Analyzing Fundamental Space-Time Tradeoffs in Inverted List Compression on Hybrid Memories

Anson Thai and Shoaib Akram

Abstract—In-memory Inverted indices are the state of the art data structure for offering fast query response times. Since DRAM capacity is limited, posting lists are compressed to reduce their storage footprint. Historically, compression has been associated with a detrimental impact on query performance due to the high latency of decompression [1]. In this paper, we revisit this issue using efficient SIMD compression algorithms and arrive at a different conclusion. Our research reveals that compression can simultaneously reduce storage costs and improve performance for single-term queries, with novel algorithms such as StreamVByte and TurboPForDelta. We find that in particular, TurboPForDelta offers high compression ratios (low memory cost) and great performance when combined with SIMD (high throughput). For conjunctive-queries, we find that there is a time-space tradeoff: we can have highly compressed indices (23.6% of the original size) on PM at the cost of 17% QPS performance. However, this consideration of performance cost does not factor in the advantages brought by reduced frequency of disk accesses thanks to compression and large PM capacity. Furthermore, we discover that these recent compression algorithms make query performance storage technology invariant, implying a potential benefit from the use of a persistent memory-backed index for increased memory capacity and persistence.

1 Introduction

Efficient integer compression is crucial for data-driven technology giants like Google, Facebook, and Twitter, which manage immense volumes of data each day. It is estimated that Google stores over 10 Exabytes of data [2]. Storage is not always cheap due to disk failures and running costs, and main memory capacity is limited [3]. This motivates the need for compression, which reduces storage costs but at the expense of slower data access due to high decompression latency [3].

Inverted indices represent one area where fast decompression is of vital importance. An inverted index is the state-of-the-art data structure for rapid full-text search, enabling users to quickly retrieve a list of documents containing a specific search term, along with their respective memory locations. In practice, the inverted index is used to locate web pages and social media posts; for instance, Twitter employs this data structure for managing tweets [4]. As users expect prompt search results, failing to deliver timely results can lead to revenue drops [1].

The state-of-the-art indexing method involves constructing an in-memory hash table and subsequently converting it into either an unsorted or sorted string table (UST/SST) [5]. Typically, portions of the UST are retained in memory to enable rapid access, as the latency associated with loading from a disk is significantly higher [5]. However, main memory capacities are limited and costly compared to disk storage, and inverted index sizes grow proportionally with datasets [1]. Compression can help to mitigate the expense of storing the index in memory. However, it has been shown that there is a performance cost associated with decompression [1].

In this paper, we explore the space-time tradeoffs associated with compressed indices hosted on an emerging storage technology called Intel Optane DC Persistent Memory (PM) as an alternative to DRAM. PM offers higher capacities than DRAM, at the expense of latency. Recent studies indicate PM is unable to provide real-time response rates [3], however, it offers much better capacity scaling than DRAM [1]. As space on PM is more cost-effective, we also investigate whether searching the index in an uncompressed format on persistent memory yields a speed improvement over decompressing from DRAM.

- *A. Thai and S. Akram are with the Australian National University, Canberra. E-mail: anson.thai, shoaib.akram@anu.edu.au*

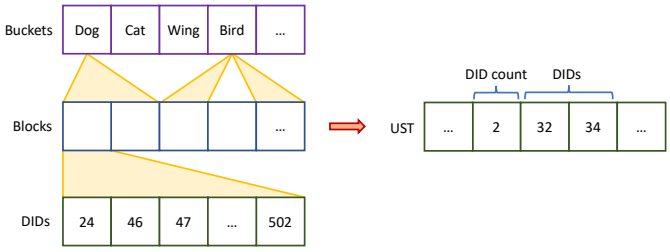


Fig. 1: Inverted Index: Hash table and UST

Previous research has shown that this is true for the Apache Lucene search engine [1], however, state-of-the-art SIMD compression algorithm were not considered. To investigate if this result is true in general, we integrate cutting-edge compression techniques into a search engine called LSIP, a versatile and expandable full-text search engine that accommodates various indexing formats and storage devices. LSIP is built on top of Psearchy, a high-performance search engine developed using the C programming language [6].

We use a combination of state-of-the-art and legacy compression algorithms to validate our results against prior research (in the related work section).

We use Intel VTune for code profiling and performance counter analysis, which incorporates Intel's top-down methodology.

2 Background

In this section we provide some background on Intel Optane DC persistent memory, inverted indices (Psearchy), pointer chasing, and variable sized blocks.

2.1 Intel Optane DC Persistent Memory

Intel Optane DC Persistent Memory (PM) is an emerging storage technology which utilizes 3D XPoint, a new storage medium which "stores information as a change in the material's bulk resistance" [3]. Functionally, Intel PM is similar to DRAM, except that it offers non-volatility and better scalability; Optane is more cost effective (GB/$) than DRAM and offers higher

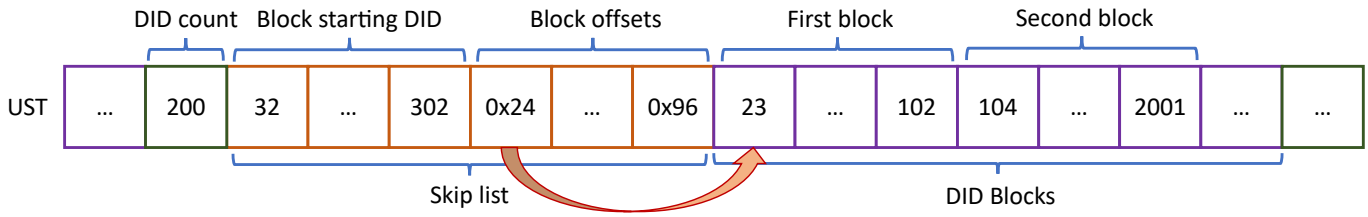


Fig. 2: UST with skip list [5]

module capacities [3]. Unlike traditional persistent storage devices, Optane PM offers byte addressability, removing read-modify-write overhead costs [3]. Unfortunately, Optane PM has higher latency (2-3 times slower) and lower bandwidth than DRAM [3].

In previous research, we have demonstrated that persistent memory can be used to optimize both indexing and query evaluation by eliminating the necessity to persist and load data from the disk, which is expensive and slow [5]. This also removes the need for costly operations, such as sorting, flushing, and merging, typically required for indexing in DRAM/SSD-based search engines [5].

2.2 In-Memory Inverted Indices

We use the LSIP search engine because it accommodates various indexing formats and storage devices. Although the original implementation supports parallel operations, in this paper we will only consider single threaded indexing.

An inverted index is an indexing system that maps terms to their respective locations in a collection of documents. The LSIP index adopts a structure similar to a log-structured merge-tree and is implemented using an in-memory hash table [3]. As depicted in Figure 1, the hash table consists of buckets and blocks. Each term has a corresponding bucket that contains pointers to the first and last blocks for that term. Every block has a pointer to the subsequent block and houses a fixed-size array of postings. Typically, a posting contains the document identifier (DID) of the document the term appears in, and its position within that document. For our use case, we have modified the posting structure by eliminating position metadata, thereby focusing our analysis primarily on DIDs. The hash table can be queried by obtaining the term's bucket through a specific hash function, then accessing the first block from the bucket and traversing the linked list of blocks to generate a posting list.

After indexing the hash table, it is converted into a sorted or unsorted string table (SST/UST). An SST is a file containing a series of sorted key-value pairs, whereas a UST includes unsorted key-value pairs. In this paper, we use the UST format as its indexing time is significantly shorter. As shown in Figure 1, a UST stores the DID count for each term, followed by a list of DIDs. We employ a Berkeley DB (BDB) dictionary to store offsets. When querying the UST using a search term, the BDB is used to obtain the offset of the desired posting list if the term is present in the dictionary. This offset allows for the retrieval of the posting list for the given term [3].

The LSIP index operates as follows:

1) In the first stage, each document is assigned a DID, and a mapping from the DID to the document's location on a file system is stored in a did-to-file map.
2) The indexer iterates over the words in the documents. For each word, a hashing algorithm is used to obtain an offset to its corresponding bucket in the hash table. The bucket contains a reference to a linked list of blocks, which contain postings for the associated term. A posting is created for the term, and inserted into the last block. If the block is full, it is inserted into a new block, which is attached to the end of the linked-list and becomes the new last block.
3) Once the hash table index is built, it is transformed into a UST.
4) Finally, we use a flush operation to ensure that the data is persisted.

We implement a skip list similar to the Powturbo index scheme [7] to improve conjunctive query performance. The data structure allows the query evaluator to skip decompression of blocks when unnecessary. The skip list is integrated with the UST format as shown in Figure 2. For each term, we store the first DID of each block and the block offsets in the skip list. If the term has only one block (or partially filled block), we do not use a skip list.

2.3 Single Term Queries

The evaluation of single term queries involves locating and serving all postings which match the query term. We serve single term queries by first obtaining the offset of the posting list for that term via a dictionary lookup. From the offset we obtain the posting block list for the term. We traverse each block utilizing the skip list and decompress it directly into the result buffer.

2.4 Conjunctive Queries

Serving a conjunctive query involves finding the intersection (AND) of postings for two terms, i.e., postings for the query terms which contain matching DIDs. In general, conjunctive queries tend to be the most expensive step in query processing [2].

In general, fast algorithms such as binary or finger search to perform intersections for posting lists [1]. However, this is inefficient for compressed posting lists since it requires decompressing all blocks before performing the intersection. We implement a skip list to speed up intersection queries by only decompressing blocks when necessary.

When serving conjunctive queries, we first obtain posting list offsets for both terms from two dictionary lookups. From the offsets, we obtain the number of postings and a list of posting blocks for each term. We compare two blocks at a time, and decide to decompress and intersect their postings when the ranges of possible DIDs in each block overlaps. The ranges are obtained from the starting DID of the current and next blocks, which reside in the skip list. In the absence of an overlap, we advance one of the lists to the next block.

2.5 SIMD

SIMD stands for single instruction, multiple data. As its name suggests, it involves applying the same operation on multiple pieces of data. SIMD instructions are special instructions which allow the CPU to execute a single operation on multiple data elements in parallel [8]. SIMD enhances performance by reducing the cost and complexity of managing multiple instruction

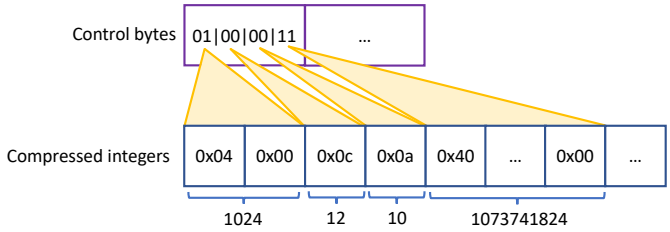


Fig. 3: Streamvbyte scheme [12]

streams to a single stream [9]. The data-parallelism provided by SIMD occurs at the instruction level, enabling these instructions to perform as quickly as scalar instructions [9]. In the best case scenario, this can result in a potential speedup proportional to the vector length [9]. However, the actual speedup may vary significantly based on the memory operations required to transfer data [9]. In addition, starting address alignment may limit the potential speedup; the initial array address in memory should be a multiple of the SIMD length to obtain the best results [9].

To implement SIMD instructions in C or C++ on Intel processors, we use special functions called intrinsics to access lower level vector instructions without writing assembly code [9]. Intel CPUs have special register sets for vector instructions including 128-bit wide XMM registers and 256-bit YMM registers [10].

2.6 VByte

VByte (also known as VarInt) is a popular encoding for compression that is used in search engines such as Lucene [11]. Integers are compressed at byte granularity, utilizing 7 bits of each byte to store the integer's data bits starting from the least significant bit [8]. The remaining 8th bit is utilized as a continuation flag [8]. If the flag is set, it signifies that the integer requires more than 7 bits for representation, and another byte is used to store the integer, and so on [8].

VByte decoding performs best when all integers are compressed to the same size [11]. However, performance suffers heavily from branch mispredictions when compressed integers have different byte sizes [11].

We use TurboVByte in our experiments, which is a fast implementation of VByte provided by Powturbo [7].

2.7 StreamVByte

StreamVByte is a state-of-the-art compression algorithm that offers much faster decompression speeds than VByte [4]. Firstly, it uses 128-bit SIMD instructions to decode chunks of 4 integers at a time. Secondly, 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 [4]. This keeps the processor busy and allows it to determine the position of the next chunk of integers to be decoded without decoding the current chunk [4]. In VByte, the continuation flag bits are interleaved with the data bits, which introduces data dependencies and can increase pipeline stalls [4].

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

We use StreamVByte library by Lemire. [14]

2.8 TurboPFor

TurboPFor is a compression algorithm that offers state-of-the-art decompression speeds while maintaining good compression ratios [13]. We first explain the TurboPFor256 algorithm which uses 256-bit SIMD registers, then extend our discussion to TurboPForDelta256 which extends the algorithm with differential coding.

The TurboPFor256 algorithm encodes integers in blocks of 256 integers; the trailing block may have less than 256 integers [13].

Before each block is a header which specifies its type [13], the four types are:

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

For each block of integers, the algorithm chooses the block type which maximises compression (compression is prioritized over speed) [7]. 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 (variable byte) is used for large integers with non-uniform exceptions (exceptions are explained later) [13].

We now explain each block type, using examples from [13].

The constant block is used when all elements in the block have the same value (Figure 4). The value is stored only once, and the bit width is set to the minimum number of bits that are required to represent this value.

Bit packing is a technique that encodes integers within the range $[0, 2^b)$ where b is the bit width of the encoded integers [11]. The bits of each compressed integer are packed together in memory as shown in Figure 4. If we have 256 integers in the range [0, 8], then the largest number 8 requires 3 bits to represent. Therefore we encode all of the integers with 3 bits.

The bit packing with exceptions (bitmap) block uses the widely known PFor (patched frame of reference) compression scheme [13]. It uses a smaller bit width for bit packing, meaning that the most-significant bits of some large integers are cut off [13]. The algorithm chooses an optimal bit width b such that the majority of integers are smaller than 2^b [14]; in particular, it chooses the bit width that results in the best compression [7]. meaning The missing information 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 (Figure 4) [13]. For the bit packing with exceptions (bitmap) block in Figure 4, we decode the third integer out2 by obtaining the lower portion from the normal bit packing block (000b). We obtain the position of the upper portion in the exception bit packing block from index 2 of the exception bitmap (10110b), and combine them to obtain 10110000b. The total number of exceptions is obtained by summing the 1s in the exception bitmap.

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, as shown in Figure 4.

In the TurboPForDelta256 scheme, differential encoding is applied before performing the TurboPFor256 algorithm to improve the compression ratio. In differential encoding, we store the differences between successive integers instead of the integers themselves [8]. Note that the list must be sorted. This results in better compression because the difference between adjacent integers is generally smaller than the value of each

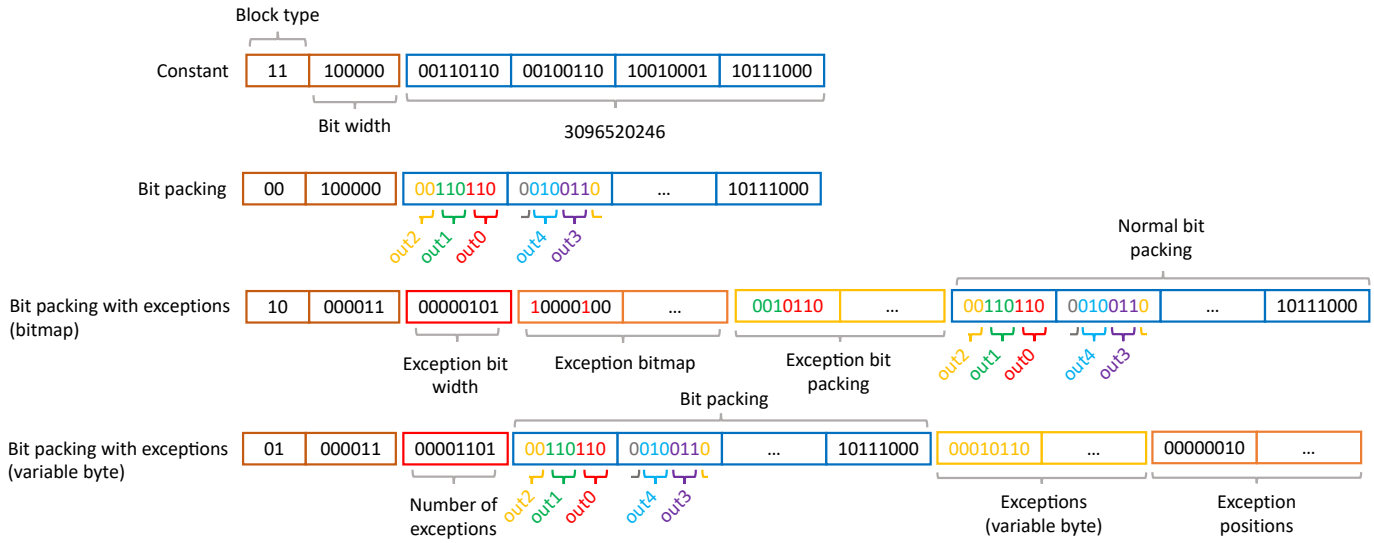


Fig. 4: TurboPFor256 scheme [12]

integer [8]. We can retrieve the original integers after decompression by calculating the prefix sum [8].

2.9 SIMD Bit Packing

Bit packing can be optimized by using SIMD to unpack multiple integers at once. We consider the 128 bit case (for 128-bit SIMD registers) for simplicity. The compressed integers are packed into a series of 4 consecutive 32-bit words in a round robin fashion as shown in Figure 5 [11]. When the current series is filled completely, the left over bits "spill over" to the next series [11]. To decode these integers, we load the first series (top row of Figure 5) into the SIMD register. We then apply shift and mask operations repeatedly to extract 4 integers at a time [11]. Once completed, we load the next series and apply similar operations. The exact shifting and mask operations depend on the bit width and is slightly more complex when the bit-width is not a divisor of 32. A more detailed explanation is provided by Lemire and Boytsov [11].

3 Experimental Setup

The parameters of our server is outlined in Table 1. We use a dual-socket Dell PowerEdge R740 with configured with DRAM, NVM and SSD.

3.1 Configurations

In this paper we examine five indexing configurations on PM and DRAM:

- **Base**: default - No compression.
- **StreamVByte**: Compressed using StreamVByte which uses 128-bit SIMD registers [15].
- **VByte**: Compressed using a fast implementation of VByte by Powturbo [7].
- **PForDelta**: Compressed using TurboPForDelta, which uses the PFor scheme with differential encoding [7].
- **PForDelta256**: Compressed using TurboPForDelta256, which uses 256-bit SIMD registers. Integers are compressed in groups of 256 and the TurboPForDelta algorithm is used to encode trailing blocks with less than 256 integers [7].

System	
Operating System	Ubuntu 18.04.1 Linux OS (5.4.0 kernel)
Hardware	Dell PowerEdge R740 Server
Processor	
Processors	Intel Xeon Gold 6252N
Number of cores	48 physical cores (96 logical)
Core frequency	2.3 GHz
Issue width	4-wide
ROB size	128 entries
Branch predictor	hybrid local/global predictor
Max. outstanding	48 loads, 32 stores, 10 L1-D misses
Cache Hierarchy	
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	shared 36 MB, 64 way, 30 cycle
DRAM	
Capacity	400 GB
Bus frequency	800 MHz (DDR 1.6 GHz)
Bus width	64 bits
Channels	6
Ranks	1 rank/channel
Banks	8 banks/rank
NVM	
Capacity	1.5 TB
Hardware	Intel Optane Persistent Memory
SSD	
Capacity	1 TB
Hardware	3.5-Inch, Seagate, SATA (6 Gbps)

TABLE 1: Target system parameters.

3.2 Index Formation

Our data set is a file constructed from Wikipedia's English corpus (sourced from the Luceneutil website [6]) containing 5,000,000 lines; each line is 1 KB in size. The corpus is stored in DRAM on the tmpfs filesystem to ensure that reading the corpus is not the bottleneck [1]. We build the index using 1 indexing thread for all configurations.

Since the TurboPFor256Delta algorithm encodes integers in blocks of 256 integers [13], we choose a block size of (256+1) (the extra integer is stored in the skip list).

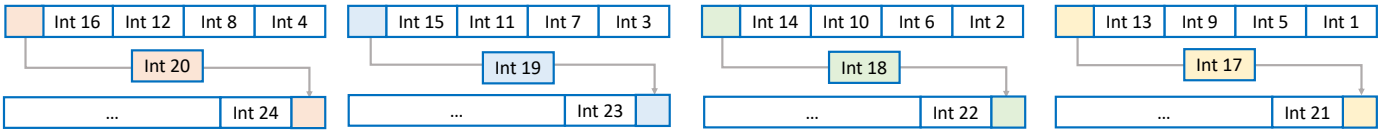


Fig. 5: SIMD bit packing format [11]

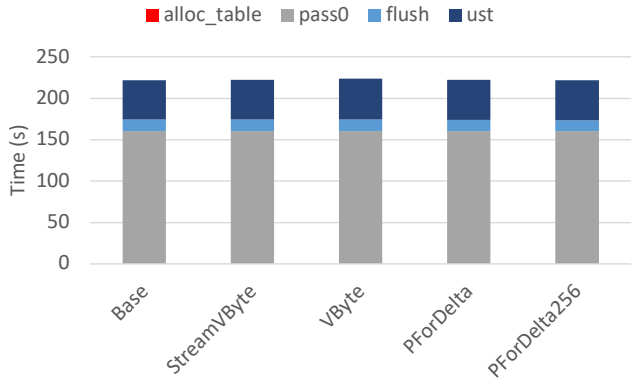


Fig. 6: Indexing time breakdown.

3.3 Query Formation

We have three query types: high, medium and low frequency. We test both single term and conjunctive queries. For each configuration, we run experiments with 10,000 terms of each query type with 10 repeats.

3.4 Measurement Methodology

For the indexer, we obtain measurements for each component of indexing time. We also measure the compressed size of the index. We evaluate the indexing performance based on execution time and compressed size.

From the query evaluator, we obtain measurements for queries per second (QPS) and tail latency. We use VTune to find the proportion of pipeline slots are front-end, bad speculation, back-end (memory and core) and retiring bound using Intel's top-down methodology for microarchitectural analysis. We also use VTune to find the proportion of clock ticks that are L1, L2, L3 and external memory bound. Afterwards, we scale these proportions with performance counters to obtain the breakdown in terms of the total number of pipeline slots or clockticks.

4 Evaluation

We now provide an evaluation for the proposed optimisation against the baseline and state-of-the-art.

4.1 Indexing Performance

We break indexing time into four components:

- **alloc_table**: Allocating persistent memory for the hash table (buckets, blocks and postings).
- **pass0**: Retrieving buckets and blocks and inserting postings into the table.
- **ust**: Transforming the hash-table into the compressed UST format.
- **flush**: Persisting the data.

Figure 6 shows the indexing time break down for our configurations. We observe that the indexing times with compression are very similar to the baseline, indicating that the cost of compression is negligibly small.

We note that UST creation, ust, constitutes only 22% of the total indexing time on average. Indexing the hash table, pass0, is the largest component of indexing time, accounting for 72% of the total indexing time on average. The flush component constitutes 6% on average, and alloc_table time is negligible.

We observe that VByte has the highest compression cost, the ust component is around 1.9 seconds longer than Base. The next highest is PForDelta, with a 0.7 second difference, followed by PForDelta256 (0.5), and StreamVByte (0.001).

Algorithm	Compressed size (bytes)	Compression ratio (%)
Base	2,983,757,756	100%
StreamVByte	2,429,982,074	81.4%
Vbyte	2,680,618,256	89.8%
PForDelta	703,301,122	23.6%
PForDelta256	703,301,122	23.6%

TABLE 2: Compressed index size (100% = uncompressed)

TABLE 2 shows compressed index sizes in bytes and corresponding compression ratios. We see that PForDelta and PForDelta256 offer the best compression ratios, reducing the index to 23.6% of its original size. The reason is two-fold: first, they use differential encoding to encode the differences between successive integers (which are generally smaller than the original integers), and second, they use the PFor encoding which has bit-granularity instead of byte-granularity (VByte and StreamVByte). Streamvbyte is the second most space efficient at 82%, followed by Vbyte at 90%.

4.2 Query Evaluation

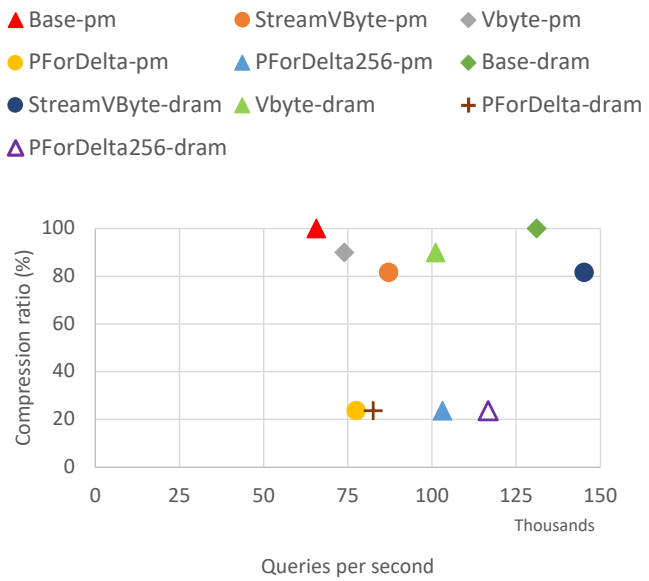
We measure queries per second (QPS) for low, medium and high frequency queries. To analyse the time-space tradeoff, we plot QPS against index compression ratio for single-term and conjunctive queries (Figure 7 and Figure 10).

We use Intel VTune to conduct microarchitectural analysis for each configuration, showing the number of pipeline slots that are: (1) retiring - devoted to useful work, (2) front-end bound - unutilized due to the front-end being unable to adequately supply its back-end, (3) bad speculation bound - wasted as a result of incorrect speculations, (4) memory bound - stalled due to memory operations such as load or store instructions, and (5) core bound - bound by computational resources (Figure 8). We also investigate the number of clockticks bound by stalls on L1, L2, L3 and external memory (Figure 9).

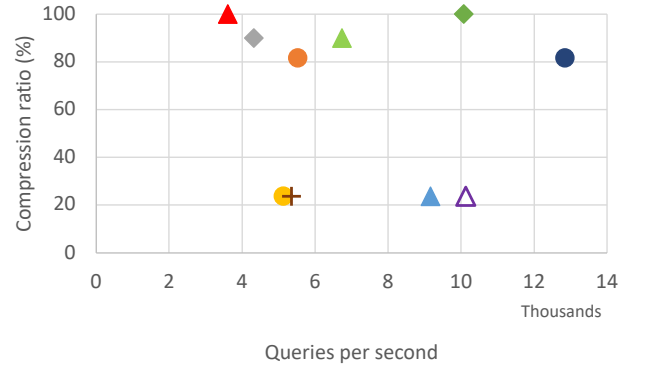
4.2.1 Single-Term Queries

For single-term queries (Figure 7), we observe that DRAM consistently outperforms PM. This is expected because NVM latency is higher and bandwidth is lower than DRAM. Our analysis from (Figure 8a) shows that performance is primarily limited by memory constraints. For Base, Streamvbyte and VByte, there is a notable difference between the memory bound portion for PM and DRAM (Figure 8a). Figure 9a shows that a significant amount of these memory bound constraints are attributable to external memory (i.e. DRAM/PM).

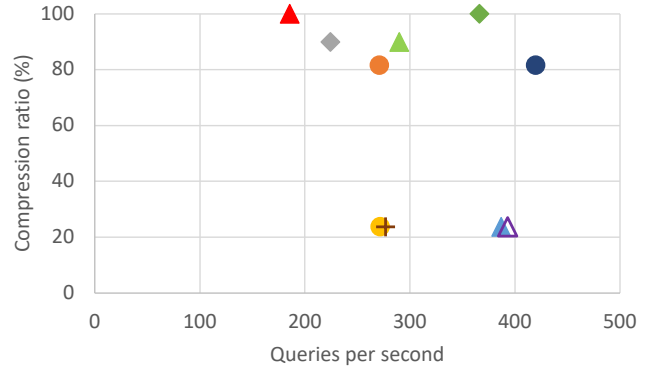
The DRAM/PM performance gap is relatively small for PForDelta configurations (Figure 7). This gap becomes smaller as we increase the query term frequency. For high frequency queries, the performance is nearly identical - PForDelta256-dram is only 1% faster than PForDelta256-pm. From Figure 9a,



(a) Low frequency



(b) Medium frequency



(c) High frequency

Fig. 7: Time-space tradeoff of (a) low (b) medium and (c) high frequency single-term queries (100% = uncompressed).

we can see that this is because the memory-bound proportion is similar between PM and DRAM. Furthermore, Figure 9a reveals that the majority of the memory-bound segment is constrained by the L1 cache rather than the external memory - this explains why performance is memory-technology invariant for PForDelta configurations.

We find that the external memory bound proportion decreases as the index compression ratio improves (Figure 9a). Among different configurations, Base exhibits the highest external memory bound figures, followed by VByte, StreamVByte, and both PForDelta configurations (which are mostly L1 bound). This is because higher levels of compression mean that less data needs to be retrieved from memory, which reduces the pressure on memory bandwidth. As a result, we see that compression can actually improve QPS performance even with decompression overhead - e.g. PForDelta256-pm significantly outperforms Base-pm (Figure 7). In fact, all configurations that compress the index outperform base-pm (Figure 7).

We observe that SIMD boosts performance significantly. In Figure 7, PForDelta256-dram has 1.4x PForDelta-dram's QPS performance for low frequency queries. We also see the same trend for StreamVByte and VByte, however, we should note that StreamVByte has adds more optimisations than just SIMD. For high and medium frequency queries, we observe that configurations that do not use SIMD perform considerably worse than base-dram, e.g., VByte and TurboPForDelta.

We observe that StreamVByte-dram is the fastest for all frequency types (Figure 7). This is because it uses both SIMD (for fast decompression) and compression to maximise performance (less memory bound). StreamVByte-dram outperforms PForDelta256-dram even though the latter has a much better compression ratio (81.4% vs. 23.6%), reducing the memory bound portion. This is because the cost of decompression for good compression is generally higher. For instance, PForDelta256 uses differential encoding on top of TurboPFor256 encoding to further decrease the compressed size which incurs additional overhead when decompressing.

PForDelta256-pm is not only one of the fastest variants, but also has very good compression ratios (comparable performance to PForDelta256-dram) (Figure 7). It is the only PM configuration which outperforms a DRAM configuration. This finding leads to an important insight - the combination of SIMD and high compression on PM provides better performance than no compression on DRAM. This means that we can exploit the benefits of PM (persistence and high capacity) without the cost of higher latency. SIMD hides decompression costs and provides greater throughput, while good compression relieves pressure on external memory. Without SIMD (PForDelta-pm), we see performance cannot match Base-dram (Figure 7). For high frequency queries, PForDelta256-pm is 5% faster than Base-dram while compressing the index to 23.6% of its original size (Figure 7).

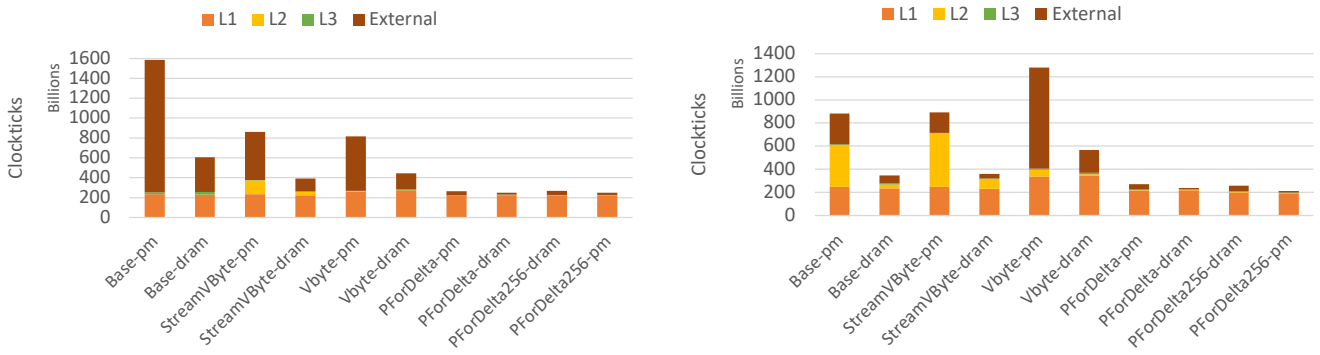
The QPS performance of PForDelta256-pm is only 8% lower than StreamVByte-dram (Figure 7). There are some important tradeoffs to consider here - PForDelta256-pm is slightly slower than StreamVByte-dram, but offers much better index compression (81.4% vs. 23.6%) and additional benefits from PM (persistence and high capacity). This means that with PForDelta256-pm, we can hold much more of the index on PM (due to both compressed size and higher capacities), and avoid retrieving data from disk (which is much slower) as often.

We find that PForDelta256 (both PM and DRAM) performs better relative to Base-dram as we increase the query term frequency. This may be because higher frequency terms have more postings, which improves the compression performance

(a) Single-term queries

(b) Two-term conjunctive queries

Fig. 8: Top-down analysis for (a) single-term and (b) two-term conjunctive high frequency queries.



(a) Single-term queries

(b) Two-term conjunctive queries

Fig. 9: Number of cycles stalled on L1, L2, L3 and external memory for (a) single-term and (b) two-term conjunctive high frequency queries.

of differential encoding - this in turn helps to further reduce the memory bound portion.

We observe that the performance gap between PM and DRAM decreases as we increase the query term frequency. We believe this is a result of improved prefetching capability which helps to hide memory latency. Higher frequency search terms have more posting blocks, which gives the prefetcher more time to operate and learn memory access patterns.

	L		M		H	
	PM	DRAM	PM	DRAM	PM	DRAM
Base	9.9E-05	3.7E-05	1.2E-03	4.3E-04	7.0E-02	4.2E-02
StreamVByte	6.5E-05	2.9E-05	7.6E-04	3.2E-04	5.0E-02	3.8E-02
VByte	8.2E-05	5.6E-05	9.7E-04	6.6E-04	5.8E-02	4.8E-02
PForDelta	7.6E-05	7.3E-05	8.0E-04	7.8E-04	4.6E-02	4.5E-02
PForDelta256	4.6E-05	4.1E-05	4.1E-04	4.0E-04	3.9E-02	3.8E-02

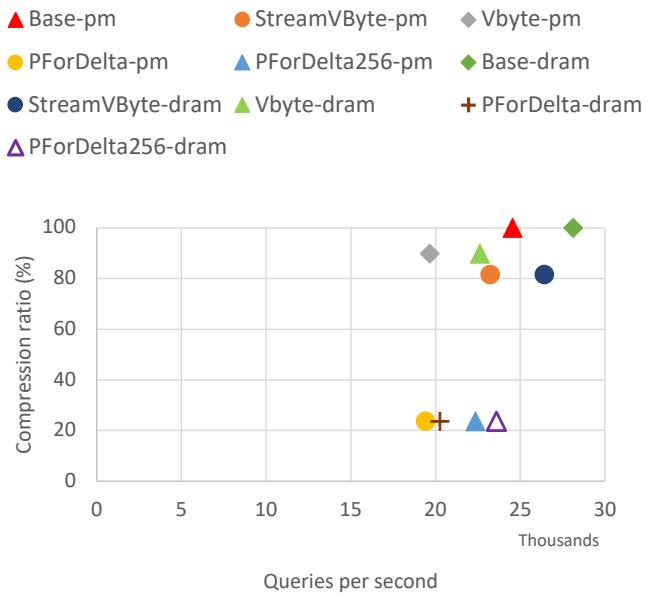
TABLE 3: Tail Latency for low (L), medium (M) and high (H) single-term queries in seconds.

TABLE 3 contains 99th percentile tail latency results for single term queries. We can see that on DRAM, StreamVByte has the lowest tail latency figures. PForDelta256 ties with StreamVByte for H queries on DRAM. We find that PM generally has has worse tail latency than DRAM, however, the difference is small for PForDelta and PForDelta256. On PM, PForDelta256 has the lowest tail latency figures for all query types, followed by StreamVByte (for L and M queries) or PForDelta (for H queries). VByte and Base have relatively poor tail latency results on PM.

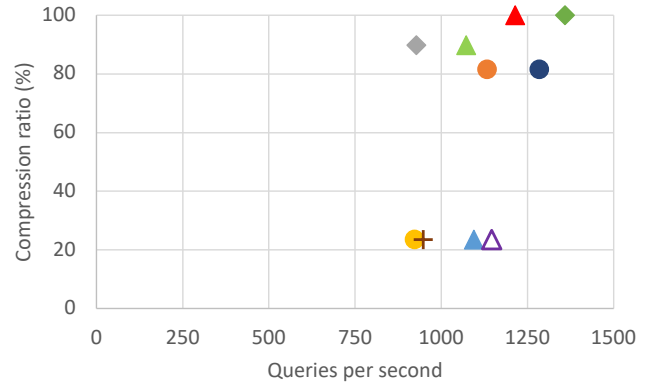
4.2.2 *Two-Term Conjunctive Queries*

Similar to single-term queries, we also find there is a performance gap between DRAM and PM, which decreases as compression ratios improve (Figure 10). This is because better compression reduces proportion of pipeline slots that are memory bound (Figure 8b). Compared to single-term queries, the difference is not as great because the memory bound proportion is much smaller (Figure 8b). This is because the majority execution time is spent on the intersection algorithm, not decompression, due to the skip list which reduces the number of postings that are loaded/decompressed from memory. The intersection algorithm is less memory bound and more speculation and front-end bound as it involves much more branching (Figure 8b). As a result, the data points are relatively close in Figure 10. Since the memory bound portion is already small and has little room for improvement even if we improve compression ratios.

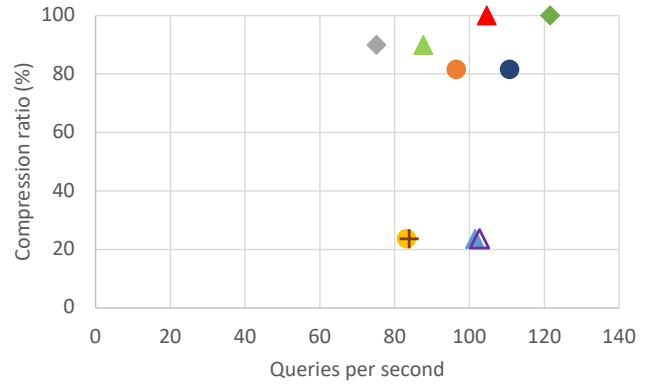
As with single-term queries, SIMD improves QPS (Figure 10). However, the difference is not as large as it was for single-term queries because SIMD is only involved in decompression, which is separate from the intersection algorithm (the main component of execution time). This means that faster decompression speeds do not have as large of an impact on performance.



(a) Low frequency



(b) Medium frequency



(c) High frequency

Fig. 10: Time-space tradeoff of (a) low (b) medium and (c) high frequency two-term conjunctive queries (100% = uncompressed).

We observe that Base-dram is the best performing configuration for conjunctive queries (Figure 10). Base has much higher performance here because it performs the intersection algorithm directly on the uncompressed UST, while other configurations must decompress the DIDs of each term to a DRAM buffer before intersecting them.

For conjunctive queries, the performance of Base-pm improves significantly relative to other configurations (Figure 10). As mentioned earlier, this is because the performance is not as limited by memory, meaning that higher compression ratios are less impactful. It is faster than all non-SIMD configurations, which agrees with the results of Chilukuri et al. [1]. However, we find that Base-pm has 5% lower QPS performance than StreamVByte-dram which employs SIMD.

Chilukuri et al. [1] find that PM is a strong candidate for hosting indices in uncompressed format. Our results indicate that compressed indices in PM may be an even better alternative. For high frequency queries, the QPS performance of PForDelta256-pm is only 3% worse than Base-pm and very similar to PForDelta256-dram Figure 10. Given that PForDelta256-pm provides very high compression ratios with a very small performance cost, it is definitely the more advantageous alternative.

For conjunctive queries, we find that PForDelta256-pm performs worse than Base-dram, which is different to our result for single-term queries. This leads to another important tradeoff to consider: at the cost of 8% QPS performance compared to StreamVByte-dram (fastest compressed index on dram), or a 17% QPS performance compared to Base-dram (fastest uncompressed index on dram), we gain a higher memory capacity, persistence and better compression. With both high PM capacity and good compression, disk accesses will be reduced significantly.

VByte-pm is the worst performing configuration. From Figure 8b, we can see that VByte-pm has a relatively high front-end and memory bound component. Most of the memory bound component is attributed to external memory (Figure 9). This is expected because VByte is known to suffer heavily from branch mispredictions which hurt memory prefetching.

	L		M		H	
	PM	DRAM	PM	DRAM	PM	DRAM
Base	2.0E-04	1.7E-04	2.6E-03	2.4E-03	5.9E-02	4.4E-02
StreamVByte	2.1E-04	1.8E-04	2.8E-03	2.5E-03	6.3E-02	5.0E-02
VByte	2.6E-04	2.2E-04	3.4E-03	3.0E-03	8.6E-02	6.8E-02
PForDelta	2.5E-04	2.4E-04	3.5E-03	3.4E-03	6.2E-02	6.1E-02
PForDelta256	2.1E-04	2.0E-04	2.9E-03	2.8E-03	5.2E-02	5.1E-02

TABLE 4: Tail Latency for low (L), medium (M) and high (H) conjunctive queries in seconds.

TABLE 4 contains 99th percentile tail latency results for conjunctive queries. We can see that Base has the lowest figures on DRAM, followed by StreamVByte. On PM, PForDelta256 has the lowest tail latency fast for H queries, however, Base has lower tail latency for L and M queries. VByte and PForDelta have the worst tail latency results on both PM and DRAM. Compared to tail latency results for single-term queries, the figures are relatively closer together.

5 Related Work

Related work can be divided into two categories: (1) the use of persistent memory for storing compressed inverted indices, and (2) the use of emerging memories, including other non-volatile memory (NVM) technologies, for high-capacity data stores

In most related work, Chilukuri et al. evaluate the performance of compressed and uncompressed indices on DRAM

and PM for the Apache Lucene search engine [1]. They find that uncompressed indices on PM can provide better performance than compressed Indices on DRAM for non-SIMD compression algorithms [3].

In related research, Akram evaluates various hybrid memory and storage configurations for search using inverted indices [3]. He finds that the indexing time can be improved with hybrid configurations (DRAM, NVM, SSD) at low core counts. Our work is the first to evaluate a fully persistent in-memory index.

Yang et al. provide practical insight into the behavior of persistent memory [6], providing guidelines for effective use. They investigate Optane PM by measuring and evaluating its performance relative to DRAM (latency and bandwidth). They discuss methods for maximising the performance of Optane PM.

Recent work proposes NVM-based software stacks for databases, key-value stores, hash tables, in-memory caches, and filesystems [16]–[26]. However, despite their ubiquity, little prior work focuses on exploiting emerging storage for hosting search indices. In the closest related work, Akram et al [3] evaluate Intel Optane PM for various roles (memory expansion, persistent storage, universal memory) in traditional batch search. Their work motivates rethinking the software stack for search with PM hardware. Our work proposes a new stack for real-time search.

Inverted indices are similar to log-structured merge (LSM) trees. Prior work extends LSM-tree-based key-value stores [16]–[19] to exploit heterogeneous storage. Existing efforts try to establish NVM as a new tier in the storage hierarchy, which does not fully exploit NVM's potential as byte-addressable memory. NoveLSM [16] uses an NVM-backed Memtable, while SLM-DB [17] places a global index in NVM and maintains single-level SSTables, unlike traditional LSM.

MatrixKV [18] exploits NVM for fine-grained column compaction. SpanDB [19] exploits different types of SSDs to offer cost efficiency. Our work uses NVM as a first-class citizen in the address space, placing the entire inverted index directly in a managed NVM heap. Other works propose NVM-based filesystems [23], [24]. redesigns a complete service over hybrid memory. Others reconsider service components, e.g., caching over hybrid memory [26].

Recent work exploits fast storage for enabling terabyte-scale managed heaps in big data frameworks [27]. They use *mmio* to grow the heap over a fast storage device (NVM or NVMe SSDs). Our work complements TeraHeap, and we leave evaluating over managed heap backed by fast storage to future work.

uses DRAM and NVM-backed hash tables as index segments. Recent work optimizes hash tables for NVM [28]–[34]. They mitigate NVM writes, and store meta-data in DRAM. Other techniques limit the size of in-DRAM inverted index to mitigate pressure on DRAM [35].

6 Future Work

We identify three directions of future work. First, in this paper, we have explored merge-based intersection methods. However, for bitmap compression methods, intersections can be performed using bit-wise computations which are more efficient and can improve performance [14].

Second, we will explore SIMD to further accelerate bit-wise computations, improving performance. Our rigorous performance counter analysis will direct how best to utilize wide registers in SIMD in the future. Note that although bitmap methods do offer faster intersection performance, they generally have more space overhead and slower decompression [14].

Third, we will implement and evaluate the performance of a hybrid system which uses SSD, DRAM and PM. Unlike today's systems that read index from SSD into DRAM cache, we can build a system that (1) brings compressed data from SSD into DRAM, (2) resolves the query from DRAM, and (3) instead of throwing away the compressed data, stores it in PM to avoid later reads from DRAM. This system will highlight the potential benefits of our approach.

7 Conclusion

Our research conclusively demonstrates that the utilization of SIMD compression algorithms, in particular StreamVByte and TurboPForDelta256, can simultaneously reduce storage cost and enhance the performance of single-term queries. While there is a minor decompression cost associated with conjunctive queries, the trade-off is a significant increase in compression, PM capacity, and persistence. Additionally, we have uncovered that the query performance of a TurboPForDelta-compressed index is storage technology invariant. We conclude that employing a persistent memory-backed index with SIMD-compression is optimal for single-term queries, and a strong candidate for conjunctive-queries.

References

[1] A. Chilukuri and S. Akram, "Fast and scalable text search using non-volatile main memory," 2022.

[2] S. Scargall, *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[3] S. Akram, *Exploiting Intel Optane Persistent Memory for Full Text Search*. New York, NY, USA: Association for Computing Machinery, 2021, p. 80–93. [Online]. Available: https://doi.org/10.1145/3459898.3463906

[4] D. Lemire, N. Kurz, and C. Rupp, "Stream vbyte: Faster byte-oriented integer compression," *Information Processing Letters*, vol. 130, pp. 1–6, 2018.

[5] A. Thai and S. Akram, "Persistent in-memory, text inversion," 2021.

[6] J. Yang *et al.*, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[7] Powturbo, "Turbopfor-integer-compression," 2023, accessed: 2023-05-27. [Online]. Available: https://github.com/powturbo/TurboPFor-Integer-Compression

[8] D. Lemire, L. Boytsov, and N. Kurz, "Simd compression and the intersection of sorted integers," *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, 2016.

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

[10] Intel Corporation, "Intel intrinsics guide," 2023, accessed: 2023-05-27. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[11] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.

[12] D. Lemire, "Stream vbyte: breaking new speed records for integer compression," 2017, accessed: 2023-05-27. [Online]. Available: https://lemire.me/blog/2017/09/27/stream-vbyte-breaking-new-speed-records-for-integer-compression/

[13] M. Stapelberg, "Turbopfor: the fastest integer compression," 2019, accessed: 2023-05-27. [Online]. Available: https://michael.stapelberg.ch/posts/2019-02-05-turbopfor-analysis/

[14] J. Wang *et al.*, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[15] D. Lemire, "Streamvbyte: Fast integer compression in c using the streamvbyte codec," 2018, accessed: 2023-05-27. [Online]. Available: https://github.com/lemire/streamvbyte

[16] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 993–1005. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/kannan

[17] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "SLM-DB: Single-Level Key-Value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 191–205. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[18] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 17–31. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/yao

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

[20] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "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)*, 2020. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[21] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "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*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 194–209. [Online]. Available: https://doi.org/10.1145/3447786.3456237

[22] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid pmem-dram key-value store," in *VLDB*. ACM, 2021, pp. XXX–XXX. [Online]. Available: https://doi.org/10.14778/3461535.3461543

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

[24] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 460–477. [Online]. Available: https://doi.org/10.1145/3132747.3132770

[25] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 427–439. [Online]. Available: https://doi.org/10.1145/3297858.3304077

[26] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "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)*. USENIX Association, Jul. 2021, pp. 821–837. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kassa

[27] I. G. Kolokasis, G. Evdorou, S. Akram, C. Kozanitis, A. Papagiannis, F. S. Zakkak, P. Pratikakis, and A. Bilas, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 694–709. [Online]. Available: https://doi.org/10.1145/3582016.3582045

[28] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 8, p. 1147–1161, apr 2020. [Online]. Available: https://doi.org/10.14778/3389133.3389134

[29] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: An experimental evaluation," vol. 14, no. 5, p. 785–798, jan 2021. [Online]. Available: https://doi.org/10.14778/3446095.3446101

[30] L. Vogel, A. van Renen, S. Imamura, J. Giceva, T. Neumann, and A. Kemper, "Plush: A write-optimized persistent log-structured hash-table," vol. 15, no. 11, p. 2895–2907, jul 2022. [Online]. Available: https://doi.org/10.14778/3551793.3551839

[31] B. Lu, X. Hao, T. Wang, and E. Lo, "Scaling dynamic hash tables on real persistent memory," *SIGMOD Rec.*, vol. 50, no. 1, p. 87–94, jun 2021. [Online]. Available: https://doi.org/10.1145/3471485.3471506

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

[33] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 461–476.

[34] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, ser. FAST'19. USA: USENIX Association, 2019, p. 31–44.

[35] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He, "Mercury: A memory-constrained spatio-temporal real-time search on microblogs," in *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 172–183.