Exploiting Intel Optane Persistent Memory for Full Text Search

Shoaib Akram
Australian National University
Canberra, Australia

Abstract

In our information-driven societies, full-text search is ubiquitous. *Search is memory-intensive.* Quickly searching massive corpora requires building indices, which consumes big volatile heaps. *Search is storage I/O-intensive.* Limited main memory necessitates writing large partial indices on non-volatile storage, where they finally live in merged form. These indices reside in memory, in full or in part, during query evaluation. Memory and I/O intensity make it hard to index and search content rapidly and efficiently. On the hardware side, the recently introduced Intel Optane DC persistent memory (PM) offers byte-addressability, high capacity, and non-volatility. This paper evaluates and exploits Optane PM for text indexing and search on multicore platforms.

We identify essential structures in inverted indices (hash table, merge tree, and key-value store), where they reside (memory or storage), and key operations over them (sort, flush, and merge). We allocate these structures in DRAM, Optane PM, and block storage by modifying an existing search engine. We then evaluate a myriad of hybrid memory and storage configurations. Our findings include: ① careful placement of index structures across DRAM, Optane PM, and SSD, speeds up indexing with a single core compared to a high-performance baseline but does not scale to many cores, ② crash-consistent indexing with Optane PM is feasible without incurring a high overhead, and ③ the tail latency of the longest multi-term conjunctive queries is lower with a PM-backed index than an SSD-backed one. This paper opens up persistent memory to a practical role in full-text search.

CCS Concepts: • **Information systems → Search engine indexing**; **Phase change memory**.

Keywords: Text search, inverted index, persistent memory

ISMM '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8448-3/21/06…$15.00
https://doi.org/10.1145/3459898.3463906

ACM Reference Format:
Shoaib Akram. 2021. Exploiting Intel Optane Persistent Memory for Full Text Search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3459898.3463906

1 Introduction

Text search powers vital services in the modern world. Popular engines, such as Google and Bing, deliver new information to individuals and enterprises every day. Equally prevalent is searching short, rapidly generating social media content (e.g., Facebook and Twitter) in real-time. Due to its wide use, improving search speed and efficiency is urgent.

Trivial search tools serially scan each word in a corpus to match documents to queries. However, for large corpora, such as the world wide web or real-time tweets, full-text search consists of (1) text indexing and (2) query evaluation.

Text indexing builds an inverted index of the corpus that maps terms (words) to postings (document identifiers). Indexing speeds up query evaluation, but unfortunately, it is memory and storage I/O-intensive. Building an index requires large heaps, and storing it results in large amounts of storage I/O. Also, evaluating search queries in real-time demands fast access to inverted indices. The current practice is to cache large indices in memory [20]. On the hardware side, persistent memory (PM) offers byte-addressability, scalable capacity, and non-volatility. This paper exploits and evaluates Intel Optane DC PM for text indexing and search.

Building and storing inverted indices on persistent memory has four advantages. ① To store words and postings temporarily, indexers require large hash tables, well-suited to scalable memories. ② Periodically, indexers sort the tables and flush them to a storage device, where they merge them into a large index. The costly I/O due to flushing and merging is avoided by direct access to fast persistent memory. Bypassing the OS buffer cache, the resulting stack is also clean, simple, and efficient. ③ Current indexers waste significant effort to offer fault-tolerance over traditional storage [18]. Persistent memory offers, *instant-on* and *crash-consistent* preservation of state. More importantly, programmers can reason the guarantees for fine-grained persistence from the user space without the intricacies of coarse-grained block updates and OS caching [11, 12, 40]. ④ The query evaluators today either read the index from a solid-state drive (SSD), risking poor response time, or keep them in main

memory, wasting DRAM capacity. Keeping inverted indices in persistent memory introduces novel space-time tradeoffs.

We introduce Intel Optane PM in a commodity server and empirically evaluate the rich space of hybrid memory and storage configurations for text indexing and search. Empirical evaluation of nascent technologies can betray intuition. Consider, for example, a configuration with a DRAM-backed hash table, PM-backed postings, and SSD-backed dictionary. This configuration outperforms more straightforward ways of exploiting persistent memory. Exhausting the rich space demands a fast and flexible text indexing and search tool. We build upon a high-performance and native (C++) search engine, namely Psearchy [9, 31]. Prior art uses it in industrial-strength search services [21, 44].

We modify the engine to build, update, and store inverted indices across different memory and storage types. Specifically, we place the critical (volatile) index structures in DRAM or PM and non-volatile ones in PM or an SSD. (Our server has an Intel Optane NVMe SSD.) We focus on execution time and scalability (indexing) and tail latency (search) of different query types. We discover and report new, sometimes surprising, performance versus device-type and capacity tradeoffs. We propose and evaluate crash-consistent text indexing at a finer granularity. Also, we use hardware performance counters to gain insight into the behavior of Optane PM. Finally, we uncover performance pathologies in the state-of-the-art PM software.

In summary, this work is the first to evaluate Intel Optane PM as the main memory, as an extension of the main memory, persistent storage, and a universal memory (main memory and storage) for full-text search.

The main findings of our evaluation are:

- Persistent memory slows down indexing with one thread by more than 30% as the primary and universal memory, compared to a DRAM-SSD composite and stock search engine. Poor scaling with rising core count renders its main and universal roles at a supreme disadvantage.
- With careful placement of the hash table, the partial and merged postings, and the dictionary, a hybrid approach with DRAM and PM, speeds up single-threaded indexing by 20%. At high core count, this hybrid solution performs only as well as the stock.
- For text indexing, the premium for fine-grained crash consistency with persistent memory is between 18% to 30%. This overhead due to persist-ordering instructions, surprisingly, diminishes at high core count.
- Updating an existing PM-backed index is 10× faster than an SSD-backed index. The primary advantage is due to the removal of filesystem I/O operations.
- The PM-aware key-value storage engines perform up to 60% worst than Berkeley DB (BDB) for dictionary storage. The analysis points to the untapped potential for performance gains over the BDB engine.
- The tail latency distribution for single-term queries is similar with PM and SSD-backed indices. However, persistent memory reduces the tail latency of the longest-running multi-word conjunctive queries substantially. This trend is due to the access patterns the query workloads generate.

2 Background

We provide background on our baseline search engine and justify our choice. We provide a tutorial on the inverted index and briefly discuss Intel Optane DC persistent memory.

2.1 Baseline Search Engine

Our baseline search engine is Psearchy [9, 31, 44]. It consists of a parallel, scalable text indexer and query evaluator written in C++. It powers OverCite in prior work [44], a distributed, community-driven variant of CiteSeer [21]. Prior work also exploits its distributed hash tables (DHTs) to evaluate peer-to-peer search [31]. Prior efforts make it multicore scalable [9], and rigorously evaluate its scalability [2]. We download Psearchy from the MOSBENCH suite [10].

We use Psearchy for three reasons. ① It is written in C/C++, which eases integration with persistent memory libraries, such as Intel PMDK [13]. ② Managed search engines, such as Solr [17] and Elasticsearch [16], complicate performance analysis, especially with hybrid memories. Specifically, controlling the perturbation due to garbage collection and just-in-time compilation is tedious [6, 7, 25]. ③ Teasing apart the indexing time is critical for understanding the interaction between text search and Optane PM. This deconstruction is challenging with industrial-strength engines. The closest effort provides a coarse-grained breakdown of query evaluation only [26].

2.2 Inverted Index Tutorial

We show the high-level structure of an inverted index and discuss an efficient way for constructing such an index.

2.2.1 Structure. An inverted index consists of a dictionary of terms (words) and postings. Postings record the position and frequency of words in documents. The dictionary maps terms to posting lists. Postings are linked together in an ordered or unordered list. Figure 1 shows an example inverted index for a corpus with two documents. Each word has one or more postings that record its position and frequency in each document. Psearchy stores postings as unordered lists. Lucene-based engines offer skip lists to jump over posting lists and improve search speed [45, 49]. The query evaluators iterate over postings to rank matching documents [44].

2.2.2 Construction. The indexing engine in Psearchy is multithreaded. We show the per-core data structures it uses in Figure 2. A document map contains document ids and their location on a filesystem. Modern indexing engines for full-text search use a structure similar to a log-structured merge-tree (LSM). During indexing, an in-memory hash table, also called memtable, first records the words and their

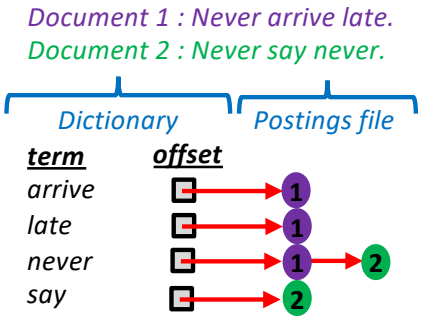


Figure 1. High-level structure of an inverted index.

posting lists. Typically, the memtable is not searchable [38]. On exhausting the memtable, the indexer writes its contents, including the terms and postings on persistent storage. The table is then available for reuse. In popular engines, the terms (keys) and posting lists (values) are organized into key-value pairs and kept in SSTable-like sorted files. These files are also called segments, or partial postings file (PPF) [38, 44]. Unlike Lucene, the partial files in Psearchy are not searchable. The indexer merges several PPFs into a binary file. Merging also results in a dictionary that tells the queries at which offset in the sorted file they need to look for a term.

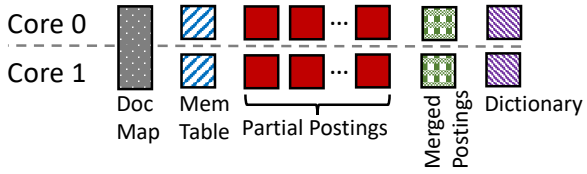


Figure 2. The global and per-core data structures for indexing in Psearchy.

Psearchy indexes incoming documents in three stages (Figure 3). The single-threaded *Name-to-docId* stage assigns each document an integer docId, which it inserts into a shared work queue. Psearchy stores the mapping of document names (and their location on a filesystem) to docIds in a Berkeley DB (BDB) file. We show this structure as a document map in Figure 2.

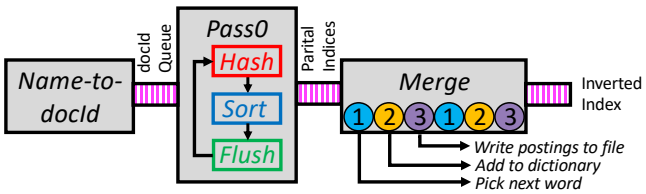


Figure 3. The stages in the construction of an inverted index.

The next stage is *Pass0* in which a per-core indexer makes the first pass over each new document. Specifically, each core picks a docId from the shared queue, parses it, and notes each word's positions in a hash table. The table consists of several buckets. Each bucket stores a unique word (term) and pointers to the first and the last block. A block contains a pointer to the next block and pointers to a series (n) of postings. (In the baseline engine, n is equal to 128.) A large n risks wasting unused block memory, and a small n risks instantiating too many blocks. The hash table uses open addressing and inserts a new bucket in one of several locations. When the hash table is full, the indexer sorts the table (qsort from the C library) and flushes the sorted table to block storage. The resulting PPF resembles an SSTable and forms the first level of LSM. The memtable absorbs random writes in DRAM and transforms them into sequential writes on storage, improving I/O throughput.

During the next stage, namely *Merge*, each core merges all PPFs into a per-core postings file. Merging also results in a BDB dictionary. The merge procedure either creates a new index from scratch or updates an existing index.

We next discuss the I/O behavior of flush and merge operation. Both operations are I/O-intensive. Eliminating the I/O overhead, including the system call, slow device access, context switch, is one of the advantages of Optane PM.

Flushing hash table to disk. The flush procedure writes the in-memory hash table as a PPF on block storage. This operation results in sequential write I/O. The procedure picks a bucket and first writes the term, then an integer n (number of postings), and then the term's postings. The indexer opens a file stream and calls OpenBSD putc() for writing each posting list. The writes first arrive in a C library buffer (DRAM). On exhausting the table's flush, the indexer closes the file, which implicitly calls fflush(), flushing file contents to the OS buffer cache. The indexer does not call fsync(), avoiding device writes. The OS performs device I/O asynchronously. These details are critical for interpreting the results of performance evaluation against Optane PM.

Merging partial indices. The merge procedure reads a term from one of the many PPFs, calling fread() on an open file stream and checking which other files contain the same term. Merging may benefit from the OS buffer cache. It then reads the number of postings (n) stored in each partial file, copying character-wise all postings to a single (merged) postings file. It writes each term's postings count in the merged file to ease query evaluations. The procedure also inserts the term and an offset to the posting list in a BDB file. Once merged, the indexer closes the index, flushing writes to the buffer cache without ensuring crash consistency.

2.3 Simplifying Assumptions

Text search is highly inter-disciplinary. For example, our engine supports the semantic cleanup of search queries through a stop word file and relevance-aware document ranking. However, we ignore architectural issues due to distribution and focus on indexing and search on a single multicore machine. Prior work from Twitter evaluates similarly [32]. On

the query evaluation side, we leave out load balancing and caching for frequent queries.

2.4 Intel Optane DC Persistent Memory

A non-volatile DIMM (NVDIMM) connects to the memory bus, similar to conventional DRAM. Intel Optane DC PM is the most scalable and cost-effective NVDIMM to date. It exploits a new storage medium called 3D XPoint, which stores information as a change in the material's bulk resistance [30]. 3D XPoint is more scalable than DRAM, and the current NVDIMM capacity is up to 512 GB (8× that of DRAM.) Communication with Optane DIMM is mediated by the processor's integrated memory controller (iMC). iMC uses a new DDR-T (64-byte) interface. DDR-T enables asynchronous command and data timing. Once a request reaches the Optane DIMM, a module controller reads and writes to the actual media at a 256-byte line granularity. A 16 KB write combining buffer merges adjacent lines to mitigate high media latency [57].

Optane PM has two operation modes. The *memory* mode turns DRAM into a direct-mapped cache for PM, and the host memory controller transparently manages the DRAM cache. Unlike the memory mode, the *App Direct* mode exposes the hybrid of DRAM and Optane PM to the software. The PM media is abstracted by a light-weight filesystem [48, 56]. Our evaluations use the *App Direct* mode.

Recent work finds Optane PM between 2×-3× slower than DRAM [57]. More slow are random accesses compared to sequential accesses. Optane PM's bandwidth is also lower than DRAM. Its read-to-write bandwidth ratio is higher than DRAM. Interleaved Optane PM delivers higher bandwidth than non-interleaved one. More background on Intel Optane PM is found in recent work [1, 57].

3 Baseline Characterization

This section characterizes the baseline Psearchy across four dimensions: ① multicore scalability of indexing, ② teasing apart indexing overheads, ③ latency of index updating, and ④ tail latency distribution of query evaluation. Our primary goal in this characterization is to pinpoint potential acceleration targets for Optane PM.

Figure 4 (a) shows the time in seconds for indexing the large dataset. We allocate the hash table in DRAM and store the postings (partial and merged) on an SSD. (See Section 5 for details about our datasets, search workloads, and methodology.) We show the indexing time with increasing core count. Psearchy's indexer is fast and takes only 500 seconds to index a 20 GB corpus. We observe good scaling up to eight logical cores with the indexing time reducing by 5.8×. The indexer's scalability is limited beyond eight cores. Still, indexing the large corpus takes under one minute with 32 cores. We also show the breakdown of single-threaded indexing time into five components in the same figure. (Refer to Figure 3 and Section 2 for the meaning of each component.)

We observe that assigning docIds to incoming documents (DocId) is fast (4% of total time). The indexer spends a substantial 22% of the time looking up the hash table (Hash) and inserting new entries in it. Sorting the table (Sort) takes up the most time (35% of total). Flushing partial postings on SSD (Flush) takes 24% of the indexing time. Merging the partial postings into an inverted index (Merge) consumes 17% of the time. The C memory allocator's overhead is negligible.

The breakdown of the indexing time changes at high core count (not shown). The docId assignment is single-threaded, and with 32 cores, it consumes 40% of the time. Both Hash and Sort make up a smaller percentage of time (13% and 28% respectively). Each core in multicore execution instantiates a fixed-size hash table. With more cores and the same dataset size, parsing and looking up words in the hash table, flushing, and sorting are parallelized, thus reducing the total indexing time. Flushing makes up 12% of the indexing time with 32 cores. Flushing leads to sequential disk writes and thus benefits from multicore execution. Finally, merging takes up 7% of the indexing time with 32 cores.

Quickly updating an existing index is critical [45]. Figure 3 (b) shows the update time with an SSD-backed index. We use a single thread to add a document to the large and the small dataset's index. We observe substantially high update latencies. Updating an index involves: (1) reading the old index into memory and (2) writing the new index on storage. The OS buffer cache hides the latency of a slow storage device. However, the filesystem I/O operations waste precious CPU cycles. Merging consumes most of the update time.

So far, we have analyzed the performance of the Psearchy's indexer. We now discuss tail latency distribution for search queries served by the Psearchy's search engine. We use the S1 query workload that sends 10,000 single-term queries to a server with our large SSD-backed inverted index. (We will evaluate multi-term query workloads in a later section.) Figure 4 (c) shows the distribution of tail latency. Each query is served by one instance of a single-threaded Psearchy query evaluator. The total query workload saturates our 96-core server. We observe that around 12% of the queries resolve in 10 milliseconds (ms) or less. Around 50% of the queries take less than or equal to 100 ms. Finally, the query evaluator resolves 95% of the queries in one second or less. Less than 1% of the queries run for 10 seconds or more. Our 90th percentile tail latency matches the one reported in recent work with an industrial-strength search engine [22, 23]. The longer tail in our evaluation is due to scoring and reporting all matches.

Acceleration opportunity. Our analysis reveals three opportunities to exploit Optane PM in full-text search. The first two pertain to indexing and the third to query evaluation. ① Flushing partial postings to a storage device and later merging them consumes between 20% (32 cores) and 40% (one core) of the execution time. Backing the partial postings by Optane PM can speed up flushing and merging. ② The

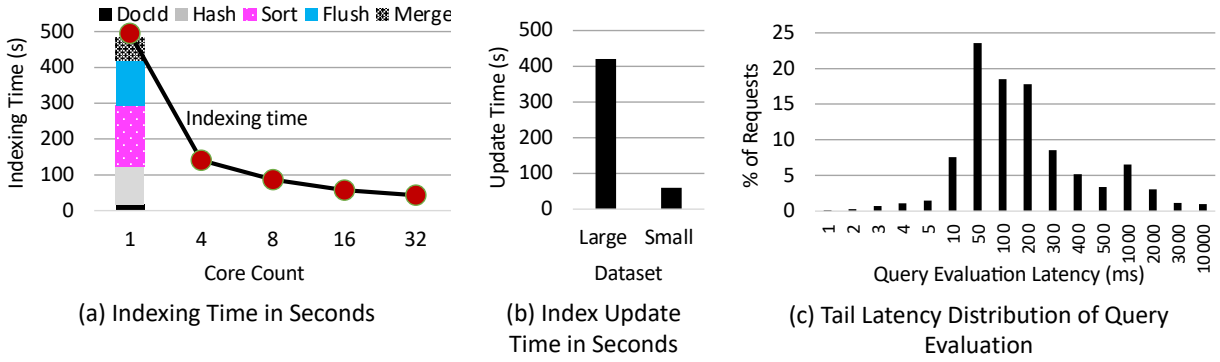


Figure 4. Characterizing the baseline indexing and search engine: (a) Multicore scalability and teasing apart the indexing time with a single thread, (b) Index update time in seconds, (c) Tail latency distribution of single-term queries in the S1 workload.

capacity advantage of Optane PM provides an opportunity to allocate a much larger hash table in memory. ③ The query evaluator can respond faster with PM-backed indices. We particularly anticipate a significant reduction in tail latencies for queries that generate random storage I/O.

4 Exploiting Persistent Memory

We now discuss incorporating Intel Optane persistent memory (PM) in full-text search. Not all configurations lead to a performance gain. Our objective is to explore the rich space of configurations that a system with hybrid DRAM and Optane PM and traditional storage offers.

4.1 Indexing with Optane PM

4.1.1 Optane PM as Main Memory. The prime advantage of Optane PM as the main memory is its high capacity. This high capacity is beneficial for allocating the per-core hash table that temporarily stores words and postings. We allocate this per-core hash table in Optane PM. Compared to a DRAM baseline, we use tables between four to sixteen times larger. We then evaluate if PM's high capacity reduces the indexing time.

The hash table is split into four contiguous regions. Each region is dynamically allocated. The four regions contain hash table buckets, linked lists of blocks with pointers to postings, the posting lists, and the words in documents. The indexing engine takes a user-specified heap size (e.g., one gigabyte), splits it into four parts, and allocates each part to a separate region. When one of the regions is full, the indexer flushes the table to disk.

Different datasets stress a different table region. For instance, a dataset in which a few terms repeat in one billion documents stresses the postings region. The region that stores the words stays mostly unoccupied. On the other hand, a large corpus in which each document contains unique terms stresses all regions. Realistic datasets fall between the two extremes. The nature of the dataset thus impacts the sorting and flushing frequency. Both these operations impact the indexing performance with Optane PM. In particular, sorting requires frequent main memory accesses as it compares words stored in different buckets in the hash table.

Similarly, the flush operation first reads hash table entries in a user-level buffer from which it copies the entry into a separate disk-bound buffer. Overall, the tradeoff between PM capacity and indexing performance is not a simple one. We empirically analyze this tradeoff in this work.

In addition to the hash table, the indexer allocates several small data structures, for example, strings for storing temporary file names, the work queue, and temporary buffers for holding dictionary entries. We allocate these structures in DRAM in evaluating Optane PM as the main memory. Storing them on PM-backed files increases fragmentation and meta-data overhead.

Memory allocation. The C library malloc() is unavailable for PM allocations. Therefore, to allocate tables in Optane PM, we first pre-allocate files on it. We find that allocating large files by writing zeros in Optane PM is prohibitive performance-wise. We instead use posix_fallocate() to create files of a specific size. We then map the newly created file in virtual memory by calling mmap() with the MAP_SHARED and MAP_SYNC flags.

4.1.2 Optane PM as a Memory Extension. Our baseline search engine flushes partial postings to block storage. The reason is limited main memory (DRAM). Across different datasets, we observe the volume of partial postings to be around 60% of the dataset volume. A 1 TB dataset would require a massive 600 GB DRAM footprint. In essence, DRAM alone does not scale to large datasets.

Instead of flushing the partial postings to block storage, we store them in Optane PM instead. One option is to change the indexer's design and use a single large hash table, obviating the need to flush entirely. However, this is challenging for two reasons: (1) large PM-backed tables increase lookup latency, (2) they demand large amounts of virtual memory. Similar to baseline Psearchy, we use a fixed-size DRAM hash table, and once it is full, we flush its contents to a separate file.

To flush the contents, we first allocate a small file on Optane PM and map it into the indexer's virtual address space. We grow this file as flushing progresses, calling ftruncate().

The flush operation in the baseline engine essentially serializes the hash table buckets and writes them to a storage device. It first writes the term, then the postings, followed by meta-data. Figure 5 lists the xwrite() procedure in Psearchy that writes words and postings data (as bytes) to a file. The procedure writes n bytes, pointed to by xptr, to a file stream, namely fp unless it encounters the end of the file. The bytes first arrive in a library buffer and then move to the OS buffer cache. Writing more than one byte in a single call does not impact performance. In fact, for frequent short writes, the OpenBSD fwrite(), is slower than putc().

```
1 int xwrite(const void *xptr, int n, FILE *fp) {
2  const char *ptr = (const char *) xptr;
3  for(int i = 0; i < n; i++) {
4   if(putc(ptr[i], fp) == EOF)
5    return(EOF);
6  }
7  return(i);
8 }
```

Figure 5. Writing partial postings to file streams in Psearchy.

```
1
2 int op_xwrite(const void *xptr, int n, char **fp) {
3  const char *ptr = (const char *) xptr;
4  for(int i = 0; i < n; i++) {
5   *(*fp + i) = ptr[i];
6  }
7  *fp += n;
8  return(i);
9 }
```

Figure 6. Writing partial postings to memory-mapped Optane PM files in modified Psearchy.

Next, the op_xwrite() procedure in Figure 6 stores bytes to Optane PM. This procedure receives a pointer to the location holding the memory-mapped file's address. It writes n bytes to PM and also advances the file pointer by n bytes.

Implementation lessons. Migrating from file streams to memory-mapped (PM) files demands care. Correctly advancing the memory-mapped file's pointer is vital. Sometimes, the same pointer requires context-specific conversion to the correct data type. Manually placing end-of-file markers in text manipulation is another source of errors.

4.1.3 Optane PM as Persistent Storage. So far, we have explored Optane PM as the main memory or its extension. We now explore it as fast storage for persistent data. Our indexer merges the partial postings into an inverted index. This index consists of a binary postings file and a dictionary. We use Optane PM as fast storage in two ways: (1) storing the merged binary file and (2) replacing the Berkeley DB store (dictionary) with persistent memory stores.

We now discuss the merge operation with our modifications. There are two scenarios: (1) the partial postings reside on block storage, (2) the partial postings reside in persistent memory. In the first case, the indexer gathers the postings for each word from multiple files, reading them into DRAM, one word (and its postings) at a time, and then writes them to Optane PM. On the PM media, this process is write-intensive. In the second case (partial postings in Optane PM), the indexer reads these partial indices from Optane PM and writes the merged index. Each core merges its partial indices. A mix of reads and writes stress PM's limited bandwidth.

We also replace the BDB dictionary with a persistent memory key-value store. We use the Intel-backed pmemkv store in this work. The pmemkv store provides various storage containers (or engines). The concurrent non-volatile hash map (CMap) engine provides crash consistency across key and value updates. On Intel 64, persisting a cache line requires one flush instruction, e.g., clflush_opt and one store fence (sfence) [42]. This *persist-ordering* instruction sequence (so named because it imposes an order on the writes to persistent memory) introduces extra overhead [29]. To relax crash consistency, we also experiment with a volatile hash map (VCMap), a fairer comparison to the native engine that does not guarantee crash consistency. We also use a third storage engine, namely Hollow, to evaluate the minimum overhead of a PM-backed dictionary.

Optane PM as persistent storage offers fine-grained crash consistency. The CMap-backed dictionary updates are instantly visible, transactional, and crash consistent. However, for this to be useful also requires crash-consistent flushing and merging (see Figure 3). In this work, we explore one approach for a crash-consistent indexing scheme. The design space is rich, but our primary interest is to uncover the overhead due to persist-ordering instructions. We leave more sophisticated schemes for future work.

Crash consistent indexing scheme. To flush the partial postings in a crash-consistent way, we call Intel PMDK's pmem_persist() after the flush operation in Pass0 in Figure 3. (pmem_persist() implies clflush_opt and sfence.) We pass the starting address and size of the partial postings file. We also write a unique marker to the partial file. On recovery, the indexer ignores partial files without this marker and reprocesses the associated documents. We then set and persist single-bit entries in a PM map to remember the fully processed documents. If this last step is interrupted, the indexer reprocesses some of the documents. Otherwise, in recovery mode, the indexer processes the unprocessed documents.

To merge partial indices, the indexer first writes the term to a binary file, then inserts it in the dictionary, and then writes the postings to the file. This mechanism is shown in Figure 3. We switch the ordering of the dictionary insertion

and postings' write for instant-on visibility. Otherwise, a valid entry in the dictionary leads the query evaluator to an incomplete and possibly non-existent posting. Next, to flush the postings from the cache lines, we use pmem_persist() and pass the starting address and the byte count. We also persist offsets to remember the last processed term in each partial file. We use a valid bit to ensure the atomic durability of offsets. The recovery protocol reads the offsets and replays the last merge operation. It locates the offset to the final merged file from the dictionary. Merging then continues as in the normal indexing mode.

Native Psearchy neither guarantees instant-on preservation of index entries nor crash consistency. A local or remote process accesses the fully merged index from the buffer cache or storage. The OS flushes its cache asynchronously. One can imagine scenarios where a query evaluator reads two versions of the index, before and after the crash. Unlike PM, fine-grained and instant-on indexing in the stock engine is tedious. A straightforward solution of inserting fsync() after line 7 in Figure 5 substantially slows down indexing.

4.1.4 Optane PM as Universal Memory. Persistent memory is byte-addressable and non-volatile. These features make our systems closer to having universal memory. Unfortunately, its high latency and limited bandwidth are likely to degrade performance over a classical storage stack. We evaluate Optane PM as universal memory. We allocate the hash table and then sort it in place in Optane PM. Large hash tables induce high latency. Therefore, we copy the table into a file that we map in virtual memory. We merge the indices in PM and use pmemkv to store the dictionary.

We envision two challenges for Optane PM as universal memory. ① The indexing engine generates a mix of read and write traffic, especially during the flush and merge operations. PM's bandwidth should be able to cope with a heavy workload of reads and writes. ② The in-memory hash table is semantically a temporary structure, not requiring non-volatility. Prior work shows a tradeoff between PM write latency and retention time [60]. The latency of updating the hash table on Optane PM is likely to be significant. Unfortunately, in today's PM offerings, a knob to trade non-volatility off for lower write latency is unavailable.

4.2 Query Evaluation with Optane PM

We also exploit Optane PM for query evaluation. We place the inverted index in PM and evaluate the tail latency distribution for different query types. We compare these distributions to ones with DRAM and SSD-backed indices.

On receiving a query request, the Psearchy query evaluator first reads the dictionary entry. It then opens a file stream and verifies that the matched term resides in the postings file at the offset returned by the dictionary. It uses a combination of fread() and fseek(). The evaluator then maps the postings in virtual memory and evaluates the query, giving each document a score based on terms' location and frequency.

To evaluate queries with Optane PM, we eliminate the file I/O operations and instead map the postings file upfront to read both the term and postings. The baseline evaluator aggressively prefetches in DRAM the index pages required for serving a query. We disable this feature for query evaluations with PM-backed indices.

5 Experimental Methodology

We discuss our evaluation platform, datasets, memory and storage configurations, and measurement methodology. We also discuss how we form query evaluation workloads.

5.1 Platform

Our server is a dual-socket Dell PowerEdge R740 running the Ubuntu 18.04.1 Linux OS (5.4.0 kernel). Each processor is an Intel Xeon Gold 6252N operating at 2.3 GHz with 48 physical cores (96 logical) and a 36 MB shared Level-3 cache. Each host iMC supports six memory channels. Each memory channel is attached to a 32 GB Micron DDR4 DIMM and a 128 GB Intel Optane DIMM. The system thus has approximately 400 GB of DRAM and 1.5 TB of Optane PM. The system has a 1.5 TB Intel Optane PCI Express NVMe SSD (DC P4800X). Interestingly, both the SSD and NVDIMMs use 3D XPoint as a storage medium but sit behind different interfaces. The system also has a 1 TB, 3.5-Inch, Seagate, SATA (6 Gbps), hard drive, capable of 7200 rotations per minute.

5.2 Datasets and Ingestion

We download and use Wikipedia's English corpus from the Luceneutil website [37]. They provide a line-terminated (1 KB each) file of the corpus. We extract lines to individual files, which Psearchy reads. Indexing the entire corpus is tedious. To evaluate the rich space of memory and storage configurations, we construct a small and a large dataset: (1) 2 GB volume with 500,000 files (small), and (2) 20 GB volume with five million files (large). For the large dataset, the posting lists in the resulting index consume approximately 12 GB, the document map takes up 310 MB, and the dictionary is 170 MB in size. The document map stores file names, which takes up more space than dictionary terms.

We store the corpus in DRAM on the tmpfs filesystem. Using tmpfs ensures reading the corpus is not the bottleneck. Removing this bottleneck is good practice for evaluating search engines [36]. Also, new content resides in DRAM in contemporary real-time search [45, 59].

5.3 Configurations

We evaluate six configurations on our multicore server with hybrid DRAM and Optane PM and a disk and SSD. Table 1 lists the configurations and where they place the hash table, partial postings, and the (final) inverted index. The last column shows the role of Optane PM in each configuration. First, stock runs the unmodified indexer with a DRAM-backed hash table, SSD-backed postings, and inverted index. Unless otherwise stated, we use a per-core hash table of 1 GB.

The remaining five configurations use Optane PM. Two configurations, namely table-pm and pm-only allocate the table in Optane PM. Furthermore, pm-only uses Optane PM for storing the partial postings and the index. As the name suggests, it evaluates a system with Optane PM only in a universal role. Flush-pm uses Optane PM only for storing partial postings in a DRAM extension. Hybrid and hybrid+ place the table in DRAM and store the postings (partial and merged) in Optane PM. (They are named to reflect PM's dual use as extension and storage.) Hybrid stores the dictionary in Optane PM, and hybrid+ stores it in an SSD-backed BDB dictionary. Besides, we evaluate cc-hybrid, a crash-consistent variant of hybrid.

Table 1. Our evaluated configurations.

Name of	Placement of **Table**, **Post**ings, and **Dict**ionary				Role of
Configuration	Table	Partial Pt	Merged Pt	Dict	Optane PM
stock	DRAM	SSD	SSD	SSD	*none*
table-pm	PM	SSD	SSD	SSD	*main memory*
pm-only	PM	PM	PM	PM	*universal*
flush-pm	DRAM	PM	SSD	SSD	*extension*
hybrid	DRAM	PM	PM	PM	*ext + storage*
hybrid+	DRAM	PM	PM	SSD	*ext + storage*

5.4 Measurement Methodology

We use execution time to quantify the indexer's performance. We perform each experiment eight times and report the arithmetic mean. Our measurements are statistically sound, and the coefficient of variation is less than 1% across eight runs. For query evaluation, we report the tail latency distribution from executing 10 K concurrent user-facing queries. We perform each experiment multiple times, confirming the statistical soundness of our observations. We clear the page and directory caches before each experiment.

We use best practice guidelines for Optane PM use from recent work [57]. ① We use interleaved PM. ② For indexing, we bind threads, and memory (both volatile and non-volatile) to CPU 1, avoiding non-uniform memory accesses. To expose Optane PM to our search engine, we use the ext4-DAX filesystem [48]. ext4-DAX bypasses the buffer cache and provides direct access to Optane PM. It also disables data journaling and eliminates other metadata writes.

5.5 Query Formation

We use two query types: (1) single-word queries (S1) and (2) multi-word conjunctive queries with two words (M2). Our query evaluation workloads are homogeneous. We do not mix S and M queries. To construct the workload for the evaluator, we pick a random word (S1) or words (M2) from the list of commonly used English words [15]. We validate critical findings for workloads with up to 100 K queries. For ease of experimentation, especially with disk-backed indices, we show results with 10 K query workloads. Multiple queries run concurrently. A single query is resolved sequentially.

Our query evaluator obtains the matching documents by traversing the posting lists of each keyword. For M2, the evaluator finds and reports the matching documents by intersecting the posting lists of multiple terms. The evaluator scores each result based on document ranks, the file offset where the term appears, the closeness of terms in the document. Prior literature has more details [9, 44]. We set stopearly to 0, reporting all matches. The average rate of zero matches for S1 and M2 queries are 0% and 1%, respectively.

6 Evaluation Results

This section evaluates hybrid memory and storage across three dimensions: ① Performance and multicore scalability of full-corpus indexing. ② Latency for updating an existing index. ③ Tail latency distribution of two query workloads. Unless otherwise stated, we use the large dataset.

6.1 Full Corpus Indexing

We first discuss the results for indexing an entire corpus.

6.1.1 Single-Threaded Performance. We evaluate the different roles of Optane PM for text indexing. Figure 7 shows the indexing time with one core normalized to pm-only. Our two datasets reveal similar findings. We observe that both table-pm and pm-only are 30% slower than stock. This slowdown is because Optane PM has higher latency and lower bandwidth than DRAM. Compared to direct PM accesses in table-pm and pm-only, the baseline engine uses the OS buffer cache. With current technology, the main and the universal roles of Optane PM do not perform well. It is critical to place data structures in hybrid memory and storage carefully.

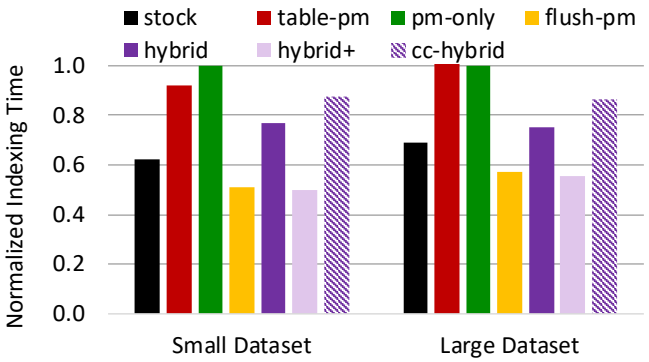


Figure 7. Single-threaded indexing time for all configurations normalized to pm-only.

Flushing the partial postings to Optane PM, as flush-pm does, results in a speed-up of 43% over pm-only and 17% over stock. However, it consumes DRAM, Optane PM, and SSD capacity. Although inefficient, spreading the memory and I/O activity across three devices results in a performance gain. Unfortunately, a hybrid DRAM and Optane PM approach,

namely hybrid performs, 9% worst than stock. We expect better performance with a hybrid approach. We find that replacing the PM-backed dictionary in hybrid with an SSD-backed BDB dictionary (hybrid+) results in a 20% speed-up compared to stock. (Section 6.1.4 investigates the performance pathologies in PM-aware dictionaries.) Overall, for single-threaded indexing, the best-performing configurations, flush-pm and hybrid+, both use Optane PM.

Besides performance, PM offers instant-on preservation of the state and crash consistency. The final configuration in Figure 7, namely cc-hybrid, offers crash-consistent index updates, but at a cost, a 15% slowdown over hybrid, 56% over hybrid+, and 25% over stock.

We next tease apart the single-threaded indexing time into various components and then discuss multicore scalability.

6.1.2 Breaking Down Indexing Time. We break down the single-threaded indexing time into different components to better understand the impact of Optane PM. Figure 8 shows the five components of indexing time, with each component normalized to the total indexing time with pm-only. From the bottom, DocId takes up 4% of the indexing time with stock. For configurations with a PM-aware dictionary, e.g., pm-only and hybrid, this time increase 5×. Also, there is a 44% increase in Hash in table-pm and pm-only due to high PM latency. For the same two configurations, Sort increases 35% due again to the PM-backed table. Flush is highest in table-pm because the indexer flushes a PM-backed table to an SSD. Compared to table-pm, flushing the table is 2× faster with pm-only. Flushing in pm-only is essentially an in-persistent-memory copy. Flush is lowest in hybrid and hybrid+ (15% of indexing time) because of fast DRAM-PM copies through pointer manipulations. With cc-hybrid, flushing takes 23% longer due to persist-ordering instructions. Next, Merge is lowest in hybrid+, consuming 7% of the indexing time. Merging is also fast in flush-pm in which the indexer reads PM-backed partial postings and writes them to an SSD-backed binary file. Merging is 2× slower in pm-only and hybrid, mainly due to a PM-aware dictionary. Merging takes the longest in cc-hybrid due to crash-consistent index updates. Overall, faster single-threaded indexing with PM is due to accelerated flushing and merging of partial postings.

6.1.3 Multithreaded Scalability. We now evaluate the multicore scalability of full-corpus indexing. Figure 9 shows the indexing time with increasing core count normalized to one core. Unfortunately, beyond eight cores, table-pm and pm-only scale poorly. The execution time with pm-only from eight cores to 32 cores increases by 1.8×. Therefore, Optane PM, as the main or universal memory, does not gracefully handle concurrency. The three hybrid configurations scale up to 16 cores, reducing indexing time by 4×. From 16 to 32 cores, they do not further reduce the indexing time. The reason for their poor scalability is the limited bandwidth of Optane PM. Specifically, for hybrid+ with 32 cores, there is a 1.7× increase

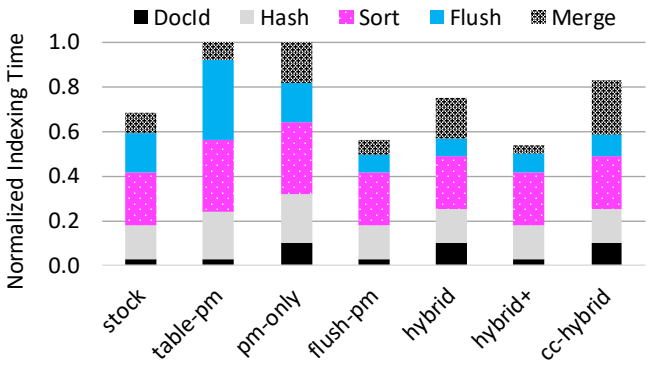


Figure 8. Breaking down single-threaded indexing time (normalized to pm-only) into various components.

in Flush, and 8× increase in Merge, compared to stock. We analyze this result further in Section 6.1.6 with hardware performance counters. The most scalable configurations are stock and flush-pm. Between one and 32 cores, flush-pm reduces the indexing time by 9.3×. Stock scales better with increasing core count, speeding up indexing by 11.5× with 32 cores than a single core.

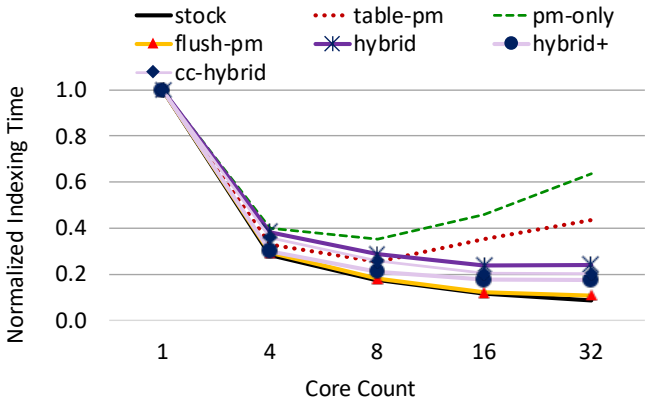


Figure 9. Scalability of indexing with increasing core count for selected configurations. The Y-axis is normalized to the indexing time with a single core.

Figure 10 shows the indexing time with different core count normalized to pm-only. We observe that flush-pm is slower (3%) than hybrid+ with one thread, but it scales better than hybrid+. With 32 cores, flush-pm is 38% faster than hybrid+. Also, with increasing core count, table-pm performs better than pm-only. This better scaling is due to a reduction in flushing and merging time compared to pm-only. We also observe that the performance gap between hybrid and cc-hybrid bridges with increasing core count. This result is surprising because cc-hybrid executes clflush and sfence instructions regardless of core count. (See Section 6.1.5 for a deeper analysis). Finally, we observe that as core count grows, stock performs better than hybrid+, eliminating the single-core advantage of hybrid+ over stock.

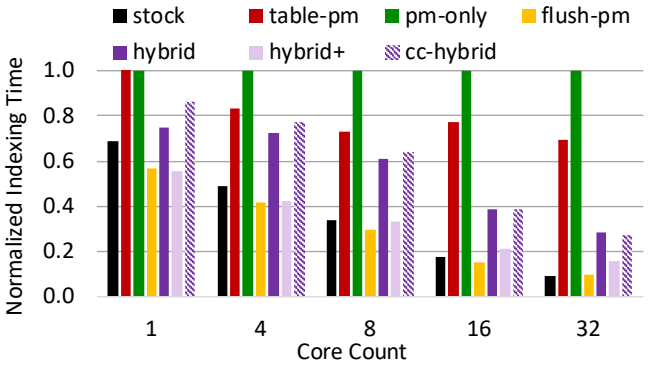


Figure 10. Normalized indexing time with increasing core count (n). The Y-axis is normalized to n-core pm-only.

PM's capacity advantage. We evaluate PM's capacity advantage as the main memory. We increase the hash table's capacity in table-pm up to 16× compared to stock. We use eight cores, beyond which table-pm does not scale. The slowdown with table-pm is 2× with a similarly-sized hash table. With a 16× advantage, table-pm is still 1.6× slower than stock. PM's capacity advantage cannot make up for its high latency and low bandwidth relative to DRAM.

6.1.4 Performance of PM-aware Stores. We now investigate the performance of PM-aware key-value storage engines for dictionary storage. Figure 11 shows the change in execution time with three pmemkv storage engines. We normalize to execution with hybrid+, that uses a BDB dictionary, at the same core count. The concurrent hash map (CMap) engine has a high overhead, especially with increasing core count. It increases indexing time by up to 84%.

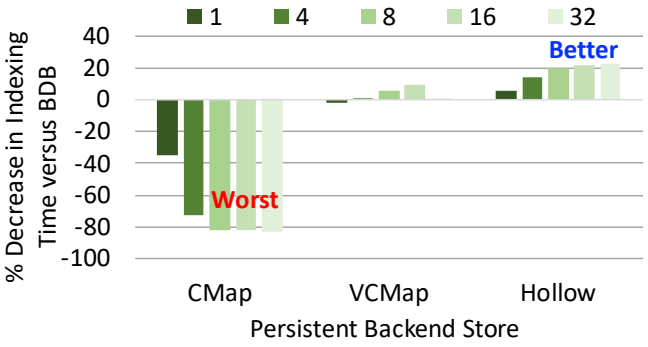
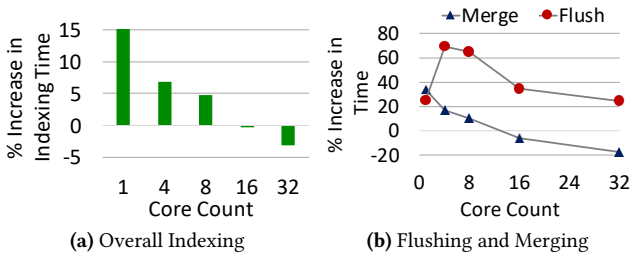


Figure 11. Showing the change in indexing time with three pmemkv storage engines across different core count.

The CMap engine guarantees crash-consistent dictionary updates. To relax the consistency requirement, we also evaluate the volatile concurrent hash map or VCMap. We find that, for multicore indexing, VCMap reduces indexing time over CMap (hybrid+). Each indexing thread has a private dictionary, and multicore execution distributes the dictionary workload across cores. One caveat in our VCMap evaluation is that we retain the term dictionary after indexing. The original VCMap requires closing and deleting the database. Surprisingly, closing the database increases the indexing time by up to 2×, and the DocId time 7×. We also evaluate Hollow in Figure 11. This engine measures only the interfacing (binding) overhead as internally, it drops information. Overall, depending on the engine, the indexing time with 32 cores is between 84% worst to 23%, better than hybrid.

6.1.5 Overhead of Crash Consistency. We investigate in detail the impact of crash-consistent indexing. Figure 12 (a) shows the increase in the indexing time with cc-hybrid compared to hybrid. The 15% increase (one core) is due to persist-ordering instructions that the indexer execute while flushing and merging partial postings. This overhead decreases with increasing core count, as the workload is parallelized across cores. Surprisingly, the performance of crash-consistent indexing improves by 3% with 32 cores. We suspect that persisting a cache line invalidates it and prevents future writeback. Later in the execution, the cache eviction policy prioritizes such (dead) lines for a new allocation, leaving other more useful lines in the cache. We analyze this result further in Section 6.1.6 with hardware performance counters.



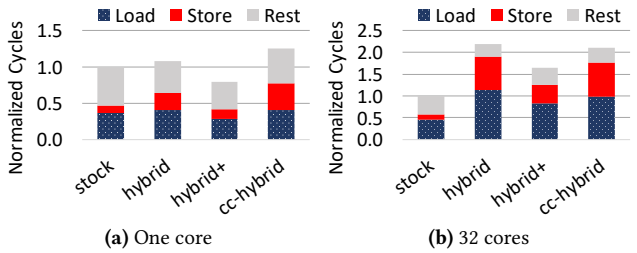
(a) Overall Indexing **(b)** Flushing and Merging

Figure 12. Performance impact of crash-consistent indexing with increasing core count: (a) Increase in total indexing time, (b) Increase in flushing and merging time.

We isolate the change in flushing and merging time with cc-hybrid compared to hybrid (similar core count) in Figure 12 (b). Compared to hybrid, the flushing time increases from 25% to 70%. The increase is highest for four cores. With more cores, the per-core flushing workload is reduced, reducing the overhead of pmem_persist calls. Merging (single-core) is 34% slower than hybrid. With 16 and 32 cores, however, crash-consistent merging is faster than hybrid.

6.1.6 Microarchitectural Analysis. We use Linux's perf utility to analyze full-corpus indexing from the processor microarchitecture perspective. We configure perf to obtain the aggregated cycles across all cores. We further break down cycles into three components: (1) all cycles during which the core is stalled and waiting for load requests to resolve (Load), (2) the core is stalled due to a fully occupied store buffer (Store), and (3) the remaining cycles (Rest).

We show in Figure 13 the breakdown of total cycles for stock and three hybrid memory configurations with one core and 32 cores. First, we observe that, with one core, compared to stock, hybrid exhibits 12% more Load stalls, and 2.25× more Store stalls. These stalls are due to the PM-aware dictionary. On the other hand, Hybrid+ exhibits fewer Load stalls over stock, but still incurs 25% greater Store stalls. Interestingly, we observe that Rest is smaller in both hybrid configurations compared to stock. Rest is smaller because PM-based configurations use memory-mapped files instead of explicit I/O. Indeed, we find that faster indexing with PM in hybrid+ (single-core) is mainly due to avoiding explicit I/O. As expected, we observe a significant (3.5×) increase in Store stalls with cc-hybrid compared to hybrid.



(a) One core **(b)** 32 cores

Figure 13. Breakdown of total cycles into memory-related stalls, normalized to the total cycles with stock.

With 32 cores, hybrid+ observes a significant increase in Load and Store stalls. This increase is due to the limited bandwidth of Optane PM. We observe a much more significant increase in Store stalls than Load ones. Figure 13 (b) also provides an insight into the reduction in indexing time with cc-hybrid at high core count compared to hybrid. As hypothesized in Section 6.1.5, the reason for the reduction in indexing time for cc-hybrid is 15% fewer Load stalls. Invalidating cache lines where newly written postings reside informs the LLC controller to reuse these (dead) lines instead of evicting other useful lines.

6.2 Index Updating

Rapidly updating an existing index is critical, especially in real-time search [45]. We evaluate the latency of updating an SSD-backed index with stock and PM-backed index with pm-only and variants of hybrid. Figure 14 shows the update time in seconds for adding between one to 10 K documents with a single thread. We use the small dataset's index, but our conclusions are unchanged for the large index.

Updating an SSD-backed index (stock) takes 60 seconds. A PM-backed index reduces this high latency by 2×. This result is consistent with faster single-threaded indexing with Optane PM. Adding a few documents to an index does not stress the temporary data structures, including the hash table and partial postings. Thus, the DRAM-PM latency gap is less

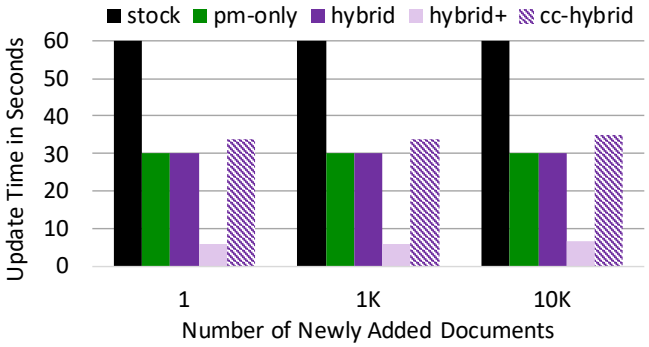


Figure 14. Showing the index update time in seconds with an SSD and a PM-backed index.

critical. All the time in updating is due to merging, which is storage I/O intensive. Thus, it happens faster on Optane PM than on the SSD. The PM latency is lower than SSD, and manipulating the memory-mapped PM-backed index is more efficient than filesystem I/O with an SSD-backed index.

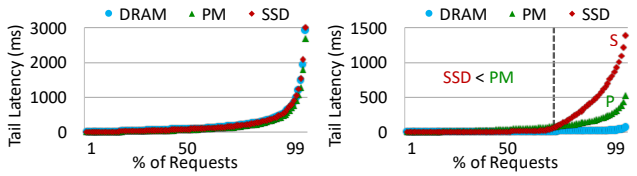
An update latency of 30 seconds is still high [45]. Hybrid+ reduces the update latency by a further 5× to only six seconds. It achieves this low latency by replacing the CMap dictionary in hybrid with a BDB dictionary. A crash-consistent PM-backed index update (cc-hybrid) takes 35 seconds, which is 16% slower than hybrid, and 71% faster than stock. Except hybrid+ reaching up to 6.6 seconds, update latencies stay stable with increasing document count.

6.3 Query Evaluation

We now discuss the performance of query evaluation. We show the distribution of tail latency across two query workloads. In addition to SSD and PM-backed indices, we evaluate queries from a DRAM-backed index. We use the tmpfs filesystem for emulating a DRAM-backed index. We always use a BDB dictionary and place it in DRAM.

The first query workload dispatches S1 queries with a single term. Analysts report that S1 queries constitute 21% of all Google queries [8]. Figure 15 (a) shows the tail latency distribution for such queries. We observe a similar tail latency distribution with SSD, PM, and DRAM-backed indices. At first, this outcome surprised us. However, we find that on a dictionary match, an S1 query generates sequential read accesses because the postings for the matching term are stored contiguously. The sequential read latency of both the SSD and the PM is much lower than random latency [50, 57]. The 99th percentile latency is three seconds. Our modified query evaluator serves 90% of the queries in under 500 ms across the three devices and a half in under 80 ms. These findings reveal a rich choice for optimizing the placement of inverted indices in data centers.

We show the 99th tail latency for the shortest 50% and the longest 50% queries in Table 2. As a reference, we also report tail latencies with a disk-based index. The SSD is slightly



(a) S1 **(b)** M2

Figure 15. Distribution of tail latency for S1 and M2 query workloads with a DRAM and an PM-backed index.

slower (10%) than PM in serving queries. The tail latency with a disk-backed index violates the real-time response time constraint. In our setup, a PM-backed index resolves queries 11% faster than a DRAM-backed index. We suspect this is due to the use of tmpfs in the DRAM evaluation. This filesystem is not optimized to handle large amounts of concurrency.

Table 2. 99th percentile tail latency (milliseconds) of shortest and longest 50% of requests.

Posting lists in	S1		M2	
	Short	*Long*	*Short*	*Long*
DRAM	90	2900	5.0	70
Optane	80	2700	40	500
SSD	90	3000	9.0	1400
Disk	20000	80000	60000	140000

The M2 queries result in a different trend. In Figure 15 (b), we divide their tail latency distribution into two regions. In the first region, the SSD is faster in serving the M2 queries. Its 99th percentile tail latency for short queries is 9 ms. Optane PM is 4.4× slower in serving the 50% shortest queries. This region continues for 70% of the queries on the horizontal axis. At that point, a different trend emerges. Optane PM now responds faster than the SSD. Initially by a small percentage margin, and ultimately, for the longest 50% of the queries, its 99th percentile tail latency is 2.8× lower.

We investigate the two regions in Figure 15 (b). First, the M2 query workload generates random read requests for the SSD and PM controllers. The read access pattern is random because the M2 (conjunctive) query evaluator first obtains the posting lists for each of the two terms. Each posting list is sorted from the rarest occurrence of a term to the more common. Then, the query evaluator advances the lists, in turn, switching between them until it finds all documents that contain both terms. Advancing two or more lists results in random read accesses.

Optane PM does not handle large amounts of concurrent random requests gracefully [57]. Thus, many queries in the left region in Figure 15 (b) suffer a high latency with a PM-backed index. The SSD-backed index partially resides in the OS (DRAM) cache. The SSD-bound queries that find postings in DRAM resolve faster. (To observe DRAM reads in all experiments, we use Intel's pcm-memory.x utility [27].) The queries that access the SSD media traverse the PCIe interconnect and resolve slower than a PM-backed index.

Finally, DRAM is the fastest to serve M2 queries. Its tail latency for the M2 query workloads is 8× and 20× lower than PM and SSD, respectively. DRAM's random access latency is lower than Optane PM and SSD. Overall, our analysis opens up new tradeoffs in placing indices in data center servers.

7 Related Work

Intel Optane DIMMs appeared two years ago. Few efforts characterize their main memory and storage roles. Yang et al. compare Optane PM's latency and bandwidth to DRAM [57]. They also characterize its persistent nature for key-value stores. Jian et al. report the performance of in-memory and embedded database workloads with Optane PM [54]. Intel Optane SSDs and DIMMs use the same 3D XPoint media. Recent work evaluates Optane SSDs for data-intensive workloads [28, 51, 58]. Our work is the first to evaluate Optane PM and SSDs for full-text search.

On the programming side, Memaripour et al. introduce Pronto, a library for transforming volatile data structures into non-volatile ones [40]. They use real persistent memory. A flurry of prior work uses emulation to propose and evaluate persistent memory libraries, filesystems, and programming models [11, 12, 14, 24, 46, 53, 55, 56, 61]. Early efforts focus on native support for persistent memory requiring manual intervention for identifying persistent data. Recently, researchers have explored managed language support for automatically persisting data [43, 52].

Prior literature discusses in detail various techniques for constructing an inverted index [62]. Real-time search is gaining attention, and especially relevant is the approach at Twitter [32]. Other works improve search latency with better index managers, specialized memory allocators, and faster query evaluators [3–5, 19, 26, 33–35, 39, 41, 47]. Our special focus in this paper is on emerging non-volatile memory.

8 Conclusion

We have now evaluated Intel Optane DC persistent memory (PM) for text indexing and search. Optane PM delivers mixed results against a traditional storage stack. At low core count, it reduces the time it takes to build or update an index. Its tail latency is lower for long-running conjunctive queries. Unfortunately, PM-backed indexing does not scale to a high core count due to limited bandwidth. Also, conjunctive queries suffer longer tail latencies for the shortest queries. Some performance pathologies exist in the PM software stack. The overhead of crash-consistent indexing is low at a high core count. Hardware and software for persistent memory must continue to evolve to realize its full potential in the future.

References

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

[2] S. Akram, M. Marazakis, and A. Bilas. 2012. Understanding Scalability and Performance Requirements of I/O-Intensive Applications on Future Multicore Servers. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. https://doi.org/10.1109/MASCOTS.2012.29

[3] N. Asadi and J. Lin. 2013. Fast candidate generation for real-time tweet search with bloom filter chains. *ACM Trans. Inf. Syst.* 31 (2013), 13. https://doi.org/10.1145/2493175.2493178

[4] N. Asadi, J. Lin, and M. Busch. 2013. Dynamic memory allocation policies for postings in real-time Twitter search. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mmining*. https://doi.org/10.1145/2487575.2488221

[5] Hannah Bast and Björn Buchhold. 2013. An Index for Efficient Semantic Full-Text Search. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. https://doi.org/10.1145/2505515.2505689

[6] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2004.1317436

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

[8] Conor Bond. 2020. 27 Google Search Statistics You Should Know in 2019. https://www.wordstream.com/blog/ws/2019/02/07/google-search-statistics

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores

[10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2021. MOSBENCH. https://pdos.csail.mit.edu/archive/mosbench/

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 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)*. https://doi.org/10.1145/1961295.1950380

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

[13] Intel Corporation. 2021. pmem.io: Persistent Memory Programming. https://pmem.io/pmdk/

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/2592798.2592814

[15] EF. 2021. 3000 most common words in English. https://www.ef-australia.com.au/english-resources/english-vocabulary/top-3000-words/

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

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

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

[19] L. Gao, Y. Wang, Dong sheng Li, Junming Shao, and Jingkuan Song. 2017. Real-time social media retrieval with spatial, temporal and social constraints. *Neurocomputing* 253 (2017), 77–88. https://doi.org/10.1016/j.neucom.2016.11.078

[20] Radu Gheorghe, Matthew Lee Hinman, and Roy Russo. 2015. *Elasticsearch in Action*.

[21] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. 1998. CiteSeer: An Automatic Citation Indexing System. In *Proceedings of the Third ACM Conference on Digital Libraries (DL)*. https://doi.org/10.1145/276675.276685

[22] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2775054.2694384

[23] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3123939.3123956

[24] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378472

[25] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Priented Applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1028976.1028998

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

[27] Intel. 2021. Processor Counter Monitor (PCM). https://github.com/opcm/pcm

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

[29] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783761

[30] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010). https://doi.org/10.1109/MM.2010.24

[31] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David R. Karger, and Robert Morris. 2003. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *Peer-to-Peer Systems II*. https://doi.org/10.1145/2808194.2809489

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

[33] L. Lin, Xiaohui Yu, and N. Koudas. 2013. Pollux: towards scalable distributed real-time search on microblogs. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.1145/2452376.2452416

[34] A. Magdy, M. Mokbel, S. Elnikety, S. Nath, and Yuxiong He. 2014. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. *IEEE 30th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/ICDE.2014.6816649

[35] Giorgos Margaritis and S. Anastasiadis. 2014. Incremental Text Indexing for Fast Disk-Based Search. *ACM Trans. Web* 8 (2014), 16:1–16:31. https://doi.org/10.1145/2560800

[36] Michael McCandless. 2021. Apache Lucene performance on 128-core AMD Ryzen Threadripper 3990X. https://www.ef-australia.com.au/english-resources/english-vocabulary/top-3000-words/

[37] Michael McCandless. 2021. Luceneutil: Lucene benchmarking utilities. http://blog.mikemccandless.com

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

[39] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. 2001. Building a Distributed Full-Text Index for the Web. *ACM Trans. Inf. Syst.* 19, 3 (2001), 217–241. https://doi.org/10.1145/502115.502116

[40] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378456

[41] G. Mishne, Jeffrey Dalton, Zhenghua Li, A. Sharma, and J. Lin. 2013. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. *ArXiv* abs/1210.7350 (2013).

[42] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3037697.3037730

[43] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314608

[44] Jeremy Stribling, Jinyang Li, Isaac G. Councill, M. Frans Kaashoek, and Robert Morris. 2006. OverCite: A Distributed, Cooperative Citeseer. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

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

[46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1961296.1950379

[47] Y. Wang and J. Lin. 2015. The Feasibility of Brute Force Scans for Real-Time Tweet Search. *Proceedings of the 2015 International Conference on The Theory of Information Retrieval* (2015). https://doi.org/10.1145/2808194.2809489

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

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

[50] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[51] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. 2019. Exploiting Intel Optane SSD for Microsoft SQL Server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. https://doi.org/10.1145/3329785.3329916

[52] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. 2020. GCPersist: An Efficient GC-Assisted Lazy Persistency Framework for Resilient Java Applications on NVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3381052.3381318

[53] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3173162.3173201

[54] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3297858.3304077

[55] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[56] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132761

[57] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang

[58] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5, 1, Article 4 (Feb. 2020). https://doi.org/10.1145/3372783

[59] Xi Yang. 2021. Private communication.

[60] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. 2016. Mellow Writes: Extending Lifetime in Resistive Memories Through Selective Slow Write Backs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3007787.3001192

[61] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2694344.2694370

[62] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. 38, 2 (2006). https://doi.org/10.1145/1132956.1132959