Optimizing a Hash Table-Based Inverted Index for Persistent Memory

Anson Thai and Shoaib Akram

Abstract—The state-of-the-art implementation of full-text search on persistent memory involves creating an in-memory hash table from the data set, then transforming this in-memory hash table into a sorted/unsorted string table (UST/SST). The transformation is required due to poor query evaluation performance of the hash table. We build the hash table on Optane DC persistent memory to achieve state-of-the-art indexing and query evaluation performance [10]. Persistent memory is an emerging hardware technology which offers many advantages over traditional DRAM, with the most crucial being non-volatility and scalability. In this paper, we explore an optimisation of an hash table which utilises variable sized blocks, and evaluate its performance against the state-of-the-art. We find that this optimisation significantly improves query evaluation performance, providing equivalent performance to the UST index for most query types. We present our optimised hash table as an appealing alternative to a UST index; it allows us to eliminate an expensive indexing step while retaining excellent query evaluation performance.

1 Introduction

The development of fast and efficient search is of great interest to companies such as Google, Facebook and Twitter which must handle large amounts of data. The implementation of full text search for large corpora is a two step process which involves both indexing and query evaluation. The purpose of an index is to enable fast and efficient query evaluation. Given a search term, an index can be traversed to produce a list of documents which contain the term and their locations on disk. The state-of-the-art approach for indexing on persistent memory involves building an in-memory hash table, then transforming it into a sorted/unsorted string table (UST/SST). This transformation step is necessary since searching the hash table is slow.

In this paper, we explore an alternative design for the hash table that uses variable sized blocks to improve query evaluation performance. We find that with this optimisation, the hash table performs just as well as a UST for most query types. This means that we can avoid transforming the hash table into a UST, but still retain excellent evaluation performance.

Eliminating this transformation opens up new opportunities for real time search. Real time updates to the index are easier to facilitate since it is easier to append new terms or postings to a hash table than to a UST. The index can be efficiently searched in real time because the query evaluator does not have to wait for the UST transformation to finish.

We build this new design upon a high-performance search engine called Psearchy [3], implemented in the C programming language.

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” [1]. 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 module capacities [1]. Unlike traditional persistent storage devices, Optane PM offers byte addressability, removing read-modify-write overhead costs [2]. Unfortunately, Optane PM has higher latency (2-3 times slower) and lower bandwidth than DRAM [5].

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

We use PM to build an in-memory hash table quickly and efficiently. The special properties of PM mean that operations such as sorting, flushing and merging can be eliminated from the indexing process. These operations are required for DRAM/SSD based search engines [10].

We use PM to improve query evaluation performance since we do not have to load partial indices from secondary storage, which is a much slower than accessing the index directly on PM [10].

2.2 Indexing with Persistent Memory

Psearchy is the baseline search engine (written in C). Although the original implementation supports parallel operations, in this paper we will only consider single threaded indexing. The Psearchy engine is used because it is written in C, which simplifies integration with PM libraries such as Intel PMDK.

An inverted index is an indexing system which maps terms to their locations in a set of documents.

The Psearchy index follows a similar structure to a log-structured merge-tree and is implemented using an in-memory hash table [1].

As shown in Figure 1 The hash table is made up of buckets and blocks. We have a bucket for each term, which contains a pointer to a first and last block for that term. Each block contains a pointer to the next block, and contains a fixed-sized array of postings. A Posting records information about the location of the word: the id of the document in which it is contained and its position inside the document. We can query the hash table by obtaining the bucket for the term using some hash function. We can get the first block from the bucket, then traverse the linked-list of blocks to produce a posting list.

After indexing, the hash table is transformed into a sorted/unsorted string table (SST/UST). A SST is a file which contains a set of sorted key-value pairs. A UST is the same as a SST, except the key-value pairs are unsorted. In this paper, we

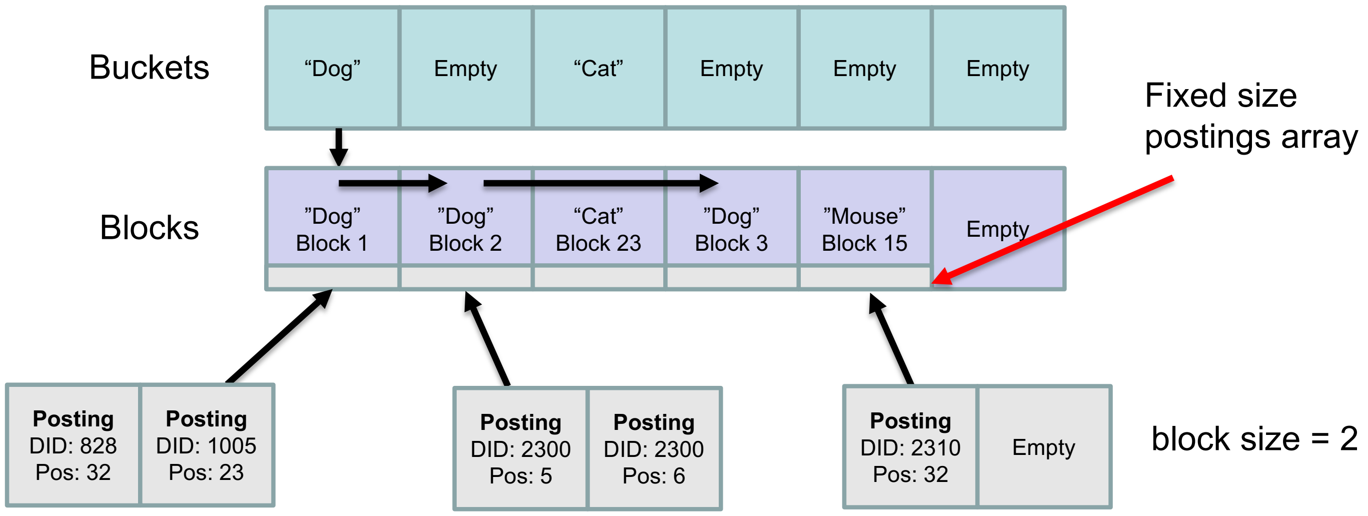

Fig. 1: Hash table design

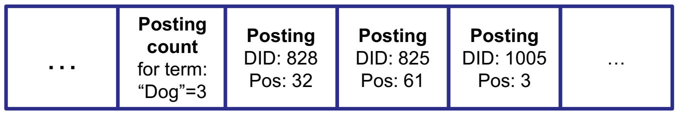

Fig. 2: UST example

focus our comparison with the UST because the indexing time is considerably shorter. We use a Berkeley DB (BDB) dictionary to store the offsets. When querying the UST with 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 is used to read the posting list for that term [2].

Figure 2 provides an example of a UST segment. In this example, we have the posting list for the term "dog", preceded by its posting count. Given the term "dog", we would use a dictionary to obtain the offset of this segment.

The state-of-the-art technique for indexing with persistent memory consists of four stages [10]:

1) In the first stage, each document is assigned a document id, and a mapping from its document id to its 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 find its corresponding bucket in the hash table using open addressing. The bucket contains a reference to a linked-list of blocks, containing the postings for the 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.

2.3 Pointer Chasing

Pointer chasing is ubiquitous in data-intensive applications such as databases and key-value stores. In our search engine, pointer chasing is required to traverse the blocks of the hash table. The blocks use a linked-list data structure; each block holds a pointer which refers to its subsequent block. In this scheme, pointer chasing refers to the act of dereferencing this pointer to retrieve the next block from memory. Traversing the blocks is slow because irregular memory access is an expensive

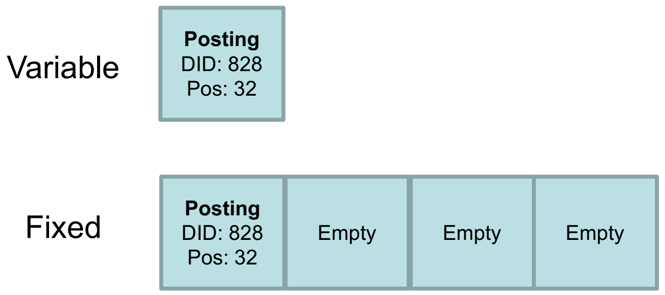

Fig. 3: Fragmentation example

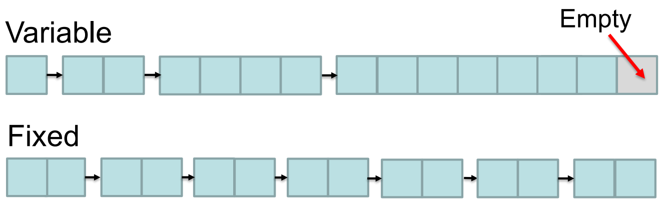

Fig. 4: Pointer example

operation. With irregular access patterns, the out-of-order execution feature of the cpu is limited, and cache and TLB misses are more frequent, resulting in worse latency and throughput [9]. For opposing reasons, traversing an array of postings is fast because we have a linear (regular) access pattern and spatial locality. Therefore, we would like to reduce the number of block traversals as much as possible by "trading" them for less expensive posting traversals. This can be achieved by simply increasing block sizes, however this leads to a trade-off between space and performance; having larger block sizes results in higher fragmentation. A UST structure solves this problem by essentially allocating each term a block size corresponding to its posting count, however, the transformation process is expensive. We can solve this problem efficiently used variable sized blocks, discussed in the next section.

2.4 Variable Sized Blocks

In a hash table with fixed sized blocks, newly allocated blocks have the same capacity. A block's capacity refers to the number of postings it can hold. In this paper, we introduce variable sized blocks to the hash table; when a block reaches its capacity, we create a new block which is twice as large as the previous one. We will refer to the hash table with variable sized blocks as VSB, and the hash table with fixed sized blocks as FSB for conciseness.

One of the reasons that the original hash table design performs poorly is the large number of block traversals required when evaluating a query. In the previous section, we explain how pointer chasing (which is used for block traversal) is an expensive operation. VSB improves query evaluation performance by increasing block sizes, thereby reducing the total number of blocks for each term, resulting in fewer block traversals. This is shown in Figure 4, where we compare the number of traversals for a hash table with fixed and variable sized blocks. Although both have the same number of postings, we have to chase 6 pointers for FSB, but only 3 for VSB.

We originally hypothesized that VSB would reduce fragmentation in the hash table, since it means we allocate less space for unpopular terms. If we allocate a small block when we first encounter a term, it means we reduce over-allocation for very infrequent terms, e.g. terms that appear only one (see Figure 3). In Figure 3, we have a variable sized block with a starting size of 1 posting; for a fixed block size of 4, the fixed sized block allocates unnecessary space for 3 additional postings. Unfortunately, we found that VSB did not reduce fragmentation. In fact, we found that we achieved the lowest fragmentation with FSB for small block sizes.

The transformation from a hash table to a UST is an expensive operation, but it helps to eliminate fragmentation. However, it does this at the cost of flexibility; the UST is not designed to facilitate index growth.

To implement variable sized blocks, we formulated a new hash table design which separates the postings from the blocks (Figure 5). Unlike the previous design where each block contained a fixed size array of postings (Figure 1), the new design stores all of the postings in a separate array. For example, to obtain the postings for the term "dog" in Figure 5, we use a dictionary to obtain its corresponding bucket, and access the first block. The block now points to an offset in the postings array, and the block size represents the number of consecutive postings for that term after that offset. This new design allows us to easily vary the number of postings associated with each block.

2.5 Configurations

In this paper, we evaluate the evaluate our optimisation: VSB (a hash table with variable sized blocks) against a baseline hash table index with fixed sized blocks (FSB) and the state-of-the-art (UST).

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 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 of the corpus 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 for varying block sizes (1, 2, 4, 8, 16, 32, 64, 128) for both VSB and FSB, using 1 indexing thread.

3.2 Query Formation

We have three query types: high, medium and low frequency (H, M, L). We test single term queries, using 8 threads for our query evaluator. For each configuration, we run 8 experiments with 1,000 words from each query type. The query evaluator retrieves the postings and writes them to an output buffer.

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.

3.3 Measurement Methodology

From the indexer, we obtain measurements for posting utilisation, block count, and indexing time. We evaluate the indexer's performance based on the total execution time and the execution time of its components. From the query evaluator, we obtain measurements for queries per second (QPS), tail latency and average blocks traversed. We use the ext4-DAX filesystem to directly access Optane PM, bypassing the buffer cache [1].

4 Evaluation

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

4.1 Indexing Performance

The indexing time results for VSB and FSB are shown in Figure 6 and Figure 7. The time is broken down 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 UST format.
- **flush**: Persisting the data.

In Figure 6, we can see that indexing time for VSB increases with block size due to an increase in **flush** time. **flush** increases with block size because we allocate space for the postings array using the product of the number of blocks and the starting block size.

For FSB, the indexing time decreases as we increase the block size, up to around a block size of 16 (Figure 7). This decrease occurs because **ust** time improves, since we reduce the amount of block traversals when we create the UST from the hash table.

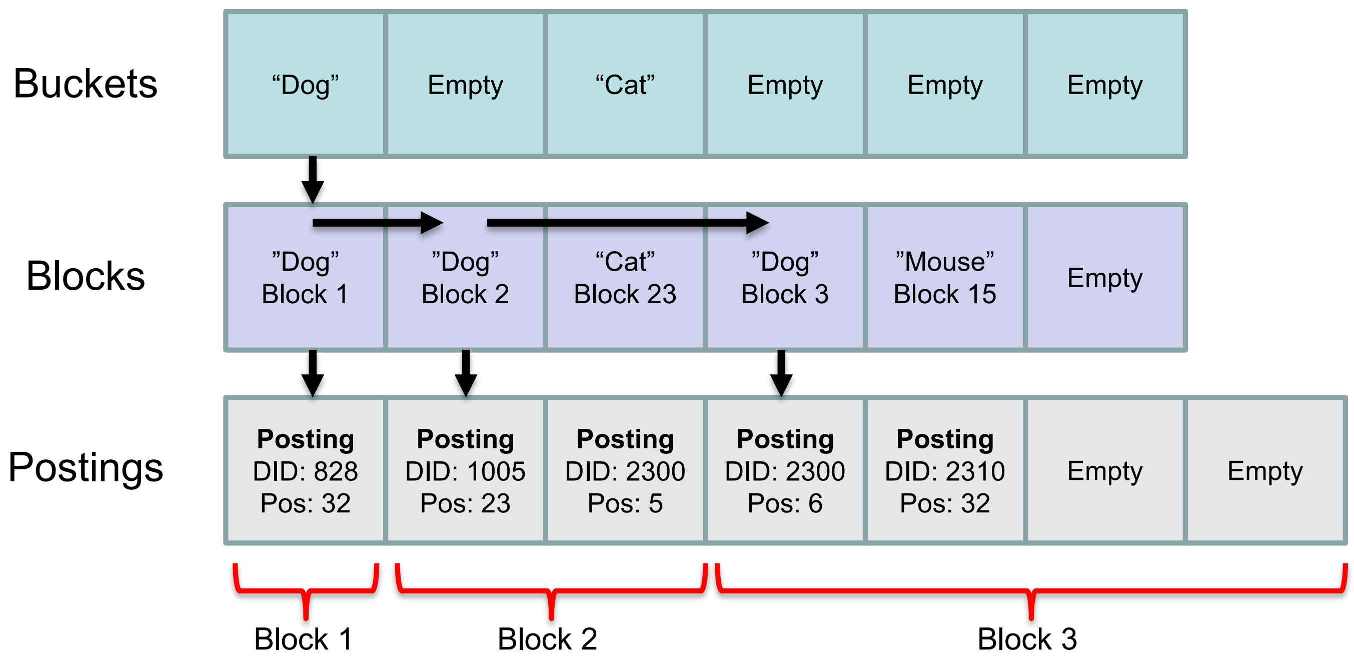

Fig. 5: New hash table design

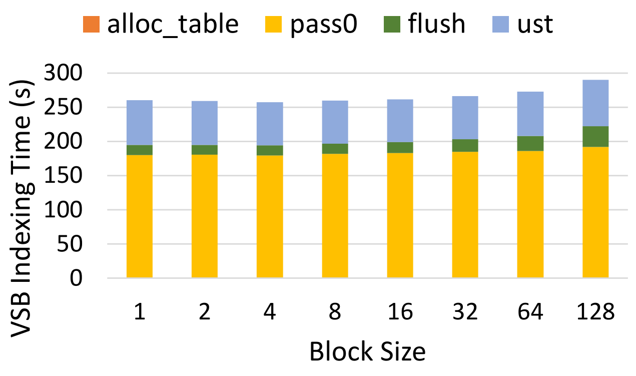

Fig. 6: Indexing time break down for VSB.

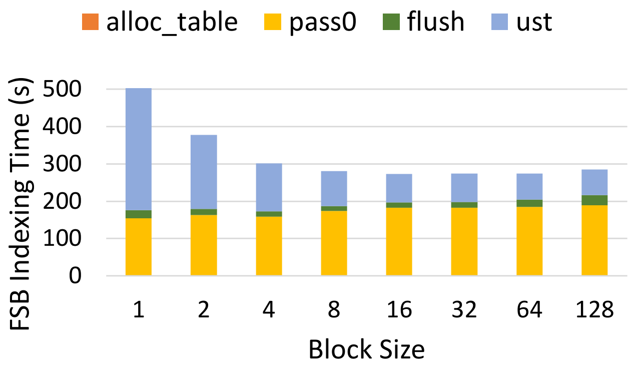

Fig. 7: Indexing time break down for FSB.

Both VSB and FSB have similar indexing times when block size reaches 128; VSB takes 290 seconds and FSB takes 285 seconds. For both, **pass0** takes up the majority of the indexing time, followed by **ust**, **flush** and **alloc_table** (which is too small to be seen). **ust** accounts for around 25% of the total indexing time.

The **pass0** time increases for both VSB and FSB with block size, however, we did not have enough time in this project to identify the cause.

4.2 Query Evaluation Performance

The queries per second (QPS) measurements for each configuration are shown in Figure 8. QPS represents the number of queries served per second. We tested high, medium and low frequency queries (H, M and L).

We found that the QPS results for VSB were on par with UST for H and M queries, but slightly worse for L queries. This result is significant because it highlights an important advantage of VSB over the state-of-the-art: reduced indexing overhead with similar query evaluation performance.

FSB performed far worse than other configurations, with around half the QPS of VSB and ust-pm with a block size of 128. FSB performs worse than VSB because there are significantly many more blocks to traverse on average, as shown in Figure 9. For example, for H queries with a block size of 128, VSB traverses 13 blocks on average, while FSB traverses around 17,000.

The tail latency results of VSB and UST are similar across all query types as seen in Figure 10. In the graphs, FSB has at least double the tail latency of VSB for all query types; this can be attributed to the increased number of block traversals.

4.3 Fragmentation

Unfortunately, our results show that VSB does not improve fragmentation. Posting utilisation is higher for FSB since the block sizes are much smaller on average, leaving less potential for unused space when we allocate a new block (Figure 11). For VSB, we have an increased risk of allocating unnecessary space since we double the block size when we allocate a new block. This result introduces an important tradeoff between performance and utilisation to consider; we can improve the posting utilisation by using small fixed-sized blocks, however, the query evaluation performance is much worse.

Posting utilisation decreases with block size for both VSB and FSB, however, they eventually converge when block size reaches 128.

Figure 12 shows the total block count for FSB and VSB. Both decrease with increasing block size; FSB is much higher

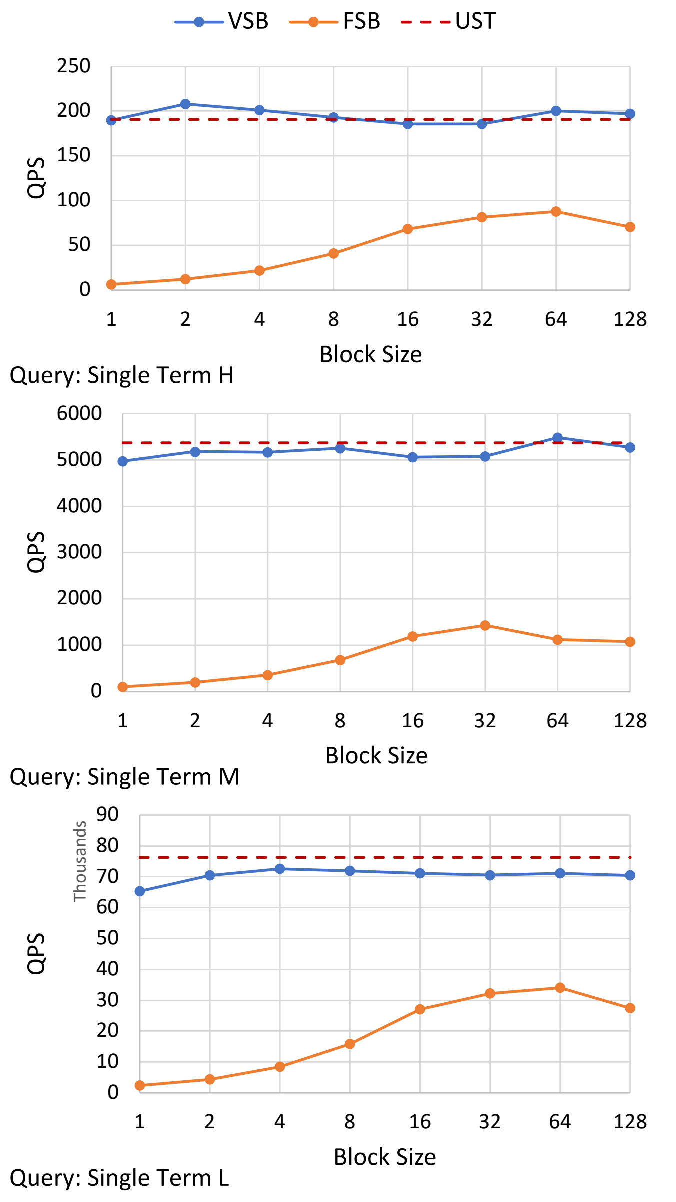

Fig. 8: QPS for H, M, L queries

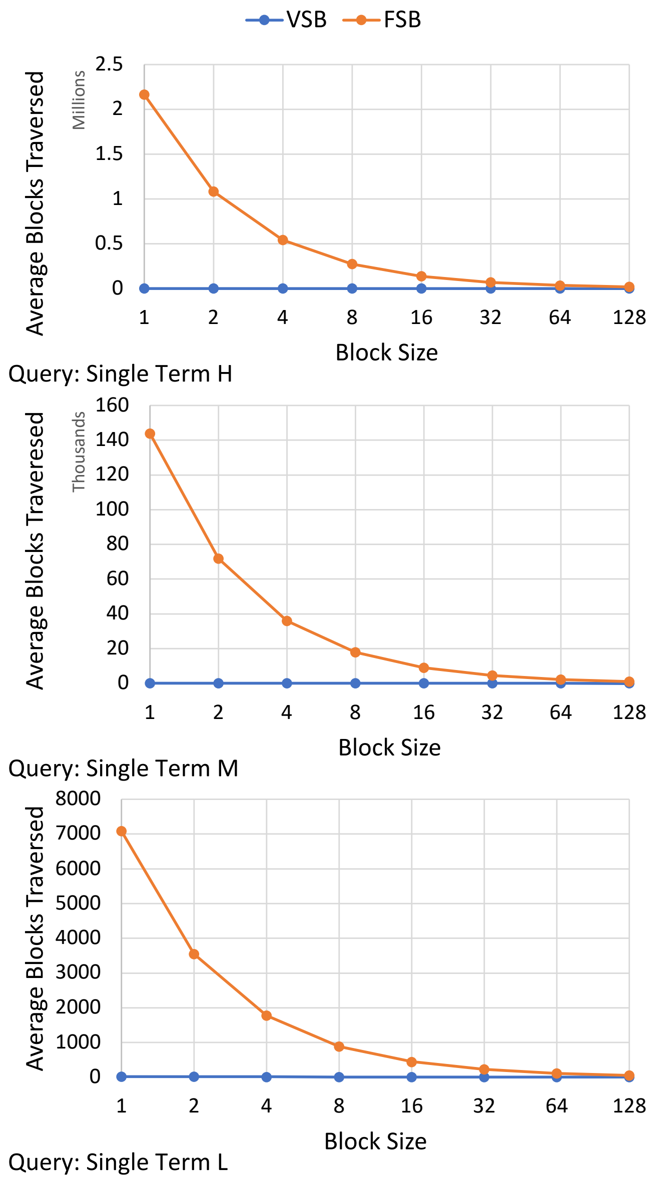

Fig. 9: Average block traversals for H, M, L queries

initially, but converges with VSB. VSB block count is less than FSB, especially for small fixed blocks sizes; using less blocks saves memory. At block size = 128, VSB block count is about 30% lower.

5 Related Work

In related research, Shoaib Akram evaluates various hybrid memory and storage configurations for search using inverted indices [1]. 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 an "empirical guide to the behavior and use of scalable persistent memory" [5]. 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.

Jia et al. perform a performance study on the application of Optane PM for a popular key-value store called RocksDB [7]. They confirm that using persistent memory results in a performance gain. They describe bottlenecks in the current LSM-tree design which inhibit us from exploiting the full potential of PM.

Busch et al. discuss Twitter's real-time search engine, Earlybird [8]. This engine provides both rapid content ingestion while "concurrently supporting low-latency, high-throughput query evaluation" [8]. This is closely related to the long-term goal of this project in implementing efficient real-time search with persistent memory.

6 Future Work

Future work for this project includes using our new design with SSD storage. Since our new design separates blocks and postings into separate arrays, we could consider storing portions of the postings array on the SSD while keeping the bucket and block arrays in memory. This is important so that we can increase the capacity of our index beyond the limits of persistent memory.

Another idea for future work is exploring new methods for implementing variable sized blocks to improve posting utilisation. For example, we could try allocating larger block sizes for frequent terms, and smaller sizes for infrequent terms.

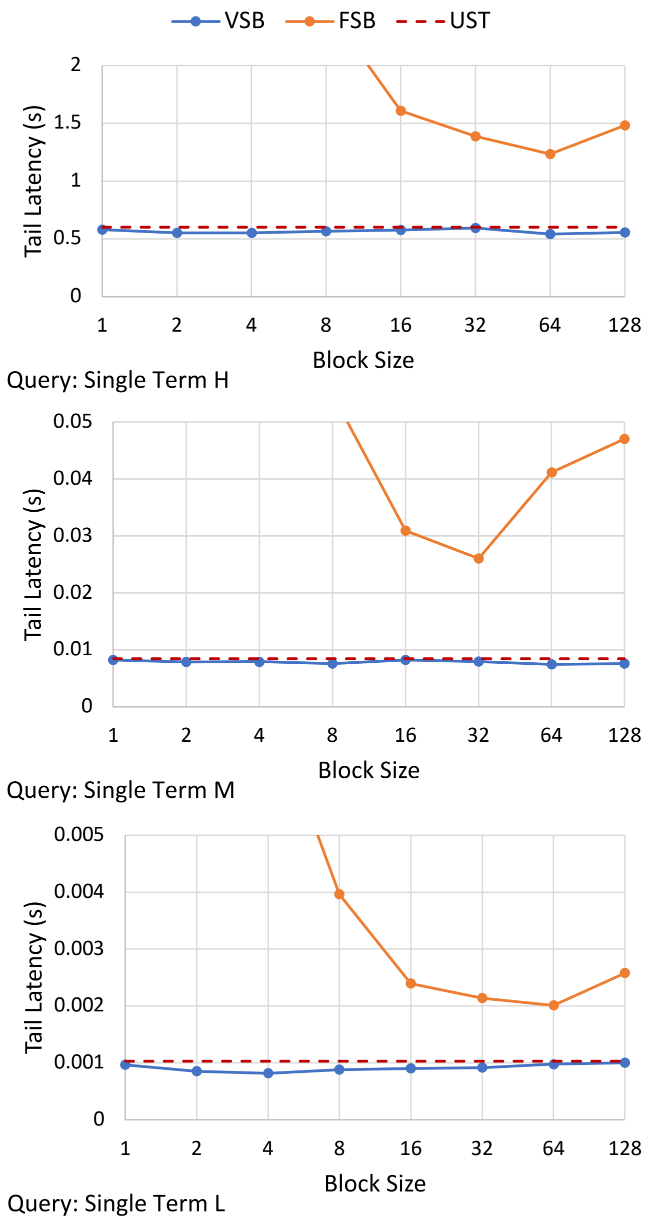

Fig. 10: Tail latency for H, M, L queries

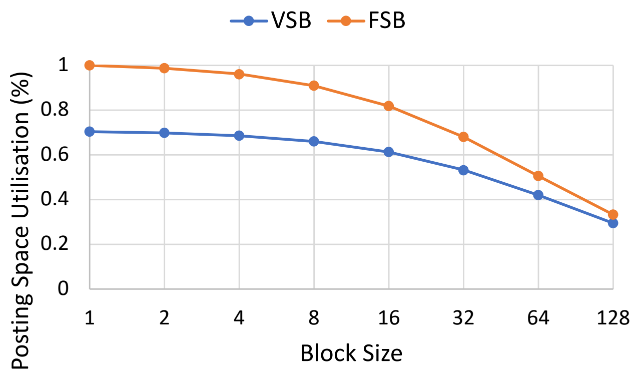

Fig. 11: Posting utilisation

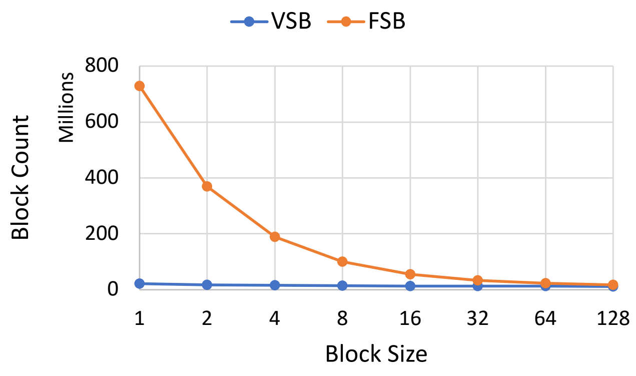

Fig. 12: Block count

7 Conclusion

The key finding of our work is that we can significantly improve the performance of a hash table index using variable sized blocks, matching the query evaluation performance of a state-of-the-art, UST index for most query types. The significance of this result is that we can eliminate an expensive step of the indexing process, while maintaining excellent query evaluation performance. This also introduces an opportunity for efficient real-time search with the hash table, since it can be built and queried in the same format in real time. Finally, we show that there is an important tradeoff between performance and utilization; using fixed sized blocks gives us better posting utilization at the cost of query evaluation performance.

References

[1] Akram, Shoaib. ”Exploiting Intel optane persistent memory for full text search.” Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management. 2021.

[2] Scargall, Steve. Programming Persistent Memory: A Comprehensive Guide for Developers. Springer Nature, 2020.

[3] Lin, Liwei, Xiaohui Yu, and Nick Koudas. ”Pollux: Towards scalable distributed real-time search on microblogs.” Proceedings of the 16th International Conference on Extending Database Technology. 2013.

[4] Lee, Benjamin C., et al. ”Phase-change technology and the future of main memory.” IEEE micro 30.1 (2010): 143-143.

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

[6] McCandless, Michael. Luceneutil: Lucene benchmarking utilities. 2021.

[7] Jia, Yichen, and Feng Chen. ”From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution.” 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2020.]

[8] Busch, Michael, et al. ”Earlybird: Real-time search at twitter.” 2012 ieee 28th international conference on data engineering. IEEE, 2012.

[9] Denning, Peter J. ”The locality principle.” In Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe. 2006.

[10] Thai, Anson and Shoaib Akram. ”Persistent In-Memory, Text Inversion.” 2021.