The Australian National University
2600 ACT | Canberra | Australia

Australian
National
University

School of Computing

College of Engineering, Computing
and Cybernetics (CECC)

APCache: An Adaptive Postings Cache in Heterogeneous Memory for Storage-Resident Search Indices

— Honours project (S2/S1 2023–2024)

A thesis submitted for the degree
Bachelor of Advanced Computing

By:
Anson Thai

Supervisor:
Dr. Shoaib Akram

January 2025

Declaration:

I declare that this work:

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

January, Anson Thai

Acknowledgements

Firstly, I would like to extend my heartfelt gratitude to my supervisor, Shoaib Akram, whose unwavering support and guidance have been invaluable throughout this journey. Shoaib's insightful discussions during our weekly meetings and his mentorship have been instrumental in shaping this project. Beyond guiding my research, he has taught me resilience and fostered skills that will undoubtedly benefit me in the future. I also thoroughly enjoyed his entertaining stories, which made our meetings both educational and enjoyable. I deeply appreciate his dedication, expertise, and commitment to the highest standards, without which this project would not have been possible.

I am also grateful to my parents for their unwavering support of my undergraduate studies and their emotional encouragement during times of stress. Their belief in me has been a constant source of strength.

Lastly, I would like to thank my friends for their consistent encouragement and unwavering belief in me. Their support has been a pillar of strength throughout this journey.

Abstract

Search engines use inverted indices to speed up query resolution. Unfortunately, indices reside on disk drives with high storage capacity but are extremely slow compared to main (DRAM) memory. Therefore, resolving full-text search queries over emerging big datasets promptly demands a large DRAM capacity for hosting inverted indices. On the other hand, preserving DRAM requires compressed indices, incurring decompression latency during query evaluation. Beyond DRAM, emerging non-volatile memory (NVM), disaggregated remote memory over fast ethernet and Infiniband, and fast PCIe NVMe SSDs promise cost-efficient (memory) capacity expansion.

We rigorously investigate space-time tradeoffs in hosting compressed/uncompressed indices in scalable and byte-addressable non-volatile memory (NVM), which offers better density but incurs high access latency. Counter-intuitively, we find NVM an attractive medium to host indices in compressed format. Specifically, state-of-the-art list compression algorithms hide NVM latency by dramatically reducing the data moved between CPU and memory (up to 5.75×). Also, queries expose spatial locality as document identifiers and term frequencies reside sequentially, which on-chip cache prefetchers exploit. Finally, SIMD accelerates decompression (up to 30.7%), bridging the response time gap between uncompressed DRAM indices (best speed) and compressed PM indices (most scalable). We substantiate our findings with a detailed top-down microarchitectural analysis. Our key findings from this detailed microarchitectural analysis include: (1) fast SIMD compression reduces the storage capacity required for indexing and enhances QPS performance on emerging memory technologies, offering a solution to alleviate DRAM pressure, (2) we demonstrate that using TurboPFor compression and its variants, the decompression overhead with NVM-backed indices results in only a 4% degradation in QPS compared to DRAM-based index, while reducing the index size by 82.7%.

Informed by our thorough analysis, we build APCache, a heterogeneous DRAM-NVM postings cache for search engines that minimizes DRAM utilization for storing disk-resident and persistent inverted indices in fast (hybrid) memory. It relies on memory-mapped I/O and operates over an unmodified storage stack, copying term postings from the OS page cache to a custom NVM cache. It requires a constant amount of DRAM to aid disk I/O, whereas the baseline system, which consists of disk and DRAM, demands

DRAM proportional to index size. APCache prefetches postings ahead of the main query thread into NVM. It employs query workload-aware prefetching, relying on the fact that today's systems buffer client requests as the client request rate is higher than the request processing rate. APCache uses an adaptive lookahead policy to prevent a high cache eviction rate.

Our detailed evaluation demonstrates that APCache achieves identical performance to the baseline system which utilizes the OS page cache. It saves DRAM capacity up to 6 GB for a 20 GB index as the baseline system (state of the art) requires OS cache proportional to index size, whereas APCache requires a fixed 50 MB OS cache for I/O. Our newly proposed system achieves higher QPS with larger cache sizes, making PM more advantageous due to its ability to offer larger capacities. Our adaptive prefetching method ensures that tail latency remains highly comparable to a system with a magnitude more DRAM capacity. APCache applies to other emerging memory technologies, such as remote CXL memory, and offers the potential to conserve costly DRAM capacity as datasets expand.

Table of Contents

1 Introduction **1**

1.1 Thesis Roadmap 4

2 Background and Motivation **5**

2.1 Indexing and Query Evaluation 5

2.2 Hybrid Memory 6

2.3 SIMD 6

2.4 Compression 7

2.4.1 Differential Encoding 7

2.4.2 VByte 7

2.4.3 StreamVByte 7

2.4.4 TurboPFor 8

2.5 BM25 Scoring 10

2.6 Control Group v2 11

3 Related Work **13**

3.1 Indices in Hybrid Memory and Fast Storage 13

3.2 Other Data-Intensive Applications over Hybrid Memory 14

3.3 Datacenter Caches 15

3.4 Persistent Memory Characterization 15

4 Index Organization and Query Evaluation **17**

4.1 Indexing 17

4.2 Query Evaluation 19

5 Evaluation Methodology **21**

5.1 Microarchitectural Analysis Methodology 21

5.1.1 Index Configurations 21

5.1.2 Intel Top-down Microarchitectural Analysis 22

5.2 APCache Evaluation Methodology 23

5.2.1 Server Platform 23

5.2.2 Datasets and Ingestion 24

5.2.3 Query Formation 24
5.2.4 Measurement Metrics and Methodology 26

6 Microarchitectural Analysis **27**
6.1 Indexing Performance 27
6.2 Index Size 28
6.3 Query Evaluation 29
6.3.1 Roadmap of Results 29
6.3.2 Analysis 30
6.3.3 Tail latency Results 33
6.4 Wikipedia Dataset 34

7 Design and Implementation of APCache **47**
7.1 Principles 47
7.2 Architecture of APCache 48
7.2.1 Baseline System 48
7.2.2 Optimizing the Baseline System 49
7.2.3 Hybrid APCache 51
7.2.4 Optimizing the Hybrid APCache 54

8 Evaluation of APCache **63**
8.1 Configurations and Parameters 63
8.1.1 Configurations 63
8.1.2 Parameters 64
8.1.3 Skip List and Dictionary Overhead 64
8.1.4 Skip List and Dictionary Location 64
8.1.5 Cache Segment Size 66
8.1.6 Prefetch Lookahead 66
8.1.7 Memory Limit for cache 67
8.2 Query Evaluation Results 69
8.2.1 Query Workloads 69
8.2.2 QPS Findings 70
8.2.3 Muticore Scalability 80
8.3 Key Takeaways 83

9 Concluding Remarks **85**
9.1 Conclusion 85
9.2 Future Work 85
9.2.1 Considering Linux Asynchronous I/O 86
9.2.2 Other Storage Technologies 86
9.2.3 Real-Time Search Engines 86
9.2.4 Slow Memory as Page Cache 86
9.2.5 JVM-Based Search Engines 87
9.2.6 Customized DRAM Cache 87

9.2.7 Eviction Algorithms 87
9.2.8 Generalizing to Other Data-Intensive Applications 88

Bibliography **89**

Chapter 1

Introduction

Search engines serve an increasingly important role in modern society. Enabling fast response times in full-text search is critical to the success of many enterprises, including Linkedin, Meta, and Twitter (Auradkar et al., 2012; Lin et al., 2012). The next frontier in text search is generative real-time AI engines and vector search. These advances will pressure search engines' memory and storage infrastructures already feeling pressure from static content on the World Wide Web. In this thesis, we aim to limit the growth (and cost) of a key hardware component in search engines, namely, DRAM main memory. DRAM is used today to host indices in memory, but we believe its growth proportional to datasets is untenable.

The critical data structure search engines use for locating documents (web pages, social media posts, or tweets) matching a word (term) is an inverted index (Zobel and Moffat, 2006). Recent work shows that fast and emerging SSDs with byte-addressable memory cannot deliver real-time response times for search (Akram, 2021a). Therefore, service providers host indices in DRAM for better service delivery (Gupta et al., 2023; Xie et al., 2018). Unfortunately, indices grow proportional to datasets, and large indices put pressure on DRAM. However, DRAM scaling cannot cope with the growth in datasets (Mutlu and Subramanian, 2014; Handy, 2018; Gupta et al., 2023; Berg et al., 2020a; McAllister et al., 2021; Chen et al., 2020; Abulila et al., 2019; Huang et al., 2015; Weiner et al., 2022), and increasing infrastructure cost (Weiner et al., 2022; Chilukuri and Akram, 2023a). The scaling issues of DRAM have been known for a while, and they continue to get worse every year.

Key Idea and Thesis: The thesis of this research is that we can overcome DRAM scaling issues by utilizing in-memory vector-powered compression coupled with emerging, cheap, and dense memory technologies to *supplement* DRAM. SIMD mitigates decompression latency while saving tremendous capacity. The resulting multi-tiered byte-addressable memory system can host large inverted indices that outgrow DRAM main memory. We can use this multi-tiered memory as a cache for a disk-resident persistent index. The key idea is to use a fixed-size DRAM cache only to facilitate I/O transfers between disk and I/O cache, limiting DRAM growth with the index size on disk. Then, we hypothesize that using SIMD-powered decompressed index on a cheaper memory technology will provide the capacity and query evaluation speed that matches a baseline system with DRAM only. The OS stack remains unmodified for maximum deployment potential.

It is not simple to build a system and demonstrate good performance in terms of query evaluation throughput and tail latency, capacity savings, and how one can leave the OS stack unmodified while still transferring blocks of inverted index from the fast tier (DRAM) to the slow tier (second memory technology). In the rest of the thesis, we show how one can build such a system at the application level and show optimizations that lead to better performance than a naive system.

The road to scaling in-memory indices to large datasets begins with compression. It especially applies to inverted indices as they consist of integer document identifiers (DIDs). State-of-the-art industrial algorithms can compress inverted indices to up to 17.3% of their original size (we verify). However, compression exposes a well-known space-time tradeoff. Despite its memory efficiency advantage, compression slows down query evaluation due to decompression cost. One possibility to avoid decompression costs is to use emerging technologies that promise high density and expand memory capacity. These include ① non-volatile memory (NVM), ② disaggregated memory, and ③ NVMe SSDs. Collectively, we call these technologies expanded physical memory. Unfortunately, these emerging alternatives are slower than DRAM, exposing high access latency to the processor. It is non-intuitive if it is better to use DRAM-backed compressed indices (capacity limited, but backed by faster memory technology) or emerging memory-backed uncompressed indices. Indeed, we believe the most scalable approach is to host compressed indices in hybrid memory if the resulting query throughput and tail latency are within acceptable margins. As a first contribution in this thesis, we show, with rigorous microarchitectural analysis, that emerging alternatives are suitable for hosting inverted indices in compressed form due to the nature of search queries and the resulting memory access patterns they generate.

In this work, we investigate these tradeoffs and bring surprising results. We use non-volatile memory (NVM) or persistent memory (PM) as our emerging memory technology, as NVM is the fastest DRAM complement to date. (We use NVM and PM interchangeably in this thesis.) Our work generalizes to other technologies. We hypothesize that NVM-backed uncompressed indices provide comparable or better QPS than DRAM-backed compressed indices. On the contrary, our results indicate that for multi-term conjunctive queries that form the majority of the queries on the World Wide Web, NVM-backed compressed indices are only up to 4% slower than DRAM-backed uncompressed indices (using the most space-efficient compression algorithms), and they result in a competitive QPS to that of DRAM-backed compressed indices. Our detailed analysis shows that inverted indices contain many sorted integers, which modern compression algorithms compress aggressively. Consequently, even for popular queries, compressed indices result in the same QPS with DRAM and NVM-backed indices.

We exploit our empirical observations to inform the design of a system called APCache capable of hosting large inverted indices in scalable memory. APCache does not require any modifications to the operating system kernel. Specifically, during query evaluation, a worker thread prefetches the compressed inverted lists from block storage using the traditional Linux storage I/O path into the page cache. The worker copies the inverted lists from the page cache into a large PM-resident postings cache. A query thread reads the posting lists from the PM cache and decompresses the postings into a DRAM buffer to serve the query requests. This offers two advantages: (1) avoiding expensive trips to block storage and (2) mitigating pressure on the DRAM cache. We exploit our performance analysis to inform decisions, for example, (1) we move compressed (not uncompressed) postings to the PM cache, noticing the small degradation in query performance between compressed/uncompressed PM-resident postings, when using the SIMD-powered compression, (2) we size the page cache just large enough to retain the compressed lists in DRAM while they are being copied into PM.

We organize PM into segments and drop entire segments from PM once the system runs out of PM capacity. We use a PM dictionary to quickly locate whether an inverted list is in PM or disk. We further utilize a PM skip list to accelerate list intersection by avoiding the compression of list elements that are not likely to result in a match. We analyze our system for performance improvement using extensive experimentation.

Our research demonstrates that SIMD compression algorithms, particularly StreamVByte, TurboPFor256, and their delta variants, reduce storage costs while enhancing query evaluation performance. We find that these algorithms deliver similar QPS performance on both DRAM and PM. Our APCache system leverages these findings to build a secondary-storage-resident index with a PM cache. By incorporating an adaptive prefetching policy, we achieve optimal performance on SSDs with a small cache size. For disk indices, prefetching improves performance to match a DRAM cache. Additionally, since QPS increases with cache size, using PM allows for a larger cache and, thus, better QPS, thanks to PM's higher capacity. Finally, using PM enables us to save DRAM capacity.

Our key findings on saving DRAM capacity are as follows.

- By incorporating an adaptive prefetching policy, we achieve optimal performance on SSDs with a small PM cache. The DRAM capacity is limited to 50 MB regardless of the dataset size.

- For disk-resident indices, prefetching with a PM cache (only 50 MB DRAM) improves performance to match that of a DRAM cache (up to 6 GB DRAM or 30% of index size). Since QPS increases with cache size, using PM allows for a larger cache and, thus, better QPS, thanks to higher PM capacity offerings.

The main contribution of this work are as follows:

- Showing through rigorous microarchitectural analysis that fast SIMD compression reduces the index storage capacity needs and enhances QPS performance on emerging memory technologies, promising to mitigate DRAM pressure. Specifically, we show that with TurboPFor and its variants, the decompression cost with PM-backed index results in only 4% QPS degradation than the DRAM-based index while reducing the index size by 82.7%.
- Designing and implementing a system called APCache that offers cost-effective caching using hybrid memory for large disk-resident indices. Our proposed system limits DRAM growth with index size (not possible today) while leaving the OS stack unmodified.
- Proposing and analyzing (adaptive) query workload-aware prefetching for search engines to reduce query search latency. Adaptive prefetching prevents needless cache evictions prevalent in today's systems.

1.1 Thesis Roadmap

This thesis is organized roughly into two parts. We first build a sound baseline system in Chapter 4. We present our newly proposed system, namely APCache in Chapter 7. We present our evaluation methodology before we discuss the design and implementation of APCache in Chapter 5. We present a rigorous evaluation of the baseline system (also before discussion of APCache) to uncover overheads in the state of the art in Chapter 6. We present a detailed evaluation of our newly proposed system in Chapter 8. Thus, our evaluation results are divided across two chapters, and the key new contribution, namely APCache is sandwiched between the two chapters. The first set of evaluation results inform the design and implementation of APCache.

Chapter 2

Background and Motivation

2.1 Indexing and Query Evaluation

The key data structure search engines use for speeding up query evaluation is an inverted index (Zobel and Moffat, 2006). It maps words (terms) to document locations. It consists of posting lists and a dictionary. The posting lists are typically stored in a postings file, and the dictionary is stored in a separate term dictionary file. A posting list contains postings, identifying documents as sorted integer IDs, and meta-data, such as term frequency and position. The term dictionary maps each term to an offset into the postings file where the posting list for that specific term starts.

Typically, the mapping from term to postings list is kept in SSTable-like sorted files merged in the background as needed (Kleppmann, 2017). The basic idea is similar to a log-structured merge (LSM) tree (O'Neil et al., 1996). In an LSM tree, the in-memory table called *memtable* ingests fresh documents, storing a mapping of words (terms) to document IDs in a hash table or sorted data structure, such as a red-black tree or prefix tree. Once the memtable reaches a threshold size, it is serialized (i.e., any virtual references are removed) and persisted to block storage. As the memtable is sorted before being persisted to disk, it is called a sorted string table or SSTable.

Evaluating search queries requires finding document IDs (DIDs) matching query terms. The parsing step identifies query terms, including any conditional operators. Single-term queries are straightforward, while boolean queries use conjunctions (AND) and disjunctions (OR) operators. The next step is the dictionary lookup. The dictionary provides offsets into the postings file. Terms and offsets are stored in the same file sorted alphabetically (Foundation, 2021a). Since the sorting is alphabetical, terms are recursively split into prefixes and suffixes, and common prefixes are only stored once. Typically, the dictionary is a hash table or a key-value store. Once the offsets into the postings file are available, the next step performs set operations (multi-term queries) to find matching IDs. Posting list traversal is a linear function of the posting list size and index size.

The naive approach for evaluating an AND (intersection) query is to scan the posting lists from left to right and match candidate IDs across the lists. On a mismatch, the *naive* algorithm advances one of the lists past the maximum document ID seen so far. Faster approaches use binary or finger search (Zobel and Moffat, 2006). Using skip lists to minimize comparisons across postings lists is also typical. (Our index design uses a skip list in this work.)

2.2 Hybrid Memory

Hybrid memory consists of fast DRAM and high-capacity but slow secondary byte-addressable memory. The secondary tier is either remote (disaggregated (Lim et al., 2009, 2012)) DRAM or another scalable technology (Kim, 2008; Lai, 2008). Although future technologies are possible, we consider NVM the second tier and use Intel Optane DC Persistent Memory (PM) as a real-world NVM prototype. A non-volatile DIMM (NVDIMM) connects to the memory bus, similar to DRAM. Intel Optane DC PM exploits 3D XPoint, which stores information as a change in the material's bulk resistance (Lee et al., 2010). Production NVDIMMs provide 8× more capacity than DRAM. It exhibits 10× lower latency than NVMe SSDs, and 2–3× higher latency than DRAM (Yang et al., 2020a). Communication with Optane DIMM is mediated by the processor's integrated memory controller (iMC) that uses a new DDR-T (64-byte) interface. Once a request reaches the Optane DIMM, a module controller reads/writes from/to the media at a 256-byte line granularity. A 16 KB write combining buffer merges adjacent lines to mitigate high media latency (Yang et al., 2020b; Akram, 2021c). Small (< 256 B) and random PM accesses are slower than large and sequential ones.

2.3 SIMD

SIMD (single instruction, multiple data) is used in compression to enhance decompression latency significantly. SIMD involves executing the same operation on multiple data items in parallel (Thai and Akram, 2023). Intel processors feature specialized processing units that operate on large vector registers, enabling simultaneous processing of all bits in the register at once (Intel Corporation, 2023). For example, with 256-bit SIMD registers, 8 integer operations can be processed using a single instruction. In the best case, we can achieve a speedup proportional to the vector length because SIMD instructions can perform at the same speeds as scalar instructions (Hager and Wellein, 2010). However, in most cases, we cannot achieve the ideal speedup because of a variety of factors, e.g., the overhead of moving data between memory and SIMD registers, especially when the address is not aligned to the SIMD length (Hager and Wellein, 2010). On Intel processors, special functions (intrinsics) are used in C/C++ to access lower-level SIMD instructions without requiring assembly code (Hager and Wellein, 2010).

2.4 Compression

Compression is crucial for reducing the size of large indices, leading to savings in memory capacity and bandwidth. Compressed indices also increase memory locality at all levels, e.g., cache and TLB locality. Here, we discuss both basic and state-of-the-art integer compression methods and encoding.

2.4.1 Differential Encoding

Differential encoding (delta encoding) is applied before performing a more complex compression algorithm to improve the compression ratio (Thai and Akram, 2023). In delta encoding, we store the differences between successive integers instead of the integers themselves (Lemire et al., 2016; Thai and Akram, 2023). This results in a better compression ratio because the difference between adjacent integers is generally smaller than the value of each integer (Lemire et al., 2016). After decompression, we retrieve the original integers by calculating the prefix sum (Lemire et al., 2016). Delta encoding works best when the integers are sorted (Thai and Akram, 2023). This requirement of having a sorted integer stream is straightforward to satisfy in search engines because documents arrive in a sorted order. (Each new document is assigned a monotonically increasing integer identifier.)

2.4.2 VByte

VByte is a very simple but efficient compression technique (Lemire et al., 2018). VByte compresses integers at byte granularity by eliminating redundant bytes containing leading zeros in binary representation (Lemire et al., 2016; Thai and Akram, 2023). For each byte of the compressed integer, the most significant (eighth) bit serves as a "continuation flag", indicating whether the number requires an additional byte. The remaining seven bits store the integer's data bits (Thai and Akram, 2023).

2.4.3 StreamVByte

StreamVByte is a state-of-the-art compression algorithm that uses the fundamental principles of VByte but achieves substantially faster decompression speeds due to two critical optimizations (Lemire et al., 2018; Thai and Akram, 2023). First, it uses 128-bit SIMD instructions to decode chunks of 4 integers at a time (Lemire et al., 2018; Thai and Akram, 2023). Second, it represents the sizes of the compressed integers in a stream of control bytes and stores them separately from the data bytes to reduce data dependencies (Lemire et al., 2018; Thai and Akram, 2023).

The second optimization improves processor utilization since the position of the next chunk of integers to be decoded can be determined without decoding the current chunk (Lemire et al., 2018; Thai and Akram, 2023). VByte interleaves continuation flag bits with data bits, introducing data dependencies that can increase pipeline stalls (Lemire et al., 2018; Thai and Akram, 2023).

Decoding with StreamVByte can be as much as six times faster than decoding with VByte across real data sets (Lemire et al., 2018).

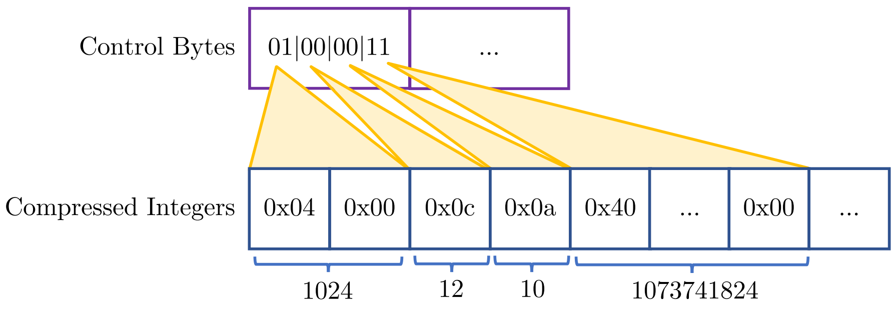


Figure 2.1: StreamVByte algorithm scheme.

Figure 2.1 shows how integers are encoded under the StreamVByte scheme (Thai and Akram, 2023). The sizes of uncompressed integers, namely 1, 2, 3, and 4 bytes, are represented by the binary codes 00, 01, 10, and 11, respectively (Thai and Akram, 2023). Each control byte decodes four integers by loading the next 16 data bytes into a 128-bit SIMD register (Lemire et al., 2018; Thai and Akram, 2023). A vector shuffle function is applied to extend the next four compressed integers across the whole vector register (Lemire et al., 2018; Thai and Akram, 2023). The exact shuffle function is obtained from a look-up table indexed by the control byte (Lemire et al., 2018; Thai and Akram, 2023).

2.4.4 TurboPFor

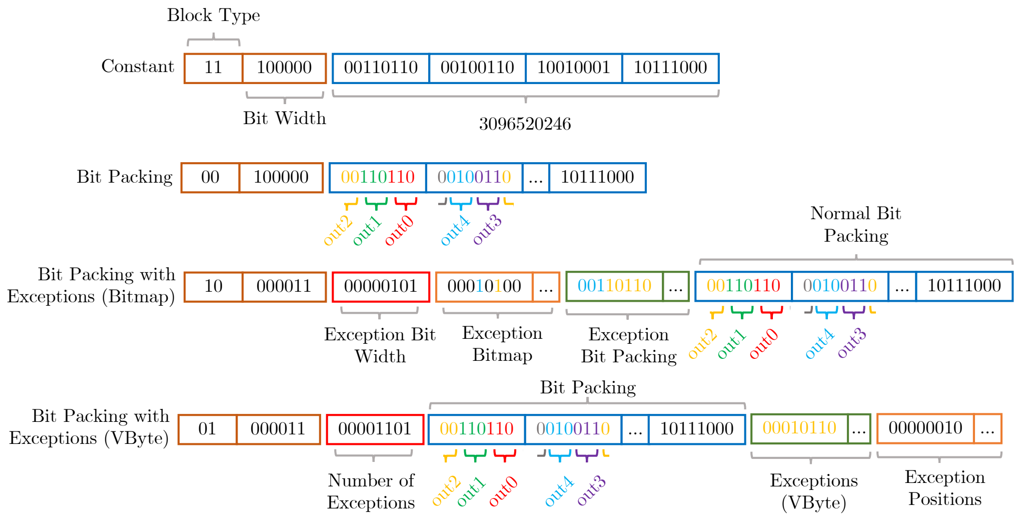


Figure 2.2: TurboPFor algorithm scheme.

TurboPFor256 offers state-of-the-art decompression speeds alongside very high compression (Stapelberg, 2019; Thai and Akram, 2023). We describe the TurboPFor encoding and how it is extended using 256-bit SIMD instructions (TurboPFor256).

In the TurboPFor scheme, integers are decoded in blocks of 256 integers (Stapelberg, 2019). The encoding scheme is depicted in Figure 2.2. Blocks are compressed using one of four methods, the type of method is indicated in the block header (Stapelberg, 2019; Thai and Akram, 2023):

- 00: bit packing
- 01: bit packing with exceptions (variable byte)
- 10: bit packing with exceptions (bitmap)
- 11: constant

The algorithm chooses the block type that maximizes compression (compression is prioritized over speed) (Powturbo, 2023; Thai and Akram, 2023). The constant and bit packing blocks are optimal for small integers. The bit packing with exceptions (bitmap) block is used for larger integers, and bit packing with exceptions (VByte) is used for large integers with non-uniform exceptions (explained shortly). (Stapelberg, 2019; Thai and Akram, 2023).

The constant block is used when all elements in the block have the same value (Figure 2.2) (Thai and Akram, 2023). The value is stored only once, and the bit width is set to the minimum number of bits required to represent this value (Thai and Akram, 2023).

Bit packing blocks are used when all integers in the block have small values. Bit packing is a technique that encodes integers within the range $[0, 2^b)$ where b is the bit width of the encoded integers, e.g., if the largest number in the block is 8, then all integers are encoded using 3 bits (Lemire and Boytsov, 2015; Thai and Akram, 2023). The bits of each compressed integer are packed together in memory (Lemire and Boytsov, 2015; Thai and Akram, 2023).

The bit packing with exceptions (bitmap) block uses the widely known PFor (patched frame of reference) compression scheme (Stapelberg, 2019; Thai and Akram, 2023). It uses a smaller bit width for bit packing, meaning that the most significant bits of some large integers are truncated (Stapelberg, 2019; Thai and Akram, 2023). The algorithm chooses an optimal bit width b such that the majority of integers are smaller than 2^b (Wang et al., 2017; Thai and Akram, 2023); in particular, it chooses the optimal bit width for compression (Powturbo, 2023; Thai and Akram, 2023). The truncated bits of large integers are bit-packed separately, requiring another bit width header for exceptions, bit-packing block for exceptions, and an exceptions bitmap for combining the split integers when decoding (Stapelberg, 2019; Thai and Akram, 2023), as shown in Figure 2.2. The number of exceptions is obtained by summing the 1s in the exception bitmap (Stapelberg, 2019; Thai and Akram, 2023).

We present a concise example for decoding a bit packing with exceptions (bitmap) block using Figure 2.2. Our goal is to decode the third integer (zero-indexed). The lower portion (in yellow) is obtained from the normal bit packing block: 000b (Thai and Akram, 2023). Since index 2 of the exception bitmap is set, we extract the upper portion from the exception bit packing block: 10110b. They are combined to obtain the decompressed integer: 10110000b.

The bit packing with exceptions (variable byte) block is similar to the bit packing with exceptions (bitmap) block, except we encode the upper portion of exceptions using VByte (Stapelberg, 2019; Thai and Akram, 2023).

To speed up decompression, TurboPFor is extended with 256-bit SIMD bit packing to unpack multiple integers at once (Stapelberg, 2019; Thai and Akram, 2023). For simplicity, we discuss SIMD bit backing with 128-bit registers. In SIMD bit packing, compressed integers are packed into a series of 4 consecutive 32-bit words in a round-robin fashion (Lemire and Boytsov, 2015; Thai and Akram, 2023). When the current series is filled completely, the leftover bits “spill over” to the next series (Lemire and Boytsov, 2015; Thai and Akram, 2023). To decode these integers, we load the first series into the SIMD register (Thai and Akram, 2023). We then apply shift and mask operations repeatedly to extract 4 integers at a time, load the next series, and apply similar operations (Lemire and Boytsov, 2015; Thai and Akram, 2023). The exact shifting and mask operations depend on the bit width and are slightly more complex when the bit-width is not a divisor of 32. More details are found in prior art Lemire and Boytsov.

2.5 BM25 Scoring

We employ a popular scoring algorithm called BM25 to rank our query results, it is the default relevancy algorithm used in Elastic Search (Connelly).

The BM25 formula is given taken from Wikipedia (Wikipedia contributors):

$$\mathrm{BM25}(q, D) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\mathrm{avg\_dl}}\right)} \tag{2.1}$$

$$\mathrm{IDF}(q_i) = \log\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}\right) \tag{2.2}$$

In Equation 2.3, q is the query, D is the document, n is the number of terms in the query q, $f(q_i, D)$ is the frequency of term q_i in document D, avg_dl is the average document length in the collection, $|D|$ is the length of document D, k_1 and b are hyperparameters, typically set to $k_1 = 1.2$ and $b = 0.75$, and $\mathrm{IDF}(q_i)$ is the inverse document frequency of term q_i (Wikipedia contributors). In Equation 2.2, N is the total number of documents in the collection and $n(q_i)$ is the number of documents containing term q_i (Wikipedia contributors).

In our datasets, we use fixed-sized documents, which simplifies the Equation 2.3 to:

$$\mathrm{BM25}(q, D) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1} \tag{2.3}$$

Note that fixed-sized documents is realistic scenario on many emerging platforms where search is important feature. For example, X (formerly Twitter) limits the total size of a tweet to some fixed threshold.

We calculate $\mathrm{IDF}(q_i)$ once for each query term q_i to eliminate redundant calculations. We use $k_1 = 1.2$.

2.6 Control Group v2

Control Group V2 (called "cgroup" and written in a non-capitalized manner) is a Linux feature that offers mechanisms for distributing system resources to groups of processes (Heo). Every process belongs to a single control group (also called "cgroup"). The behavior of processes within a cgroup are collectively bound by a shared a set of rules governing system resource constraints (Heo). Every thread within a process is assigned to the same cgroup (Heo). We use the cgroup memory controller to regulate memory usage. We configure a memory (DRAM) usage hard limit by writing the limit (in bytes) to a cgroup's memory.max interface file (Heo). The OOM killer is invoked if the memory usage reaches the limit and cannot be reduced (Heo). The cgroup memory controller accounts for the following memory usage categories. (1) userland memory consisting of the page cache and anonymous memory. (2) kernel data structures such as dentries, inodes, and TCP socket buffers (Heo).

Chapter 3

Related Work

In this section, we discuss prior work related to capacity expansion (beyond DRAM) for search engines, exploiting emerging fast storage devices (both non-volatile memory and PCIe NVMe drives) for storing inverted indices, and tradeoffs at the intersection of compression and storage technologies. We also discuss prior art related to how key-value stores and other data-intensive applications exploit emerging storage devices.

3.1 Indices in Hybrid Memory and Fast Storage

Recent work proposes NVM-based software stacks for databases, key-value stores, hash tables, in-memory caches, and filesystems (Kannan et al., 2018; Kaiyrakhmet et al., 2019; Yao et al., 2020; Chen et al., 2021; Yang et al., 2020a; Zhang et al., 2021; Benson et al., 2021; Zheng et al., 2019; Kwon et al., 2017; Xu et al., 2019; Kassa et al., 2021). However, despite their ubiquity, little prior work focuses on exploiting emerging storage for hosting search indices. In the closest related work, Akram et al. evaluate Intel Optane PM for various roles. They evaluate PM for memory capacity expansion, persistent storage, and universal memory, incorporating both roles as main memory and storage. Unlike many prior works, their focus is on traditional batch search. Their work motivates rethinking the software stack for search with PM hardware. Specifically, they find building indices on PM degrades indexing throughput due to the high latency and limited bandwidth of PM. However, they find a good role of PM in search engines. They find the performance of PM-backed inverted index for one-term queries at par with DRAM. They also find the tail latency of AND queries higher compared to state-of-the-art NVMe SSD (with buffered accesses via the page cache). Our work is different in two ways. (1) We use compressed indices which is more realistic than their uncompressed indices. We also rigorously show the QPS obtainable with DRAM and PM when compressed indices are backed by them. (2) Their system lacks a dynamic mechanism to bring index segments from storage to PM. They assumed that indices can be copied into PM using system administration tools. However this approach is unrealistic in datacenters where indices are preserved on storage, and server hardware may or may not have a second memory tier. Our software solution for multi-tiered index hosting can be deployed on systems with or without PM (or second memory tier). Also, even on baseline systems with DRAM alone, our query workload-aware prefetching mechanism presents a new contribution.

The second closely related work is a recent study on DRAM and PM-backed compressed inverted indices (Chilukuri and Akram, 2023b). They use a managed search engine, namely Lucene (Foundation, 2021b), written in Java, to evaluate the behavior of compressed/uncompressed inverted indices in on-heap and off-heap memory. They find that decompression slows down applications more than memory technology. In observing that, they agree with one of our crucial findings, but not the other. They also observe that PM is an attractive medium for storing inverted indices due to the sequential access pattern of search queries. However, they observe a higher degradation in performance (QPS-wise) between compression and no compression. First, they use a different search engine than our in-house C/C++-based engine. Second, they make limited use of SIMD due to JVM lacking full support of new SIMD intrinsics, and especially Lucene not keeping up with the latest vector additions to the Intel x86-64 ISA. Finally, they use a more complicated index design, where different portions of the index are compressed using different compression technologies. It implies that there is overhead in the application code for maintaining and tracking all these compressed portions of the index in various on-heap buffers. Our design of the search engine is highly optimized with each component of the index and its corresponding evaluation hand-tuned, using buffers only when they are necessitated. Our interest is more fundamental level to figure out the true cost of decompression, and how much of the overhead of decompression is offset by the improvement in memory locality (both cache and TLB), and also the fraction of cost offset by SIMD instructions in decompression. Note that SIMD impacts register usage and cache impacts in critical ways, so it cannot be ignored that although decompression is extra work on the compute side, it impacts cache locality in a different way, balancing the cost with an improvement in memory behavior. We should also note that the work by Chilukuri and Akram also lacks the dynamism of moving the index automatically from block storage to the second memory tier.

Overall, compared to the most related recent work, this thesis moves the body of knowledge on the performance of compressed inverted indices backed by new memory tiers in important ways.

3.2 Other Data-Intensive Applications over Hybrid Memory

Inverted indices are similar to log-structured merge (LSM) trees. Prior work extends LSM-tree-based key-value stores (Kannan et al., 2018; Kaiyrakhmet et al., 2019; Yao et al., 2020; Chen et al., 2021) to exploit heterogeneous storage. Existing efforts try to establish NVM as a new tier in the storage hierarchy, which does not fully exploit

NVM's potential as byte-addressable memory, complementing DRAM. NoveLSM (Kannan et al., 2018) uses an NVM-backed Memtable, while SLM-DB (Kaiyrakhmet et al., 2019) places a global index in NVM and maintains single-level SSTables, unlike traditional LSM. MatrixKV (Yao et al., 2020) exploits NVM for fine-grained column compaction. SpanDB (Chen et al., 2021) exploits different types of SSDs to offer cost efficiency. Other works propose NVM-based filesystems (Zheng et al., 2019; Kwon et al., 2017). Prior art also reconsiders other service components, e.g., caching over hybrid memory (Kassa et al., 2021). Recent work also optimizes hash tables for NVM (Lu et al., 2020; Hu et al., 2021; Vogel et al., 2022; Lu et al., 2021; Debnath et al., 2015; Zuo et al., 2018; Nam et al., 2019). They mitigate NVM writes, and store meta-data in DRAM.

Recent work exploits fast storage for enabling terabyte-scale managed heaps in big data frameworks (Kolokasis et al., 2023). They use *MMIO* to grow the heap over a fast storage device (NVM or NVMe SSDs). They find that preserving DRAM capacity is possible if a heap is grown over fast storage. The OS automatically prefetches heap objects that are needed by the application into the page cache. Identifying objects to move to the second heap is challenging, and the current best approach is to communicate programmer's knowledge expressed through annotations in the Java code to the Java Virtual Machine (JVM), which then transfers the transitive closure of the so-called root object to the SSD tier of memory acting as an expansion device. We leave evaluation of managed compressed indices storing over SSD tier to future work.

3.3 Datacenter Caches

Caches are widely deployed in datacenters for a range of purposes. Companies like Twitter, Facebook, and Amazon rely on caches to mitigate backend storage and database latency, improving the service experience for their customers. We believe a customized postings cache for search engines over hybrid memory is more efficient. Prior art uses and proposes a number of caches for various purposes Twitter (2023); Berg et al. (2020b); Services (2023); Sanfilippo (2023); Fitzpatrick (2023). These prior systems offer unique features and capabilities that cater to specific requirements.

3.4 Persistent Memory Characterization

Prior work characterizes Intel Optane PM for native and managed workloads (Yang et al., 2020a; Akram, 2021c). Both focus on uncovering Optane's behavior but for different workloads, and they report Optane's limited scaling with increasing thread count. Specifically, Akram (Akram, 2021c) finds that traversal algorithms (e.g., heap traversals in garbage collection) incur high latency in Optane-backed data structures compared to DRAM-backed structures. For example, they report that copying from nursery to mature space happens almost as fast with Optane as with DRAM. On the other hand, scanning objects to perform the transitive closure is slowed down significantly. In contrast, we focus on Optane as a high-capacity medium for storing uncompressed indices. Other prior efforts propose partitioning managed heaps into NVM and DRAM heaps and placing special objects in the NVM heap to improve write endurance, performance, and scalability (Akram, 2021c; Wang et al., 2019; Nguyen et al., 2016).

Chapter 4

Index Organization and Query Evaluation

Search engines use inverted indices to speed up query evaluation. For this thesis, we build an inverted index that uses several features found in state-of-the-art search engines. We decide to build an index from scratch for two reasons. First, we want to rigorously evaluate a number of compression algorithms, which the popular search engines (e.g., Lucene) do not incorporate. Lucene is written in Java, and JVM has limited support for SIMD intrinsics used by modern compression algorithms. Second, we want to use multi-tiered memory for which JVM also has limited support. We discuss the organization and design of our index and the related aspects of query evaluation here (instead of Chapter 2) because the resulting infrastructure for research on compressed indices is our contribution.

We now discuss the design of the index and the indexing and query evaluation processes for single (one-term) and conjunctive (multi-term) queries.

4.1 Indexing

Indexing consists of two stages: (1) ingest and (2) transform. In the first stage, fresh documents are ingested into an in-memory buffer. Specifically, in the ingest stage, the indexer builds a hash table of posting blocks. Inside a posting block, we store postings which contain an integer document identifier (**DID**) and frequency. Frequency represents the number of times a term appears in a document. Frequency is useful for scoring each document against a query term. The DIDs and frequencies are stored in a structure of arrays format for ease of compression. In the transform stage, the hash table is compressed and converted into the format shown in Figure 4.1. This transformation step is akin to serialization, removing all virtual references to in-memory objects. It compacts the useful fields by allocating them in contiguous storage, and makes the entire

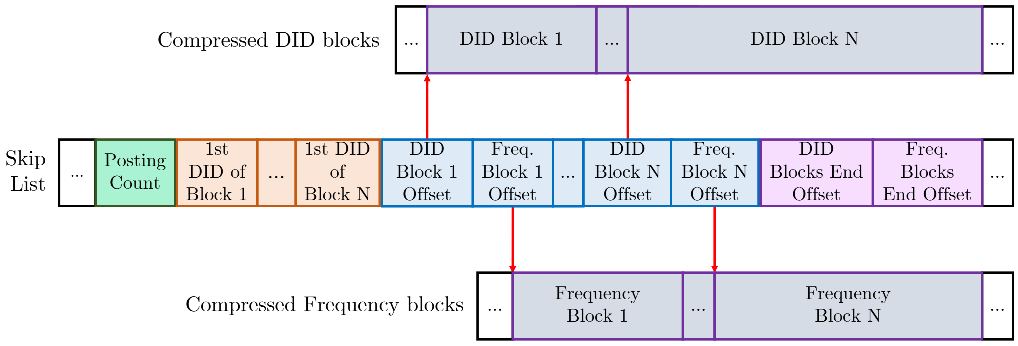


Figure 4.1: Design and organization of our inverted index on disk.

structure amenable for disk storage. Next, we explain ingestion and transformation in more detail below.

Ingestion works as follows.

1. The indexer processes a stream of documents. For each document it performs the following actions.
   a) The document is assigned a DID. This mapping from the DID to the document's location on a file system can be stored in a dictionary to retrieve the document file (Thai and Akram, 2023).
   b) The document is processed by creating postings for its terms and adding them to the hash table (Thai and Akram, 2023). We outline the process for adding a posting to the hash table below.
      i. The term associated with the posting is hashed to obtain an offset to its corresponding bucket in the hash table. The bucket contains a reference to a linked list of blocks. A block holds postings stored in a structure of arrays format (array of DIDs, and array of frequencies).
      ii. The new posting is either inserted into the last block of the list, or a new block if the last block is full. The new block is attached to the end of the list and becomes the new last block. (Thai and Akram, 2023) We aggregate postings for the same document using the frequency to avoid duplicate DIDs.

Next, the index is transformed into the format shown in Figure 4.1 as follows.

1. Separate files are created for the skip list, compressed DID blocks and frequency blocks. The skip list stores special metadata for each term, including the posting count, first DID of each block, and the term's DID and frequency block offsets in the DID and frequency block files (shown in Figure 4.1). The first DID of each block is used for the intersection algorithm and decoding blocks with compressed with delta encodings. We employ a Berkeley DB (BDB) dictionary to store the

mapping from terms to its skip list offset (where its metadata is stored). We use memory-mapped IO to write to the index files.

2. For each occupied bucket (associated with a term), we apply the following process to create the index:
    a) The mapping from the bucket's term to current the current skip list offset is stored in the skip list offset dictionary.
    b) The skip list offset is incremented to allocate space for the term's metadata, shown in Figure 4.1.
    c) For each of the term's blocks:
        i. We update the block's DID and frequency file offset in the skip list. We write the block's first DID to the skip list.
        ii. The DIDs and frequencies within the block are compressed directly to the DID and frequency block files.
        iii. The DID and frequency block file offsets are incremented.
3. The final DID and frequency file offset is written to the skip list.

Since the TurboPFor256Delta algorithm encodes integers in blocks of 256 integers, we use a block size of 256 (powturbo).

4.2 Query Evaluation

Now we explain the query evaluation process for this index. The query evaluator returns the top 1024 results for each query, ranked using the BM25 scoring algorithm. We utilize the output buffer array, intended for query results, as a min-heap to quickly extract and replace the lowest scoring posting.

The query evaluator handles two types of queries: single (one term queries) and AND (two-term conjunctive queries). We explain how each type of query is processed below.

The evaluation of one-term queries involves locating and serving all postings which match the query term (Thai and Akram, 2023). For each one-term query, we take the following steps to resolve the query.

1. We obtain the term's skip list position using the skip list offset dictionary.
2. The term's posting count is extracted from the skip list and used to calculate the number of blocks. For each posting block:
    a) We obtain the DID and frequency block offsets from the skip list (Figure 4.1). The DID and frequency blocks are decoded into user buffers using memory-mapped IO.

   b) The block's postings are scored using frequencies, and inserted (copied) to the min-heap (output buffer).

3. The output buffer is sorted by score.

Serving an AND query involves finding the intersection of postings for two terms, i.e., postings for both query terms which contain matching DIDs (Thai and Akram, 2023).

In general, fast algorithms such as binary or finger search to perform intersections for posting lists (Chilukuri and Akram, 2023a). However, doing this is inefficient for compressed posting lists since it requires decompressing all blocks before performing the intersection. We utilize the skip list to accelerate intersection queries by bypassing unnecessary blocks (Thai and Akram, 2023).

Our intersection is based on the powturbo algorithm (powturbo). For each AND query, we apply the following steps:

1. The skip list offsets for both terms are obtained from two lookups to the skip list offset dictionary.
2. We obtain the number of postings, first DIDs of each DID block, and DID/frequency block offsets for each term from the skip list.
3. Next, we perform skip list intersection. By using the first DID of a block and the first DID of the subsequent block, we determine the range of possible DIDs within that block. We then perform a traversal of blocks for both terms, checking if the potential DID ranges intersect. If they do, the blocks are decompressed and intersected. The traversal process begins by calculating the number of blocks for each term and initializing a counter to track the current block of each term to zero. While the block counter for both terms are less than their total block counts:
   a) If there is a DID range overlap between both current blocks, the DID and frequency blocks are decompressed into a user buffer.
   b) The DIDs of the block are intersected, and if a match is found, the posting is scored using the BM25 algorithm. If the score exceeds that of the lowest scoring posting in the min-heap (output buffer), it is inserted. During the intersection process, we keep track of our position within each block to avoid redundant work if the same block is used in the next iteration.
   c) The term with the current block having the lower upper DID range advances to the next block by incrementing its counter. If the upper DID range is the same, then both terms advance to the next block.
4. The output buffer is sorted by score.

Chapter 5

Evaluation Methodology

We now discuss our experimental methodology for (1) performance analysis of DRAM-Only and NVM-Only indices, and (2) evaluation of APCache.

5.1 Microarchitectural Analysis Methodology

We perform rigorous microarchitectural analysis of various integer compression algorithms for compressing inverted indices. Our goal is to establish if storage technology has an impact on query evaluation latency. Microarchitectural analysis relates to the hardware implementation aspects of an instruction set architecture (e.g., Intel x86-64 or ARM A32). For example, it relates to understanding the behavior of caches and out-of-order pipelining machinery—both of which are transparent to software and programmers. We discuss our evaluation methodology of the microarchitectural analysis aspect of the thesis below.

5.1.1 Index Configurations

We use several state-of-the-art compression algorithms and their (SIMD and other) variations in our experiments. These algorithms are in use in production systems, and they offer varying degrees of compression. Specifically, we experiment with the following configurations.

- *TurboPFor*: TurboPFor encoding (powturbo).
- *TurboPFor256*: TurboPFor with 256-bit SIMD bit packing.
- *TurboPForDelta*: TurboPFor with delta encoding.
- *TurboPForDelta256*: TurboPFor with 256-bit SIMD bit packing and delta encoding.

- *TurboVByte*: A state-of-the-art implementation of VByte (powturbo).
- *TurboVByteDelta*: TurboVByte with delta encoding.
- *StreamVByte*: StreamVByte encoding (Lemire et al., 2018).
- *StreamVByteDelta*: StreamVByte with delta encoding.

Most of the compression algorithms are sourced from the `TurboPFor Integer Compression` library (powturbo), except for StreamVByte and StreamVByteDelta (Lemire et al., 2018).

One concern is how to deal with frequencies as they are not monotonically increasing or in sorted order. Therefore, we cannot and do not use delta encoding for frequencies. Instead, we compress frequency blocks using the same algorithm as DIDs, but without applying delta encoding.

With these decisions, our experimental configurations and taxonomy is as follows.

- *Base*: Index is not compressed at all.
- *PForDelta*: DID blocks are compressed using TurboPForDelta, and frequency blocks are compressed using TurboPFor.
- *PForDelta256*: DID blocks are compressed using TurboPForDelta256. If a block contains less than 256 integers, it is compressed using TurboPForDelta instead. Similarly, frequency blocks are compressed using TurboPFor256, or TurboPFor for incomplete blocks.
- *SVByteDelta*: DID blocks are compressed using StreamVByteDelta, and frequency blocks are compressed using StreamVByte.
- *VByte*: The index is compressed using TurboVByte.
- *VByteDelta*: DID blocks are compressed using TurboVByteDelta, and frequency blocks are compressed using TurboVByte.

  For each configuration, we experiment with storing the index (both DIDs and frequencies) on PM and DRAM. The skip list and skip list offset dictionary are always stored in DRAM on Linux `tmpfs`.

5.1.2 Intel Top-down Microarchitectural Analysis

We use a Intel's recommended top-down methodology for performance analysis (Yasin, 2014). The method is designed for identifying performance bottlenecks in out-of-order processors using performance counters (Yasin, 2014). In out-of-order processors, instructions execute concurrently, making it difficult to pinpoint the reason for a pipeline stall. In other words, if in a cycle, the processor is under-utilized, it could be due to one of many so-called miss operations. For example, an instruction may not be found in the instruction cache, or a data object may not be found in the data cache, or else, all

Table 5.1: Formulas for top-down metrics from performance counters. (Yasin, 2014)

Metric Name	**Formula**
Clocks	*cpu_clk_unhalted.thread_p*
Slots	4 × *Clocks*
Top-Down (Pipeline Slots)	
Front-End	*idq_uops_not_delivered.core*
Bad Speculation	(*uops_issued.any* − *uops_retired.retire_slots* + *int_misc.recovery_cycles*)
Retiring	*uops_retired.retire_slots*
Backend	*Slots* − (*Front-End* + *Bad Speculation* + *Retiring*)
Backend-Bound (Clock Cycles)	
Core	*Clocks* − *uops_executed.thread_cycles_ge_3*
Memory	*cycle_activity.stalls_mem_any* + *resource_stalls.sb*
Memory-Bound (Clock Cycles)	
External	*cycle_activity.stalls_l3_miss*
L3	*cycle_activity.stalls_l2_miss* − *External*
L2	*cycle_activity.stalls_l1d_miss* − *cycle_activity.stalls_l2_miss*
L1	*cycle_activity.stalls_mem_any* − *cycle_activity.stalls_l1d_miss*

the functional units (e.g., arithmetic logic units or ALUs) are busy occupied by older instructions. We collect performance counters using the `libpfm4` library (Cohen), and use them to calculate the number of pipeline slots that are (1) retiring, i.e., devoted to useful work, (2) front-end bound, i.e., un-utilized due to the front-end being unable to adequately supply its back-end, (3) bad speculation bound, i.e., wasted as a result of incorrect speculations, (4) back-end bound, i.e., it consists of stalls related memory-bound and core-bound. The memory-bound stalls are due to instructions waiting for memory to respond. The core-bound stalls are due to the lack of functional units or reservation stations (where instructions wait for their operands to become available). Finally, we calculate the number of clock ticks that are L1, L2, L3 (these three components represent the on-chip levels of the cache hierarchy), and external memory bound. We show these calculations in Table 5.1.

5.2 APCache Evaluation Methodology

5.2.1 Server Platform

We conduct experiments on a dual-socket Dell PowerEdge R740 server running Ubuntu Linux 18.04 LTS. Each socket contains Intel Xeon Gold 6252N running at 2.3 GHz, each with 24 physical cores (48 logical cores) and 96 cores in the system. Each core has 35.75 MB of shared L3 cache and an integrated memory controller supporting six memory channels. Each memory channel connects to a 32 GB Micron DDR4 DIMM and a 128 GB Intel Optane NVDIMM. With 12× 32 GB DIMMs and 12× 128 GB Intel Optane DIMMs, the total memory capacity of the system is 384 GB of DRAM and 1.5 TB of Optane PM. The system has a 1.5 TB Intel Optane PCI Express NVMe SSD (DC P4800X). The system also has a 1 TB, 3.5-Inch, Seagate, SATA (6 Gbps), hard drive, capable of 7200 rotations per minute. We disable non-uniform memory access

Table 5.2: This table highlights key features of two storage technologies: the Seagate ST8000NM0205 Hard Disk Drive (HDD) (ALLHDD.com, 2024) and the Intel Optane Solid State Drive (SSD) DC P4800X Series (Intel, 2024).

Feature	**Seagate ST8000NM0205**	**Intel Optane DC P4800X**
Type	Hard Disk Drive (HDD)	Solid State Drive (SSD)
Capacity	8TB	1.5TB
Interface	SATA 6GBPS	PCIe 3.0 x4, NVMe
Sequential Read Bandwidth	181MB/s	2200MB/s
Sequential Write Bandwidth	167MB/s	902MB/s
Average Latency	4.16ms	$< 10/12\mu$s

(NUMA) accesses in all our experiments.

Table 5.2 highlights the key features of the two secondary storage technologies used in our experiments Seagate ST8000NM0205 (disk) and Intel Optane DC P4800X (SSD). We measure the sequential read and write speeds using the Linux `dd` utility (Baeldung, 2024).

5.2.2 Datasets and Ingestion

We evaluate our configurations across two real-life datasets. The first dataset is a 160 GB corpus we buid from the May 2022 Common Crawl archive (Crawl). The second dataset is a 30 GB corpus Chilukuri and Akram we build from the full text of English Wikipedia, with the snapshot taken in March 2021 (Chilukuri and Akram, 2023a). Both datasets limit each document to 1 KB in size.

5.2.3 Query Formation

We generate two types of queries, (1) conjunctive two-term (AND), and (2) single term (SINGLE). For both types, we create three workloads containing terms of different frequency classes reflecting their posting count. The high (H) workload consists of terms with the highest frequency in a corpus, medium (M) with medium frequency, and low (L) with the lowest frequency.

For the Common Crawl dataset, we obtain a list of the terms that appear in the dataset and their posting count. We divide the terms into different frequency classes using the following buckets.

$$\begin{aligned} \text{H} &: [1\,000\,000, \infty) \\ \text{M} &: [1\,000, 1\,000\,000) \\ \text{L} &: [0, 1\,000) \end{aligned}$$

For the Wikipedia dataset, terms are separated into three sets: high frequency terms (H), medium frequency terms (M), and low-frequency terms (L) using classification from the LuceneUtil McCandless (2021). We create one-term and two-term query workloads by randomly generating queries from the H, M, and L term sets. This results in six workloads, (1) three one-term workloads, namely L, M, and H, and (2) three two-term workloads, namely LL, MM, and HH.

For the Common Crawl dataset, we increase the size of query workloads with lower frequency terms to balance the execution time across all workloads. We use the following workload sizes for different frequency classes.

H	: 1 000
M	: 100 000
L	: 1 000 000

Finally, for the Common Crawl dataset, we use a parameter called query repetition probability (RP) to control the regularity of repeated query terms in our query workloads (Weerakkody and Akram, 2023). This feature is useful for evaluating the cache (discussed in Chapter 8). We generate workloads with different RPs for each frequency class. Specifically, we use two FIFO queues, (1) a queue containing all unique terms (for a specified frequency class), which we will call the term queue, and a queue from which we choose repeated queries, which we will call the repeat queue. We limit the repeat queue size to 100 which is the default setting, effectively setting 100 to be the maximum number of queries between any two repeating queries (Weerakkody and Akram, 2023). Given a specified workload size (number of queries), term list of all unique terms in the corpus, repeat queue size, and frequency class, a single-term workload for SINGLE queries is generated as follows.

1. All of the terms in the term list belonging to the specified frequency class are added to the term queue.
2. The term queue is randomly shuffled.
3. Until the workload reaches the desired size, we add terms to the workload:
   a) The *next term* is popped from the term queue.
      i. If the repeat queue is empty, then the *next term* is added to the workload.
      ii. Otherwise, with a probability of RP, we add a random term from the repeat queue to the workload. If not chosen from the repeat queue, the *next term* is added to the workload.
   b) The *next term* is added to the repeat queue. If the repeat queue is larger than its maximum size, the oldest term is popped from the repeat queue.

We generate two-term workloads for AND queries in a similar way, except that after shuffling the term queue, we split it into two halves. We use the two halves to generate two one-term workloads (as explained above), and then join terms in a pair-wise fashion, grouping together elements from each sequence based on their position.

We set RP for in our experiments as follows. We use an RP of 25% for the microarchitectural analysis. Furthermore, we use a range of RPs (0%, 25%, and 50%) for evaluating APCache.

5.2.4 Measurement Metrics and Methodology

On the indexing side, we report the time it takes to build the index, i.e., indexer execution time. We also collect and report the compression ratios and the compressed size of the index.

On the side of query evaluation, we report queries per second (QPS), tail latency, and microarchitectural performance counters. We collect performance counters using the `libpfm4` library (Cohen). Using performance counters and Intel's top-down methodology, we determine the number of pipeline slots that are front-end, bad speculation, back-end (memory and core), and retiring bound (Yasin, 2014). We also use performance counters to calculate the number of clock ticks that are L1, L2, L3, and external memory bound.

For the large Common Crawl dataset, we observe consistent results across several repetitions of the same experiment, i.e., the standard deviation is negligible. We run each query experiment twice, taking the results of the second repetition. We run each query experiment for the smaller Wikipedia dataset ten times and take the mean across the ten experimental runs.

We use an output buffer size of 1024 integers. The search engine returns the top 1024 posting results based on the BM25 scoring algorithm. For this purpose, it uses a min-heap with 1024 elements.

Chapter 6

Microarchitectural Analysis

In this section, we discuss motivating analysis at the microarchitecture level to understand the behavior of compressed indices across two memory technologies. This motivating analysis informs the design of our proposed postings cache, namely, APCache. We first discuss characterize the indexing process, discuss index size under different compression algorithms, and then discuss the performance of query evaluation. The former is important for picking the best compression algorithm and understanding the impacts of cache locality against our index sizes, and the latter informs the design of APCache.

6.1 Indexing Performance

The indexing process consists of two stages: (1) ingest and (2) transform. During the ingestion stage, the indexer creates a hash table that maps terms to posting blocks. In the transformation stage, this hash table is converted into the index structure described in Section 4 to enable faster lookups. We show the breakdown of indexing times into two components for our configurations in Figure 6.1.

We observe that ingestion takes up a large fraction of the indexing time, and ingestion times, as expected, are similar across all configurations. Therefore, the cost of compression is low. Nevertheless, we observe that transformation with compressed indices is faster than the baseline. This speedup in transformation is mostly because compressed indices necessitate transfers of smaller amounts of data to storage. Note that the time to store data on block storage from memory is included in the transformation step; we use SSD for block storage. Transformation with compressed indices is between 4% (PForDelta) and 20% (SVByteDelta) better than the baseline. We also observe that SVByteDelta and VByteDelta exhibit marginally better performance compared to VByte because more integers can be compressed into a single byte (deltas are smaller), resulting in a simple and predictable code path and thereby reducing branch

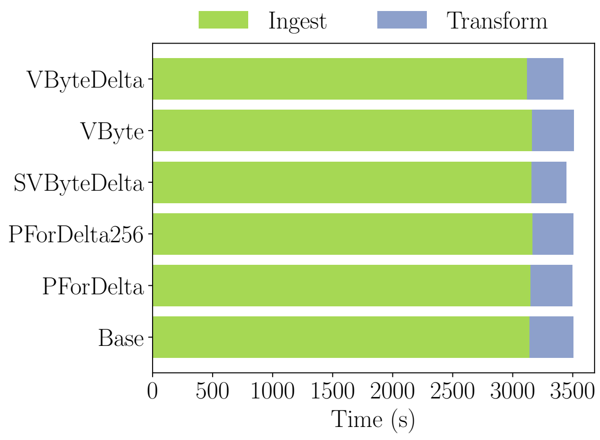


Figure 6.1: Showing the breakdown of time to build the index into ingest transform times.

misprediction penalties (Lemire et al., 2018).

6.2 Index Size

We now discuss index sizes with the baseline and different compression algorithms. Figure 6.2 shows the effectiveness of our compression algorithms in reducing the index size. Index size is broken down into sizes of the post-compression DID and frequency blocks. For all configurations, we have a 2.2 GB skip list and 1.25 GB skip list offset dictionary. It is important to note that frequency blocks are compressed without delta encoding, as delta encoding requires the integers to be monotonically increasing.

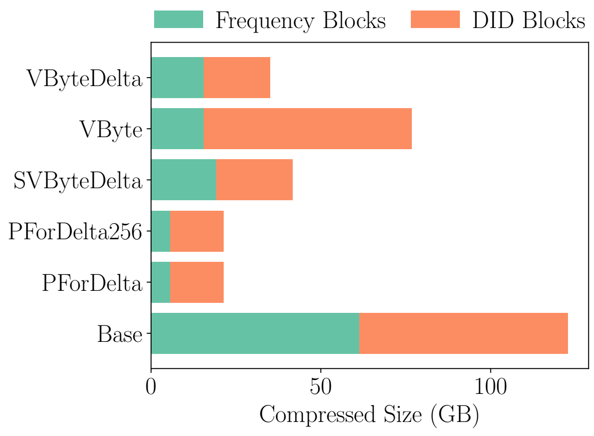


Figure 6.2: Showing the compressed index sizes and the sizes of DID and frequency blocks.

We find compression to be very effective. With the PForDelta configuration, the DID blocks and frequency blocks are compressed to 25.6% and 9.2% of their original size, respectively, as seen in Figure 6.2. We find that delta encoding significantly improves the compression ratio of TurboVByte. TurboVByte compression alone reduces the size of the DID blocks by 0.2% (Figure 6.2). TurboVByteDelta is much more effective, reducing the size by 68.2%. TurboVByteDelta compresses large numbers inefficiently but performs better with smaller deltas.

We find the frequency blocks compress extremely well for PForDelta256 and PForDelta configurations (Figure 6.2). They are compressed using the TurboPFor scheme, which uses bit packing, enabling finer-grained compression than byte-oriented algorithms.

The SVByte compresses slightly worse than VByteDelta (Figure 6.2). This result is expected as prior work reports that StreamVByte uses 0.5 to 2 additional bits per integer compared to TurboVByte (Lemire et al., 2018).

Overall, the compression ratios for PForDelta and PForDelta256 are 5.76× reduction in size compared to the baseline, and for VByteDelta, it is 3.5× compared to the baseline. Other algorithms are less effective at compressing the index. We use PForDelta256 in APCache later in the thesis, as it offers the best compression ratio and exhibits good performance because it exploits SIMD intrinsics in modern Intel processors.

6.3 Query Evaluation

We now discuss query evaluation with the index fully allocated in DRAM and PM. Our goal is to find out if a slower memory technology, such as Intel Optane PM, slows down query evaluation. If there is a slowdown, we next aim to understand the reason for the slowdown. If there is no performance degradation, we intend to find out why there is no degradation.

6.3.1 Roadmap of Results

Explaining the impact of memory technology on high-level application metrics, such as queries per second and tail latency, is challenging. For example, we need to be able to explain the changes in QPS with different compression algorithms (with or without SIMD) and relate the results with the dynamics of the microarchitecture and locality impacts. For this reason, we collect and show data at the user (high level) and microarchitectural (low level) layers. Specifically, in Figure 6.4 and Figure 6.3, we show the space-time trade-off between compression effectiveness and query performance. We show QPS achieved for various compression algorithms and their compression ratio. These figures inform us of the best tradeoff since the best compression ratio and the best QPS cannot be used simultaneously (we discuss this result in more detail later).

We measure the queries per second (QPS) for low (L), medium (M), and high (H) frequency queries for both AND and one-term (SINGLE) query types. We expect that compression slows down query evaluation due to decompression cost. Therefore, we use performance counters to conduct microarchitectural analysis to be able to explain our results. In Figure 6.5 and Figure 6.6, we show the number of pipeline slots that are: (1) retiring - devoted to useful work, (2) front-end bound, i.e., un-utilized due to the front-end being unable to adequately supply its back-end, (3) bad speculation bound, i.e., wasted as a result of incorrect speculations, (4) back-end bound, i.e., it consists of memory bound and core bound. In Figure 6.7 and Figure 6.8, we further break backend bound into (1) memory bound, i.e., stalled due to memory operations such as load or store instructions, and (2) core bound, i.e., bound by computational resources. Finally, we also investigate the number of clock ticks bound by stalls on various levels of the cache hierarchy, first level (L1), second level (L2), their level (L3), and external memory in Figure 6.9 and Figure 6.10.

6.3.2 Analysis

As explained above, discussing the various results we gather to understand QPS versus compression ratio tradeoffs in isolation of each other impedes pinpointing the reasons for differences in memory technology. We now crystallize findings below from our high-level application level metrics (QPS and compression ratio) and microarchitectural analysis.

DRAM versus PM Performance

We host the index in DRAM alone and PM alone in the experiments in this section. For AND H and M queries, we observe that the QPS performance gap between PM (index is fully hosted in PM) and DRAM (index is fully hosted in DRAM) narrows significantly for configurations with heavily compressed indices (Figure 6.3a), compared to the gap between (uncompressed) baseline DRAM and PM. For example, whereas the QPS gap for H between DRAM and PM is 21% for the baseline uncompressed index, it is only 2% with PForDelta and the SIMD variant of PForDelta. And it is only 4% for VByteDelta. We also note that the SIMD variant of PForDelta delivers a higher QPS, but we will discuss SIMD impacts in detail later.

Similarly, for M queries, the QPS gap with DRAM alone and PM alone is 26% with uncompressed indices. However, once the indices are heavily compressed, the gap is 5% for PForDelta and 10% for PForDelta256. Similarly, for VByteDelta, the difference in QPS between DRAM and PM is only 4%. L queries are short queries because the word typically has very few matching documents in the corpus. The baseline systems exhibit the same QPS. With PForDelta, PM is 9% slower than DRAM. The overhead of decompression now outweighs the locality benefits due to compression (more on this below), and we observe a different tradeoff with L queries. Overall, it is well known that search engines prioritize optimizing for tail latency, and long queries contribute to tail queries, and H queries are the longest. Therefore, for L queries, we believe compression impacts are less critical, and all systems with all configurations (compression or no compression) satisfy real-time response time constraints that search engines target.

Next, we discuss the reasons for the small gap in performance (QPS) between systems with DRAM only and PM only. The top-down analysis reveals that the retiring component (as shown in Figure 6.5 and Figure 6.6) is the same for uncompressed (baseline) DRAM and PM configurations. (We discuss AND H results here.) This component represents the number of retired instructions, and it is expected that the number of instructions executed by the baseline configurations is the same. The cycles lost to bad speculation (branch mispredictions) is the same as expected. The same is the case for the front-end bound, which represents the instruction fetch bandwidth. However, the one component where a PM-only system incurs a higher overhead than the DRAM-only system is the backend-bound component. This component is high either due to the backend of the pipeline being stalled due to missing ROB entries or lack of entries in the issue queue (reservation stations) or due to memory being unable to feed data with enough speed relative to the CPU's processing of memory instructions. In short, the component that is higher in PM only is the backend component, and our hypothesis is that this component is high due to memory latency. This component is 76% (H AND queries) higher for PM with no compression. Therefore, in the baseline system with no compression, the high latency of PM is indeed exposed to the application, which we verify later in this section.

For AND L queries, the DRAM-PM performance gap remains consistent across most configurations except for VByteDelta (Figure 6.3a). Compared to M queries, the performance gap between PM baseline and DRAM baseline is smaller. The top-down analysis in Figure 6.5 shows a similar breakdown across configurations for AND L queries, with a substantial increase in the backend component compared to AND H and M queries. This increase is likely due to more frequent non-sequential memory access patterns.

We note that for M queries, our above observations stand true. The performance gap between DRAM and PM is more pronounced for AND M queries compared to AND H queries (Figure 6.3a and Figure 6.3b). We attribute this to enhanced prefetching efficiency when the query evaluator frequently uses sequential access patterns. AND M queries involve more queries with fewer postings, which increases random access patterns.

The backend-bound component with M queries is almost twice to that in the DRAM case for M queries. However, for L queries, the backend bound component is similar in both DRAM and PM cases because memory accesses are less profound for L queries. Very few postings are traversed for L queries, as very few documents contain such words.

Many of our findings for AND queries are also true for SINGLE queries. For H queries, we still find similar performance for PM and DRAM with high compression. We find a difference with M queries, where the performance of PM and DRAM is more alike compared to AND queries, possibly due to improved prefetching. Prefetching likely improves because access patterns are more sequential and predictable, not interleaved between the postings blocks of two terms. This behavior is related to poor speculation, which is significantly higher for AND H queries (Figure 6.5 and Figure 6.6). Also, the backend bound component for SINGLE H queries is 2.3× that of DRAM only.

We observe that for compressed indices, the gap in the backend bound component between DRAM and PM is small. For example, the gap in the backend bound shrinks from 1.76× to 1.11× for PForDelta256 (AND, H). Similarly, the gap shrinks to 1.37× from 2.28× for one-term high-frequency queries. At this time, we have established that the QPS gap with DRAM and PM is diminished when compression is employed, and the reason is a reduction in backend pipeline stalls.

We note that the memory-bound pipeline slots for highly compressed configurations are comparable for both PM and DRAM (Figure 6.13a), indicating that the type of memory technology has little impact on these stalls. A further breakdown of the memory component shows that the external memory-bound component (Figure 6.14a) is significantly reduced for highly compressed configurations. This reduction explains the similar performance between PM and DRAM when indices are compressed. We believe it is due to two reasons. First, data compression decreases memory bandwidth pressure by reducing the volume of data loaded into memory. Second, it increases cache and TLB locality because more posting data fits in the cache. The benefits are akin to cache blocking in matrix multiplication, as an analogy.

The bottom line is that for AND H queries, the performance gap between DRAM and PM for PForDelta, PForDelta256, VByteDelta, and SVByteDelta is negligibly small (Figure 6.3a). These configurations utilize delta encoding to compress the index further. With these configurations, we can leverage PM's capacity advantage without sacrificing QPS performance.

SIMD Compression

We find that SIMD significantly enhances QPS performance. As shown in Figure 6.3, PForDelta256-DRAM consistently outperforms PForDelta-DRAM. For AND H queries, PForDelta256-DRAM has 30.7% higher QPS than PForDelta-DRAM. This improvement is due to SIMD accelerating decompression using vector instructions, which increases throughput.

The combination of compression and SIMD further enhances QPS performance despite the overhead of decompression. SIMD hides decompression costs and provides greater throughput, while effective compression reduces pressure on external memory. Notably, SVByteDelta-DRAM outperforms uncompressed DRAM baseline for both H and M queries (Figure 6.3a). For H queries, the QPS is 9% higher, which is unexpected given the typical additional costs associated with decoding integers. SVByteDelta achieves more retiring pipeline slots (dedicated to useful work) and compensates for the cost by reducing backend-bound slots (Figure 6.5a).

Figure 6.14a further details the reduction in backend-bound slots for H queries, showing a decrease in memory-bound pipeline slots. Additionally, Figure 6.9a breaks down memory-bound clock cycles, revealing a significant reduction in the external memory component when comparing SVByteDelta-DRAM to Base-DRAM. **This leads to an**

important insight: we can both improve performance and reduce the index size with a combination of SIMD and high compression.

Space-Time Tradeoff

We can host compressed indices on PM and achieve similar QPS performance, which is more scalable for larger indices. SVByteDelta-PM has 2.2% worse QPS performance than SVByteDelta-DRAM, but still 9.3% better performance than baseline DRAM.

PForDelta256-DRAM offers the best compression ratios while maintaining good QPS performance. It is only 4% slower than baseline DRAM, but compressing the index to 17.3% of its original size.

We analyze the time-space tradeoff between PForDelta256-PM and SVByteDelta-PM. PForDelta256-PM provides superior compression, reducing the index size to approximately half that of SVByteDelta-PM. However, QPS performance is about 10% lower because StreamVByte is more efficient. By using PForDelta256-PM, we can host larger indices in memory. When indices are too large to fit entirely in memory, they reside in secondary storage and utilize memory as a cache. Fetching postings from disk to memory adversely affects QPS. In this context, PForDelta256-PM allows more postings to be stored in the cache.

If the priority is minimizing index size, PForDelta and PForDelta256 should be preferred. If high performance (QPS) is critical, then SVByteDelta-DRAM or baseline DRAM is more suitable. **Balancing both aspects, PForDelta256 offers compelling trade-offs.** Therefore, we propose and use PForDelta256 for building APCache later in this thesis, our newly proposed caching framework with workload-aware prefetching for search engines.

6.3.3 Tail latency Results

Tail latency is an important metric for search engines. TABLE 6.1, TABLE 6.2 and TABLE 6.3 shows the 99th percentile tail latency (p99) results for AND and SINGLE queries.

For H queries, SVByteDelta has the lowest tail latency for both AND and SINGLE queries with DRAM and PM systems. Specifically, the tail latency for SINGLE, H is 201 ms on PM, and 195 ms on DRAM. Note that there is more memory bandwidth pressure on a DRAM system as it hosts the skip list and postings. Whereas, a PM system, still hosts the skip list in DRAM, while postings reside in PM.

Base (baseline uncompressed) exhibits relatively good tail latency performance on DRAM, but the worst tail latency results on PM (up to 252 ms compared to 217 ms with DRAM). This result is expected because of no compression. PForDelta256 has a slightly worse performance than SVByteDelta and has significantly lower tail latency than PForDelta. PForDelta has the worst tail latency results on DRAM.

For M queries, the tail latencies are much smaller than H. SVByteDelta has the best tail latency results, and VByteDelta has the worst. Finally, the tail latencies are much smaller for L queries than M. The results are similar for most configurations. PForDelta performs significantly worse on DRAM than other configurations.

Table 6.1: Tail latency results for H queries.

Configuration	**Tail Latency (E-1 s)**			
	DRAM		**PM**	
	AND	**SINGLE**	**AND**	**SINGLE**
Base	1.55	2.17	2.12	2.52
PForDelta	2.23	2.68	2.28	2.73
PForDelta256	1.56	2.14	1.61	2.19
SVByteDelta	1.34	1.95	1.43	2.01
VByte	1.57	2.20	2.09	2.66
VByteDelta	1.70	2.24	1.76	2.33

Table 6.2: Tail latency results for M queries.

Configuration	**Tail Latency (E-3 s)**			
	DRAM		**PM**	
	AND	**SINGLE**	**AND**	**SINGLE**
Base	1.46	2.39	2.19	2.76
PForDelta	2.64	3.05	2.73	3.11
PForDelta256	1.84	2.48	1.93	2.56
SVByteDelta	1.48	2.25	1.53	2.29
VByte	1.59	2.38	2.07	2.79
VByteDelta	4.12	4.15	4.30	4.28

6.4 Wikipedia Dataset

To substantiate our claims in this chapter, we analyze a different dataset that is more widely used for analyzing search engine performance. However, we initially found the dataset size to be small. Specifically, we experiment with a Wikipedia dataset to further validate our results. The uncompressed index size is 20.8 GB. The Wikipedia dataset's results closely resemble those we observe with the CommonCrawl dataset above.

Our findings reaffirm our previous conclusion: QPS performance is similar for both PM and DRAM under high compression. For AND H queries, QPS performance is only 0.76% lower for PForDelta256-PM compared to PForDelta256-DRAM, and 1.68%

Table 6.3: Tail latency results for L queries.

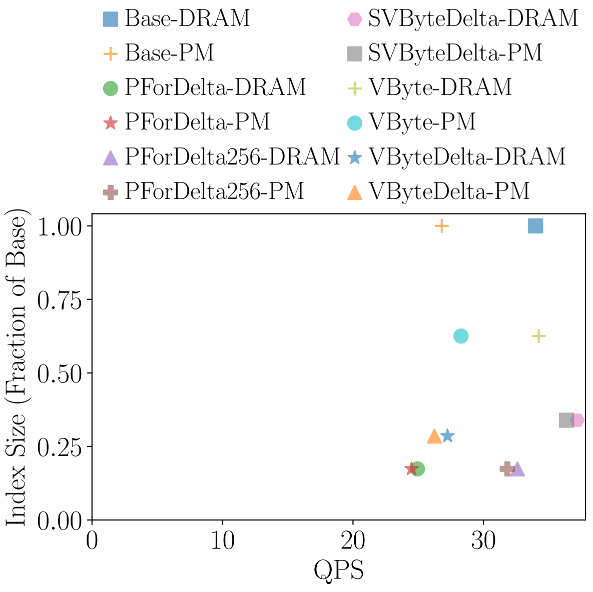
Configuration	Tail Latency (E-5 s)			
	DRAM		PM	
	AND	SINGLE	AND	SINGLE
Base	1.00	1.50	1.00	1.50
PForDelta	1.00	1.60	1.19	1.79
PForDelta256	1.00	1.60	1.19	1.72
SVByteDelta	1.00	1.60	1.19	1.69
VByte	0.91	1.50	1.00	1.60
VByteDelta	1.10	1.69	1.19	1.72

lower for SVByteDelta-PM compared to SVByteDelta-DRAM (Figure 6.11a). For SINGLE H queries, the gap is slightly smaller, QPS performance is only 0.4% lower for PForDelta256-PM compared to PForDelta256-DRAM, and 0.49% lower for SVByteDelta-PM compared to SVByteDelta-DRAM (Figure 6.11b). Figure 6.14 shows that the external memory bound component is similar for PM and DRAM configurations with high compression.

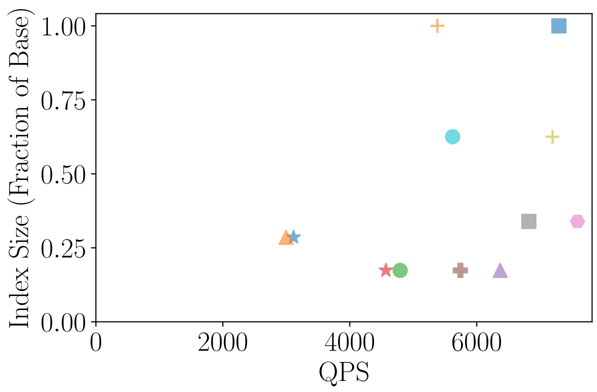
Once again, we find that SIMD enhances performance. In Figure 6.11, we see that PForDelta256-DRAM offers 15.9% better performance than PForDelta-DRAM for AND H queries and 8.3% better performance for SINGLE H queries. We observe in Figure 6.12 that retiring pipeline slots are lower for PForDelta256 than PForDelta due to fewer instructions.

We find once more that SIMD compression can outperform no compression, even when accounting for decompression costs. For AND H queries, QPS performance for SVByteDelta-DRAM is very slightly faster than Base-DRAM (Figure 6.11a), and for SINGLE H queries, it is 0.1% faster (Figure 6.11b).

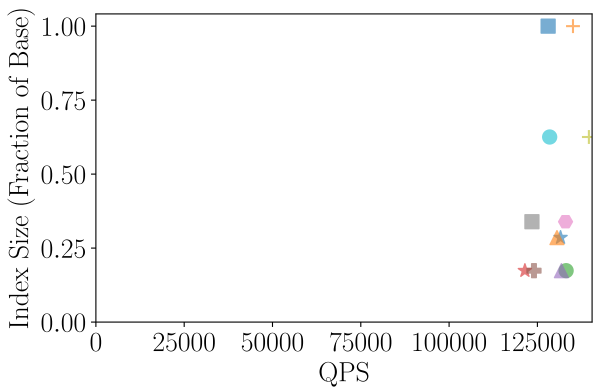
For the Wikipedia dataset, the time-space trade-off favors PForDelta256, as its performance is closer to that of SVByteDelta. QPS performance for PForDelta256-PM is 4.9% worse for For AND H queries, and 2.9% worse for SINGLE H queries compared to SVByteDelta-PM (Figure 6.11).



(a) H queries.

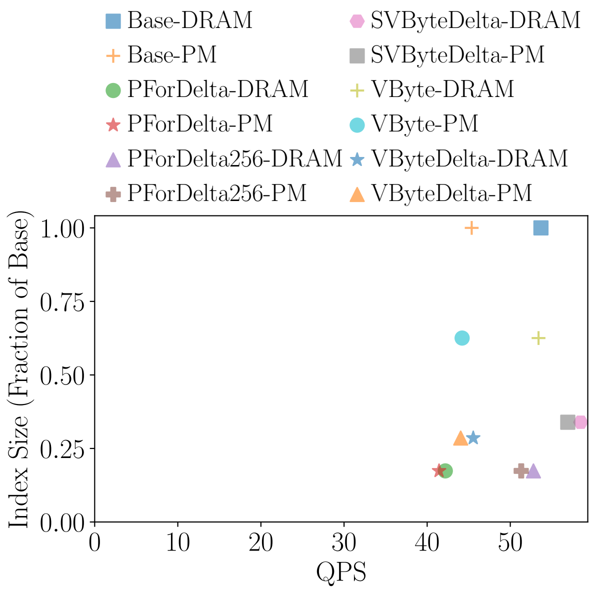


(b) M queries.

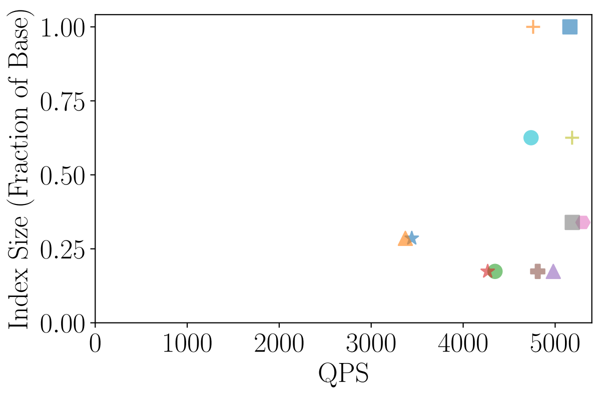


(c) L queries.

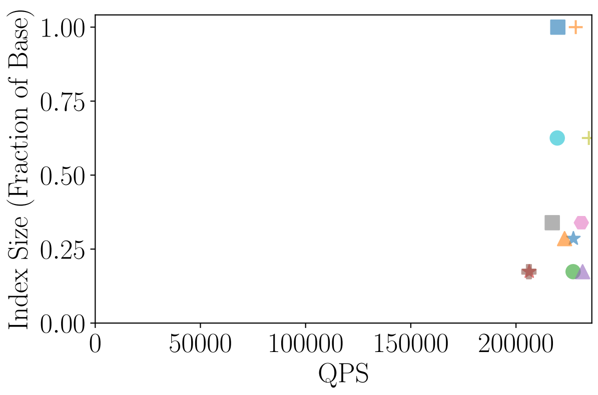
Figure 6.3: Showing the QPS versus compressed index size (as a fraction of uncompressed index size) for AND queries.



(a) H queries.

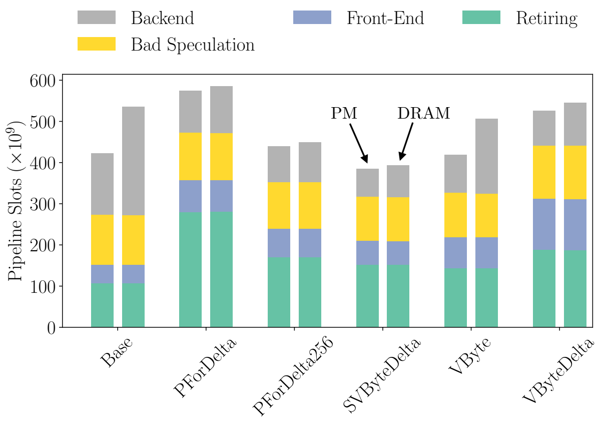


(b) M queries.

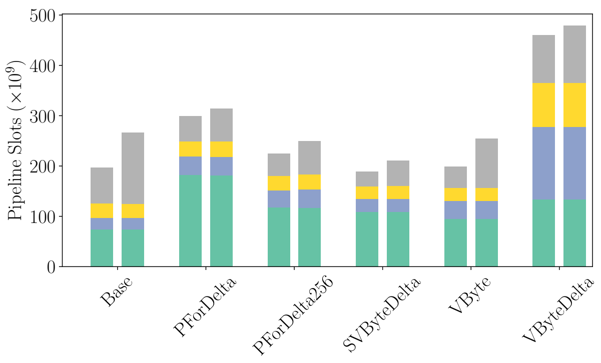


(c) L queries.

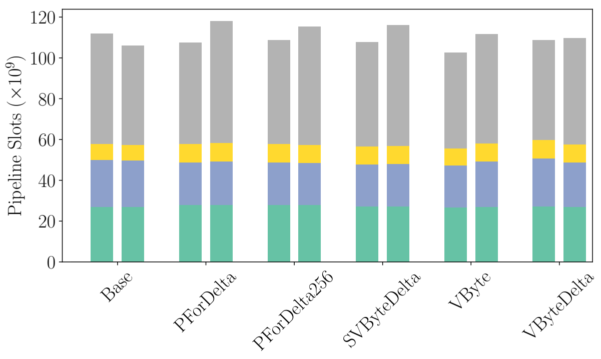
Figure 6.4: Showing QPS versus compressed index size (as a fraction of uncompressed index size) for one-term (also called SINGLE) queries.



(a) H queries.

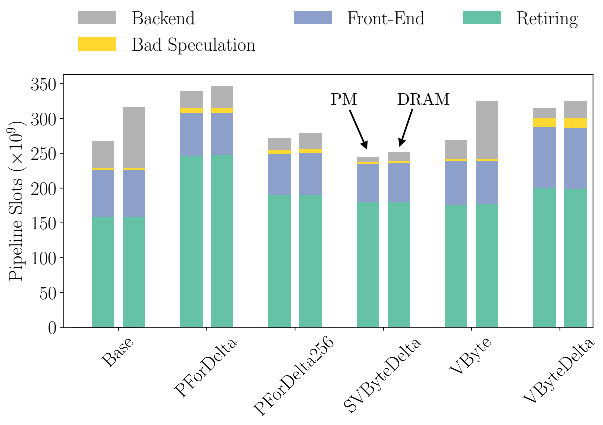


(b) M queries.

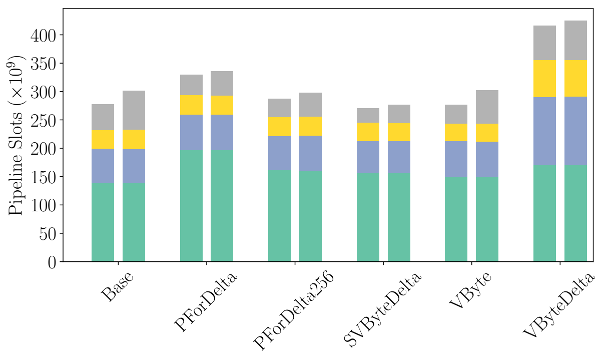


(c) L queries.

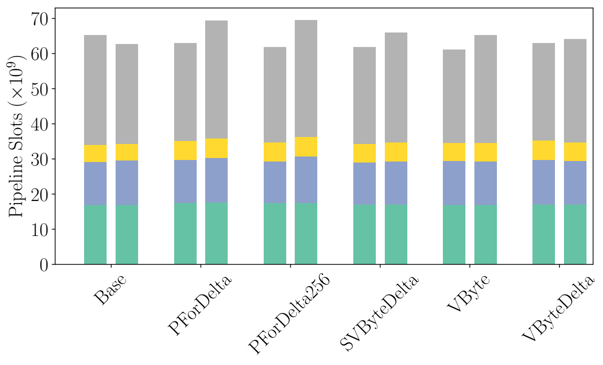
Figure 6.5: Showing the top-down breakdown for AND queries.



(a) H queries.

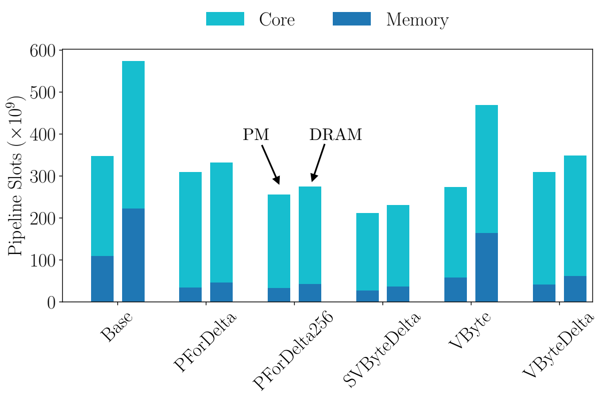


(b) M queries.

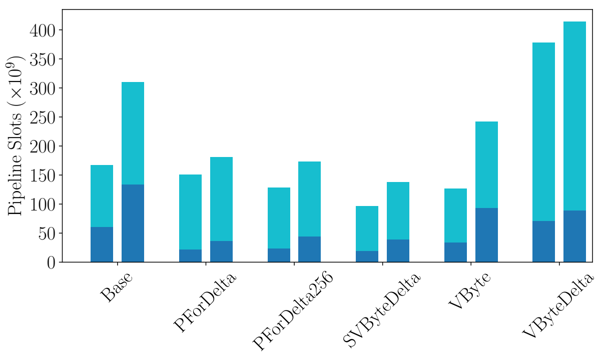


(c) L queries.

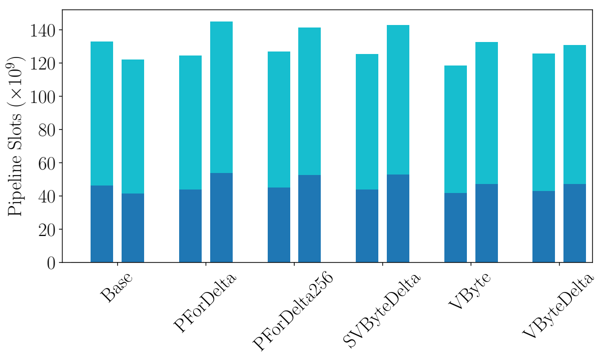
Figure 6.6: Showing the top-down breakdown for SINGLE queries.



(a) H queries.

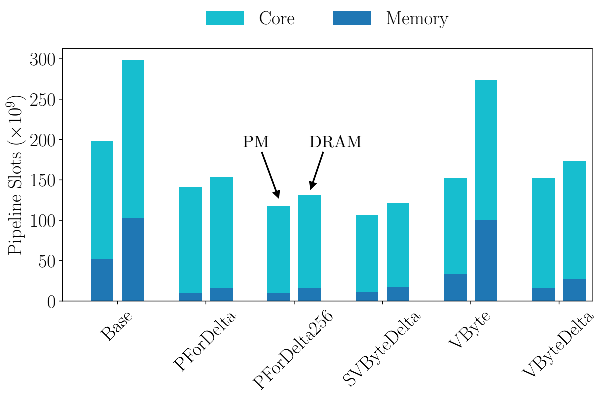


(b) M queries.

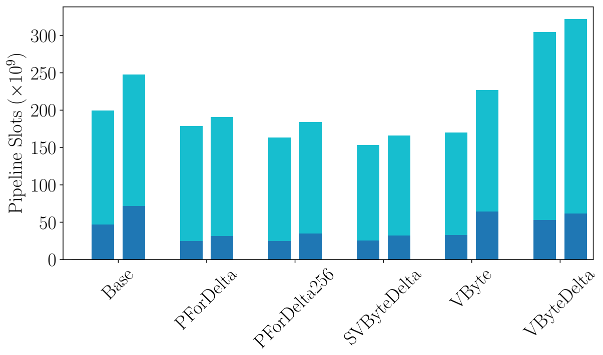


(c) L queries.

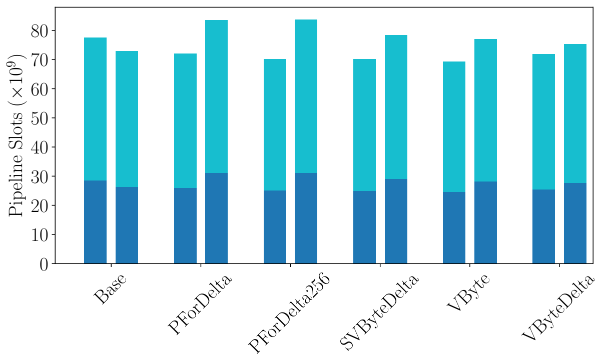
Figure 6.7: Showing the backend-bound breakdown for AND queries.



(a) H queries.

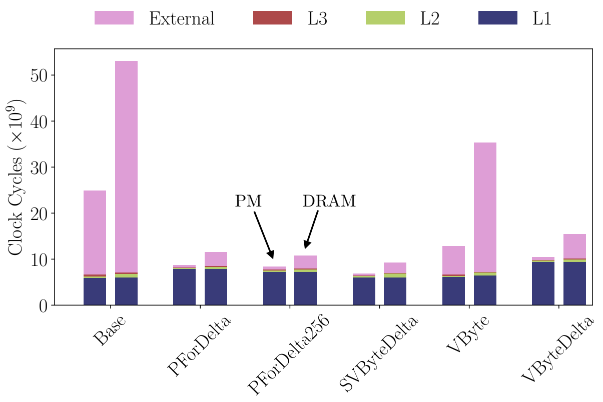


(b) M queries.

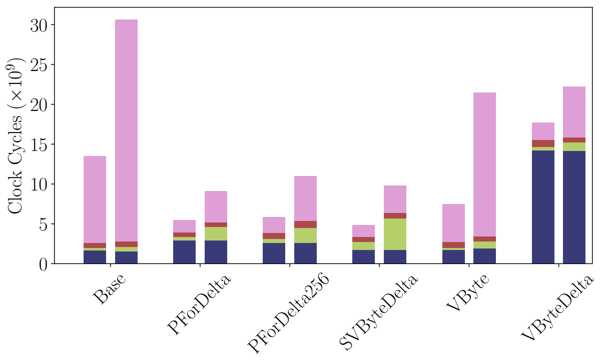


(c) L queries.

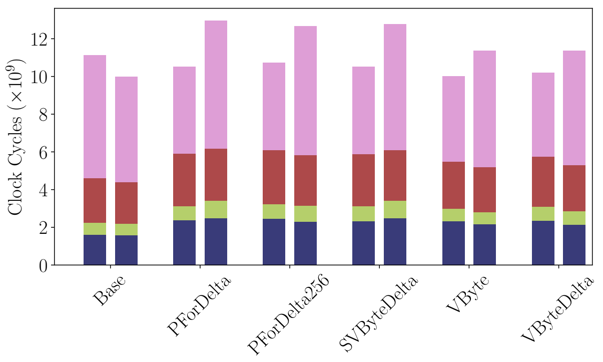
Figure 6.8: Showing the backend-bound breakdown for SINGLE queries.



(a) H queries.

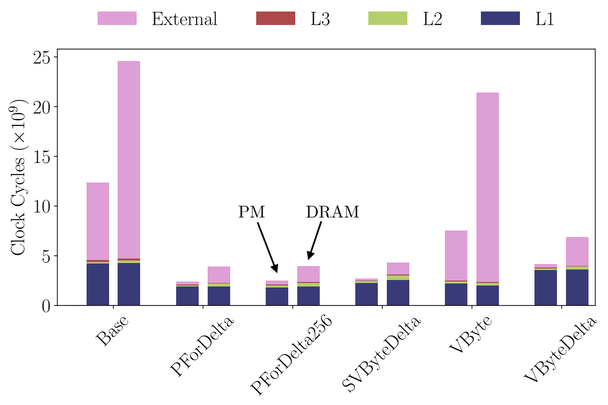


(b) M queries.

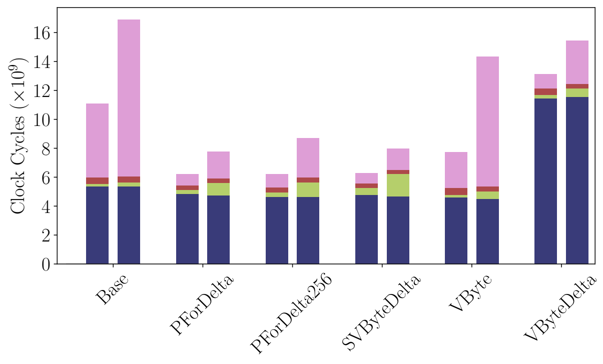


(c) L queries.

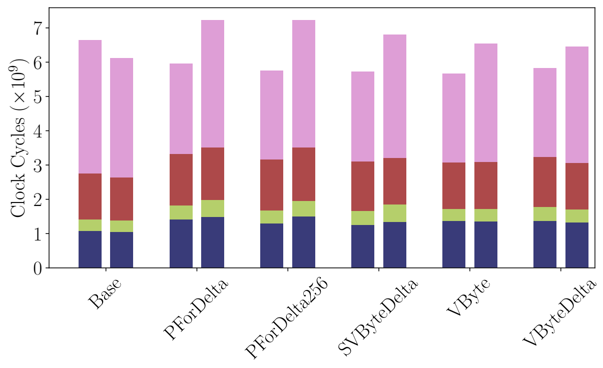
Figure 6.9: Showing the memory-bound breakdown for AND queries.



(a) H queries.

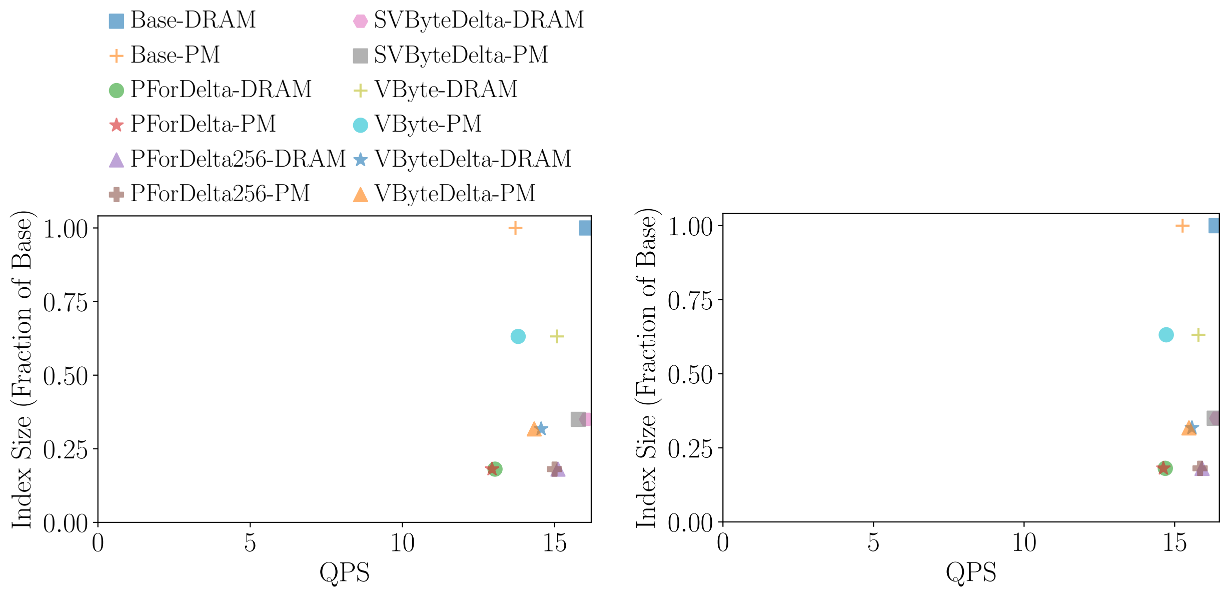


(b) M queries.



(c) L queries.

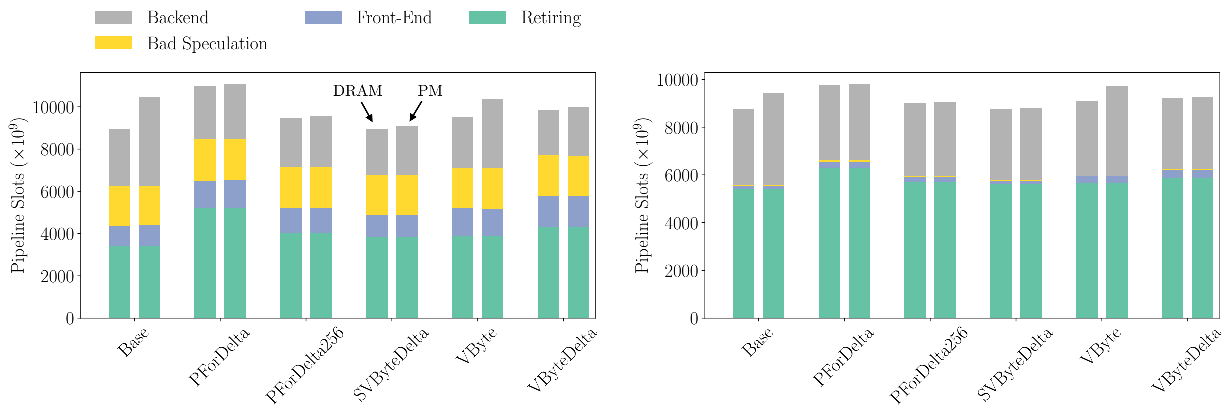
Figure 6.10: Showing the memory-bound breakdown for SINGLE queries.



(a) AND H queries.

(b) SINGLE H queries.

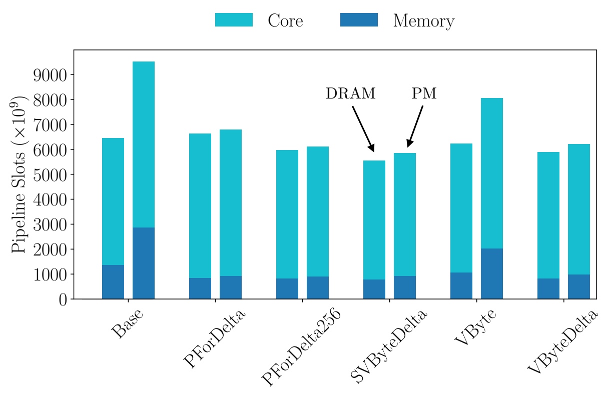
Figure 6.11: QPS vs. compressed size for the Wikipedia dataset.



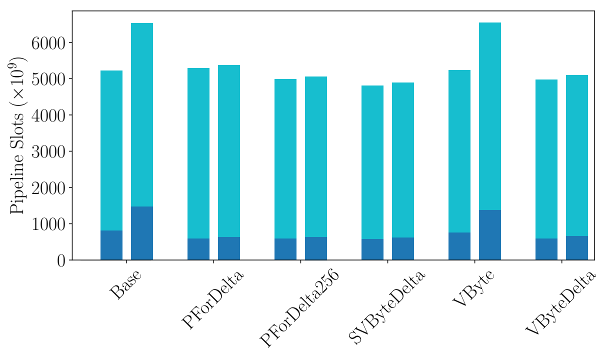
(a) AND H queries.

(b) SINGLE H queries.

Figure 6.12: Top-down breakdown for the Wikipedia dataset.

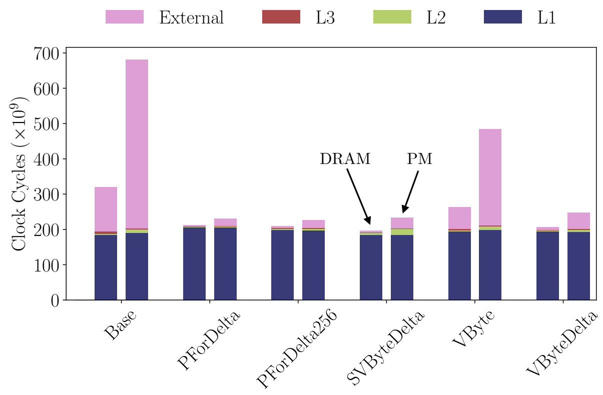


(a) AND H queries.

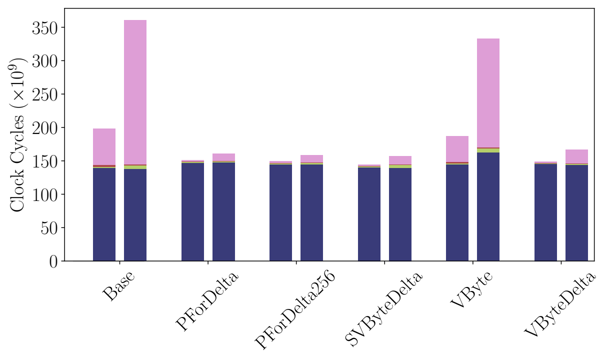


(b) SINGLE H queries.

Figure 6.13: Backend-bound breakdown for the Wikipedia dataset.



(a) AND H queries.



(b) SINGLE H queries.

Figure 6.14: Memory-bound breakdown for the Wikipedia dataset.

Chapter 7

Design and Implementation of APCache

We now present APCache, our newly proposed adaptive cache over slow memory tier (and associated machinery) for disk-resident indices that outgrow DRAM capacity by an order of magnitude. Our design goals are inspired by our investigation of the space-time tradeoffs in hosting compressed/uncompressed indices in scalable and byte-addressable NVM. Specifically, we find that compression algorithms with high compression ratios deliver similar QPS performance for DRAM and PM-backed indices. We first discuss key principles behind the design of APCache, and then provide the details of the architecture of APCache.

7.1 Principles

The design of APCache follows six principles.

Maintain a high compression ratio. The index is compressed using TurboPForDelta256, providing high performance and significant compression. Although StreamVByte and StreamVByteDelta offer better QPS performance, we choose TurboPForDelta and TurboPForDelta256 for APCache because they achieve a compressed index size of approximately half as large.

Use memory-mapped I/O. APCache leverages memory-mapped I/O (MMIO) for reading the immutable index from storage. MMIO uses virtual memory translation to read the index and performs better than system call I/O for large indices.

Keep the I/O stack unmodified. Transfer postings from the DRAM OS cache to the custom NVM cache using an unmodified storage stack for maximum applicability.

Use query-workload-aware prefetching. APCache exploits deep knowledge about query workload to prefetch postings for terms ahead of the query evaluation engine. It employs a workload-aware prefetcher with an adaptive lookahead policy to cache the necessary NVM index portions.

Use a cheap second-tier for caching. APCache uses a cheap and dense second tier of memory for hosting the index in memory. We opt for PM over DRAM for three key reasons: (1) serving queries from PM and DRAM yields similar QPS performance, (2) PM offers much higher capacities (8 times larger), allowing for larger cache sizes, which reduces costly cache misses that result in high-latency accesses to secondary storage, and (2) using PM allows us to conserve DRAM capacity by setting a hard limit on the amount of DRAM available to the program.

Use a fixed DRAM budget. APCache uses a fixed DRAM budget, and the DRAM cost does not increase proportionally to the index size.

7.2 Architecture of APCache

In this section, we discuss the architecture of APCache. We first discuss our baseline system and how we optimize the baseline by adding prefetching, and then discuss the design of our new system.

7.2.1 Baseline System

There are three important considerations in our baseline system. First, the design of the index is the same as discussed in Section 4. Specifically, the DID and frequency block files reside on secondary storage (hard disk or SSD). DID blocks are compressed using TurboPForDelta256, while frequencies are compressed with TurboPFor256.

The second important consideration is the choice of I/O mechanism. Linux broadly provides two I/O mechanisms: one based on system calls and the other on virtual memory (called memory-mapped I/O or MMIO). System calls incur an overhead, and one cannot compute over the storage-resident file. Therefore, we use MMIO to read the compressed index for storage. One concern with MMIO is that writes to disk happen in an unpredictable order, complicating the implementation of database consistency guarantees. However, it is not a concern in our work as indices are built offline in a separate step, and thus, there are no writes to the index during reads (query evaluation). Although Linux offers asynchronous APIs for I/O that can outperform MMIO in some scenarios, these asynchronous APIs complicate application design and exhibit security vulnerabilities. We leave the evaluation of new I/O APIs in Linux to future work and use the established MMIO mechanism in this work.

Finally, we must decide on the skip list and dictionary placement. In our baseline system (*Base*), we experiment with different placement strategies for the skip list and the dictionary and find that PM is the most suitable location. Consequently, we use PM as the default skip list and dictionary location in all our systems.

The Base configuration uses memory-mapped I/O to serve queries from a disk-resident index, as illustrated in Figure 7.1. As discussed above, we experiment with different storage locations for the skip list and skip list offset dictionary: ① secondary storage (same location as the index), ② DRAM, and ③ PM.

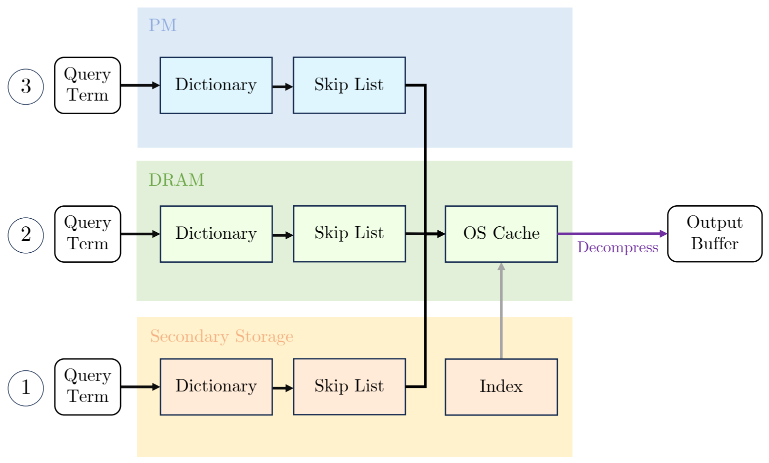


Figure 7.1: Showing the baseline system with skiplist on (1) secondary storage, (2) DRAM and (3) PM.

7.2.2 Optimizing the Baseline System

To mitigate the high latency of secondary storage, we enhance the baseline system described above to add workload-aware prefetching. We note that the prefetching mechanism we introduce is called manual prefetching. The OS has an automatic prefetching mechanism for reading pages from storage ahead of the read requests. The OS page cache works the best for sequential access patterns, but where the I/O workloads are random and mixed sequential and random, like any prefetching mechanism, the page cache cannot predict the access patterns precisely. Therefore, in this work, we introduce workload-aware prefetching, where we prefetch postings for terms for future queries.

One obvious concern with such workload-aware prefetching is predicting the query terms beforehand. We exploit the fact that typically, in systems today, client requests are queued in user and kernel buffers. For example, queuing client requests is why TCP protocol has a backlog parameter specifying the maximum number of queued connections. Sending client requests is a much cheaper operation than evaluating the same requests. Hence, requests are queued up in a typical system, and the system can thus look ahead at the requests than those currently being served.

In our system (we call it *BasePrefetch*), we employ an an additional *worker* thread to generate I/O and prefetch blocks from secondary storage into the page cache (DRAM) for the (main) query thread. We show this improved baseline system in Figure 7.2. As shown, I/O now happens concurrently with query resolution, improving the overall throughput of the search engine.

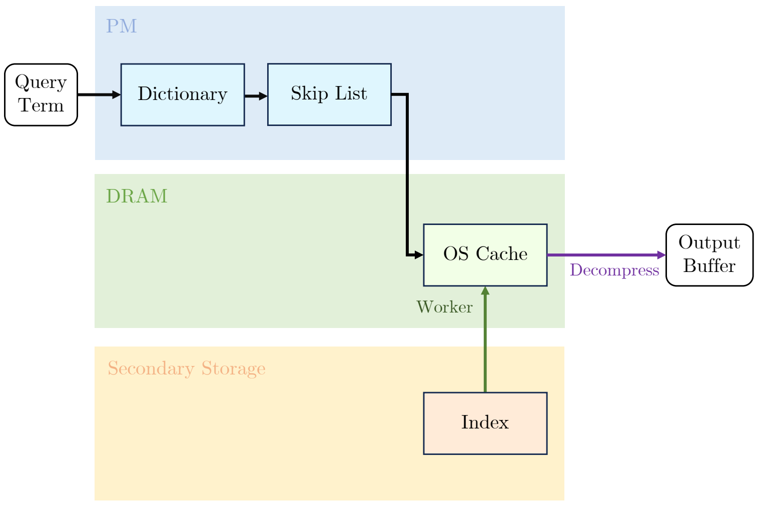


Figure 7.2: Showing the BasePrefetch system.

One important concern with prefetching for query terms ahead of the main query thread is that needless evictions could happen before the query thread resolves the (current) query. Unfortunately, we do not have control over modifying the internals of the page cache. Our principle is to leave the I/O stack and OS code unmodified to maximize deployment potential. Therefore, we solve this problem in a different way. Specifically, we limit the lookahead amount to prevent the prefetcher from getting too far ahead of the current query and causing evictions for postings before the query thread can access them, which are useful for query resolution. We use a configurable parameter that caps the maximum number of queries the prefetcher can bypass at any given time.

Our optimized baseline system, BasePrefetch, thus employs a fixed lookahead policy, maintaining a constant value for this parameter throughout the search engine's execution. The decision to employ a fixed lookahead policy is crucial due to the nature of the page cache that we use as our prefetching target. The challenge lies in not knowing when or which data has been evicted or how many segments cannot be evicted, as some page cache data is reserved for user structures.

Additionally, the worker thread prefetches the skip list offset from the skip list offset dictionary into a dedicated prefetch buffer, which avoids race conditions that could arise if both the query and worker threads simultaneously access the skip list offset dictionary.

The worker thread prefetches a term's postings into the page cache by copying portions of the index into a small fixed-sized user buffer of 4096 bytes. Compared to another prefetching method which uses posix_fadvise(), copying the file data to a user buffer guarantees that the data is brought into user space and thus into the page cache, gives us explicit control over reading and managing the data, and can be more portable as it relies on standard file I/O operations.

Finally, we must decide when queries are processed in our optimized baseline system. In our implementation, the query thread waits for a query term, or all terms of an AND query, to be prefetched before processing the query.

7.2.3 Hybrid APCache

We now discuss the design of APCache. We first discuss a naive implementation of APCache we call *Hybrid*. In the next section, we discuss an optimized variant that uses a *smart* prefetching and lookahead policy. First, we call our system Hybrid to stress the fact that the postings are cached in hybrid DRAM and PM memory. Note that in the baseline systems, the postings are only cached in the DRAM page cache. Figure 7.3 illustrates the flow of data in Hybrid.

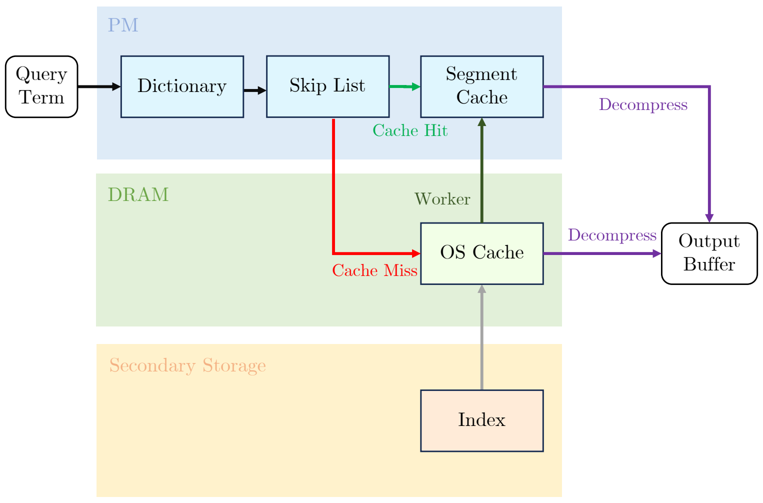


Figure 7.3: Showing the Hybrid APCache system.

We add a segmented posting cache backed by PM in Hybrid. Although any cheap (and slow) memory tier can be used in place of PM, we use it in this work due to its availability in our in-house systems. In Hybrid, we need a mechanism to move postings from the DRAM cache to our new segmented PM cache. Linux and other OSs currently do not provide a mechanism to use a multi-tiered OS page cache. Therefore, we use an additional worker thread to cache postings from DRAM to PM. Thus, we transfer postings for terms encountered by the query thread from DRAM to PM at the user level, leaving the OS unmodified. The query thread serves queries from the PM cache on a PM cache hit. However, a cache miss necessitates memory-mapped I/O to the disk-resident index. We note that in Hybrid, the size of the DRAM is fixed, which is a great advantage of our new system. As indices grow, the DRAM capacity requirements in Hybrid stay constant, placing a cap on the DRAM cost of the search infrastructure in the future.

To prevent external fragmentation, we have designed a segmented cache with fixed seg-

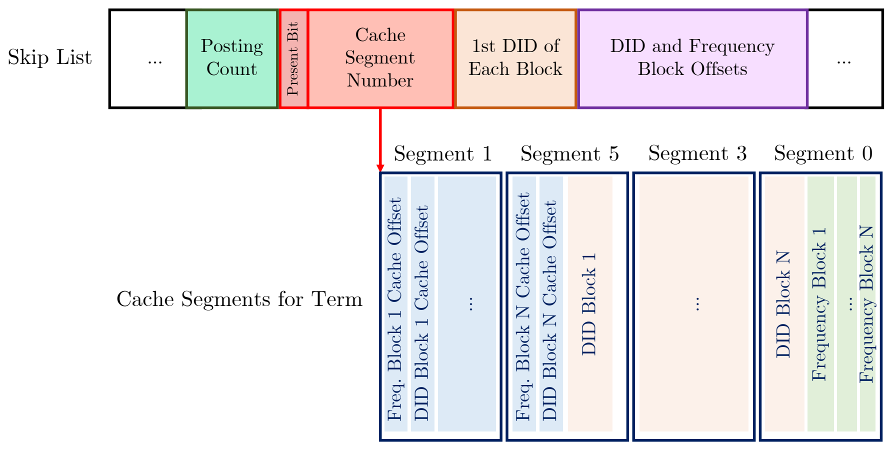


Figure 7.4: Skip list design for cache configurations.

Global Segment List

Index (Segment #)	0	1	2	3	4	5	6	7	8
Next Segment #	7	5	4	-1	-1	3	0	8	-1

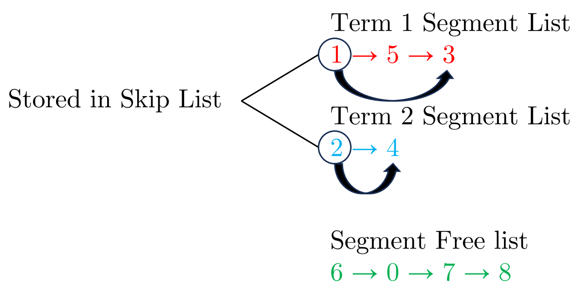


Figure 7.5: Array-based linked list design for segment lists and free list. The example shown is for a cache with eight segments.

ment sizes (see Figure 7.4). This fragmentation can occur because terms have a variable number of postings, and eviction requires removing all postings associated with a term. When evicting, the available memory may be fragmented into smaller pieces that cannot be effectively utilized for storing new terms, even though the total memory might be sufficient.

We utilize a global segment list shared among all terms in the cache. This global list includes smaller lists dedicated to each term, indicating which segments store the cached data for that term and a single list of free segments available for allocation. We implement an array-based linked list structure for storing the global segment list to optimize memory usage and avoid dynamic memory allocation. Figure 7.5 demonstrates this setup, where the array-based linked list holds segment lists for Term 1 (indices 1, 3, 5) and Term 2 (indices 2, 3), along with the segment free list (indices 0, 6, 7, 8). The number stored at a given index is the next segment in the segment list, e.g. we store the number 5 at index 1, indicating that segment 5 is the next segment in the list after segment 1. We set the next segment to -1 to indicate a segment is the last segment of the list.

Access to a term's segment list is facilitated by the array index of its first segment stored in the term's skip list section. For example, in Figure 7.5, the number 1 (for segment 1) would be stored in the skip list for Term 1. To support this, we enhance the original skip list by allocating an additional 4 bytes per term: 1 bit to indicate if the term is cached, and 31 bits to store the array index of the first segment (see Figure 7.4).

Access to the free list is achieved by storing a global pointer to the head of the free list.

When the worker thread needs additional free segments, but none are available (i.e., the free list is empty), we perform a cache eviction. We use an LRU eviction policy. We evict all segments associated with a batch's least recently used term. We track the LRU term using a doubly linked list. We move the recently touched or newly appended nodes to the beginning of the list while we select the term at the tail for eviction. Eviction involves inserting a term's segment list into the segment free list, which we achieve in constant time through pointer rearrangement. To achieve this, we store additional pointers from the first node of a term's segment list, including a pointer to the last segment. For example, in Figure 7.5, we store an additional pointer from segment 1 of Term 1 to segment 3. When evicting Term 1, we update the global free list pointer to 1 (segment 1), and set index 3 to 6, to add the original free list segments back to the free list. After eviction, the present bit in the cache that informs us whether the term is cached (in the skip list) is set to zero.

An important decision in our design is that the worker thread caches all postings for a term, including those skipped by the query thread by way of a skip list. We make this decision because for AND queries (which utilize skipping) that have two terms, the term a repeated term may be paired with may be different. This may require skipped postings that must now be loaded into the cache, which is more expensive. We notice that not many blocks are skipped, so the space overhead is not massively increased, but we gain this benefit.

An important decision in our design is that the worker thread caches all postings for a term, including those skipped by the query thread via the skip list. We made this decision for AND queries, which utilize skipping, because repeated terms may be paired with a different term. This can necessitate loading postings that were previously skipped into the cache, which is a more costly operation. However, we observed that skipping occurs infrequently, so the additional space overhead is minimal compared to the benefits gained.

Once a term is cached, we need a way to retrieve the positions of individual blocks within the cache. In our first iteration of the design of Hybrid, the block offsets in the skip list served this purpose. We instead store the new cache block offsets in the segmented PM cache, specifically in the first segment or the first few segments of a term's segment list. This decision optimizes eviction, as modifying the original offsets would require restoring them to point to storage-resident blocks, which is costly.

The query thread communicates with the worker thread about which terms to cache using a job queue. Each term encountered by the query thread enqueues the term's skip list offset. Processing each job in the queue involves the following steps.

1. The worker checks if the term's postings are already in the cache.
    a) The worker skips caching-related work if the term is already present in the cache (cache hit).
    b) If the term is not present in the cache, it is read from storage and cached as follows.
        i. Cache segments are allocated for the cache block offsets.
        ii. The term's compressed DID blocks are copied into the cache, updating the cache block offsets.
        iii. The term's compressed frequency blocks are copied into the cache, updating the cache block offsets.
        iv. The term's cache present bit in the skip list is set, and the cache segment number. The cache segment number represents the term's first segment in its segment (linked) list.
2. The worker moves the term to the MRU position in the eviction linked-list.

7.2.4 Optimizing the Hybrid APCache

We now present the design of optimized Hybrid that adds prefetching to Hybrid. Our novel prefetching mechanism adapts to the query workload and thus utilizes the PM cache in an optimal manner. Specifically, we avoid needless evictions, and also prevent needless I/O to the storage device. More importantly, we automatically tune the lookahead parameter in BasePrefetch by exploiting knowledge about the internal details of our PM cache. Eliminating a parameter and tuning it automatically eases system deployment.

We add prefetching to the Hybrid configuration to hide the latency of secondary storage. With MMIO, reading fresh postings not present in the OS cache or the PM cache would incur high latency as MMIO reads are synchronous. Synchronously reading data from storage in the application code is expensive. We call this optimized Hybrid system, *HybridPrefetch*. In this optimized system, the worker thread prefetches posting blocks into the PM cache. This system and the flow of data is shown in Figure 7.6. We also

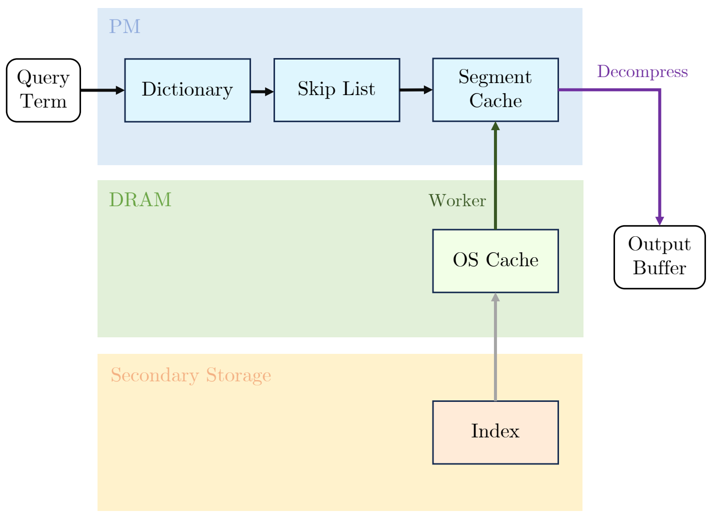


Figure 7.6: Showing the optimized APCache system called HybridPrefetch.

provide a flowchart which shows the execution path for the query and worker threads in Figure 7.7

A key difference between Hybrid and HybridPrefetch is that, unlike Hybrid, queries are exclusively served from the PM cache in HybridPrefetch. This places the burden of handling query resolution traffic on only of tier of memory (the slow tier in our case). The DRAM tier is used only to enable a path from storage device to the cache.

The query thread waits for a query term (or both terms in the case of AND queries) to be prefetched before processing the query. Similar to BasePrefetch, the worker loads the skip list offsets of the terms it processes into a shared buffer (which is later accessed by the query thread) to minimize redundant work.

Similar to BasePrefetch, we must limit the lookahead amount, i.e., the maximum number of queries the prefetcher can be ahead in order to prevent evictions of terms before the query thread can serve them. However, unlike BasePrefetch, we can enhance the lookahead policy because we have insight into the cache's internal structure. We employ an adaptive lookahead policy that dynamically adjusts the limit based on the number of available cache segments at any given time. We accomplish this dynamic limit by tracking the number of segments the worker has cached ahead of the query thread. This count provides a lower bound for the number of segments that cannot be evicted. We use a rolling sum to track this number and a FIFO queue, which stores the cache segment count for each term.

We explain adaptive prefetching through a simplified example illustrated in Figure 7.8. In this example, we show a cache with eight segments and a stream of unique query terms. We track the rolling sum and the tail term at each time step:

1. *time step = 0*: Initially, the query and worker threads are at the same position,

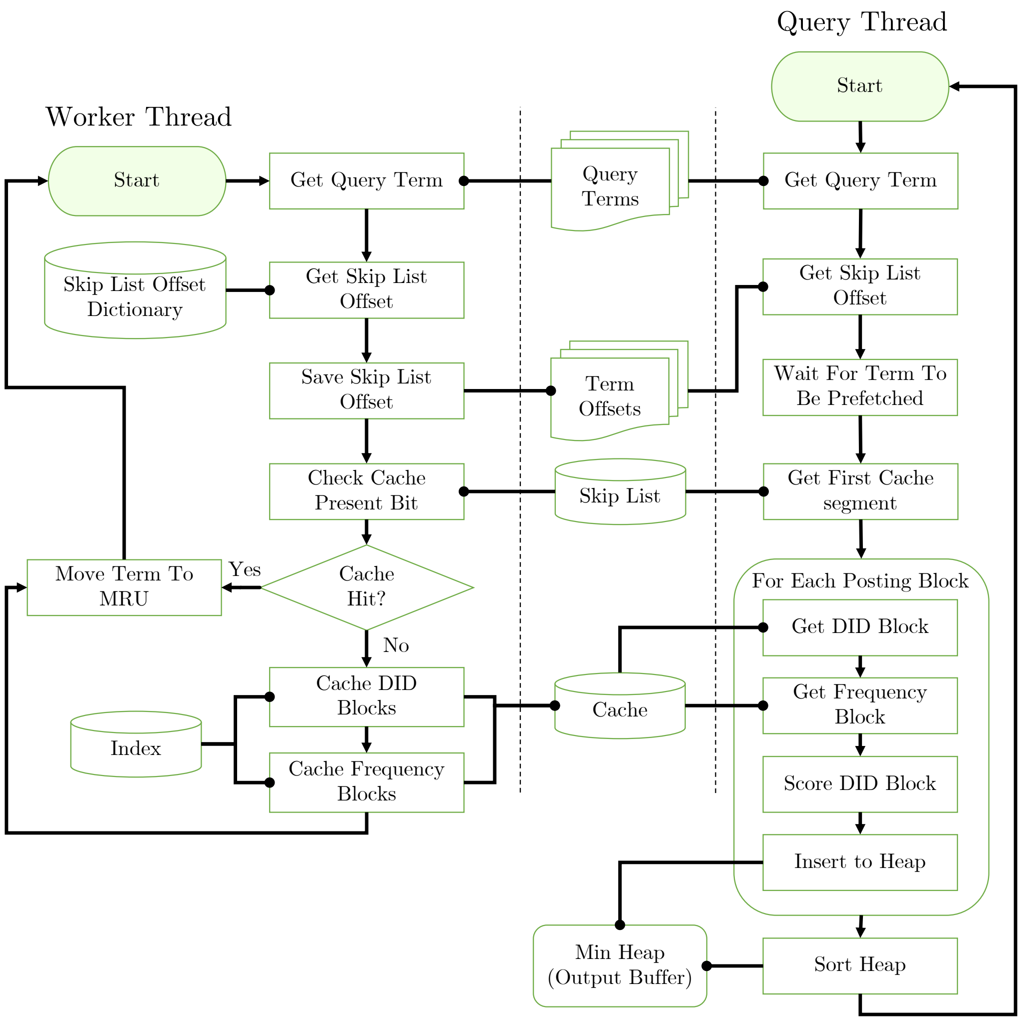


Figure 7.7: Flowchart for HybridPrefetch showing how key decisions are made and flow of data until query is resolved.

and the rolling sum is zero.

2. *time step = 1*: The worker thread prefetches the first term t0 into the cache. t0 requires four cache segments. This number is added to the rolling sum, and to the end of the segment count queue. Term t0 is added to the LRU list (for our eviction policy).

3. *time step = 2*: The query thread starts working on t0. Meanwhile, the worker thread finishes caching t1.

4. *time step = 3*: The query thread is still working on t0, and the worker thread finishes caching the first three segments of t2. At this point, the worker needs to

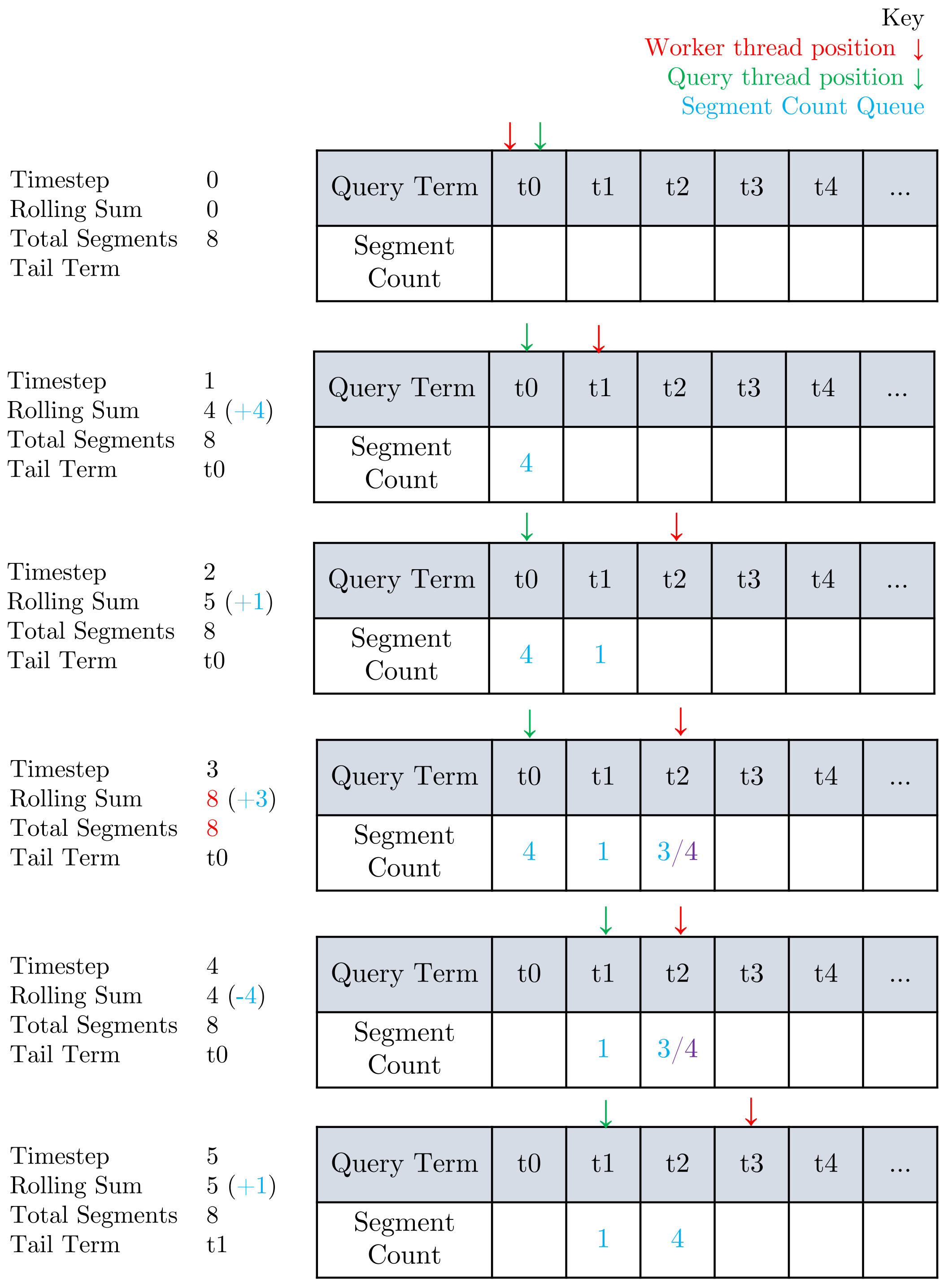


Figure 7.8: Example of adaptive prefetching for a cache with eight segments. The query and worker thread positions (arrows) indicate the current term the thread is currently operating on but has not completed.

evict the tail term (t0) to make room for the fourth segment of t2. However, the query thread is currently working on t0, and the worker must wait. The waiting condition is:

$$\textit{rolling sum} + \textit{tail term segment count} > \textit{total segments}$$

Unfortunately, we cannot use the waiting condition

$$\textit{rolling sum} > \textit{total segments}$$

because of a special case we discuss later.

5. *time step = 4*: The query thread finishes t0, pops the segment count for t0 from the front of the queue, and subtracts it from the rolling sum, freeing up segments for the worker to evict.

6. *time step = 5*: At the start of timestep 5, the waiting condition

$$4 + 1 > 8$$

is false and the worker evicts t0 and finishes caching t2 with one more segment.

Even when encountering repeated terms, we add the segment count to the rolling sum to avoid the situation in Figure 7.9. In this example, the term t0 is repeated. If we do not add to the rolling sum and segment count queue for repeated terms, we may end up with the situation depicted in time step 3, where the rolling sum is zero. Our waiting condition is false ($0 + 4 > 8$), meaning the worker thread could possibly evict the LRU term, t0, which is currently being processed by the query thread. To avoid this, the rolling sum should be incremented by 4 in time step 2.

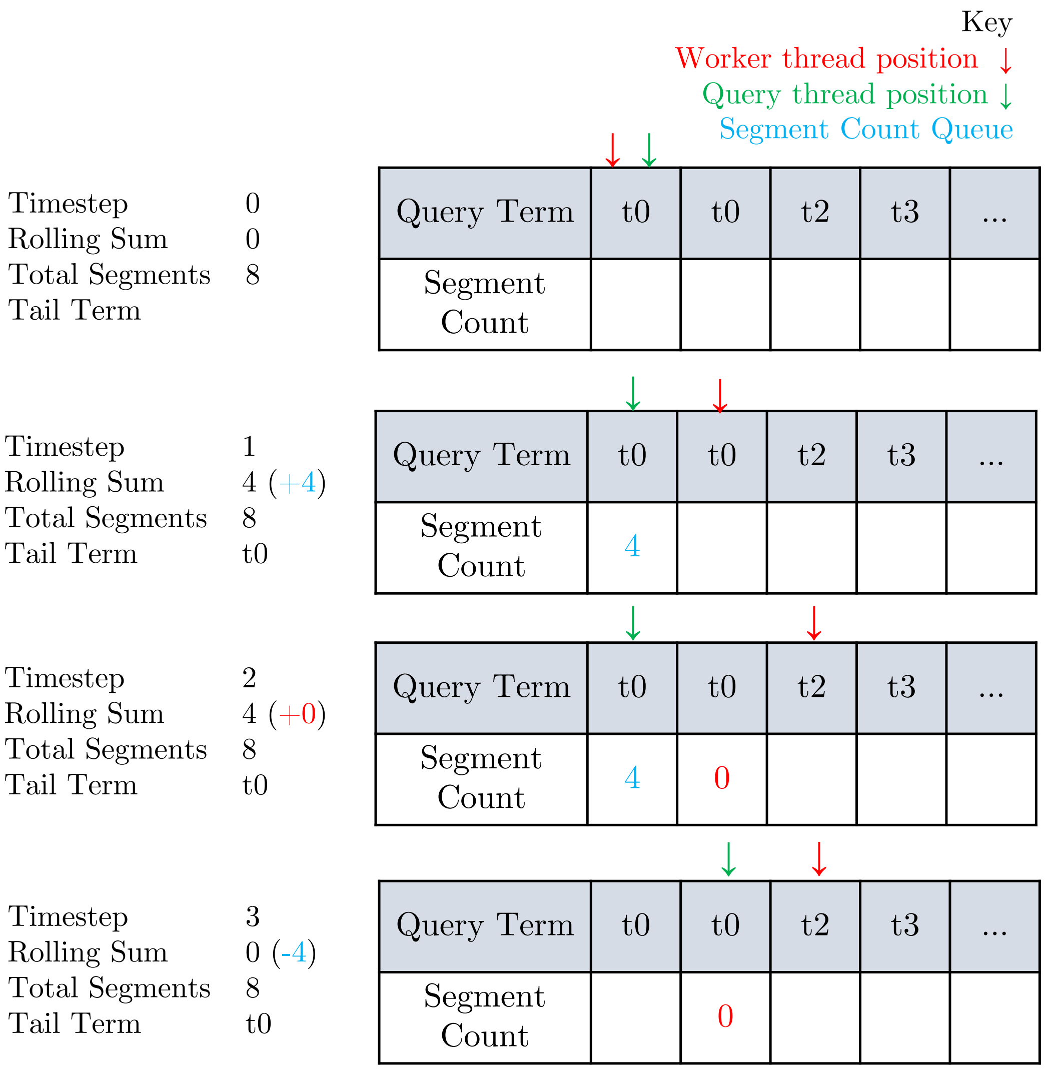


Figure 7.9: Incorrect example of segment sum for repeated terms for a cache with 8 segments. The rolling sum should be incremented by 4 in time step 2. The query and worker thread positions indicate the current term the thread is currently operating on but has not completed.

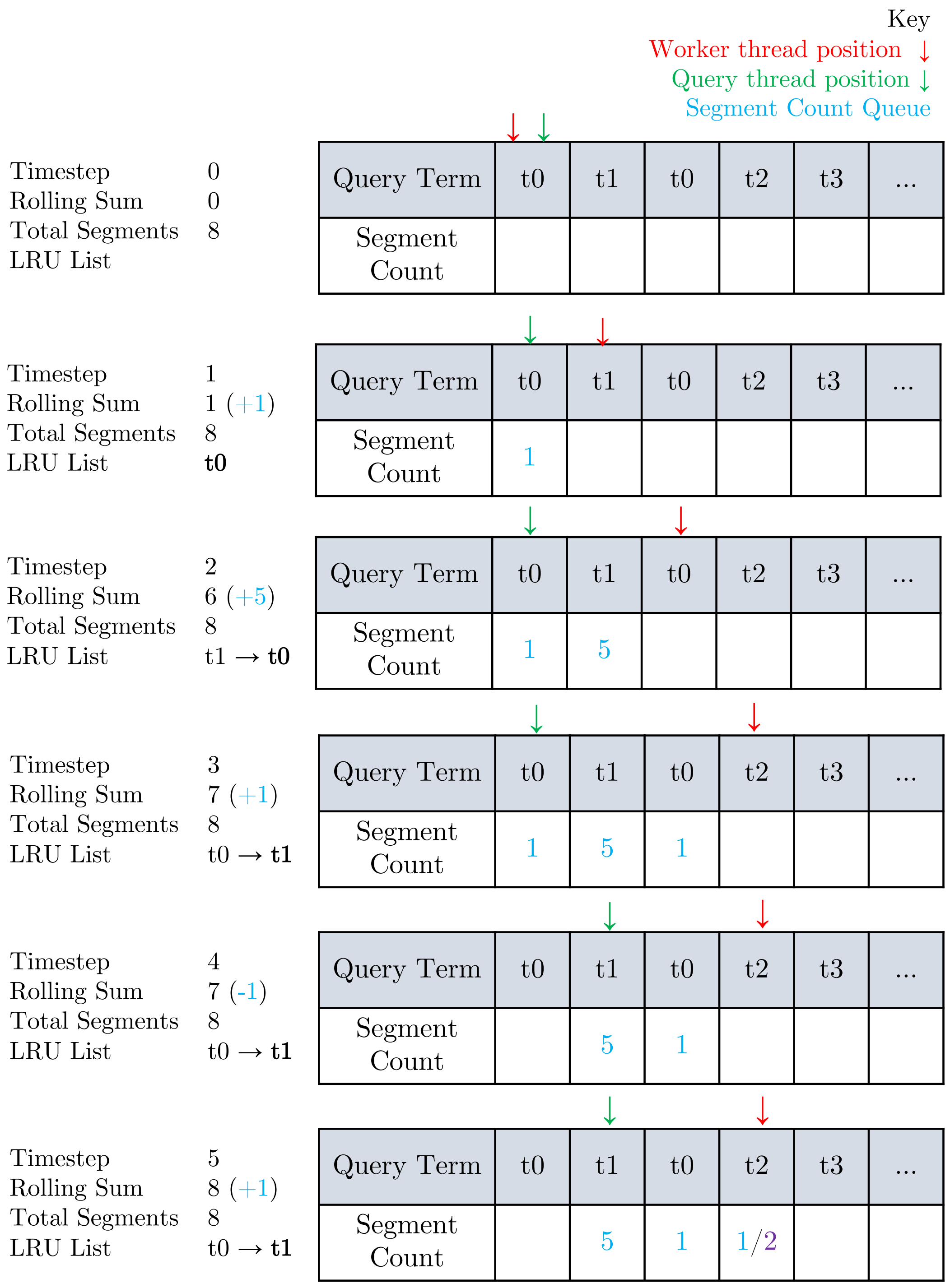


Figure 7.10: Example of special case where there is a mismatch between the evicted term and popped term from the segment count queue.

We use an example to show why the *tail term segment count* is required in the waiting condition. In Figure 7.10, we have a special case where the repeated terms cause a mismatch between the LRU term (to be evicted) and the segment count at the front of the segment count queue. In time step 3, the term t0 is repeated, making t1 the LRU term (next to be evicted). In time step 4, the query thread finishes t0, and subtracts one from the rolling sum. Finally, the worker prefetches one of two segments for t2 and needs to perform an eviction to make space for the second one. In this scenario, using the waiting condition:

$$\textit{rolling sum} > \textit{total segments}$$

which evaluates to false ($8 > 8$), will cause the eviction of t1 which the query thread is currently operating on. Instead we use:

$$\textit{rolling sum} + \textit{tail term segment count} > \textit{total segments}$$

which evaluates to true ($8 + 5 > 8$) causing the worker to wait for the rolling sum to decrease, preventing the incorrect eviction. The term t1 should only be evicted once it is processed and popped from the queue and subtracted from the rolling sum by the query thread, causing the waiting condition to be false ($3 + 5 > 8$) and allowing the eviction of t1.

In summary, the adaptive lookahead strategy allows the worker thread to adapt its lookahead arbitrarily based on the cache size and term posting block count. For example, it uses a larger lookahead for large the cache sizes or when terms have small block posting counts and vice versa. Our adaptive lookahead policy coupled with workload-aware prefetching brings a new design point for caching in search engines. We next rigorously evaluate our new cache on a variety of storage systems.

Chapter 8

Evaluation of APCache

We now present a detailed evaluation of APCache, focusing on average query performance (QPS) and tail latency, and we present results from several sensitivity studies to characterize APCache against baselines.

8.1 Configurations and Parameters

8.1.1 Configurations

We consider four configurations in our evaluation we discuss below.

- *Base*: The index is compressed using TurboPForDelta256 for DID blocks and TurboPFor256 for frequency blocks, with trailing blocks compressed using TurboPForDelta and TurboPFor.
- *Hybrid*: This configuration modifies the Base setup by caching term postings in PM. A cache miss triggers memory-mapped I/O to the disk-resident index. For each term, the query thread initiates a job for the worker thread, instructing it to transfer the term's postings to the cache using memory-mapped I/O.
- *BasePrefetch*: This configuration adds workload-aware prefetching to Base. A worker thread prefetches postings ahead of the main query thread into the page cache using a fixed look-ahead policy.
- *HybridPrefetch*: This configuration adds workload-aware prefetching to the Hybrid setup. A background thread proactively prefetches postings into the cache using an adaptive look-ahead policy. Our adaptive look-ahead policy eliminates a parameter from the baseline system, tuning it automatically.

8.1.2 Parameters

We experiment with the following key parameters.

- Lookahead (LA) is the number of queries the prefetcher can prefetch ahead of the main thread.
- Repetition probability (RP) is the characterization of the query workload, described in Section 8.8.
- Memory Limit (ML) is the memory limit we set using Cgroups. It limits the available DRAM capacity.
- Index directory is where we place the index for long-term preservation. We experiment with SSD and disk-resident indices to evaluate the performance of our system for design points with different costs.
- Segment size is the size of each PM segment. We choose a segment size for our custom cache that minimises the miss rate. We show a sensitivity analysis of this parameter later.
- Skip list and dictionary placement is where we choose to place the skip list and dictionary.

In the rest of this section, we rigorously analyze the sensitivity of our results to all parameters and experimentally choose the ones that maximize performance. We perform the rest of our evaluation using the rigorously tuned set of parameters.

8.1.3 Skip List and Dictionary Overhead

For all configurations, we have a 1.25 GB skip list offset dictionary. The skip list is 2.2 GB for Base and BasePrefetch. For Hybrid and HybridPrefetch, it is slightly larger at 2.3 GB since they require an extra 32 bits for cache metadata per term. The cache metadata overhead is small and does not increase significantly because the total number of terms in an index tend to remain relatively stable across different index sizes.

8.1.4 Skip List and Dictionary Location

We first decide the placement of the skip list and dictionary. This decision is important as our plan is to use the same placement policy for the rest of our evaluation. Our hypothesis is that placing the skip list and dictionary in a slower memory tier does not degrade the total query evaluation time. The reason is that lookups in the dictionary and skip list constitute a small fraction of the total query evaluation time. We place the skip list and dictionary in secondary storage (disk), DRAM, or PM. Figures 8.1 and 8.2 shows the results for AND and SINGLE queries, respectively.

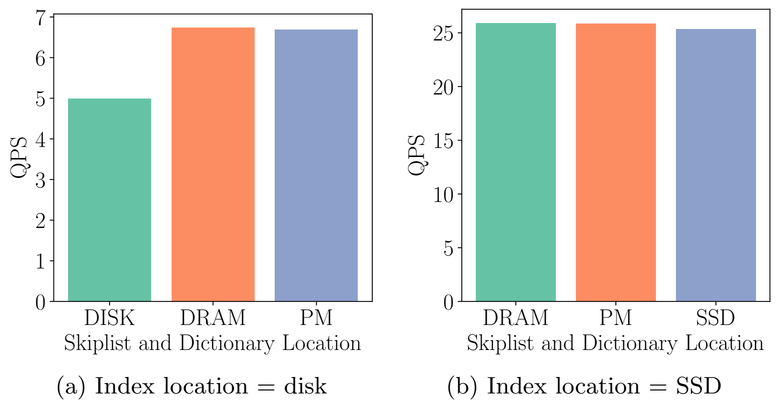


Figure 8.1: QPS vs. skip list location for AND H queries with RP = 25% and ML = 10%.

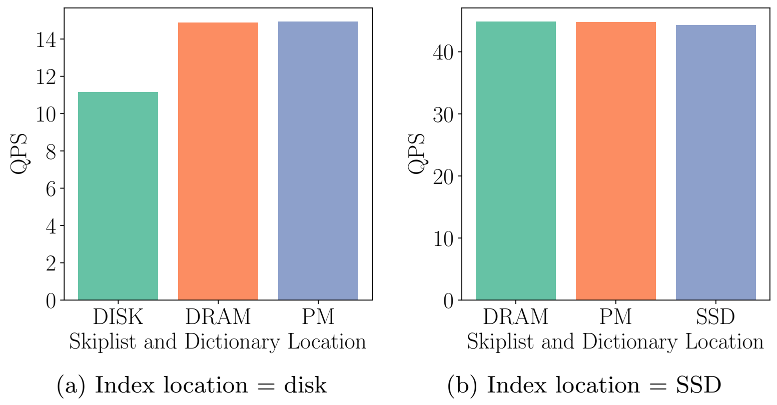


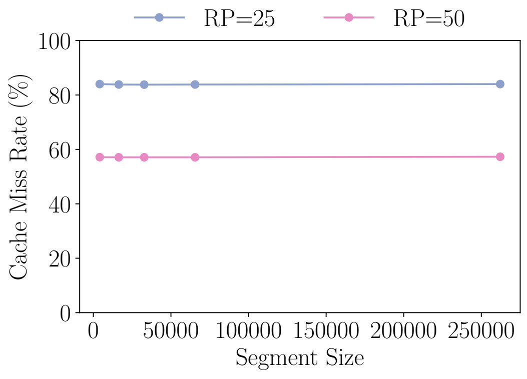
Figure 8.2: QPS versus skip list location for SINGLE H queries with RP = 25% and ML = 10%.

We observe that hosting these meta-data structures on disk results in a significant degradation in performance measured as QPS (see Figures 8.1 and 8.2). Specifically, for AND and SINGLE queries, the QPS degrades by 25.9% and 25.0%, respectively, if the skip list is placed on disk. In contrast, there is less impact on performance (QPS) if the skip list is stored on SSD due to its significantly lower latency compared to disks.

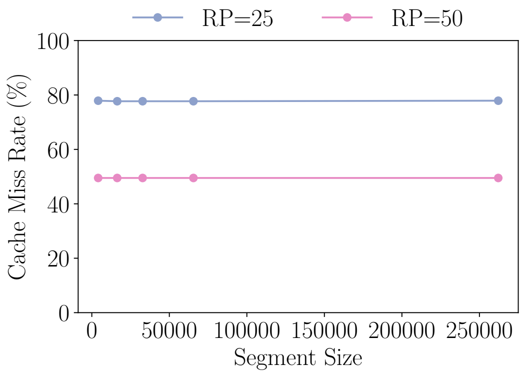
Finally, we observe comparable QPS if the skip list is backed by DRAM or PM. This observation is true for both query types, and regardless of where the index is stored (disk or SSD). From now on, we will use PM for storing the skip list and dictionary to conserve DRAM capacity.

8.1.5 Cache Segment Size

The size of the segment in our custom PM-backed postings cache is an important parameter. We now discuss the impact of segment size on the behavior of the cache. Specifically, we use cache miss rate (%) to quantify the utility of the cache. Miss rate represents how many times the request need to travel to the next slower level (e.g., disk) because the cache is unable to serve the request. This inability is due either to the item is evicted due to limited cache capacity, or the request is only seen for the first time, i.e., cold miss.



(a) AND



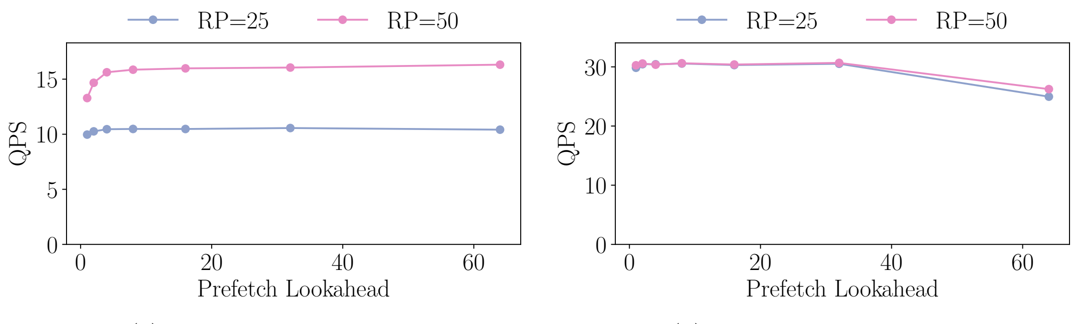
(b) SINGLE

Figure 8.3: Miss rate versus segment size for H queries with 10% cache size and RP of 25%.

We evaluate the cache miss rate of our PM cache across different segment sizes. We use a minimum segment size the same as OS page size, and the largest size as 1 MB. The segment sizes we evaluate are, 4096, 16384, 32768, 65536, 262144, and 1048576. The miss rates are measured during the second time we repeat the experiment (i.e., the system is fully warmed up), and the cache is also warm. Referring to Figure 8.3, Internal fragmentation occurs when the granularity of allocated space is too coarse, resulting in wasted space. This happens because larger segments may not be fully utilized, leaving unused portions within the allocated space. As segment sizes grow, the likelihood of having these unused portions increases, leading to less efficient use of memory and a higher miss rate. Based on our evaluation, we identify 32768 as the optimal segment size and use this setting for subsequent experiments.

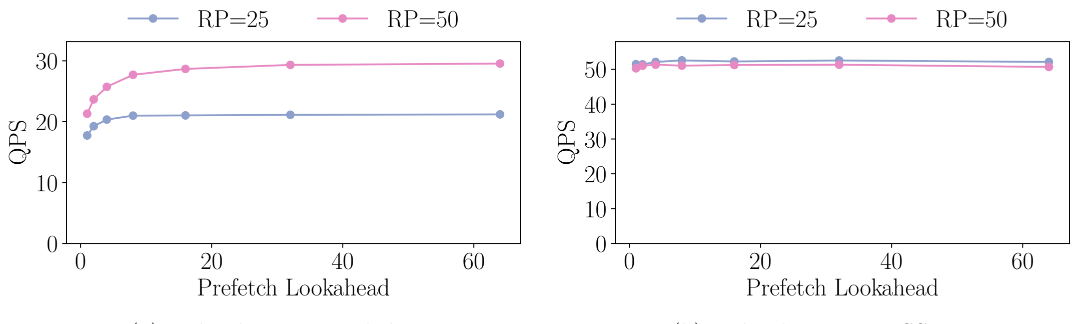
8.1.6 Prefetch Lookahead

The prefetch lookahead (LA) parameter is specific to BasePrefetch. We evaluate a number of settings for LA to pick the optimal setting as BasePrefetch is our optimized baseline comparison point.



(a) Index location = disk (b) Index location = SSD

Figure 8.4: QPS versus prefetch lookahead for H AND queries with ML = 10%, RP = 25%



(a) Index location = disk (b) Index location = SSD

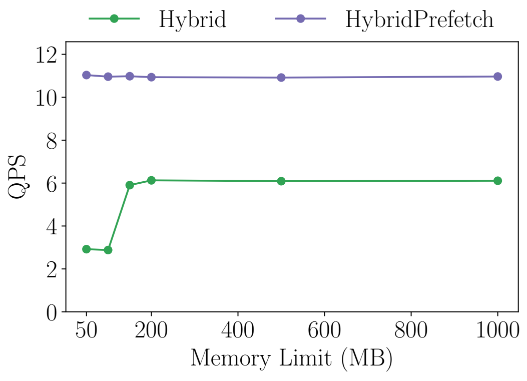
Figure 8.5: QPS versus prefetch lookahead for SINGLE queries with ML = 10%, RP = 25%

We explore different lookahead (LA) settings in Figure 8.4 and Figure 8.5. QPS increases as we increase LA and stabilizes around LA = 16. For subsequent experiments, we choose LA = 16 as the lookahead setting, even though LA = 32 shows a slight improvement (Figure 8.5a), to minimize the risk of cached terms being evicted from the page cache before the query thread can use them. We use a conservative setting as we observe cases when aggressive prefetching suddenly drop system QPS. We show one such case in Figure 8.4b. We observe quite clearly in Figure 8.4b that QPS drops when LA = 64.

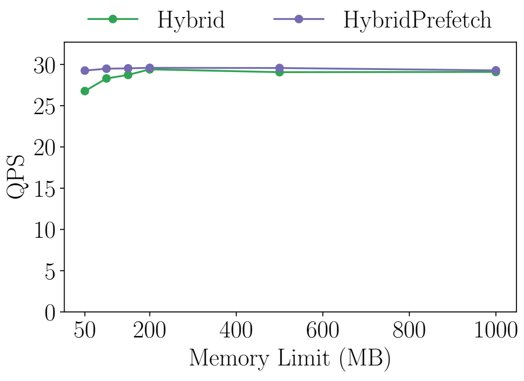
8.1.7 Memory Limit for cache

Next, we discuss the impact of DRAM capacity limitation on Hybrid configurations. This applies exclusively to Hybrid and HybridPrefetch (not Base or BasePrefetch) because we aim to replace the function of the page cache with our custom cache. Our key principle is to prevent the growth of DRAM in search engines proportional to index growth.

Specifically, we use DRAM in APCache only to create a path for data to flow from block storage to the second memory tier (PM in our case). If the DRAM size is too large, we risk increasing system costs. On the other hand, if the DRAM size is too small, we risk generating too much I/O, and degrading QPS. Ideally, we need sufficient DRAM to prevent evictions of posting blocks prior to their copy from DRAM to PM. We set up a number of experiments to rigorously find the DRAM size for our index. Other indices and indices with more documents may require tuning the memory limit. However, the fundamental principle is the same. We use a fixed DRAM budget in APCache unlike today's baseline that grow the page cache as the index grows.

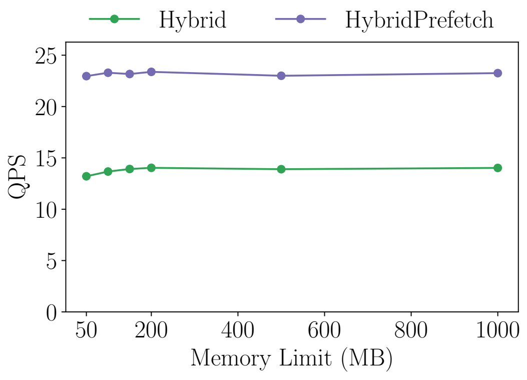


(a) Index location = disk

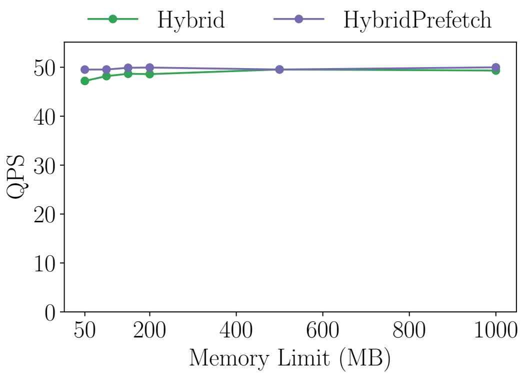


(b) Index location = SSD

Figure 8.6: QPS versus memory limit for AND H queries with 10% cache size, RP = 25%



(a) Index location = disk



(b) Index location = SSD

Figure 8.7: QPS versus memory limit for SINGLE H queries with 10% cache size, RP = 25%

We aim to minimize the memory limit for the cache configurations to conserve DRAM capacity, utilizing only what is necessary for the prefetcher to copy from the page cache to the PM cache, and in-memory data structures such as user buffers for query results, skip offset buffer, job queue (for Hybrid), cache structures (e.g. global segment list), and more. We measure QPS for various memory limits for both Hybrid and HybridPrefetch on SSD and disk.

In Figure 8.6, we show the results of our sensitivity studies for AND queries, and in Figure 8.7, we show the results for one-term queries. As we change the memory limit from 50 MB to 1 GB, we observe stable QPS for HybridPrefetch. Therefore, we adopt a 50 MB memory limit for HybridPrefetch.

For the Hybrid setup, we observe an improvement in QPS performance between 50 MB and 200 MB memory limits in Figures 8.6 and 8.7a (and up to 500 MB for SSD SINGLE queries in Figure 8.7b). Beyond these limits, performance plateaus. In Figure 8.6a, performance drops significantly at 50 and 100 MB memory limits. This likely occurs because, in the Hybrid setup, both the query and worker threads read from the memory-mapped file. The Hybrid configuration trails the query thread and can cache the postings loaded into the page cache by the query thread. When the memory limit is too small, the postings are evicted before the worker thread can process them. This issue is particularly detrimental for AND queries, as the query and worker threads could be loading postings for four different terms simultaneously (with each loading two terms).

since it retrieves postings from the memory-mapped file, which may still be in the page cache. Since our goal is to allocate just enough memory for in-memory data structures, we also adopt a 50 MB limit for the Hybrid configuration.

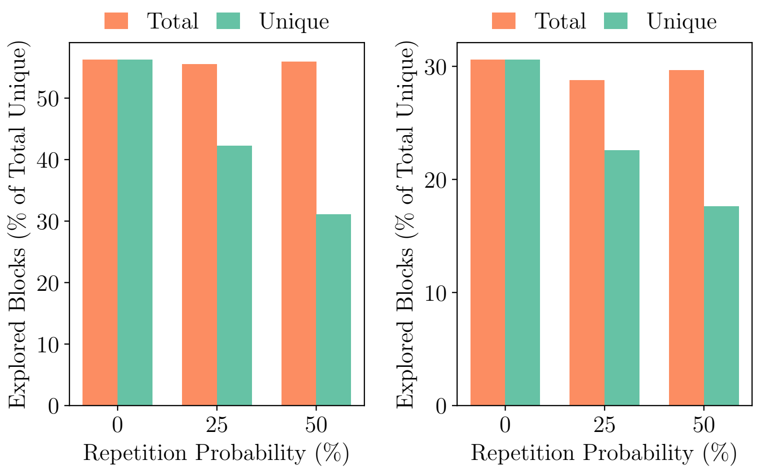
For subsequent experiments, we opt to use a 50 MB memory limit for Hybrid configurations. Experimenting with a 10 MB limit resulted in the program being terminated due to an out of memory error. We suspect this is due to search engine using user-level buffers for resolving queries, e.g., storing top results in an array for displaying to the user.

8.2 Query Evaluation Results

8.2.1 Query Workloads

We generate query workloads from the CommonCrawl 2022 dataset, creating workloads with 0%, 25%, and 50% repetition probability (RP). Figure 8.8 illustrates the portion of the index touched by AND and SINGLE H queries, showing the ratio of unique to total postings explored. In the index, there are 15.3 Billion unique postings, we call this Total Unique in Figure 8.8. In Figure 8.8, we also show Total (the total number of postings explored in the experiment including repeated but excluding skipped by the skip list), and Unique (unique postings, i.e. repeated postings are not counted) postings as a percentage of Total Unique postings.

For AND queries, approximately 56% of the index blocks are explored, and 31% for



(a) AND (b) SINGLE

Figure 8.8: Unique and total explored postings as a percentage of all postings.

SINGLE queries. As the repetition probability increases, the ratio of unique blocks to total blocks decompressed decreases, resulting in smaller working sets.

In this section, we discus results with both hard disks that are still used in datacenters and emerging PCIe NVMe SSDs, with much higher bandwidth and lower latency compared to old disks and SATA/SAS SSDs. Our aim is to show how our system behaves for a range of storage devices, and establish its robustness across a variety of scenarios.

As a reminder, Base and BasePrefetch use the OS page cache in DRAM, and Hybrid and HybridPrefetch use a fixed-size DRAM, and use PM-based cache (called APCache) for storing postings data. Thus, in Base and BasePrefetch DRAM grows proportional to dataset size, and in Hybrid and HybridPrefetch, PM grows proportional to dataset size. The results presented in this section uses H queries as they are long queries contributing to the query tail response.

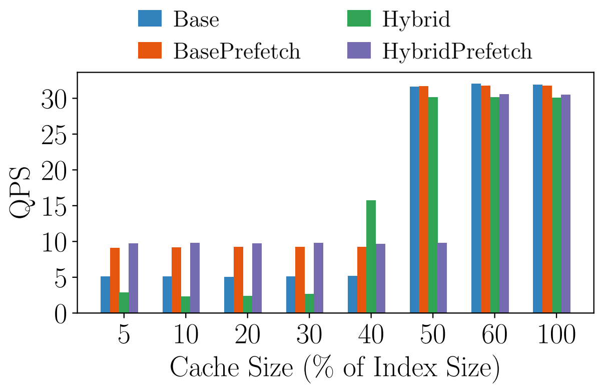
8.2.2 QPS Findings

Figures 8.9, 8.10, 8.11 and 8.12 show QPS for different cache sizes. In these figures, the cache size indicates the memory limit for Base and BasePrefetch. For Hybrid and HybridPrefetch, it represents the total cache size (number of segments multiplied by segment size), excluding the fixed 50 MB memory limit. For these experiments, we use a fixed 50 MB memory limit (ML = 5%), and a segment size of 32768 for Hybrid and HybridPrefetch. We use LA = 16 for BasePrefetch. Before each experiment, we flush the OS caches. Tables 8.1 and 8.2 show the 99th percentile tail latency results for query evaluation on disk and SSD.

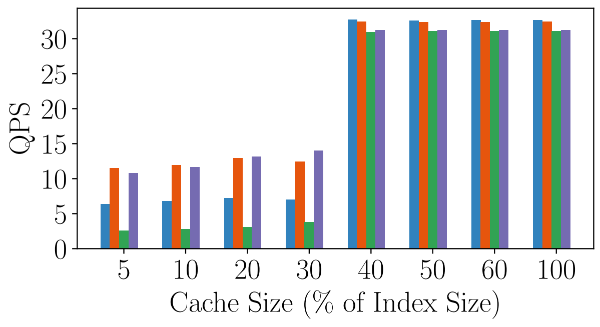
We now analyze the results obtained from different configurations. In both disk and

SSD experiments, we observe that once the cache size exceeds the working set of blocks, the QPS stabilizes. At this stage, Base and BasePrefetch exhibit identical performance, while Hybrid and HybridPrefetch show similar performance slightly below that of the Base configurations. This outcome is expected, as previously demonstrated, since DRAM indices are marginally faster than PM for PForDelta256. We notice that QPS with disks is lower than PCIe NVMe SSDs. Specifically, QPS with disk and a 5% page cache is very low (only 5 queries per second for AND queries). The QPS grows almost 6× with 50% cache. The cache size is proportional to index size. With newer and faster NVMe SSDs, this gap is only 1.25×.

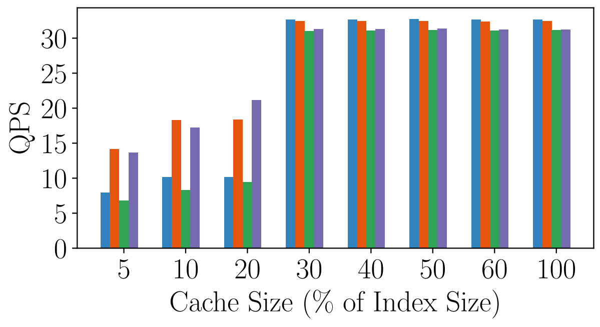
Our key observation is that as the cache size is increased, whereas Base and BasePrefetch demand more DRAM for delivering higher QPS, Hybrid and HybridPrefetch uses a fixed-size DRAM for a similar performance. The result is tremendous saving in DRAM capacity. We next discuss specific results and provide deeper insights into the behavior of APCache.



(a) RP = 0%

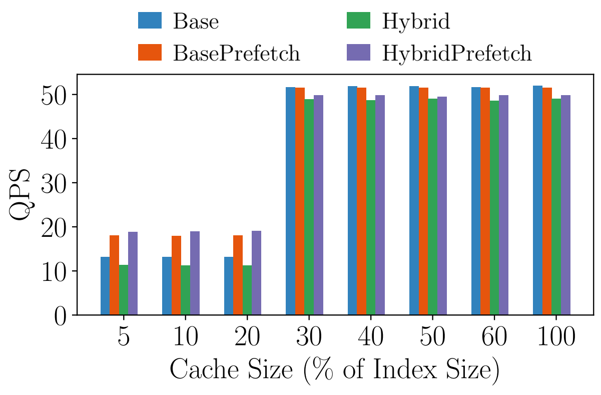


(b) RP = 25%

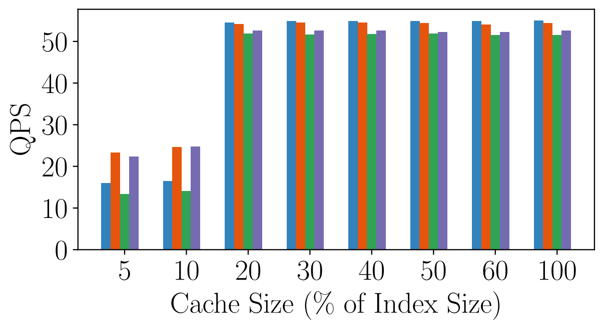


(c) RP = 50%

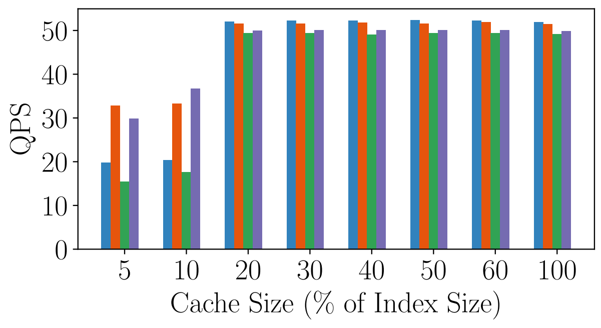
Figure 8.9: QPS versus cache size for AND H queries with index on disk.



(a) RP = 0%

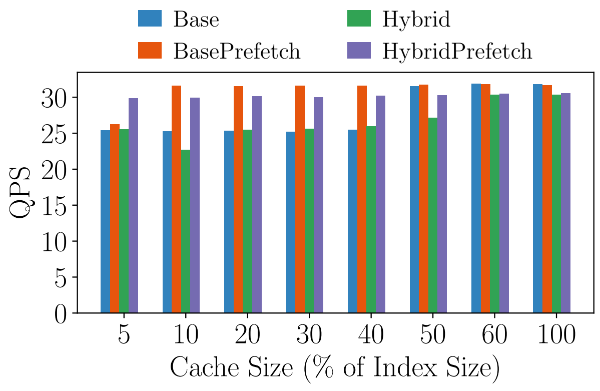


(b) RP = 25%

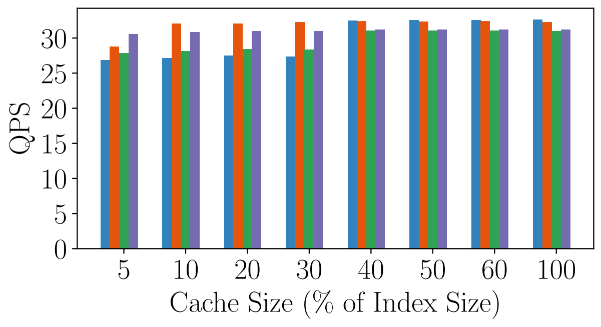


(c) RP = 50%

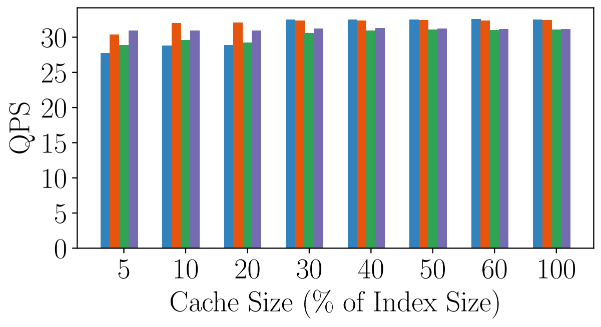
Figure 8.10: QPS versus cache size for SINGLE H queries with index on disk.



(a) RP = 0%

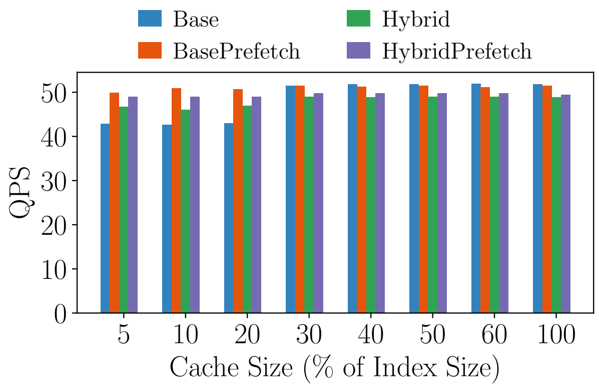


(b) RP = 25%

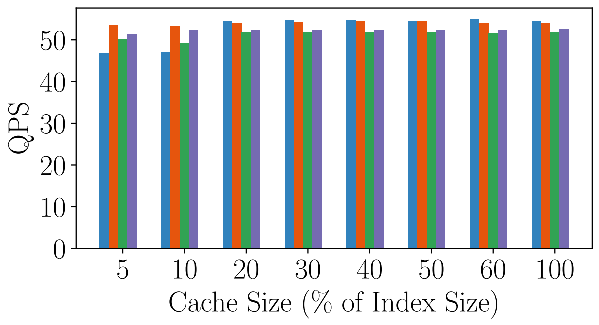


(c) RP = 50%

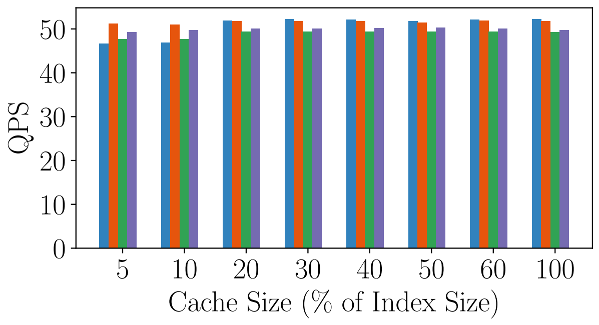
Figure 8.11: QPS versus cache size for AND H queries with index on SSD.



(a) RP = 0%



(b) RP = 25%



(c) RP = 50%

Figure 8.12: QPS versus cache size for SINGLE H queries with index on SSD.

Prefetching and Adaptive Lookahead

Prefetching improves QPS performance significantly. With SSD, the prefetch configurations can reach the optimal QPS level with small cache sizes (Figure 8.11 and Figure 8.10) because prefetching hides SSD latency, and NVMe SSDs are the fastest storage devices available today. The optimal QPS is one achieved with a DRAM-only system. For cache

Table 8.1: Showing the tail latency results for disk experiments and H queries.

		Tail Latency (s)					
		AND			**SINGLE**		
Configuration	**Cache Size (%)**	**0**	**25**	**50**	**0**	**25**	**50**
Base	5	0.87	0.76	0.69	0.45	0.29	0.39
	10	0.91	0.72	0.55	0.44	0.29	0.27
	20	0.95	0.73	0.61	0.43	0.21	0.21
	30	0.89	0.71	0.17	0.18	0.21	0.21
	40	0.89	0.15	0.17	0.18	0.21	0.21
	50	0.16	0.16	0.17	0.18	0.21	0.21
	60	0.16	0.15	0.17	0.18	0.21	0.21
	100	0.16	0.16	0.16	0.18	0.21	0.21
BasePrefetch	5	0.55	0.42	0.46	0.43	0.29	0.28
	10	0.55	0.51	0.35	0.43	0.30	0.28
	20	0.55	0.43	0.35	0.43	0.22	0.22
	30	0.57	0.43	0.17	0.18	0.21	0.22
	40	0.56	0.16	0.17	0.18	0.21	0.21
	50	0.16	0.15	0.17	0.18	0.21	0.22
	60	0.16	0.15	0.17	0.18	0.21	0.21
	100	0.16	0.15	0.17	0.18	0.22	0.22
Hybrid	5	1.92	2.38	1.03	0.97	0.53	0.57
	10	2.70	2.15	0.88	0.83	0.53	0.49
	20	2.45	2.05	0.85	0.90	0.22	0.22
	30	2.34	1.86	0.18	0.19	0.23	0.22
	40	0.61	0.17	0.18	0.19	0.22	0.23
	50	0.17	0.17	0.18	0.19	0.22	0.22
	60	0.17	0.17	0.18	0.19	0.23	0.23
	100	0.17	0.17	0.18	0.19	0.23	0.22
HybridPrefetch	5	0.52	0.48	0.47	0.40	0.35	0.28
	10	0.52	0.43	0.38	0.40	0.36	0.27
	20	0.54	0.43	0.31	0.40	0.22	0.22
	30	0.52	0.39	0.17	0.19	0.22	0.22
	40	0.52	0.17	0.17	0.19	0.22	0.22
	50	0.52	0.17	0.17	0.19	0.22	0.22
	60	0.17	0.17	0.17	0.19	0.22	0.22
	100	0.17	0.17	0.17	0.19	0.22	0.22

size of 5% with AND queries, QPS with BasePrefetch is not optimal. However, this is because the fixed look-ahead (LA) of 16 is too high. This causes prefetched blocks to be evicted before the main query thread can serve them. To verify this, we experiment with BasePrefetch and AND queries, using a cache size of 5% on SSD and reducing the lookahead amount from 16 to four. We find that QPS increases from 26.22, 28.76, and 30.34 to 31.69, 32.39, and 32.36 for RP = 0%, 25%, and 50%, respectively, verifying that the fixed LA of 16 is the issue. For HybridPrefetch, we have consistent performance for all cache sizes. This result is because we use an adaptive LA, meaning that the prefetcher adjusts its pace based on the relative position of the query thread and the available cache capacity.

On disk, the performance of HybridPrefetch is similar to BasePrefetch and markedly better for RP = 0% and 25% AND queries when the cache size is smaller than the working set. HybridPrefetch uses a DRAM size (also called memory limit or ML) of 50

Table 8.2: Showing the tail latency results for SSD experiments and H queries.

		Tail Latency (s)					
		AND			**SINGLE**		
Configuration	**Cache Size (%)**	**0**	**25**	**50**	**0**	**25**	**50**
Base	5	0.21	0.18	0.20	0.21	0.22	0.21
	10	0.21	0.18	0.20	0.21	0.23	0.21
	20	0.21	0.18	0.18	0.21	0.21	0.21
	30	0.21	0.19	0.17	0.18	0.21	0.21
	40	0.21	0.16	0.17	0.18	0.21	0.21
	50	0.16	0.16	0.17	0.18	0.21	0.21
	60	0.16	0.16	0.17	0.18	0.21	0.21
	100	0.16	0.16	0.17	0.18	0.21	0.21
BasePrefetch	5	0.22	0.19	0.19	0.18	0.21	0.22
	10	0.16	0.16	0.17	0.18	0.21	0.22
	20	0.16	0.16	0.17	0.18	0.22	0.22
	30	0.16	0.16	0.17	0.18	0.21	0.22
	40	0.16	0.16	0.17	0.18	0.21	0.21
	50	0.16	0.16	0.17	0.18	0.21	0.22
	60	0.16	0.16	0.17	0.18	0.22	0.21
	100	0.16	0.16	0.17	0.18	0.21	0.21
Hybrid	5	0.24	0.20	0.21	0.21	0.24	0.23
	10	0.28	0.20	0.18	0.21	0.24	0.23
	20	0.25	0.19	0.20	0.23	0.22	0.22
	30	0.23	0.19	0.18	0.19	0.22	0.22
	40	0.23	0.17	0.18	0.19	0.22	0.22
	50	0.21	0.17	0.18	0.19	0.22	0.22
	60	0.17	0.17	0.18	0.19	0.22	0.22
	100	0.17	0.17	0.18	0.19	0.22	0.23
HybridPrefetch	5	0.17	0.17	0.18	0.19	0.22	0.22
	10	0.18	0.17	0.17	0.19	0.22	0.22
	20	0.17	0.17	0.17	0.19	0.22	0.22
	30	0.18	0.17	0.17	0.19	0.22	0.22
	40	0.18	0.17	0.17	0.19	0.22	0.22
	50	0.18	0.17	0.17	0.19	0.22	0.22
	60	0.17	0.17	0.17	0.19	0.22	0.22
	100	0.17	0.17	0.18	0.19	0.22	0.22

MB, which means we can achieve excellent performance while saving DRAM capacity. We attribute part of this result to HybridPrefetch's adaptive lookahead, which allows it to fetch more segments due to its adaptive limit. Figures 8.13 and 8.14 show the lookahead delta, illustrating how far ahead the prefetcher operates at various points during the experiment. The method used to generate the query workload results in numerous repeated queries at the beginning, despite term randomization. HybridPrefetch capitalizes on this by prefetching further ahead. In Figure 8.14, when we allow BasePrefetch unlimited prefetching, it can prefetch more queries but still fewer than HybridPrefetch. We observe this because BasePrefetch cannot determine if a term is in the cache or evicted from the page cache, so repeated queries are prefetched again, slowing down the prefetcher. BasePrefetch does not skip cached queries because we lack visibility into the internal structures of the page cache. Enabling unlimited prefetching boosts BasePrefetch QPS from 18.41 to 19.46 for the experiment in Figure 8.14. For RP = 25%

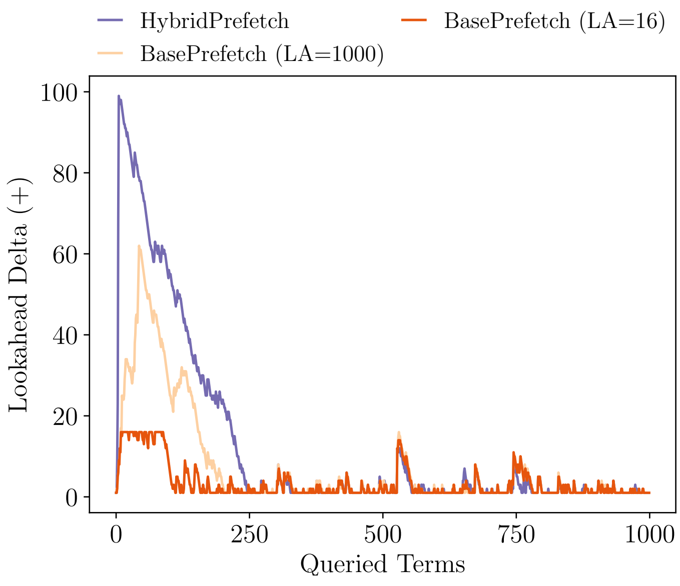


Figure 8.13: Lookahead delta for 20% cache size, RP = 50%, AND H queries.

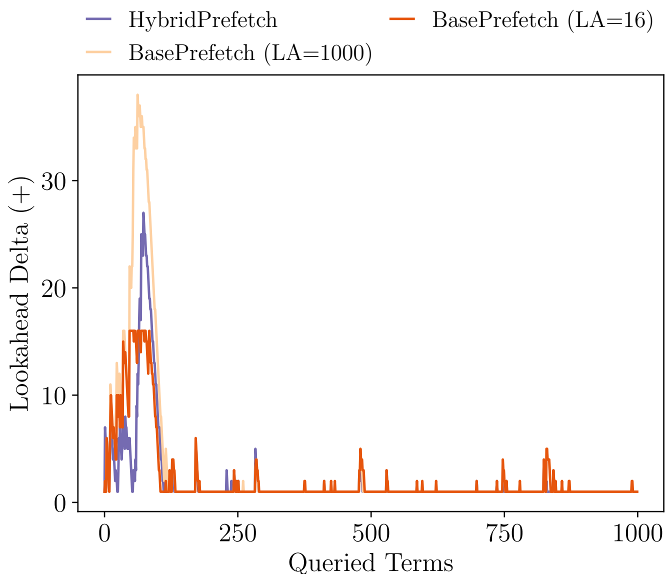


Figure 8.14: Lookahead delta for 20% cache size, RP = 25%, AND H queries.

in Figure 8.13, BasePrefetch actually prefetches faster than HybridPrefetch.

Disk versus SSD

On disk, there is a significant drop in performance when the cache size is small, unlike the consistent performance observed on SSD. This discrepancy arises because disk latency is much higher than that of SSD.

For disk (Figure 8.9 and 8.10), it is evident that larger RP improves caching performance. The increase in QPS with cache size becomes more gradual as we increase the RP.

SINGLE versus AND Queries

For disk, the difference between Base and BasePrefetch is more pronounced for AND queries than for SINGLE (Figure 8.9 and 8.10). This is likely because of improved disk access patterns. The prefetcher reads from the disk sequentially, first fetching the DID blocks and then the frequency blocks. In contrast, Base fetches DIDs and frequencies using an interleaving pattern, which is likely slower.

Tail Latency

As expected, tail latency decreases as we increase the cache size (TABLE 8.1 and 8.2). Prefetching improves tail latency significantly. The tail latency of BasePrefetch and HybridPrefetch is similar. For SSD, the tail latency quickly plateaus for BasePrefetch and HybridPrefetch.

Anomalies

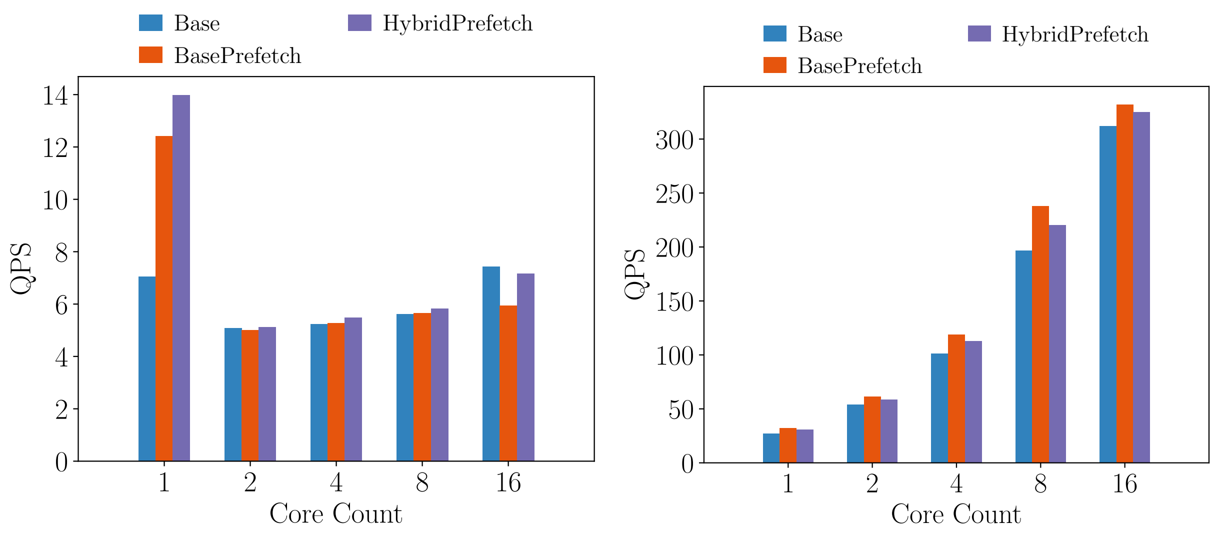
For AND queries on disk (Figure 8.9), Hybrid performs surprisingly well for cache sizes of 40 and 50. However, this result is misleading because the worker thread, responsible for copying to the cache, lags behind the main thread. During the second measurement repeat, the worker has not finished processing jobs from the first repeat, which means some terms from the beginning of the first repeat have not been evicted. This issue cannot be resolved by having the query thread wait for the cache worker to complete all jobs between repeats; the same lagging issue persists because the worker thread is still behind. We include this result to highlight a flaw in the Hybrid configuration.

In Figure 8.9, we see that Hybrid performs poorly compared to Base. Because Hybrid lags behind the query thread, it may access different parts of the disk compared to the query thread at any given point, leading to concurrent reads from the disk: with small cache sizes, the worker thread may lag so far behind that the required terms (brought in by the query thread) are evicted from the page cache. However, for RP = 50%, Hybrid does not lag behind because repeated terms allow it to catch up to the query thread, resulting in improved performance.

On disk, for AND queries with RP = 0% and 50% cache size, we observe that HybridPrefetch QPS performance does not increase as with the other configurations (Figure 8.9a). The reason for this increase in performance with other configurations is that all postings fit in the cache. However, for HybridPrefetch, the cache offsets are stored within the cache itself, which reduces its effective capacity. With RP = 50%, the cache is not large enough to accommodate all postings. We estimate that the cache offsets occupy approximately 4.5% of the cache size, considering that each block requires an

8-byte cache offset, and the average size of a fully occupied compressed block is 177.9 bytes with PForDelta.

8.2.3 Muticore Scalability



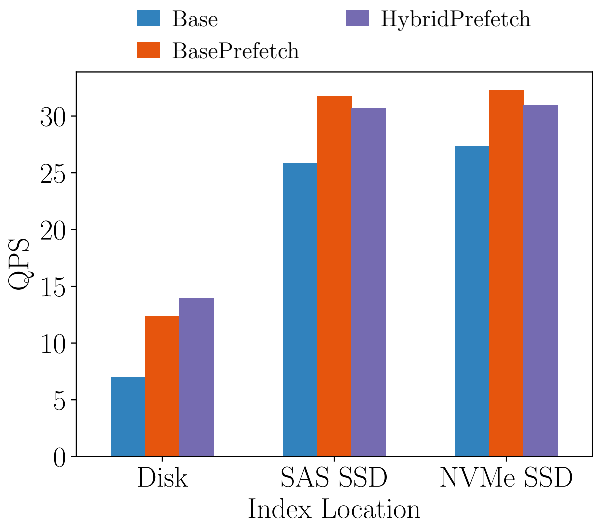
(a) Disk (b) SSD

Figure 8.15: Multicore scaling for SSD and disk, for AND H queries with RP = 25% and 30% cache size. The QPS is summed across cores.

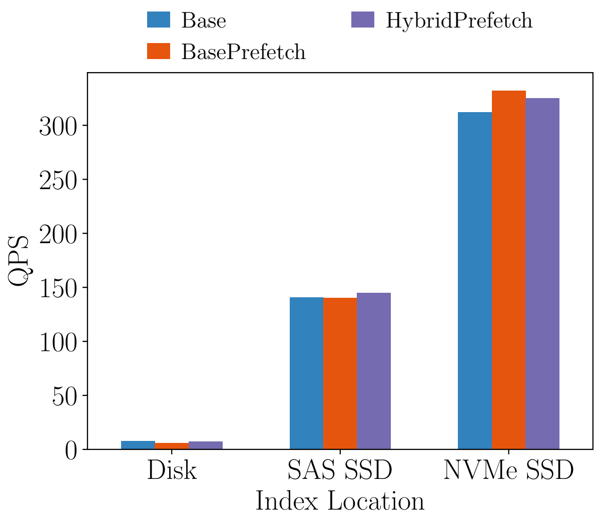
Figure 8.15 shows how QPS performance scales with core count. Each core has a query thread that operates on a shard of the index (we duplicate the index). For BasePrefetch and HybridPrefetch, we have a worker thread for each core. We eliminate the Hybrid configuration, which exhibits poor performance.

In Table 8.1 and 8.2, we present the average tail latency across all cores at different core counts. This data, which has significant implications for system performance optimization, provides a comprehensive view of the performance characteristics of different configurations.

For disk (Figure 8.15a), our analysis reveals that prefetching techniques (HybridPrefetch and BasePrefetch) offer the best performance at lower core counts but show limited benefit with higher core counts. We also observe a sharp decline in QPS when increasing from 1 to 2 cores, likely due to core interference from reading different parts of the disk, which operates optimally with sequential reads. As the number of cores increases beyond 2, QPS performance gradually improves, though with diminishing returns. At 16 cores, Base performance matches that of a single core, while the performance of HybridPrefetch and BasePrefetch significantly deteriorates. Table 8.1 indicates that tail latency rises proportionally with core count, further highlighting the performance trends.



(a) 1 core.



(b) 16 cores.

Figure 8.16: QPS performance for 1 and 16 cores for different index locations with RP = 25% and 30% cache size. The QPS is summed across cores. We introduce a new storage technology, namely SAS SSD, and refer to our original SSD as NVMe SSD.

For disk, the tail latency is quite similar across configurations. At 16 threads, tail latency is extremely high, reaching 16.81 seconds for BasePrefetch and 15.0 seconds for HybridPrefetch.

For SSD (Figure 8.15b), we see that QPS increases with core count with diminishing returns. Prefetching significantly boosts at a single core level but shows limited scalability beyond 2 cores. HybridPrefetch is marginally better than BasePrefetch.

For SSD, with 16 cores, BasePrefetch achieves 12× the QPS performance of 1 core; however, Table 8.2 shows that tail latency nearly doubles. The tail latency is quite similar across configurations and grows with core count.

For SSD, all schemes exhibit excellent scalability with core count, especially Base and BasePrefetch. HybridPrefetch, although starting strong at a single core, falls slightly behind as the core count increases.

In Figure 8.16, we compare QPS performance at 1 and 16 cores across different index locations. We introduce a new storage technology: SAS SSD, and refer to our original SSD as NVMe SSD.

For 1 core, we observe a significant improvement in performance from Disk to SAS SSD, with similar performance between SAS SSD and NVMe SSD (Figure 8.16a). This is because SAS SSD is notably faster than disk.

Table 8.3: Showing the tail latency for multicore disk experiments for AND H queries with RP of 25% and cache size of 30%.

Configuration	**Cores**	**Tail Latency (s)**
Base	1	0.68
	2	2.23
	4	4.42
	8	8.49
	16	14.31
BasePrefetch	1	0.49
	2	2.30
	4	4.58
	8	8.52
	16	16.81
HybridPrefetch	1	0.48
	2	2.53
	4	4.64
	8	8.79
	16	15.00

Table 8.4: Showing the tail latency for multicore SSD experiments for AND H queries with RP of 25% and cache size of 30%.

Configuration	**Cores**	**Tail Latency (s)**
Base	1	0.19
	2	0.19
	4	0.20
	8	0.21
	16	0.27
BasePrefetch	1	0.16
	2	0.16
	4	0.17
	8	0.17
	16	0.29
HybridPrefetch	1	0.17
	2	0.18
	4	0.19
	8	0.19
	16	0.28

For 16 cores, we find that QPS increases significantly as we move from Disk to SAS SSD and then to NVMe SSD(Figure 8.16b). As core counts increase, older SSDs (SAS) are less scalable, while newer SSDs (NVMe) maintain strong performance. This improvement is due to the significant increase in bandwidth; we measure the sequential read bandwidth of these storage devices: Disk operates at a bandwidth of 181 MB/s, SAS SSD at 793 MB/s, and NVMe SSD at 2200 MB/s. These bandwidth limitations significantly hinder performance at large core counts due to the concurrent reading from the storage technology by many threads.

8.3 Key Takeaways

Our key takeaways from these results are:

- For SSD indexes, we can achieve maximum performance by using a small amount of DRAM for the cache and prefetching to hide SSD latency. We can achieve a similar but slightly worse performance using prefetching with PM instead of DRAM.
- For indices on disk, prefetching improves QPS but is not enough to hide disk latency. In this scenario, PM provides similar performance for the same cache size. It also has the advantage of managing the internal components and using adaptive prefetching to fully utilize the available cache capacity, providing better performance in some scenarios.
- For indices on disk, QPS increases with cache size. Given that PM capacities are much higher, we should PM for a larger cache, which leads to better performance.
- Since we manage the cache directly in HybridPrefetch, we can implement an adaptive lookahead. Fixed prefetching is unreliable because too large a lookahead can cause excessive prefetching for small caches, while too small can limit performance for low-frequency queries. With our custom cache, we can prefetch more efficiently and skip prefetching terms already in the cache.
- We can achieve excellent QPS performance with just 50 MB of DRAM, conserving DRAM capacity while maintaining high performance. This is only feasible with a PM cache when using cgroups because, if our custom cache were on DRAM, the memory limit could cause portions of it to be swapped out to secondary storage. Since cgroups do not account for PM, this issue does not arise.

Chapter 9

Concluding Remarks

In this section, we present our key conclusions, and discuss directions for future work.

9.1 Conclusion

Our research demonstrates that SIMD compression algorithms, particularly StreamVByte, TurboPFor256, and their delta variants, can reduce storage costs while enhancing query evaluation performance. We find that these algorithms deliver similar QPS performance on both DRAM and PM. Our APCache system leverages these findings to build a secondary-storage-resident index with a PM cache. By incorporating an adaptive prefetching policy, we achieve optimal performance on SSDs with a small DRAM-PM cache, where the DRAM component is only 50 MB. (For our workloads and I/O pressure, NVMe SSDs do not require a large total cache size.) For disk indices, prefetching improves performance to match the DRAM OS cache. Additionally, since QPS increases with cache size, using PM allows for a larger cache and, thus, better QPS, thanks to PM's higher capacity. Finally, using PM enables us to save DRAM capacity. Future systems should use a fixed-size DRAM mitigating DRAM growth with datasets size and use prefetching and cache on a cheaper (albeit slower) memory technology. Finally, leaving the OS I/O stack unmodified is useful to maximize production deployment.

9.2 Future Work

We envision several directions for future work. More specifically, we envision opportunities at the OS, hardware (storage technology), and application levels.

9.2.1 Considering Linux Asynchronous I/O

In this work, we have considered MMIO for reading the index from block storage. Emerging Linux APIs, such as asynchronous I/O using the `io_submit()` system call, offer to exploit the high device bandwidth of today's SSDs. In one system call, and thus one entry to the kernel, the application can submit a batch of requests to the storage device. The kernel maintains two queues, a submission queue and a completion queue, and the application can poll for events in the completion queue without entry into the kernel. This new API requires rethinking of the application code but promises high performance if the implementation is able to exploit the parallelism of the device. In the future, we would like to use this asynchronous API to enable the worker thread to submit I/Os by using the lookahead mechanism in APCache. This way, the worker thread copying compressed blocks from DRAM to the second tier can happen concurrently with device I/O.

9.2.2 Other Storage Technologies

In this work, we have limited our analysis to Intel Optane persistent memory, using it as a slow second tier. Unfortunately, Intel Optane PM is now discontinued. On the other hand, several alternatives are emerging to complement DRAM as mediums for memory capacity expansion. Among the most promising ones, disaggregated memory via the emerging CXL interface or remote memory using fast network cards are promising directions. Today, it is faster to access remote memory via multi-gigabit Ethernet links than block storage. Hence, remote memory can be shared between several applications in the data center and used on a need basis. We intend to use remote and disaggregated memory for storing inverted indices in the future. Our expectation is that block storage to remote memory transfers through the page cache will be feasible.

9.2.3 Real-Time Search Engines

In this work, we have limited our analysis to batch search, in which indices are built offline, and the contents of the corpus are known ahead of time. Real-time search involves ingesting fresh data (e.g., tweets) and evaluating queries on that data concurrently, leading to complex synchronization and storage challenges. Compression in real-time is challenging as queries may arrive as soon as data is ingested and is in the middle of being transformed into compressed form. Also, it is not straightforward to copy freshly ingested blocks to the second tier while queries are running concurrently. We leave it to future work to build a dynamic search engine that ingests fresh data and serves it in real-time across two-tiered memories and also stores indices in a compressed format.

9.2.4 Slow Memory as Page Cache

In this work, we have followed the principle of leaving the Linux kernel unmodified for maximum deployment potential. A key challenge in APCache is the application logic for copying compressed blocks from DRAM to PM and the associated code to perform prefetching from user space. Hence, it implies that every data-intensive application aiming to use APCache-style caching on the second tier needs to invest effort repeatedly to rehash APCache code to fit its purpose. If the OS can offer a two-tiered page cache, all applications can benefit from the ideas used by APCache. For Linux to spread the page cache across two memory technologies is not an easy task. It remains an open challenge, and we leave it to future work. We note that if ideas in APCache can be moved inside the Linux kernel, the design of the search engine and other applications wanting to use APCache can be simplified.

9.2.5 JVM-Based Search Engines

Our baseline search engine is developed in a native language runtime (C/C++). Programmers today prefer to develop search and analytics applications in Java and run them on top of the Java Virtual Machine (JVM). JVM offers a managed heap that acts as a sandbox, offering memory safety and security and lifting the burden of manual memory management from the shoulders of programmers. Implementing APCache in JVM-based search engines comes with challenges. For instance, copying compressed blocks from the JVM heap to the second tier (assuming the blocks reside on the heap) is challenging. Introducing the slow tier to the JVM is also challenging. Also challenging is how best to partition the DRAM budget between the fixed-size page cache to facilitate I/O and the managed heap. One would need to analyze the garbage collection (GC) overhead and I/O overhead, how much the worker thread lags behind (or is ahead of) the main query threads, and adjust the memory partitions accordingly. We leave all such problems to future work.

9.2.6 Customized DRAM Cache

In this work, we refrain from changing the internal design and organization of the OS page cache. The page cache is designed to store disk-resident data at the granularity of pages. In the future, we aim to study the design of a customized page cache for search engines that use compressed blocks. We believe operating at the granularity of block sizes (not disk blocks but compressed blocks) and exposing those blocks to the application-level skip list would lead to more efficient I/O and fast memory to slow memory transfers. No prior work studies a customized OS cache for search engines. One complication is that Linux internally performs all memory operations at the granularity of pages. This work would require rethinking how application code interfaces with the kernel, tackling the impedance mismatch between application expectations for interactions with the kernel, what the kernel considers a convenient granularity internally, and the granularity of the compressed blocks stored on disk.

9.2.7 Eviction Algorithms

Our work only evaluates the LRU eviction algorithm for the PM-resident cache in APCache. In the future, we aim to perform a study to find the best eviction algorithm for compressed blocks on PM. Specifically, we aim to understand how other eviction algorithms, including ones based on machine learning approaches, interact with our prefetching mechanism. Prior work on datacenter caches reports several eviction algorithms (Song et al., 2020; Janardhan and Bhardwaj, 2019; Berger, 2018; Fedchenko et al., 2019; Belady, 1966; Friedman, 2001; Jiang and Zhang, 2002). We leave analyzing them in detail to future work.

9.2.8 Generalizing to Other Data-Intensive Applications

In this work, we only consider search engines as a target application for APCache. NoSQL stores, key-value stores, graph stores, and emerging vector databases exhibit similar concerns. Specifically, they put pressure on DRAM and would benefit from a second tier. They use different forms of compression, which we aim to study in the future. One aspect that is similar between search engines and these emerging databases is that the memory requirements grow proportional to the dataset sizes on block storage. We expect the specifics of the prefetching algorithms to require adaptions. We leave investigating other data-intensive applications and applying APCache to them to future work.

Bibliography

Abulila, A.; Mailthody, V. S.; Qureshi, Z.; Huang, J.; Kim, N. S.; Xiong, J.; and Hwu, W.-m., 2019. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19 (Providence, RI, USA, 2019), 971–985. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3297858.3304061. https://doi.org/10.1145/3297858.3304061. [Cited on page 1.]

Akram, S., 2021a. *Exploiting Intel Optane Persistent Memory for Full Text Search*, 80–93. Association for Computing Machinery, New York, NY, USA. ISBN 9781450384483. https://doi.org/10.1145/3459898.3463906. [Cited on page 1.]

Akram, S., 2021b. *Exploiting Intel Optane Persistent Memory for Full Text Search*, 80–93. Association for Computing Machinery, New York, NY, USA. ISBN 9781450384483. https://doi.org/10.1145/3459898.3463906. [Cited on page 13.]

Akram, S., 2021c. Performance evaluation of intel optane memory for managed workloads. *ACM Trans. Archit. Code Optim.*, 18, 3 (Apr 2021). doi:10.1145/3451342. [Cited on pages 6, 15, and 16.]

ALLHDD.com, 2024. Seagate st8000nm0205 new sealed - allhdd.com. https://www.allhdd.com/seagate-st8000nm0205-hard-disk-drive/. [Cited on page 24.]

Auradkar, A.; Botev, C.; Das, S.; De Maagd, D.; Feinberg, A.; Ganti, P.; Gao, L.; Ghosh, B.; Gopalakrishna, K.; Harris, B.; Koshy, J.; Krawez, K.; Kreps, J.; Lu, S.; Nagaraj, S.; Narkhede, N.; Pachev, S.; Perisic, I.; Qiao, L.; Quiggle, T.; Rao, J.; Schulman, B.; Sebastian, A.; Seeliger, O.; Silberstein, A.; Shkolnik, B.; Soman, C.; Sumbaly, R.; Surlaker, K.; Topiwala, S.; Tran, C.; Varadarajan, B.; Westerman, J.; White, Z.; Zhang, D.; and Zhang, J., 2012. Data infrastructure at linkedin. In *2012 IEEE 28th International Conference on Data Engineering*, 1370–1381. doi:10.1109/ICDE.2012.147. [Cited on page 1.]

Baeldung, 2024. Testing disk performance on linux. https://www.baeldung.com/linux/disk-performance-test. [Cited on page 24.]

Belady, L. A., 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5, 2 (1966), 78–101. doi:10.1147/sj.52.0078. [Cited on page 88.]

Benson, L.; Makait, H.; and Rabl, T., 2021. Viper: An efficient hybrid pmem-dram key-value store. In *VLDB*, XXX–XXX. ACM. doi:10.14778/3461535.3461543. https://doi.org/10.14778/3461535.3461543. [Cited on page 13.]

Berg, B.; Berger, D. S.; McAllister, S.; Grosof, I.; Gunasekar, S.; Lu, J.; Uhlar, M.; Carrig, J.; Beckmann, N.; Harchol-Balter, M.; and Ganger, G. R., 2020a. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 753–768. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/berg. [Cited on page 1.]

Berg, B.; Berger, D. S.; McAllister, S.; Grosof, I.; Gunasekar, S.; Lu, J.; Uhlar, M.; Carrig, J.; Beckmann, N.; Harchol-Balter, M.; and Ganger, G. R., 2020b. The cachelib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20. USENIX Association, USA. [Cited on page 15.]

Berger, D. S., 2018. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18 (Redmond, WA, USA, 2018), 134–140. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3286062.3286082. https://doi.org/10.1145/3286062.3286082. [Cited on page 88.]

Chen, H.; Ruan, C.; Li, C.; Ma, X.; and Xu, Y., 2021. SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 17–32. USENIX Association. https://www.usenix.org/conference/fast21/presentation/chen-hao. [Cited on pages 13, 14, and 15.]

Chen, Y.; Lu, Y.; Yang, F.; Wang, Q.; Wang, Y.; and Shu, J., 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20 (Lausanne, Switzerland, 2020), 1077–1091. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3373376.3378515. https://doi.org/10.1145/3373376.3378515. [Cited on page 1.]

Chilukuri, A. and Akram, S., 2023a. Analyzing and improving the scalability of in-memory indices for managed search engines. ISMM 2023. [Cited on pages 1, 20, and 24.]

Chilukuri, A. and Akram, S., 2023b. 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. [Cited on page 14.]

Cohen, W. libpfm4. https://github.com/wcohen/libpfm4. [Cited on pages 23 and 26.]

Connelly, S. Practical bm25 - part 2: The bm25 algorithm and its variables. https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables. [Cited on page 10.]

Crawl, C. May 2022 crawl archive now available. https://commoncrawl.org/blog/may-2022-crawl-archive-now-available. [Cited on page 24.]

Debnath, B.; Haghdoost, A.; Kadav, A.; Khatib, M. G.; and Ungureanu, C., 2015. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15 (Monterey, California, 2015). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2819001.2819002. https://doi.org/10.1145/2819001.2819002. [Cited on page 15.]

Fedchenko, V.; Neglia, G.; and Ribeiro, B., 2019. Feedforward neural networks for caching: N enough or too much? 46, 3 (jan 2019), 139–142. doi:10.1145/3308897.3308958. https://doi.org/10.1145/3308897.3308958. [Cited on page 88.]

Fitzpatrick, B., 2023. Memcached. https://memcached.org/. [Cited on page 15.]

Foundation, T. A. S., 2021a. Lucene™ release docs. https://lucene.apache.org/core/documentation.html. [Cited on page 5.]

Foundation, T. A. S., 2021b. Welcome to apache lucene. https://lucene.apache.org/. [Cited on page 14.]

Friedman, J. H., 2001. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29, 5 (2001), 1189 – 1232. doi:10.1214/aos/1013203451. https://doi.org/10.1214/aos/1013203451. [Cited on page 88.]

Gupta, S.; Oh, Y.; Yan, L.; Sutherland, M. J.; Bhattacharjee, A.; Falsafi, B.; and Hsu, P., 2023. Astriflash: A flash-based system for online services. In *Proceedings of the High Performance Computer Architecture (HPCA)*. [Cited on page 1.]

Hager, G. and Wellein, G., 2010. *Introduction to high performance computing for scientists and engineers*. CRC Press. [Cited on page 6.]

HANDY, J., 2018. Emerging memories today: Why emerging memories are necessary. https://thememoryguy.com/. [Cited on page 1.]

HEO, T. *Control Group v2 — The Linux Kernel documentation*. Linux Kernel. https://docs.kernel.org/admin-guide/cgroup-v2.html. [Cited on page 11.]

HU, D.; CHEN, Z.; WU, J.; SUN, J.; AND CHEN, H., 2021. Persistent memory hash indexes: An experimental evaluation. 14, 5 (jan 2021), 785–798. doi:10.14778/3446095.3446101. https://doi.org/10.14778/3446095.3446101. [Cited on page 15.]

HUANG, J.; BADAM, A.; QURESHI, M. K.; AND SCHWAN, K., 2015. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15 (Portland, Oregon, 2015), 580–591. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2749469.2750420. https://doi.org/10.1145/2749469.2750420. [Cited on page 1.]

INTEL, 2024. Intel® optane™ ssd dc p4800x series 1.5tb 1/2 height pcie x4 3d xpoint™ product specifications. https://ark.intel.com/content/www/us/en/ark/products/97159/intel-optane-ssd-dc-p4800x-series-1-5tb-1-2-height-pcie-x4-3d-xpoint.html. [Cited on page 24.]

INTEL CORPORATION, 2023. Intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed: 2023-05-27. [Cited on page 6.]

JANARDHAN, V. AND BHARDWAJ, A., 2019. Predictive Caching@Scale. USENIX Association, Santa Clara, CA. [Cited on page 88.]

JIANG, S. AND ZHANG, X., 2002. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. SIGMETRICS '02 (Marina Del Rey, California, 2002), 31–42. Association for Computing Machinery, New York, NY, USA. doi:10.1145/511334.511340. https://doi.org/10.1145/511334.511340. [Cited on page 88.]

KAIYRAKHMET, O.; LEE, S.; NAM, B.; NOH, S. H.; AND RI CHOI, Y., 2019. SLM-DB: Single-Level Key-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 191–205. USENIX Association, Boston, MA. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet. [Cited on pages 13, 14, and 15.]

KANNAN, S.; BHAT, N.; GAVRILOVSKA, A.; ARPACI-DUSSEAU, A.; AND ARPACI-DUSSEAU, R., 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 993–1005. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/kannan. [Cited on pages 13, 14, and 15.]

Kassa, H. T.; Akers, J.; Ghosh, M.; Cao, Z.; Gogte, V.; and Dreslinski, R., 2021. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 821–837. USENIX Association. https://www.usenix.org/conference/atc21/presentation/kassa. [Cited on pages 13 and 15.]

Kim, K., 2008. Future memory technology: challenges and opportunities. In *2008 International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*, 5–9. doi:10.1109/VTSA.2008.4530774. [Cited on page 6.]

Kleppmann, M., 2017. *Designing Data-Intensive Applications*. O'Reilly, Beijing. ISBN 978-1-4493-7332-0. https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/. [Cited on page 5.]

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. [Cited on page 15.]

Kwon, Y.; Fingler, H.; Hunt, T.; Peter, S.; Witchel, E.; and Anderson, T., 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17 (Shanghai, China, 2017), 460–477. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3132747.3132770. https://doi.org/10.1145/3132747.3132770. [Cited on pages 13 and 15.]

Lai, S., 2008. Non-volatile memory technologies: The quest for ever lower cost. In *2008 IEEE International Electron Devices Meeting*, 1–6. doi:10.1109/IEDM.2008.4796601. [Cited on page 6.]

Lee, B. C.; Zhou, P.; Yang, J.; Zhang, Y.; Zhao, B.; Ipek, E.; Mutlu, O.; and Burger, D., 2010. Phase-change technology and the future of main memory. *IEEE Micro*, 30, 1 (Jan. 2010). doi:10.1109/MM.2010.24. [Cited on page 6.]

Lemire, D. and Boytsov, L., 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45, 1 (2015), 1–29. doi:https://doi.org/10.1002/spe.2203. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2203. [Cited on pages 9 and 10.]

Lemire, D.; Boytsov, L.; and Kurz, N., 2016. Simd compression and the intersection of sorted integers. *Software: Practice and Experience*, 46, 6 (2016), 723–749. [Cited on page 7.]

Lemire, D.; Kurz, N.; and Rupp, C., 2018. Stream vbyte: Faster byte-oriented integer compression. *Information Processing Letters*, 130 (2018), 1–6. [Cited on pages 7, 8, 22, 28, and 29.]

Lim, K.; Chang, J.; Mudge, T.; Ranganathan, P.; Reinhardt, S. K.; and Wenisch, T. F., 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09 (Austin, TX, USA, 2009), 267–278. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1555754.1555789. https://doi.org/10.1145/1555754.1555789. [Cited on page 6.]

Lim, K.; Turner, Y.; Santos, J. R.; AuYoung, A.; Chang, J.; Ranganathan, P.; and Wenisch, T. F., 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, 1–12. doi:10.1109/HPCA.2012.6168955. [Cited on page 6.]

Lin, J.; Lok, P.; Larson, B.; Gade, K.; Luckenbill, S.; and Busch, M., 2012. Earlybird: Real-time search at twitter. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. doi:10.1109/ICDE.2012.149. [Cited on page 1.]

Lu, B.; Hao, X.; Wang, T.; and Lo, E., 2020. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13, 8 (apr 2020), 1147–1161. doi:10.14778/3389133.3389134. https://doi.org/10.14778/3389133.3389134. [Cited on page 15.]

Lu, B.; Hao, X.; Wang, T.; and Lo, E., 2021. Scaling dynamic hash tables on real persistent memory. *SIGMOD Rec.*, 50, 1 (jun 2021), 87–94. doi:10.1145/3471485.3471506. https://doi.org/10.1145/3471485.3471506. [Cited on page 15.]

McAllister, S.; Berg, B.; Tutuncu-Macias, J.; Yang, J.; Gunasekar, S.; Lu, J.; Berger, D. S.; Beckmann, N.; and Ganger, G. R., 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21 (Virtual Event, Germany, 2021), 243–262. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3477132.3483568. https://doi.org/10.1145/3477132.3483568. [Cited on page 1.]

McCandless, M., 2021. Luceneutil: Lucene benchmarking utilities. http://blog.mikemccandless.com. [Cited on page 25.]

Mutlu, O. and Subramanian, L., 2014. Research problems and opportunities in memory systems. *Supercomputing Frontiers and Innovations*, 1, 3 (Oct 2014). [Cited on page 1.]

Nam, M.; Cha, H.; Choi, Y.-R.; Noh, S. H.; and Nam, B., 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19 (Boston, MA, USA, 2019), 31–44. USENIX Association, USA. [Cited on page 15.]

Nguyen, K.; Fang, L.; Xu, G.; Demsky, B.; Lu, S.; Alamian, S.; and Mutlu, O., 2016. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 349–365. USENIX Association, USA. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen. [Cited on page 16.]

O'Neil, P.; Cheng, E.; Gawlick, D.; and O'Neil, E., 1996. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33, 4 (jun 1996), 351–385. doi:10.1007/s002360050048. https://doi.org/10.1007/s002360050048. [Cited on page 5.]

powturbo. Turbopfor: Fastest integer compression. https://github.com/powturbo/TurboPFor-Integer-Compression. [Cited on pages 19, 20, 21, and 22.]

Powturbo, 2023. Turbopfor-integer-compression. https://github.com/powturbo/TurboPFor-Integer-Compression. Accessed: 2023-05-27. [Cited on page 9.]

Sanfilippo, S., 2023. Redis. https://redis.io/. [Cited on page 15.]

Services, A. W., 2023. Amazon elasticache. https://aws.amazon.com/elasticache/. [Cited on page 15.]

Song, Z.; Berger, D. S.; Li, K.; and Lloyd, W., 2020. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 529–544. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi20/presentation/song. [Cited on page 88.]

Stapelberg, M., 2019. Turbopfor: the fastest integer compression. https://michael.stapelberg.ch/posts/2019-02-05-turbopfor-analysis/. Accessed: 2023-05-27. [Cited on pages 9 and 10.]

Thai, A. and Akram, S., 2023. Analyzing fundamental space-time tradeoffs in inverted list compression on hybrid memories. [Cited on pages 6, 7, 8, 9, 10, 18, 19, and 20.]

Twitter, 2023. Pelikan cache. https://pelikan.io/. [Cited on page 15.]

Vogel, L.; van Renen, A.; Imamura, S.; Giceva, J.; Neumann, T.; and Kemper, A., 2022. Plush: A write-optimized persistent log-structured hash-table. 15, 11 (jul 2022), 2895–2907. doi:10.14778/3551793.3551839. https://doi.org/10.14778/3551793.3551839. [Cited on page 15.]

Wang, C.; Cui, H.; Cao, T.; Zigman, J.; Volos, H.; Mutlu, O.; Lv, F.; Feng, X.; and Xu, G. H., 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI 2019 (Phoenix, AZ, USA, 2019), 347–362. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3314221.3314650. https://doi.org/10.1145/3314221.3314650. [Cited on page 16.]

Wang, J. et al., 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. [Cited on page 9.]

WEERAKKODY, C. AND AKRAM, S., 2023. Hybrid memory-aware query caching for real-time search. [Cited on page 25.]

WEINER, J.; AGARWAL, N.; SCHATZBERG, D.; YANG, L.; WANG, H.; SANOUILLET, B.; SHARMA, B.; HEO, T.; JAIN, M.; TANG, C.; AND SKARLATOS, D., 2022. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22 (Lausanne, Switzerland, 2022), 609–621. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3503222.3507731. https://doi.org/10.1145/3503222.3507731. [Cited on page 1.]

WIKIPEDIA CONTRIBUTORS. Okapi BM25 - Wikipedia. https://en.wikipedia.org/wiki/Okapi_BM25. Accessed: May 12, 2024. [Cited on page 10.]

XIE, Z.; CAI, Q.; CHEN, G.; MAO, R.; AND ZHANG, M., 2018. A comprehensive performance evaluation of modern in-memory indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 641–652. doi:10.1109/ICDE.2018.00064. [Cited on page 1.]

XU, J.; KIM, J.; MEMARIPOUR, A.; AND SWANSON, S., 2019. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19 (Providence, RI, USA, 2019), 427–439. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3297858.3304077. https://doi.org/10.1145/3297858.3304077. [Cited on page 13.]

YANG, J.; KIM, J.; HOSEINZADEH, M.; IZRAELEVITZ, J.; AND SWANSON, S., 2020a. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang. [Cited on pages 6, 13, and 15.]

YANG, J.; KIM, J.; HOSEINZADEH, M.; IZRAELEVITZ, J.; AND SWANSON, S., 2020b. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang. [Cited on page 6.]

YAO, T.; ZHANG, Y.; WAN, J.; CUI, Q.; TANG, L.; JIANG, H.; XIE, C.; AND HE, X., 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 17–31. USENIX Association. https://www.usenix.org/conference/atc20/presentation/yao. [Cited on pages 13, 14, and 15.]

YASIN, A., 2014. A top-down method for performance analysis and counters architecture. 35–44. doi:10.1109/ISPASS.2014.6844459. [Cited on pages 22, 23, and 26.]

Zhang, W.; Zhao, X.; Jiang, S.; and Jiang, H., 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, 194–209. ACM. doi:10.1145/3447786.3456237. https://doi.org/10.1145/3447786.3456237. [Cited on page 13.]

Zheng, S.; Hoseinzadeh, M.; and Swanson, S., 2019. Ziggurat: A tiered file system for Non-Volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 207–219. USENIX Association, Boston, MA. https://www.usenix.org/conference/fast19/presentation/zheng. [Cited on pages 13 and 15.]

Zobel, J. and Moffat, A., 2006. Inverted files for text search engines. *ACM Comput. Surv.*, 38, 2 (jul 2006), 6–es. doi:10.1145/1132956.1132959. https://doi.org/10.1145/1132956.1132959. [Cited on pages 1, 5, and 6.]

Zuo, P.; Hua, Y.; and Wu, J., 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18 (Carlsbad, CA, USA, 2018), 461–476. USENIX Association, USA. [Cited on page 15.]