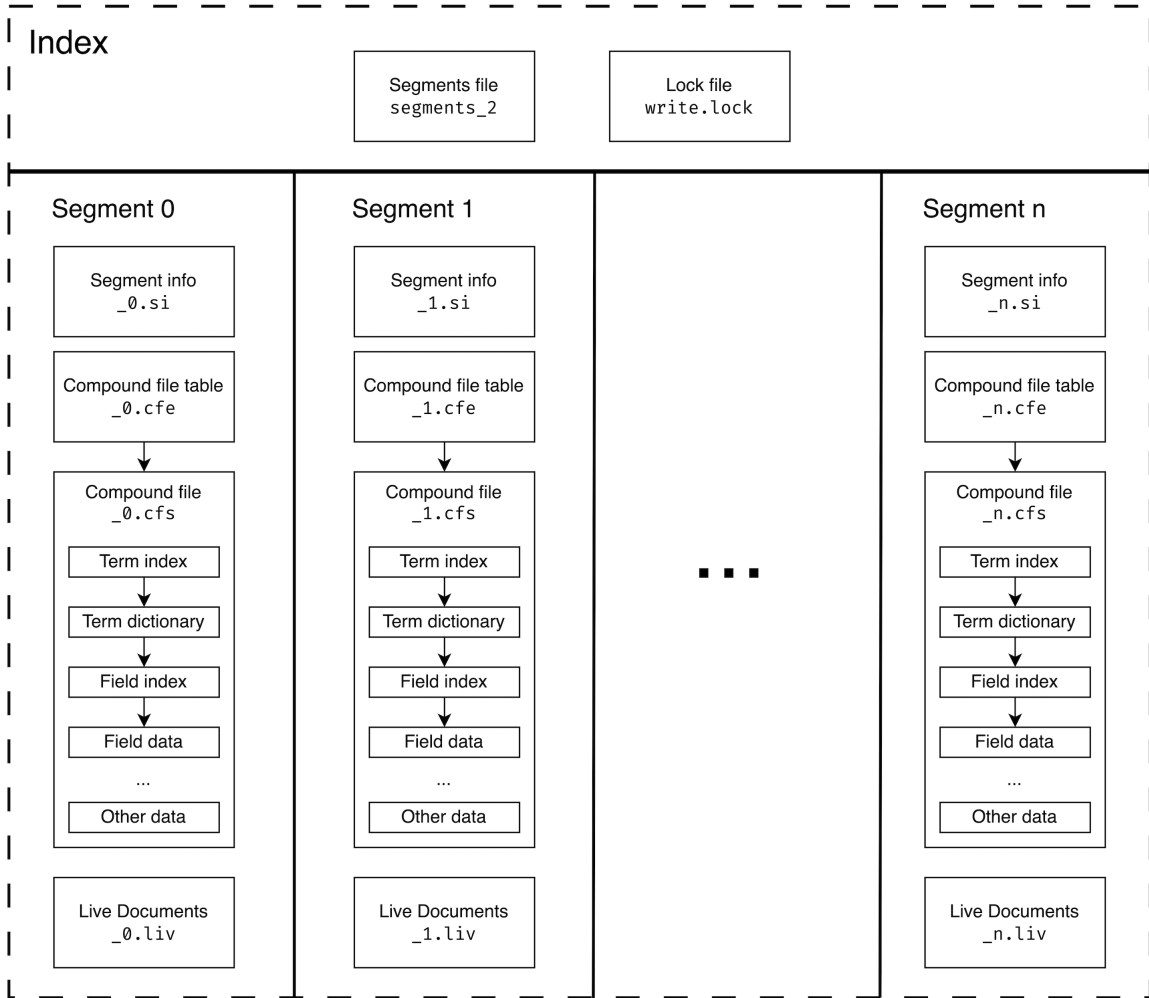


Figure 1: Structure of a Lucene index

In this project, we implement a new `PMemDirectory` subclass of `Directory`. The structure of Lucene’s index remains the same as in Figure 1, but all of these files are now stored in a single persistent memory `Heap`. Access and modification of this heap is managed with the Low-Level Persistence Library (LLPL) ([Intel, 2022](#)).

This report will first discuss LLPL and its performance characteristics before describing how we implemented the `PMemDirectory` subclass. We will finish with an analysis of the performance of our implementation. Instructions and scripts for running this work are included in the git repository of this project.

Related work

There have been some prior studies on the characteristics of persistent memory (and Optane in particular), as well as some research on how these relate to Lucene.

The initial model used for building persistent memory applications was to treat Optane like slightly slower DRAM. In fact, as mentioned by [Pydipaty and Saha \(2018\)](#), standard practice in evaluating persistent memory when it was not available was to simply use DRAM.

In their paper, [Yang et al. \(2020\)](#) found that Optane’s performance characteristics are more complicated than this. Only some of this research is relevant to our project, since we are accessing persistent memory only through LLPL and we are also not dealing with concurrency. The two most relevant findings from this paper are:

- Random accesses smaller than 256 bytes incur a performance penalty in terms of bandwidth.
- The interleaving of Optane DIMMs results in performance degradation if access sizes fit particular patterns that cause contention at the integrated memory controller (e.g. access sizes close to 4 KB, the Optane interleaving size).

[Kolokasis et al. \(2023\)](#) have conducted extensive work into developing a system that enables the Java Virtual Machine (JVM) to take advantage of larger capacity (but potentially slower) memory devices and expand the managed Java heap. While this project doesn’t make direct use of this work, our implementation uses a similar idea: we use a second, larger capacity, persistent memory heap to store a Lucene index.

In an effort to apply emerging storage devices to Lucene, [Pydipaty and Saha \(2018\)](#) conducted an experiment with DRAM set up to emulate persistent memory. In this experiment, they stored a Lucene index on a filesystem backed by emulated persistent memory, without modifying the internal structure of Lucene. They compared this to the traditional method of using an SSD-backed filesystem and found that a few specific workloads saw an improvement. However, their conclusion was that future work could see larger improvements if parts of Lucene were redesigned — this is what we attempt in this project.

[Chilukuri and Akram \(2023\)](#) have also done work on using persistent memory to improve Lucene. They compared multiple methods of storing indexes in both DRAM and persistent memory, as well as considering the performance improvements that could be gained from using uncompressed indexes (taking advantage of the larger capacity of persistent memory), rather than the compressed indexes traditionally used. They found that uncompressed indexes stored in persistent memory were faster than compressed indexes stored in DRAM, and concluded that a hybrid system using a small DRAM nursery and large uncompressed persistent memory index in the managed heap would offer query times similar to a DRAM-backed uncompressed index, but be much more feasible due to larger potential memory capacity. We use this “direct postings format” as a comparison point for the query evaluation performance of our implementation (see [comparison to direct postings format](#)).

Characteristics of LLPL

As outlined in [related work](#), there is already some research into the performance characteristics of Optane. In particular, it is clear that the “slow DRAM” model is not quite accurate enough for many purposes. Here, we discuss how this manifests through LLPL.

Overview of LLPL functionality

In LLPL, Heaps can be created which present an API that is, at a high level, similar to that provided by `malloc` and its related functions in the C programming language. Figure 2 gives an example of an LLPL Heap.

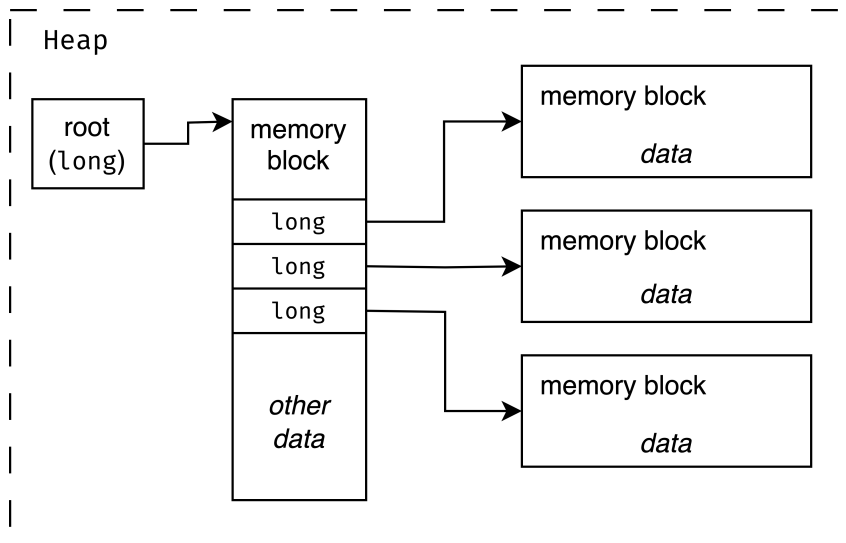


Figure 2: Structure of a Lucene index

After allocating a **Heap** which can be either fixed-size or dynamic, we can request **MemoryBlocks** of a given size. Arbitrary bytes can be read from and written to locations in these memory blocks. Additionally, each of these memory blocks has a “handle”, stored in a Java `long`. By storing these handles in memory blocks, we can create pointers between the blocks.

Additionally, the **Heap**’s metadata contains a “root”, which can store a single `long`: this is usually used to store the handle of a memory block.

Methodology

In this section, all tests are run on an Intel Xeon Gold 6252N with 128 GB Interleaved Optane DCs and a 4GB DRAM Java heap. We use the server compiler without tierd compilation (`-XX:-TieredCompilation -server`) and to ensure that garbage collection would not interfere with the results, we use the no-op epsilon garbage collector with `-XX:+UseEpsilonGC -XX:+AlwaysPreTouch`. Computation and memory accesses for each test are bound to a single NUMA node.

Timing behaviour

Before timing LLPL calls, we first need to determine the granularity and error in the actual timing method we use. We calculate elapsed time by calling `System.nanoTime()` before and after an experiment. To test this call itself, we call `System.nanoTime()` twice in succession and measure the difference in time. Performing this 1000 times gives us the results in Figure 3. We can see that the timing calls have very little overhead (always less than a millisecond), even without warmup runs. Some calls do occasionally take longer, but the overhead is still always less than half a millisecond.

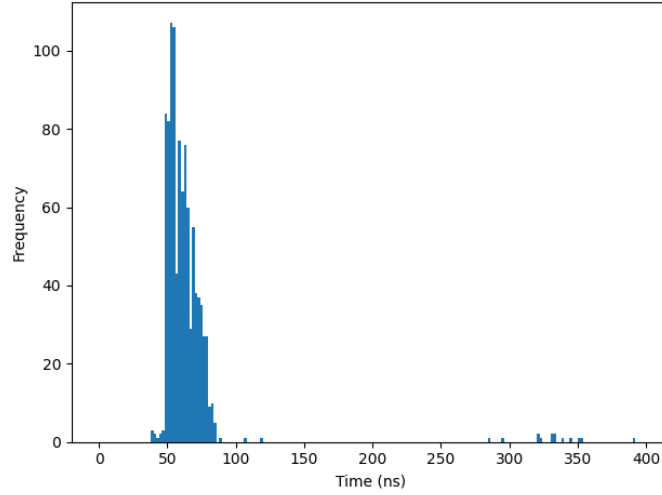


Figure 3: Histogram of the overhead of 1000 `System.nanoTime()` calls

Heap creation

We now consider how long it takes to create a heap of a fixed size. The results for creating a heap of various sizes are shown in Figure 4. For heaps that are less than a gigabyte, creation takes around 70 ms. When heaps get very large, the cost increases, but given this is a one-time cost, this is still a very cheap operation.

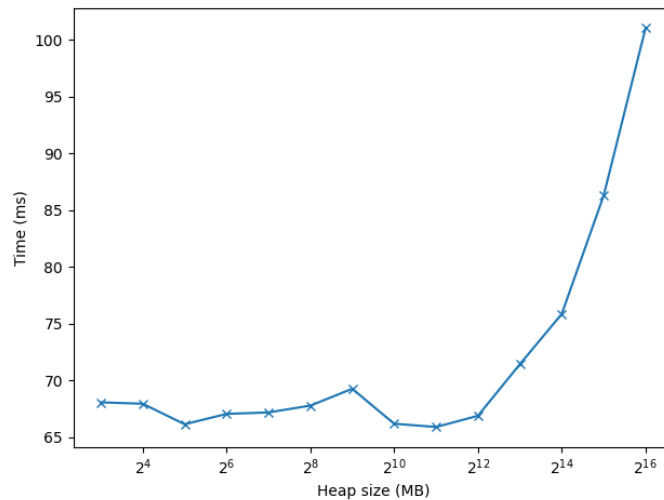


Figure 4: Time to create a fixed-size heap

Memory allocation

In LLPL, we can allocate either normal or compact memory blocks. Normal memory blocks store an extra `long` corresponding to the size of the memory block in their metadata, and compact memory blocks don't have any extra metadata.

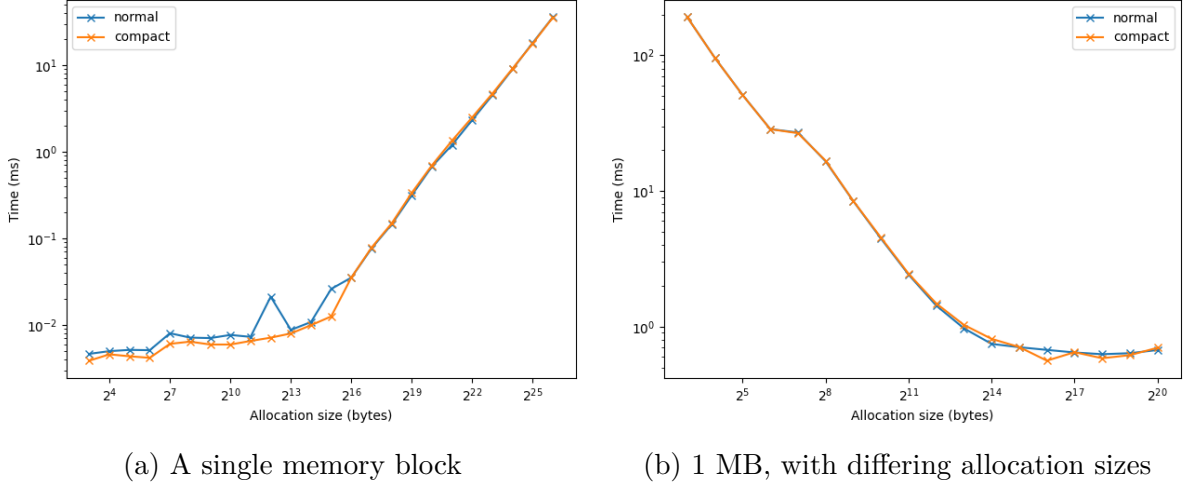


Figure 5: Time to allocate persistent memory

Figure 5a shows the time taken to allocate a single block of memory with respect to its size. We can see that large allocations scale linearly with their size. We also note that allocating memory of a “round” size is important; allocating a normal memory block with a size of 4 KB (the size of the interleaving of the Optane we are using) causes a slightly larger allocation, leading to a drastic performance decrease for that size, presumably due to the fact that this allocation will now be spread across two Optane DIMMs.

Figure 5b shows the time taken to allocate 1 MB of persistent memory using repeated allocations of a given size. As expected due to the presence of overhead in the allocation process, less large allocations are better than many small allocations. Note though that this effect doesn't seem to be present as long as the allocations are at least 4 KB. In fact, repeated compact allocations of 4 KB were the fastest way to allocate 1 MB.

Writing to a memory block

We now consider writing to persistent memory. [Yang et al. \(2020\)](#) found that the best access pattern to Optane is sequential access, and failing that, random access of size at least 256 bytes. For this reason, we test the following three memory access patterns in LLPL.

1. Random access at the byte level
2. Random access at the 256-byte level (choosing random 256-byte blocks and writing to them sequentially)
3. Sequential access

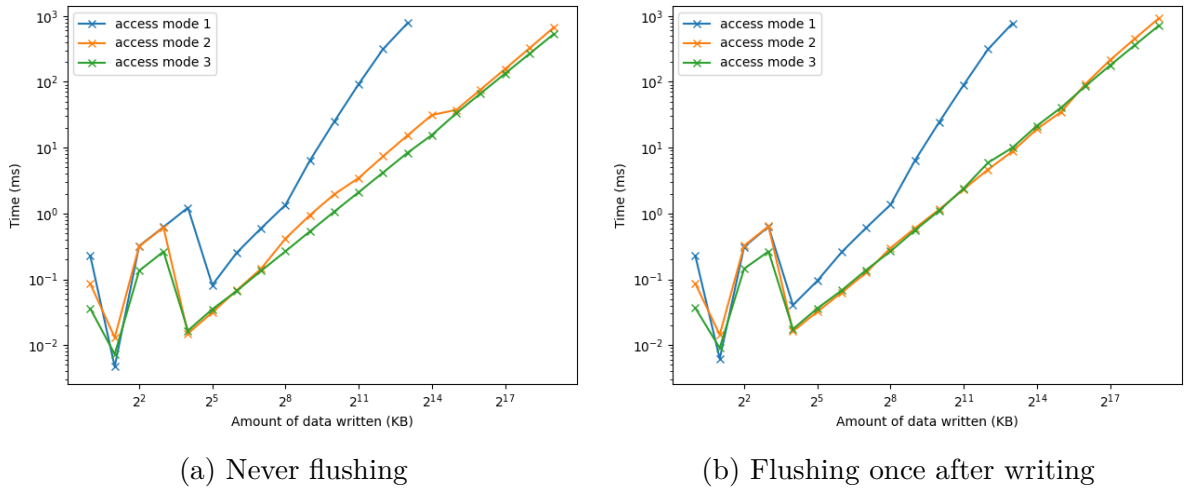


Figure 6: Time taken to write data using three different access patterns

Figure 6 shows our results. Even with warmup runs and averaging over many runs, we see inconsistent results for small writes. [Yang et al. \(2020\)](#) explain that Optane’s XPBuffer is able to coalesce writes of less than 16 KB, which is likely the cause of the strange behaviour of writes smaller than this.

One data point that behaves very differently depending on whether we are flushing is a 16 KB write with access mode 1; flushing actually makes this test faster. We hypothesise that this is because the flush before the test runs will clear the XPBuffer, allowing the test’s writes to be coalesced efficiently.

For large writes though, the trend is clear. Random writes at the byte granularity are far slower than sequential writes, and random writes at the 256-byte granularity are slightly slower than sequential writes. This aligns with the findings of [Yang et al. \(2020\)](#).

Reading from a memory block

We now consider reading from a memory block. We use the same test access patterns as we did when evaluating write performance. The results can be seen in Figure 7.

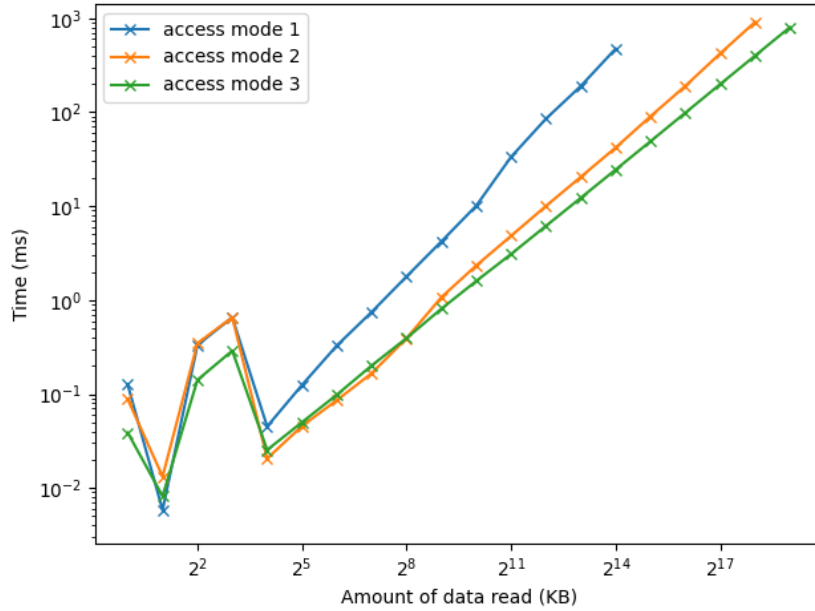


Figure 7: Time taken to read data using three different access patterns

Like when writing, we see inconsistent behaviour for reads less than the XPBuffer. Also like when writing, random reads at the byte level perform much worse than random reads at the 256-byte level and sequential reads.

Building a persistent memory index for Lucene

To change the underlying storage of a Lucene index, we subclass the `Directory` class to create a new `PMemDirectory`.

We encountered technical difficulties using LLPL in Lucene. Refer to the `dev-instructions` folder of the git repository for an explanation of how this was overcome.

The interface of a Lucene Directory

When subclassing `Directory`, a certain interface needs to be implemented, and corresponding subclasses of `IndexOutput` and `IndexInput` need to be made. This is shown in Figure 8.

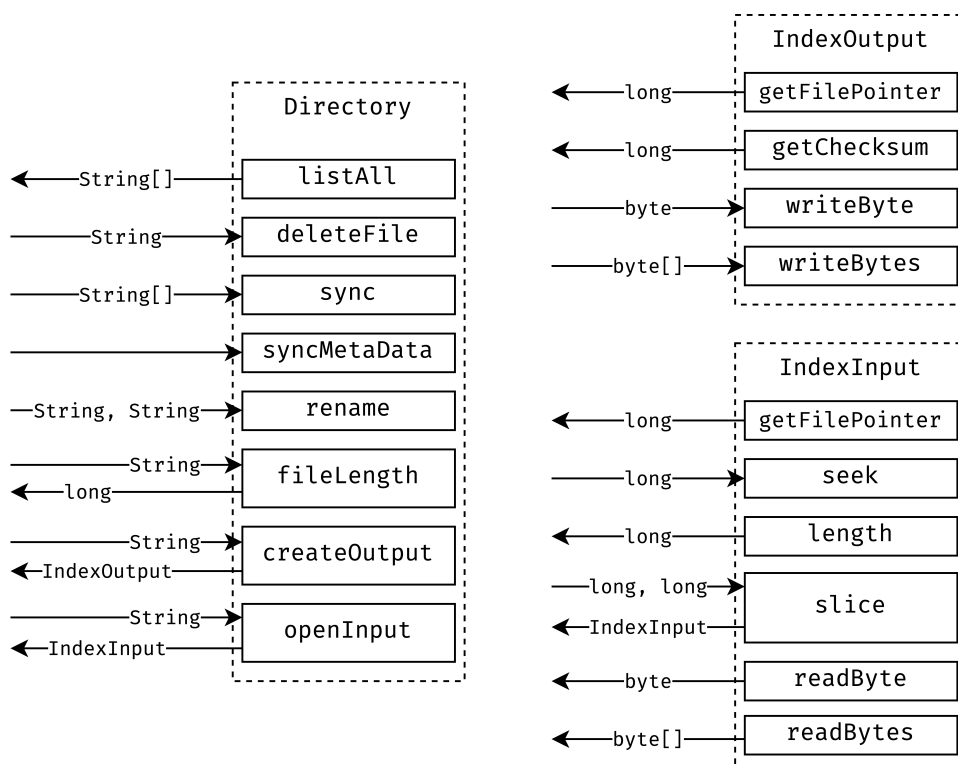


Figure 8: A simplified representation of the interface of a Lucene Directory

At a high level, a `Directory` must maintain a list of “files” that make up a Lucene index. When adding a file, it provides an `IndexOutput` object which is used for writing to the file in a completely sequential manner. When reading an existing file, it provides an `IndexInput` object which is used for reading from the file (random access is allowed using `seeks`, and `IndexInputs` can be `sliced` into smaller `IndexInputs`). Files are never modified, only deleted, and files never need to be read until after writing has finished.

Implementing a Directory using persistent memory

The overall structure of our `PMemDirectory` implementation is shown in Figure 9. All the data contained in the index is stored in persistent memory. When a `PMemDirectory` is created in Lucene, it is given the path of the index (a single persistent memory heap). The only data it stores is the `LLPL Heap` object.

The root memory block of the heap is a hash map that maps file names to file entries. A file entry stores its name and a pointer to the “header” memory block of the file. In our implementation, the hash map employs quadratic probing to insert and lookup elements, and the hash map is doubled in size through a reallocation whenever an entry is not found or cannot be inserted after 5 probes.

The “header” memory block of each file stores its length and a pointer to its data. This extra layer of indirection is useful since it allows the file data to be reallocated (doubled in size) to a new memory block when its capacity is reached without looking up the corresponding entry in the hash map.

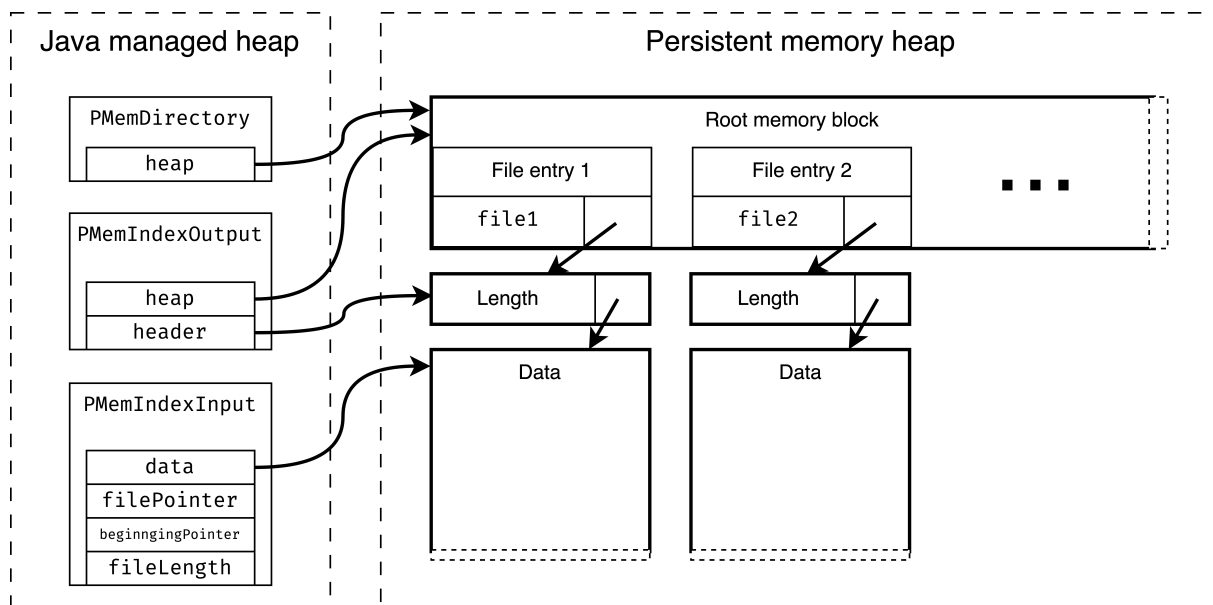


Figure 9: Structure of a `PMemDirectory`

There are three Java classes that we need to subclass to implement a Lucene `Directory`. Each of these classes stores handles to memory blocks on this persistent heap.

`PMemDirectory` stores a reference to the `Heap` so that it can access the root memory block to manage the hash map entries.

`PMemIndexOutput` stores a reference to the header of the file it refers to, and updates the length in the header as it writes to the file.

`PMemIndexInput` stores a pointer to the data block of the file it is reading from, as well as `longs` that store information about where in the file it is reading from, and the start and length of the file. These values are used to implement the slicing operation without copying any data in memory.

Opportunities for improvement in PMemDirectory

There are three main areas for improvement that we have identified in our implementation of PMemDirectory. Some of these improvements would require changes outside of the PMemDirectory class itself.

Better awareness of the Lucene index structure

Our current implementation is almost completely unaware of how it is used by Lucene. All files are treated exactly the same and so no special optimisations can be made. With deeper understanding of Lucene's index structure (see [Figure 1](#)), there is potential for optimisation. For example:

- With knowledge of how large certain files are likely to be (based on their file type), unnecessary allocations can be reduced.
- Some files store the names of other files as a way of referring to them. Instead of this, Lucene's algorithms could directly store the handles of the file header memory blocks. This would reduce indirection and therefore random data accesses.

Better awareness of persistent memory

Because of the complex performance characteristics of Optane (discussed in [characteristics of LLPL](#)), there are likely inefficiencies in our data structures that could be improved:

- As discussed in the section on [memory allocation in LLPL](#), inefficiencies in allocations can be introduced if they do not align with Optane's 4 KB interleaving. We double our memory blocks in size when we reallocate, but they have an extra 8 bytes of metadata. This is likely to be degrading our performance.
- Because of constant reallocation, when the index has a large number of files, accesses are likely to not exhibit much locality, which is important for persistent memory.

Better use of the managed heap

While the purpose of this project was to move data structures from DRAM to persistent memory for capacity reasons, it is important to recognise that DRAM (being faster than persistent memory) should still play an important role in the memory hierarchy of any application. In our implementation, very little data is stored in the managed Java heap. Frequently accessed metadata such as file lengths or the root hash map would probably benefit from being stored in DRAM, and even file data could potentially benefit from being cached in DRAM in some way.

Performance analysis

While the focus of this project was on the designing and implementing **PMemDirectory**, we do perform some benchmarking of our implementation. We first compare our query evaluation performance with that of the direct postings format use by [Chilukuri and Akram \(2023\)](#) before evaluating overall performance against the most common index storage format: a filesystem.

Comparison to direct postings format

The direct postings format involves storing uncompressed indexes in managed memory and then using these to evaluate queries. The long-term index storage is similar to using **FSDirectory** in a disk-backed filesystem, so in this section we only consider query performance.

The direct postings format answers queries in two stages. First, it loads the entire index into the managed heap and uncompresses it. Then, it responds to queries directly using this uncompressed format.

To perform a like-for-like comparison to our **PMemDirectory** implementation, we use the following experimental setup:

- Both approaches are bound to a single NUMA node.
- Both approaches use `-XX:-TieredCompilation` and `-server`, the strongest JIT compiler, and five warmup runs are conducted before the actual benchmark is run so that JIT compilation can occur.
- Both approaches run a set of 48,510 low-frequency queries searching an index created from 4.8 GB of Wikipedia article clippings.
- We vary the buffer sizes used for both approaches to see how they respond (the indexes are created with specific buffer sizes, which affect their segment size).
- The indexes are never merged.
- For a fair comparison, both approaches use only persistent memory and not DRAM. The direct postings approach uses a 20 GB managed heap (which the index is loaded into), and **PMemDirectory** uses a 12 GB managed heap, and the index is stored in an 8 GB persistent memory (non-managed) heap.

The results for this experiment are shown in Figure 10. Changing the buffer size of the indexing process results in changing the segment size of the index (and therefore the number of segments). We see that our implementation has similar performance to the direct postings approach for a large number of segments, but is up to twice as fast for a smaller number of segments. We hypothesise that our slower performance when there are more segments is due to locality issues in **PMemDirectory**; our current implementation makes no effort to ensure that reallocations occur in a way that keeps files close together.

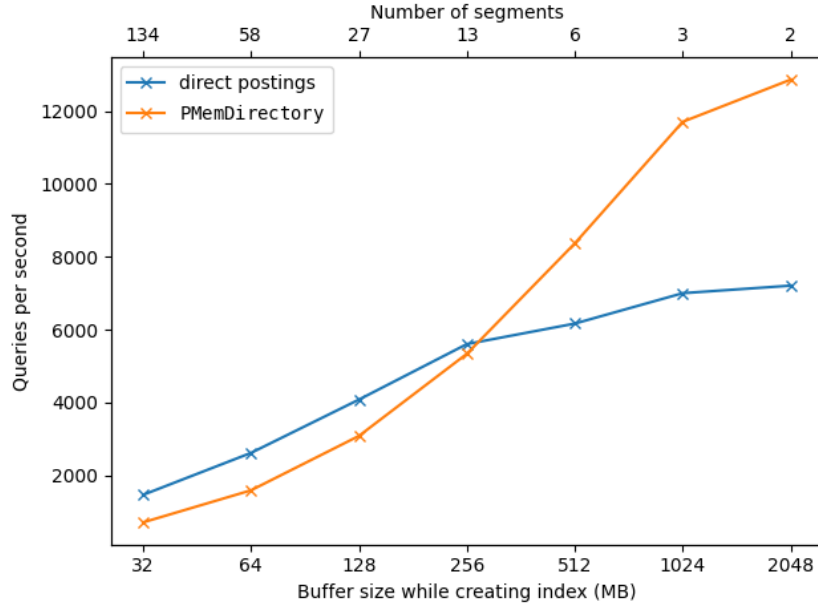


Figure 10: Query evaluation performance of direct postings format versus PMemDirectory

Our implementation has another advantage over the direct postings method: it doesn't require that the index be loaded from memory first. In the tests conducted in the above experiment, it took on average 3 seconds for the PMemDirectory to open the index to be ready for searching, whereas the direct postings method took an average of 37 seconds. While this one-time cost would be acceptable for applications where the index is largely static, it is unacceptable for real-time search. Every time the index is updated, the current direct postings implementation has to incur this load cost.

Comparison to FSDirectory

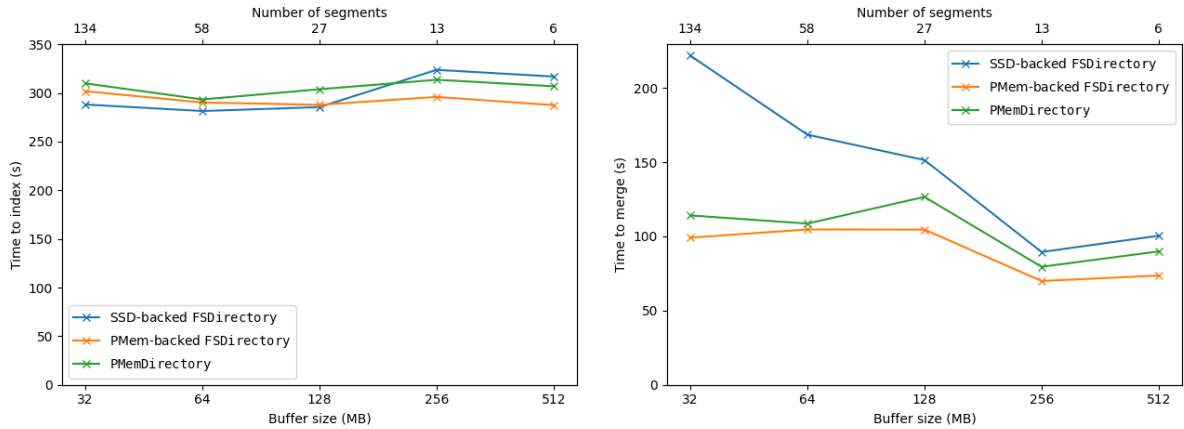
The standard way to store an index using Lucene is on an SSD-backed filesystem using the `FSDirectory` class. Here we compare `PMemDirectory` to this approach, and also to the approach of simply using `FSDirectory` on a persistent-memory backed filesystem (the configuration which [Pydipaty and Saha \(2018\)](#) attempted to emulate).

We use the same dataset and query workload as in the previous section, and we similarly bind all tests to a single NUMA node and use `-XX:-TieredCompilation -server`.

To limit the effects of the OS page cache (which for real applications wouldn't be large enough to hold much of the index), we run these benchmarks with memory restricted to 2 GB. Of this memory, 1700 MB¹ is used by the Java managed heap (which is located on DRAM, as normal).

Building the index

Figure 11 shows the time taken to build the index for our dataset. For consistency, we disable automatic merging and instead perform only one merge at the end of the indexation process.



(a) Building the index with no merging

(b) Merging the index into one segment

Figure 11: Time taken to build and merge an index using three different approaches

From Figure 11a, we can see that even with no memory remaining for the page cache, the time to build the index is similar for all methods. This makes sense as most of the work building the index is computational rather than IO-based. From Figure 11b, we can see that merging is different. The need for high memory bandwidth during the merging process makes persistent memory faster. This appears to be particularly true for indexes with many segments, indicating that the bottleneck of an SSD-backed `FSDirectory` is the number of repeated file operations.

Our `PMemDirectory` implementation is still slower than an `FSDirectory` stored on persistent memory, which indicates that there are inefficiencies in our implementation. With improvements to our design, our approach should be able to outperform an `FSDirectory` stored on persistent memory since we would be able to optimise for a specific use case, rather than rely on a generic filesystem implementation.

¹This was chosen as the largest possible amount without resulting in the process running out of memory.

Evaluating queries

When evaluating queries, as well as restricting process memory, we also purge the OS page cache before starting the process. This is because processes are able to use pages that have been cached by previous processes without this usage counting towards the memory restriction we impose.

We also found that with our memory-restricted setup, an SSD-backed `FSDirectory` was more than 100 times slower than the persistent memory implementations. This makes sense as without the OS page cache, SSD accesses (which are serialised in this workload) are drastically slower than persistent memory accesses. For this reason, we exclude this configuration from our results.

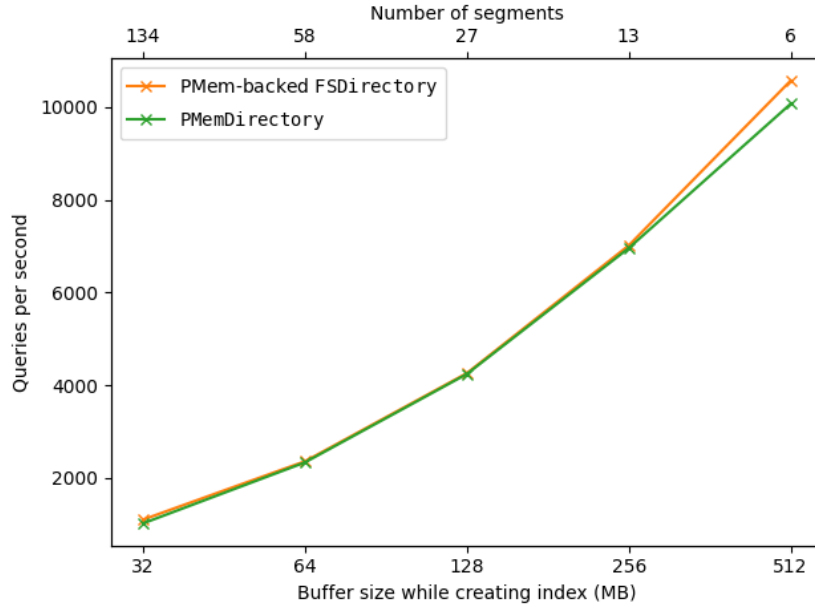


Figure 12: Query evaluation performance of direct postings format versus `PMemDirectory`

Figure 12 shows our results comparing the query bandwidth of our implementation with the naive persistent memory implementation. The performance is almost identical. We can see that our implementation is slightly slower when there are a small number of segments. This makes sense: when the number of files is fewer, the filesystem-based implementation won't suffer as much from the overhead of file operations.

Note that this is the opposite of what we saw when we compared to the [direct postings approach](#), where our implementation was *faster* for fewer files. This indicates that there is an opportunity to improve `PMemDirectory` by combining attributes of both the direct postings format and this naive approach.

Conclusion

In this project, we analysed LLPL and its performance characteristics before applying this to build `PMemDirectory`, a class of Lucene that allows storing indexes on persistent memory.

While we were unable to outperform the naive approach of storing a normal (`FSDirectory`) Lucene index in a directory backed by persistent memory, our approach showed promise. Our implementation's performance when building an index with no merging and when evaluating queries was almost identical, and the merging process was only slightly slower. The [many opportunities for improvement](#) in our design lead us to believe that our work could outperform the naive approach with modifications.

Our work showed promising characteristics when compared to the direct postings format, which stores uncompressed index data directly in the managed Java heap. Our implementation was able to compete with and actually outperform the direct postings format when the number of segments in the index was small. The fact that our implementation performed relatively worse with larger numbers of segments further indicates that there are specific areas that can be improved in our design.

References

- APACHE, 2023. Lucene. <https://github.com/apache/lucene>. Version 9.6.0.
- CHILUKURI, A. AND AKRAM, S., 2023. Analyzing and improving the scalability of in-memory indices for managed search engines. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2023 (Orlando, FL, USA, 2023), 15–29. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3591195.3595272. <https://doi.org/10.1145/3591195.3595272>.
- INTEL, 2022. LLPL. <https://github.com/pmem/llpl>. Version 1.2.1.
- KOLOKASIS, I. G.; EVDOROU, G.; AKRAM, S.; KOZANITIS, C.; PAPAGIANNIS, A.; ZAKKAK, F. S.; PRATIKAKIS, P.; AND BILAS, A., 2023. Teraheap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023 (Vancouver, BC, Canada, 2023), 694–709. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3582016.3582045. <https://doi.org/10.1145/3582016.3582045>.
- PYDIPATY, R. AND SAHA, A., 2018. On using non-volatile memory in apache lucene. (04 2018).
- YANG, J.; KIM, J.; HOSEINZADEH, M.; IZRAELEVITZ, J.; AND SWANSON, S., 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 169–182. USENIX Association, Santa Clara, CA. <https://www.usenix.org/conference/fast20/presentation/yang>.