BOSS: Bandwidth-Optimized Search Accelerator for Storage-Class Memory

Jun Heo, Seung Yul Lee, Sunhong Min, Yeonhong Park, Sung Jun Jung, Tae Jun Ham, Jae W. Lee

Seoul National University

{j.heo, triomphant1, sunhongmin, ilil96, miguel92, taejunham, jaewlee}@snu.ac.kr

***Abstract*—Search is one of the most popular and important web services. The *inverted index* is the standard data structure adopted by most full-text search engines. Recently, custom hardware accelerators for inverted index search have emerged to demonstrate much higher throughput than the conventional CPU or GPU. However, less attention has been paid to addressing the memory capacity pressure with inverted index. The conventional DDRx DRAM memory system significantly increases the system cost to make a terabyte-scale main memory. Instead, a shared memory pool composed of *storage-class memory (SCM)* devices is a promising alternative for scaling memory capacity at a much lower cost. However, this SCM-based pooled memory poses new challenges caused by the limited bandwidth of both SCM devices and the shared interconnect to the host CPU. Thus, we propose BOSS, the *first* near-data processing (NDP) architecture for inverted index search on SCM-based pooled memory, which maintains high throughput of query processing in this bandwidth-constrained environment. BOSS mitigates the impact of low bandwidth of SCM devices by employing early-termination search algorithms, reducing the footprint of intermediate data, and introducing a programmable decompression module that can select the best compression scheme for a given inverted index. Furthermore, BOSS includes a top-k selection module in hardware to substantially reduce the host-accelerator bandwidth consumption. Compared to Apache Lucene, a production-grade search engine library, running on 8 CPU cores, BOSS achieves a geomean speedup of 8.1× on various complex query types, while reducing the average energy consumption by 189×.**

***Index Terms*—full-text search, inverted index, near-data processing, storage class memory**

I. Introduction

Full-text search is one of the most popular web services. To provide users with quality service, efficient processing of the inverted index is necessary, which is a fundamental data structure for managing document information. Over decades, these demands have motivated numerous research works, such as compression schemes for the inverted index [14], [26], [30], [68], [73], [77] and optimization techniques targeting both general-purpose CPUs [25], [41], [53], [57], [64], [72] and GPUs [15], [29], [43], [63], [66], [67], [76]. Recently, hardware accelerators [34], [69] have been proposed to address the architectural limitations of CPU or GPU-based systems, hence providing much higher throughput and energy efficiency.

While these hardware/software techniques effectively improve query throughput, much less attention has been paid to address the memory capacity issue with search. Due to its sheer volume, the inverted index is usually partitioned into multiple *shards* and distributed across multiple nodes [13], [16], [45]. While the size of an inverted index is ever-growing with a steady flood of web documents, the conventional *scale-out* approach to scale memory capacity by deploying more nodes is not cost-effective as this adds not only more DRAM DIMMs but also CPU cores. Thus, this solution charges a super-linear increase in the system cost to the increase in memory capacity. For example, doubling the memory capacity by adding more CPU sockets can increase the system cost by a factor of three [75].

The emergence of commercially available storage-class memory (SCM) such as Intel Optane DC Persistent Memory Module (DCPMM) [36] provides an opportunity to alleviate this cost of memory capacity scaling. SCM is a new tier in the memory hierarchy targeted to bridge the gap between memory (DRAM) and disk. While SCM's performance is worse than DRAM in terms of both latency and bandwidth [22], it has a significant advantage over DRAM for capacity. For example, Intel Optane DCPMM can support up to 512 GB per channel [9], which is 4× larger than the maximum DRAM capacity per channel. Furthermore, a large number of SCM DIMMs can be organized into a memory pool [20], [37], [44] and connected to a CPU socket via a byte-addressable cache-coherent interconnect such as Compute Express Link (CXL) [8] and Gen-Z [7]. The capacity scaling of this memory pool is virtually unlimited via *memory disaggregation* without requiring additional CPU sockets, thus minimizing the increase in the system cost.

Although cost-effective for capacity scaling, this SCM-based memory pool introduces new challenges for architecting a high-throughput full-text search system. Specifically, two different bandwidth limitations have a negative impact on query serving performance. First, the bandwidth of SCM device itself is several-fold lower than that of a DRAM device [70]. Furthermore, the interconnect bandwidth to the host CPU is much lower than that of DRAM channels to yield a much lower bandwidth-to-capacity ratio. Without addressing this bandwidth problem, scaling out the memory pool may introduce a severe bandwidth bottleneck.

Therefore, we propose BOSS, a **b**andwidth-**o**ptimized **s**earch

accelerator targeting storage-class memory. To reduce the data movement between the host CPU and the SCM memory pool (i.e., shared interconnect bandwidth consumption), we adopt the near-data processing (NDP) paradigm to place BOSS in the memory pool, filtering out most of the traffic to the host CPU. To further cut back the shared interconnect bandwidth consumption, BOSS integrates top-k selection module into its pipeline so that only the list of the top-k documents, instead of the entire unsorted list, is transferred to the CPU as an outcome. In addition, to prevent the low bandwidth of SCM devices from becoming the performance bottleneck, BOSS integrates three techniques to save SCM device bandwidth: (i) hardware skip mechanism that only examines relevant data, (ii) minimizing the volume of intermediate data in multi-term query processing, and (iii) programmable decompression module that can select the best compression scheme for a given inverted index.

In summary, this paper makes the following contributions:

- We identify a memory capacity problem in scaling an inverted index. This work is the *first* proposal to host the inverted index in SCM-based pooled memory for cost efficiency.
- We propose BOSS, a specialized NDP architecture for efficient inverted index search, which accelerates the entire search processing in a bandwidth-efficient manner.
- We provide a detailed evaluation of BOSS using a cycle-level simulator as well as synthesizable RTL written in Chisel. BOSS achieves a geomean 8.1× speedup over Apache Lucene running on 8 CPU cores across various complex queries while saving energy consumption by 189× at TSMC 40nm technology node.

II. Background

A. Storage Class Memory

Storage Class Memory (SCM) is a new tier in the memory hierarchy to bridge the gap between memory (DRAM) and disk (HDD/SSD). There are many variants of SCM utilizing different materials and device technologies, such as memristor [39], [71], STT-RAM [32], [59], [74], phase change memory (PCM) [35], [52], [54], and so on. While being byte-addressable and non-volatile, SCM's performance is worse than DRAM in terms of both latency and bandwidth [22]. SCM often features asymmetric read and write bandwidth with much slower writes than reads [18]. For example, the read latency of Intel's Optane DC Persistent Memory Module (DCPMM) is about 3× slower than DRAM, while its read and write bandwidth is about 3× and 6× lower than DRAM, respectively [36], [70].

Despite this performance penalty, SCM is getting a spotlight as a cost-effective alternative to DRAM for higher capacity, especially in a data center environment [18]. For example, Intel Optane DCPMM can support up to 512 GB per channel which is 4× larger than the maximum DRAM capacity per channel. Moreover, this capacity gain of SCM comes at a much lower cost per bit [62] than DRAM. This makes SCM attractive for big data workloads that require large memory.

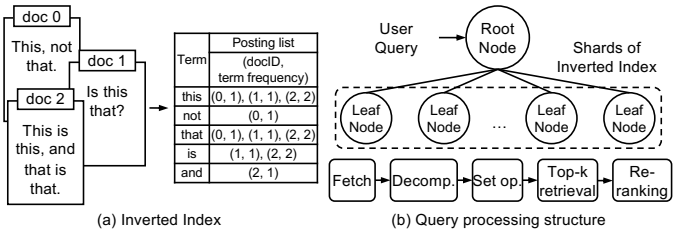

Fig. 1. Query processing using an inverted index

B. Inverted Index Search

Inverted Index. An inverted index is the standard data structure to effectively manage document information in full-text search engines. It is a set of key-value pairs called *posting lists*. A *posting list* relates each term (key) to a list of documents that contain it (value). Each posting list keeps all the unique document identifiers (docIDs), sorted in order, that contain the corresponding term, often together with additional information such as term frequency, document length, and term's position in the document. Here we focus on the case where a posting list is comprised of a docID and the term frequency, which can be expressed as a tuple (docID, term frequency) as in Figure 1(a). An inverted index is usually prepared offline before a query is served. Once created, the inverted list is a (mostly) read-only data structure [19].

Compression for Inverted Index. Since the size of the inverted index is often huge owing to the scale of web documents, it is a common practice to compress it to minimize storage overhead. Each posting list is divided into multiple fixed-size *blocks*, and deltas are computed between two consecutive docIDs in each block. Compression is applied to these deltas instead of the large docIDs themselves to save storage. Some widely used compression schemes are Bit-Packing (BP) [40], VariableByte (VB) [26], PForDelta (PFD) [77], OptPForDelta (OptPFD) [68], Simple16 (S16) [73], and Simple8b (S8b) [14]. Detailed explanations of these schemes are available in Section VI.

Full-text Search Serving System. A typical query serving workflow is comprised of five steps: 1) fetching necessary posting lists in a compressed form from memory, 2) decompressing the posting lists, 3) executing set operations, 4) retrieving top-k results based on scores, 5) (optional) re-ranking. These steps are illustrated in Figure 1(b).

A modern search engine like Google employs multiple distributed nodes to hold the inverted index. It consists of a root node and many leaf nodes (Figure 1(b)). Once a user query is passed down to the root node at the back-end, it is dissected into one or more terms whose corresponding posting list is scattered over multiple leaf nodes [16]. Typically, the inverted index is divided into multiple disjoint partitions, or *shards*, according to the intervals of docIDs [19]. Each leaf node holds a distinct shard and operates only on its shard. Thus, the entire query processing is fully parallelized across leaf nodes [16], [60]. At each leaf node, each term's compressed posting list resides in memory, and relevant posting lists are fetched and decompressed for a given user query.

Then, proper set operations are performed according to the query type. For instance, if the query is "this OR that" (Union query) in the Figure 1(a), posting lists for each term are merged to perform a *union* operation, resulting in docIDs of 0, 1, and 2. If the query is "this AND is" (Intersection query), docIDs that appear in both posting lists are chosen to perform an *intersection* operation, which is 1 and 2. For union, a simple merge sort suffices as the posting lists are already sorted. For intersection, a set of common documents in the two input posting lists are returned through membership testing (i.e., for each docID in the posting list A check if the docID exists in the other posting list B). When the sizes of the posting lists vary greatly, Small-versus-Small (SvS) intersection algorithm [25] is used for efficiency. This algorithm performs a set operation starting from the smallest two lists, hence reducing the total number of comparisons in membership testing.

The ranking is necessary to sort out the most relevant documents. Modern search engines often use a multi-stage ranking algorithm to first retrieve candidate documents, which are re-ranked in the next stage to improve the quality of search while satisfying a tight constraint of response time [27], [47]–[49], [51]. The first stage usually adopts a simple bag of words, a scoring function for fast retrieval of top-*k* candidates. In this work, we adopt the Okapi BM25 (Best Matching) ranking function, which is used by many production search engines [55]. The relevance score of document D in BM25 for a given query Q containing terms $q_i, ..., q_n$ is as follows:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b\cdot\frac{|D|}{avgdl})}$$

$$\text{IDF}(q_i) = \ln\left(\frac{N-n(q_i)+0.5}{n(q_i)+0.5}+1\right)$$

Note that the *query-score* of a document for a given query is the sum of *term-scores* for all terms contained in the query. The term frequency (tf) $f(q_i, D)$ represents the number of times a query term q_i appears in document D. The term $|D|$ and $avgdl$ means the length of document D and the average document length in the document corpus, respectively. k_1 and b are constants usually chosen as $k_1 \in [1.2, 2.0]$ and $b = 0.75$. IDF(q_i), or inverse document frequency of the query term q_i, measures the rareness of a term across the document corpus. In a nutshell, as the term appears more frequently in a relatively short document and as the term is rarer throughout the document corpus, the term has a greater impact on the overall score of the document. In the second stage, various sophisticated re-ranking algorithms are applied, possibly over multiple passes [28], [31], [46]. Recently, neural ranking models [27], [47], [49] have been introduced, which further refine the results of the first stage. Thus, BOSS leaves this second, re-ranking stage to software, while covering all the prior stages up to the first top-k candidate retrieval stage.

C. Memory Capacity Scaling for Inverted Index Search

Scaling Memory Channels. With an ever-growing volume of contents on the web, the demands for additional memory capacity in data centers are stronger than ever [42]. One straightforward approach to scaling memory capacity is scale-out by adding more CPU sockets, which provide more DRAM bandwidth and capacity. However, this accompanies a super-linear increase in the system cost with more CPU cores [50], [75]. To obtain 10 TB of memory, we need a few tens of CPU sockets, assuming each CPU socket can hold up to a few hundreds of GBs of DRAM.

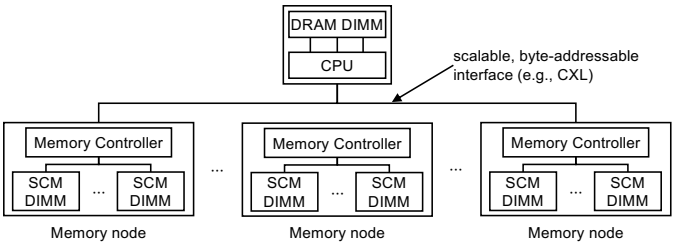

Fig. 2. SCM-based pooled memory architecture

The SCM-based memory system can alleviate this problem of an increased hardware cost as it can accommodate the same capacity with much fewer CPU sockets. For example, Intel Optane DCPMM requires only one-eighth of the CPU sockets compared to the DDR4 DRAM-based memory system. For the same capacity of 10 TB, we only need 4 CPU sockets assuming a single socket has six memory channels, and a 512GB Optane DCPMM is attached to each channel.

SCM-based Pooled Memory. SCM-based pooled memory provides a very cost-effective option for scaling memory capacity. Figure 2 shows an example of SCM-based pooled memory architecture. A memory pool consists of multiple memory nodes, each of which is comprised of multiple SCM DIMMs with a memory controller. The memory nodes are connected to one or more CPU sockets via a memory-semantic cache-coherent interconnect such as CXL [5] and Gen-Z [7]. We can place as many memory nodes as we want in a single memory pool for memory capacity scaling. A performance issue might arise as the number of memory nodes increases, as they all share the same link to the host CPU with a fixed bandwidth (e.g., 64 GB/s for a single CXL link), reducing the memory bandwidth-to-capacity ratio.

D. Hardware Acceleration of Inverted Index Search

Commodity CPUs and GPUs are not an ideal platform for executing inverted index search [34]. Since the memory hierarchy of CPUs is originally designed to boost performance by exploiting data reuse, it becomes a source of inefficiency as accessing an inverted index features very little data reuse. GPU-based acceleration of inverted index search is also challenging due to the limited memory capacity (only up to tens of GBs) of a modern GPU. This incurs a significant overhead for transferring data between the host CPU memory and the GPU memory over a narrow PCIe channel. Thus, hardware accelerators with a custom memory hierarchy have emerged as a viable alternative to those commodity platforms [34], [69]. Among related works, IIU [34], a specialized hardware architecture for processing essential operations of inverted index search, including decompression, set operations (intersection,

union), and scoring, is the most relevant one. The main limitation of IIU is that it results in suboptimal performance on SCM-based memory systems, a cost-effective way to meet the large capacity requirements for a large-scale search. Specifically, i) IIU's intersection incurs frequent random memory accesses with its binary-search-based intersection algorithm, and ii) its union algorithm ends up retrieving much more data from the memory than required without a pruning mechanism. Finally, IIU only supports a very specific compression scheme that is tied to its hardware design, which limits its applicability. BOSS, however, carefully addresses the limitations of IIU and enables the efficient hardware-accelerated search on a cost-efficient SCM-based memory system.

III. Strategies for Bandwidth Savings in BOSS

The primary design objective of BOSS is bandwidth efficiency for both SCM devices and the shared interconnect to the host CPU. Existing accelerators for inverted index search, such as Pinaka [69] and IIU [34], assume high-bandwidth DRAM-based memory systems, which would perform sub-optimally on the SCM-based pooled memory. Efficient utilization of the scarce memory bandwidth is a key differentiator of BOSS from the existing accelerators. We will discuss high-level strategies for saving the shared interconnect bandwidth to the CPU (Section III-A) and saving the SCM device bandwidth (Section III-B). The implementation of these strategies is described in greater detail in Section IV.

A. Saving Host Interconnect Bandwidth Consumption

Near-data Processing. BOSS is a near-data processing architecture where an inverted index search is performed right next to the SCM DIMMs, not on the host CPU. Thus, BOSS can fully exploit the internal bandwidth of the SCM DIMMs of the same memory node. It contrasts with a single large accelerator deployed on the host side, which can fetch data only as much as the bandwidth of the shared interconnect regardless of the number of aggregated memory nodes in the memory pool.

Hardware Support for Top-*k* Selection. Existing accelerators [34], [69] do not cover the top-k selection operation in their computing units. Instead, they only perform set operations for the inverted index and store the results in the host memory, leaving the task of top-k selection to the host CPU. If deployed on the SCM-based pooled memory, these host-side accelerators output a scored, yet unsorted, list of documents in memory, which should be transferred back to the host CPU for sorting and top-k selection. By integrating top-k selection logic into its computing units, BOSS greatly reduces the volume of the intermediate results that need to be read by the CPU. The top-k list is just a tiny fraction of the entire inverted index in the pooled memory. k is usually small since most users browse only the first few pages of search results [27], [38], [48]. This enables the memory pool to scale-out further without introducing a performance bottleneck of the shared interconnect.

B. Saving SCM Device Bandwidth Consumption

Early Termination. The existing accelerators [34], [69] consume a large amount of DRAM bandwidth for query serving as they exhaustively score all the documents that satisfy the query conditions. However, *early termination (ET)* techniques allow us to skip a large portion of the documents for scoring that satisfy certain criteria. The key idea of ET is that, if a document has no chance to get into the final top-k result, we can skip the evaluation of this document. More specifically, it estimates an upper-bound query-score of a document and skips it if the upper-bound is smaller than the smallest query-score in the current top-k documents (which we call *current cutoff*).

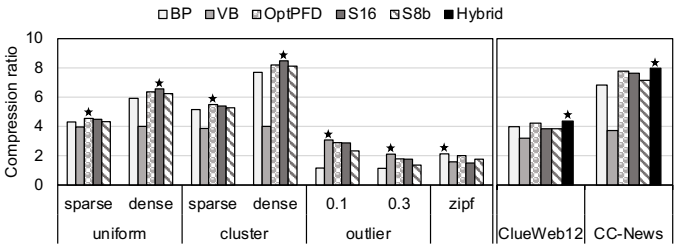

Fig. 3. The compression ratio of several schemes with synthetic datasets and real-world datasets. Higher is better. "Hybrid" applies the best compression scheme for each posting list for the entire dataset. Star indicates the best compression scheme for each dataset.

ET techniques are especially effective for OR queries, which often have redundant computation. BOSS integrates ET into its hardware pipeline for a union at two different points: (i) before fetching compressed data and (ii) before scoring documents. The former shares similarity to BlockMaxWAND [30] and interval-based pruning [24] to estimate the score upper-bound at a block level. In contrast, the latter takes WAND [21], which estimates the upper-bound query-score at a document level.

Multi-term Query Processing Optimization. Supporting a multi-term query (i.e., processing more than two terms) efficiently in hardware is crucial to reduce bandwidth usage. Pinaka [69] uses a naïve merge tree to perform multiple set operations at once. For an intersection query with multiple terms, it is not an effective solution. As Pinaka uses a simple merge, it needs to load the posting lists for *all* the terms constituting a query. However, performing iterative intersections using the Small-versus-Small (SvS) algorithm reduces the size of the posting list as an intersection operation always take a common subset of the two original posting list. Unlike Pinaka, IIU [34] adopts the SvS algorithm for the multi-term intersection query. However, it generates unnecessary memory accesses to load/store intermediate data. Instead, BOSS implements a pipelined intersection that performs multiple intersections at once, obviating the need for storing intermediate data in memory. This effectively reduces the wasted bandwidth for accessing both posting lists and intermediate data.

Programmable Decompression Module. Figure 3 compares the compression ratio of seven synthetic datasets using five different compression schemes implemented with [6]: BP [40], VB [26], OptPFD [68], S16 [73], S8b [14]. We also apply compression schemes on two real-world datasets, ClueWeb12 [3] and CC-News [4]. We use a *hybrid* approach, which applies the best compression scheme for each posting list for the real-world datasets. Since OptPFD outperforms PFD, we only consider the former. All the synthetic data streams are comprised of

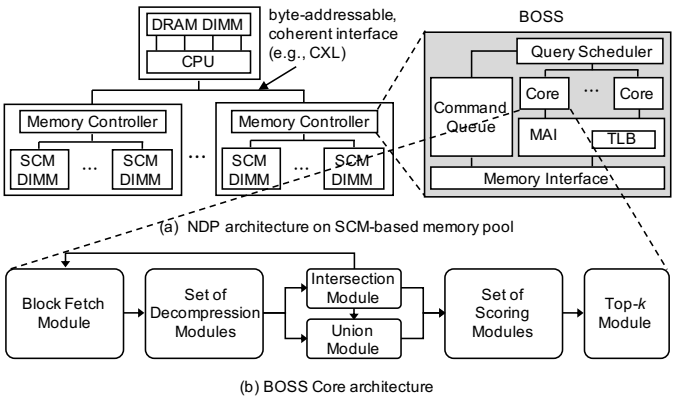

Fig. 4. Overall structure of BOSS

10M integers. We uniformly pick integers over a given range which is from 0 to $2^{28}-1$ for sparse and from 0 to $2^{26}-1$ for dense to make *uniform* streams. *Cluster* streams also consist of uniformly picked integers but from randomly chosen clusters, not from the whole range. For *outlier* streams, we pick integers following the normal distribution with a mean of 2^5 and a standard deviation of 20 but with 10% and 30% of outlier values. The distribution of *zipf* stream follows Zipf's law [12]. The figure shows that the best compression scheme differs depending on the characteristic of the input stream. To achieve the highest compression ratio, BOSS supports various compression schemes by dynamically re-configuring part of its datapath inside the decompression module. This approach is hardware-efficient as it reuses the same hardware primitives for executing different decompression schemes. Furthermore, a new decompression scheme can also be supported if it can be expressed by composing those primitive units.

IV. Design of BOSS

A. Overview

Overall Structure. Figure 4 shows the system architecture of BOSS. We design BOSS to handle the whole inverted index search pipeline except for the final re-ranking stage which will be handled in software. BOSS is placed in the memory controller of each memory node and consists of several BOSS cores and peripherals. When a search query arrives from the host CPU, BOSS first buffers the query in its command queue. The query scheduler is in charge of assigning queries to BOSS cores. Once scheduled, the cores in BOSS perform pipelined execution of fetching posting lists, decompression, set operation, scoring, and top-*k* selection. Figure 4(b) depicts the dataflow within a BOSS core. If BOSS core needs to access SCM memory during query processing, Memory Access Interface (MAI) handles all memory requests including address translation through a local TLB. Note that no remote access is necessary as a BOSS core operates only on the shard in the local node. Finally, the top-*k* results are delivered to the host via the shared, byte-addressable interconnect.

Index Structure and Per-block Metadata. The inverted index is a sorted list of posting lists in the lexical order of the indexed terms. Each element of the posting list is composed of a tuple of docID and term frequency (*tf*) for BM25 scoring. Deltas (also known as *d-gaps*) are computed between two consecutive docIDs, and the deltas, instead of raw docIDs, are compressed for compression efficiency [65]. We use the hybrid approach, to minimize the space overhead. A posting list is divided into *blocks* of 128 values except for Simple16, which has a variable number of values per block according to their distribution. Each posting list maintains its own record of metadata for efficient skipping and decompression. For efficient block skipping, it holds the first (4B) and the last (4B) uncompressed docID of the block, the maximum term-score (4B) in the block, and the address offset of the compressed block (4B). For decompression, it contains the number of elements in the block (7 bits), encoded bit-width (5 bits), and the offset of the first exception value and index (12 bits). In total, the size of the metadata per block is 19B. This information is used by various compression schemes supported by BOSS.

B. BOSS Query Execution Flow

BOSS supports two types of queries (i.e., union and intersection) and handles the query differently depending on the query type. Note that a more complex (mixed) query can be created by composing them. When the two types of queries are mixed, BOSS executes the query according to the priority of the set operation. In what follows, we describe the execution flow of each query type.

Union Query. The union module supports a 4-way merge, and the BOSS core processes up to 4 terms for a union at once. Union of more than 4 terms is processed using multiple BOSS cores. Thus, there is no iterative loop within a single core when processing a union query. The block fetch module fetches block streams of all the terms being processed together. Block skipping is applied to save SCM bandwidth by inspecting the metadata of individual blocks. Once blocks are fetched, the BOSS core subsequently performs decompression, union, scoring, and top-k selection to generate the outcome.

Intersection Query. Unlike a union query the BOSS core processes only a pair of terms at a time for an intersection. For a 3- or 4-term intersection query, the BOSS core iteratively fetches, decompresses and performs intersection. For example, for a 3-term intersection query with terms A, B, and C, the BOSS core first computes the instruction of two posting lists for terms A and B. Then, the intersection of two posting lists (A∩B) is fed back to the block fetch module (Figure 4(b)). The block fetch module fetches blocks for term C to compute intersection with A∩B. The rest of the process (decompression, intersection, scoring, top-*k*) is the same as a 2-term intersection query. This execution flow significantly improves fetching efficiency. We discuss this block skipping mechanism in greater detail in Section IV-C (Block Fetch Module).

Mixed Query. For a mixed query having both unions and intersections, BOSS performs *intersections* first. For example, for a 3-term mixed query A∩(B∪C), the BOSS core performs two intersections first and then a union between two intermediate results ((A∩B)∪(A∩C)). This is a decision of BOSS for bandwidth and storage efficiency for intermediate results as

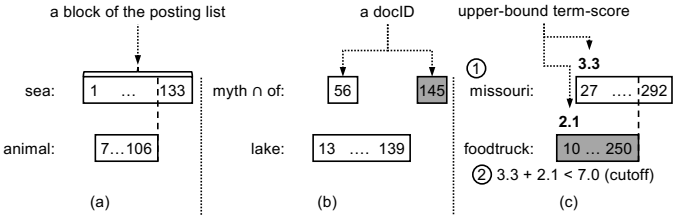

Fig. 5. Examples of block fetch operations: (a) animal ∩ sea, (b) myth ∩ of ∩ lake, (c) missouri ∪ foodtruck

an intersection always yields a smaller posting list than the original input posting lists.

C. BOSS Core

This section explains the details of individual building blocks of the BOSS core. As shown in Figure 4(b), it includes the following six modules: the block fetch module, the decompression module, the intersection module, the union module, the scoring module, and the top-k module.

Block Fetch Module. The block fetch module selectively fetches *candidate blocks* from the posting lists of the given terms to reduce the volume of block loads. The candidate blocks are blocks that can contain *candidate documents*, ones that satisfy the query condition and can potentially be included in the final top-k outcome. For intersections, candidate documents should exist in *all* input posting lists; for unions, it suffices for them to exist in *any* of the posting lists.

For an intersection query, there is an *overlap check unit*, which checks the query condition. For each block, the overlap check unit inspects the block metadata (the first and the last docID fields in particular) to check an overlap with the other input posting list. Figure 5(a) and (b) illustrate running examples of this overlap checking operation. Figure 5(a) shows an intersection between *animal* and *sea*. As the block from *animal* overlaps with the block of *sea*, these blocks are considered as candidate blocks and loaded. Figure 5(b) shows an intersection of three terms: *myth*, *of* and *lake*. Since this 3-term query is processed in a pipelined manner, intermediate result docIDs (56 and 145 in this example) from an intersection of *myth* and *of* are fed back to the block fetch module. Since the next block from term *lake* overlaps with one of the intermediate results (i.e., 56), this block is considered as a candidate block.

Unlike an intersection query, the query condition for union is satisfied by all documents included in the input posting lists. Thus, the query condition cannot reduce the volume of block loads for unions. Instead, BOSS applies an early termination (ET) algorithm inspired by BlockMaxWAND [30] and interval-based pruning [24]. To be considered for inclusion in the final top-k outcome, a document must have a higher score than the current *cutoff* to become a candidate document. Since computing the exact score of a document requires the full computation of the pipeline, we use the block's maximum term-score (or a sum of them for a multi-term query) as an upper-bound of the exact score. This criterion is examined by a *score estimation unit* within the block fetch module. It calculates the upper-bound query-score of a document by summing the maximum term-scores of the blocks that overlap

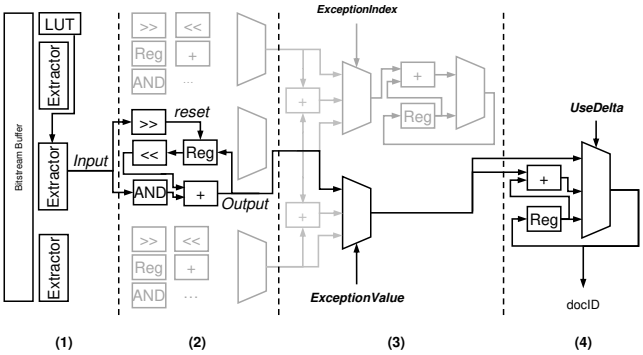

Fig. 6. Four-stage structure of the decompression module

with the document. This information can be retrieved by the overlap check unit discussed in the previous paragraph. If the upper-bound query-score is smaller than the cutoff (i.e., the score of the lowest-rank one in the current top-k list), there is no chance for this document to make a final top-k entry. Thus, BOSS can safely skip this block with no candidate document.

Figure 5(c) illustrates an example of a union of two terms (*missouri* and *foodtruck*). The score estimation unit calculates the upper-bound query-score for the shaded block for term *foodtruck*. The upper-bound query-score for those documents whose docIDs range from 10 to 26 will be 2.1, and those ranging from 27 to 250 will be 5.4 (=3.3+2.1). Since these upper-bound query-scores are smaller than the current cutoff (7.0), the shaded block will not be loaded. If the maximum term-score for the shaded block were 4.0 (instead of 2.1), the documents whose docIDs fall between 27 and 250 would become candidate documents (3.3+4.0=7.3). In this case, both the shaded block and the corresponding block from term *missouri* will be loaded for the downstream execution path.

Decompression Module. Figure 6 shows the structure of the decompression module. By analyzing multiple popular compression schemes used for inverted indexes we came up with a canonical structure composed of the following four stages, which can flexibly support multiple compression schemes: (1) individual payloads are extracted from the serialized bitstream; (2) extracted payloads get manipulated differently according to each compression scheme; (3) checks if the current payload is an exception and handles it; (4) if the compression scheme uses delta encoding, the payload is added to the previous value to obtain compressed docID. The key observation is that the datapath is nearly the same for all those compression schemes except for the second stage. Based on this finding, the second stage is designed to be programmable by specifying the connections between the inputs and outputs of the primitive units using an array of MUXes and DEMUXes. The other three stages have a fixed datapath with configurable parameters to reduce the hardware cost. The programming interface to configure the decompression module is further elaborated in Section IV-D (Configuring Decompression Module) with a running example.

Intersection Module. The intersection module consists of three intersection units, each of which is a simple 2-way merger augmented with a comparator to select docIDs present in both

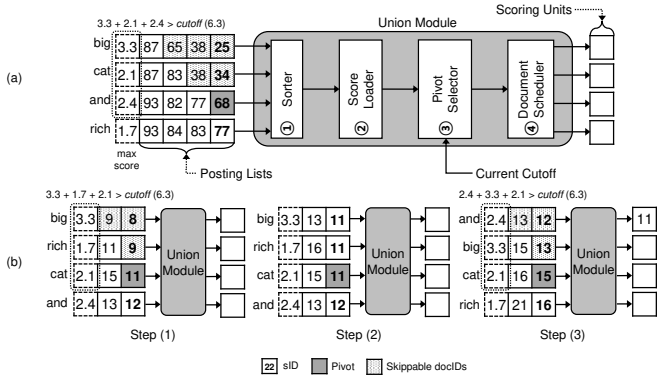

Fig. 7. Structure and operations of union module. sID (smallest unevaluated DocID) of each posting list is shown in bold, the *pivot* is shaded dark and skippable docIDs are dotted.

posting lists. As the intersection unit can only process two terms (posting lists) at once, it takes multiple passes to process 3- or 4-term intersections by looping over fetching, decompression, and intersection stages as discussed in Section IV-B. This iterative process is fully pipelined, so there is neither the performance penalty nor the need to spill the intermediate results to SCM memory and refill them. This fine-grained pipelining for 3- or 4-term queries facilitates the operation of the block fetch unit as it can perform more aggressive filtering of the posting list blocks if the list of decompressed docIDs is available. Since this information is not available during the first iteration, BOSS selects the smaller of the two input posting lists to increase the efficacy of the SvS membership testing.

Union Module. The union module implements the WAND algorithm [21] in hardware. WAND effectively skips scoring computation of those documents whose upper-bound query-score is below the cutoff score, making it impossible for them to be included in the final top-k outcome. It estimates the upper-bound of query-score for the document with the smallest docID yet to be evaluated. We denote this docID with sID. The aforementioned upper-bound for each sID can be calculated by summing up the maximum term-score for the entire posting list for all terms whose sID is smaller than the term being evaluated. The intuition here is that if $sID(X)$ of term X is greater than that of term Y ($sID(Y)$), there is a possibility of term Y also appearing in document $sID(X)$, but not vice versa. Next, WAND selects a $pivot$, which is the smallest sID whose upper-bound query-score is above the current *cutoff*. Then, all documents whose docID is smaller than the pivot are guaranteed to have a query-score smaller than the current cutoff and hence can be safely skipped without scoring.

Figure 7(a) shows a running example. In this case, the $pivot$ is sID of *and* (68). Posting lists are sorted in increasing order of sID for the sake of illustration. Here all docIDs yet to be evaluated can only appear in the terms before the pivot term (i.e., *big* and *cat*). Thus, their query-scores are upper-bounded by 5.4 (=3.3+2.1). Thus, all documents whose docIDs are below the pivot (shown in dotted gray boxes such as 25, 34, 38, and 65) can be safely skipped without calculating the query-score. A document is scored only if it is the pivot and actually present in other terms whose sID is smaller than the pivot (i.e., *big* and *cat* in this example).

Figure 7(a) also shows the hardware structure of the union module. The head value of each posting list queue (number in bold) is equivalent to the sID of each term. Skippable docIDs represent documents in the posting list queues whose docID is smaller than the pivot. The union module operates as follows. ① The sorter defines the order of $sIDs$. ② A score loader fetches pre-calculated upper-bound of the query-score for each sID from a lookup table according to the output of the sorter. Pre-calculation is possible because this module uses the maximum term-score of the entire posting list, but not each block, where the unique combinations for the upper-bound query-score are limited to 16 (=2^4) for 4-way unions. Pre-calculation can be performed at the start of each query. ③ The pivot selector chooses the *pivot* by comparing the upper-bound query-score of each sID with the current *cutoff*. ④ The document scheduler either sends the *pivot* to an available scoring module or pops those documents from the queue whose $sIDs$ are smaller than the *pivot*. Figure 7(b) illustrates this process step-by-step. In Step (1), sID of *cat* (11) is selected as the pivot as the estimated upper-bound (3.3 + 1.7 + 2.1 = 7.1) is greater than the current cutoff, which is assumed to be 6.3. Thus, the documents whose docIDs are smaller than 11 (i.e., documents 8 and 9) are discarded from the posting list queue. In Step (2), 11 is still the pivot, and it appears in all terms above it. Thus, this document is forwarded to the scoring unit as shown in Step (3).

Scoring Module. BOSS adopts the widely used Okapi BM25 metric for scoring. To reduce the runtime computation overhead, BOSS pre-computes an invariant portion of the scoring function (shown in Section II-B) at an indexing time. Specifically, for each document, BOSS calculates all sub-expressions of BM25 except the term frequency (tf) and stores them as metadata. Using this metadata, three arithmetic operations (a division, a multiplication, and an addition) are sufficient at runtime to obtain the term score. Such pre-computation will increase the per document metadata by 4B. The scoring module utilizes a single fixed-point divider, a single fixed-point multiplier, and two fixed-point adders: one for computing a term score and the other for accumulating the term scores. The final score is sent to the top-k module.

Top-*k* Module. To return the final top-k documents, BOSS utilizes a priority queue with k entries, each of which contains two fields: docID, query-score. The priority queue is sorted in descending order of the query-score. When a new docID arrives with its query-score, the top-k module inserts it into the hardware priority queue. BOSS adopts a shift register-based implementation [58], where a document entry is a unit of shifting. When a new entry is inserted, it is broadcast to all the entries in the queue for each entry to make a local decision about whether to remain in the same position, shift left, or load the incoming entry. By default, k is set to 1000 in BOSS.

On-chip Buffers. The BOSS core uses on-chip buffers to hold intermediate data in the pipeline and collect the final top-k results. Here is a list of buffers being used by individual

modules: (i) the block fetch module uses 288B to hold the address and metadata of each posting list; (ii) the four instances of the decompression module uses 1024B to store the target compressed block; (iii) the intersection and union modules use 192B for intermediate docIDs received from the previous module; (iv) the four instances of the scoring module takes about 2KB to temporarily hold docIDs and tf to be scored; (v) the buffer used to store the top-k results is 8KB. In summary, a BOSS core uses about 11KB of SRAM for on-chip buffers.

D. System-level Issues

Offloading API. BOSS exposes two intrinsic functions for offloading: `init`() and `search`().

```
void init(file indexFile, file configFile)
```

`init`() sets up a communication pipe between the host and BOSS through a memory-mapped register. It loads the inverted index file (`indexFile`) from disk to SCM memory pool. It also parses the configuration file (`configFile`) and sends the encoded configuration to BOSS.

```
val search(string qExpression, val compType[16], size_t
     nTerm, addr listAddr[16], addr resultAddr, val
    resultSize)
```

`search`() is invoked to offload a query request to BOSS. The type of query for each term is specified as a string by the `qExpression` argument. A user can define a query using query terms, round brackets, logical operators (AND/OR). To distinguish query terms from other operators, a user needs to put query terms in quotation marks. For example, a user can specify a query as "A" AND ("B" OR "C"). The API parses the received `qExpression`, converts it to an appropriate sequence of set operations, and sends it to BOSS. `compType` specifies the compression scheme of each posting list. This parameter allows a user to select the best one among the pre-defined compression schemes for each posting list. The number of terms in the query is given by `nTerm`. The `listAddr` argument gives a list of the starting addresses for all posting lists. The `resultAddr` stores the address to return the top-k results. The `resultSize` represents the size of memory reserved for storing the results. The list size for `qType`, `compType`, and `listAddr` is 16 as BOSS can support queries containing up to 16 terms.

Configuring Decompression Module. The decompression module can be flexibly re-configured using a configuration file. For example, Figure 8 shows a configuration for a VariableByte (VB) compression scheme [26]. For brevity, some components such as the current index counter or exception index comparator are omitted. The configuration file is divided into four sections, corresponding to the four stages of the decompression module presented in Section IV-C. For all stages except for Stage 2, simple parameter setting suffices. For Stage 2, the configuration file specifies the connections between the primitive units in a similar style to the structural modeling of a hardware description language such as Verilog and Chisel. Figure 6 actually visualizes the datapath configuration of the decompression module using this configuration file.

```
1     // Stage 1
2     // 0~15 for header encoded variable bit-length
3     Extractor[0].use = 0
4     Extractor[1].use = 1
5     Extractor[2].use = 0
6     Extractor[1].headerLength = 0
7     // Stage 2
8     RegInit( Reg, 0, reset )
9     reset := SHR(Input, 0x7)
10    wire1 := AND(Input, 0x7F)
11    wire2 := SHL(Reg, 7)
12    wire3 := ADD(wire1, wire2)
13    Reg  := wire3
14    Output := wire3
15    Output.valid := wire2
16    // Stage 3
17    ExceptionValue = ExceptionIndex = 0
18    // Stage 4
19    UseDelta = 1
```

Fig. 8. Example configuration file for VariableByte [26]

Address Translation. An `init`() call sends physical-to-virtual address mapping information of the inverted index to Memory Address Interface (MAI) in BOSS. We assume that each memory node in the pool has four 512GB DIMMs with 2TB of physical address space. By using 2GB huge pages, which is a common practice for running workloads with large memory footprints [33], BOSS makes its local (duplicate) TLB cover the entire physical address space with 1K entries. This prevents a TLB miss from generating additional memory access and/or host intervention for page table walks and so on.

Multi-term Query with More Than 4 Terms. One BOSS core can natively support queries having up to 4 terms. When the number of terms is greater than 4, the query terms are divided and sent to multiple BOSS cores. The mergers in the intersection/union module of these BOSS cores can be flexibly connected to another merger to support multi-way set operations. Thus, four BOSS cores can process queries with up to 16 terms in hardware. This covers over 99% of use cases in common full-text search engine queries like TREC queries [43], [69], but BOSS offers ways to handle queries with more than 16 terms. The host first divides the query into several subqueries and allocates memory space to store the intermediate results of the subqueries. BOSS then processes each subquery without pruning or top-k selection, and stores all intermediate results in the host memory. Finally, the host processes gathered data to retrieve the final output.

V. Evaluation

A. Methodology

Evaluation Model. We evaluate BOSS by comparing it against Apache Lucene [1] and IIU [34]. Apache Lucene is a production-grade search engine library driving many popular web services such as Twitter, Instagram, Netflix, and Ebay [2], [23]. IIU is a state-of-the-art inverted index search accelerator. Table I summarizes the configuration we use for the three schemes. The number of cores used for these schemes is fixed to 8. To estimate the performance of BOSS and IIU, we implement

TABLE I
HARDWARE METHODOLOGY

Host Proessor	
Core	Intel Xeon Scalable Processor 8280M @ 2.70GHz
L1 $	32KB I-cache, 32KB D-cache
L2 $	28MB (Private)
L3 $	38.5MB (Shared, Unified)
Hosts Memory System	
Organization	DDR4 2666 ECC REG, 6 channels, 384GB Intel Apache Pass Memory, 6 channels, 1.5TB
Bandwidth	140.76 GB/s (23.46 GB/s per channel) 39.6 GB/s (6.6 GB/s per channel)
BOSS Configuration	
BOSS	8 BOSS Cores @ 1.0GHz
BOSS Core	1 Block fetch module, 4 Decompression modules, 1 Intersection module, 1 Union module, 4 Scoring modules, 1 Top-k module
BOSS Memory System	
Organization	SCM, 4 channels
Bandwidth	Read bandwidth: 25.6GB/s (sequential), 6.6GB/s (random) [70] Write bandwidth: 2.3GB/s [70]

TABLE II
QUERY TYPES

Type	Number of Terms	Operation
Q1	1	A
Q2	2	A AND B
Q3	2	A OR B
Q4	4	A AND B AND C AND D
Q5	4	A OR B OR C OR D
Q6	4	A AND (B OR C OR D)

a cycle-level simulator integrated with DRAMSim2 [56]. We assume Intel Optane DCPMM as a baseline memory module for all three schemes to carefully validate its behaviors reported in an experimental study [36], [70] by adjusting the DRAMSim2 timing parameters. For IIU, we ignore the top-k selection time. For area and power estimation, we implement BOSS with Chisel3 [17], compile it to Verilog, and synthesize the Verilog code using Synopsys Design Compiler with a TSMC 40nm standard cell library.

Workloads. We conduct experiments on two web datasets: CC-News [4] and ClueWeb12 [3]. CC-News is a publicly available dataset, which contains news articles crawled from news sites through CommonCrawl. ClueWeb12 [3] is crawled by Hetrix web crawler and made public by CMU for research purposes. For each posting list (corresponding to each term) in the two web datasets, we find the best compression scheme among the five (BP [40], VB [26], OptPFD [68], S16 [73], S8b [14]) in advance and use the best for BOSS. As for query, we randomly select 100 1-term, 2-term, and 4-term queries (total 300 queries) from TREC 2006 and 2005 Terabyte Track dataset [11]. As the TREC dataset does not specify query type, we randomly assign a type for each query, as shown in Table II.

B. Performance Results

Query Throughput Analysis Using Multiple Cores. Figure 9 and Figure 10 show query throughput of BOSS and IIU with varying numbers of computation cores on the two datasets, normalized on 8-thread Lucene running on a CPU with 8 cores. In general, the query throughput of BOSS is superior over both Luecne and IIU. BOSS with 8 cores achieves 7.54× and 8.7× average query throughput improvement over the Lucene baseline for ClueWeb12 and CC-News datasets, respectively.

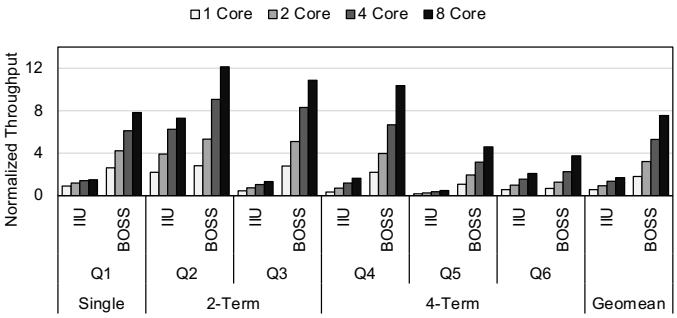

Fig. 9. Multi-core throughput analysis (ClueWeb12) (normalized to Lucene with 8 cores on SCM)

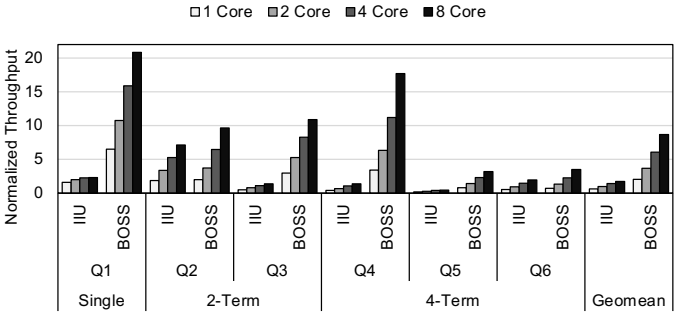

Fig. 10. Multi-core throughput analysis (CC-News) (normalized to Lucene with 8 cores on SCM)

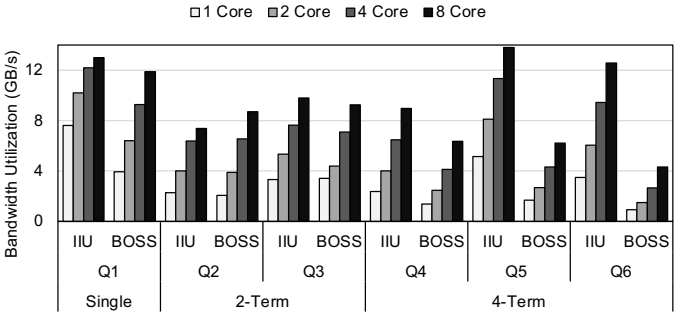

Fig. 11. Bandwidth utilization (ClueWeb12)

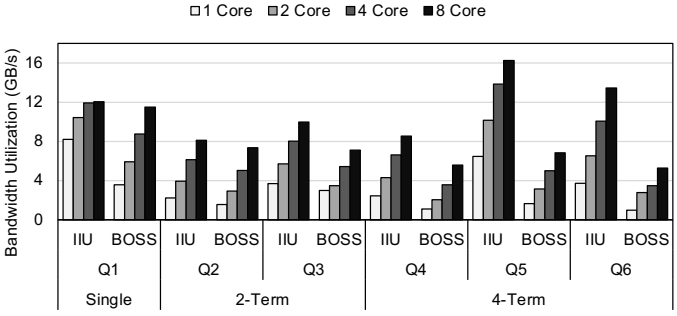

Fig. 12. Bandwidth utilization (CC-News)

In contrast, the average query throughput of IIU with 8 cores is 1.69× and 1.75× over Lucene, respectively. IIU shows a decent performance on Q2, unlike in other queries. This is because IIU uses a membership testing strategy, which can skip unnecessary blocks, similar to the overlap check between block and document used in BOSS for an intersection query. However, contrary to BOSS, which uses a sequential search, IIU uses binary search, which generates random access. As the latency of random access is longer than that of sequential

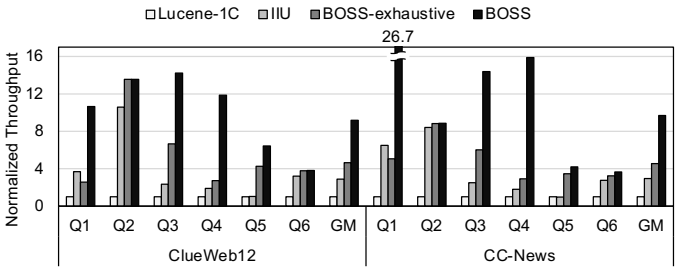

Fig. 13. Singe core throughput analysis (normalized to Lucene with 1 core on SCM)

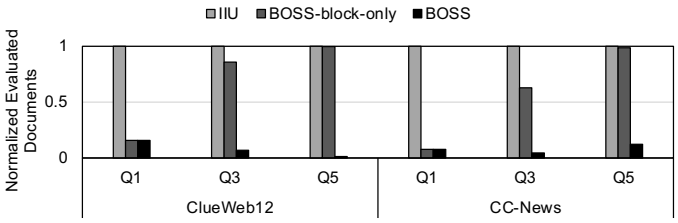

Fig. 14. Normalized evaluated documents

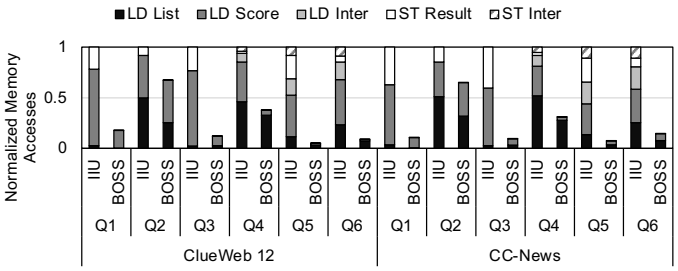

Fig. 15. Normalized memory access count

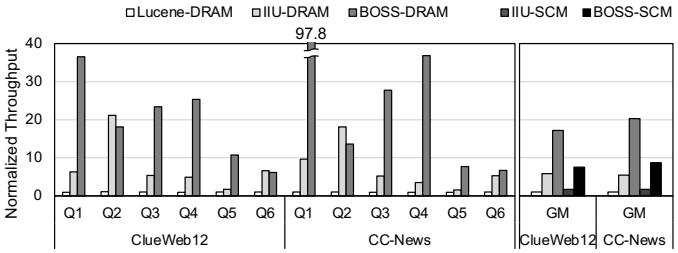

Fig. 16. Performance comparison between Lucene, IIU, and BOSS on DRAM and SCM (normalized to Lucene with 8 cores on SCM)

access for SCM [36], IIU performs worse than BOSS.

Figure 11 and Figure 12 show the bandwidth utilization of BOSS and IIU. For all queries, except for Q2 of ClueWeb12, BOSS has substantially less bandwidth consumption than IIU while maintaining a 4.7× higher query throughput than IIU. Even for Q2 of ClueWeb12, BOSS has a higher bandwidth efficiency than IIU as BOSS maintains a 1.7× higher throughput. The superior bandwidth efficiency of BOSS leads to more scalable performance than IIU as we increase the number of cores. Due to the difference in bandwidth efficiency, IIU hits the maximum performance with fewer cores than BOSS; as the aggregate bandwidth of SCM devices scales in the future, BOSS can utilize additional cores much more effectively than IIU to yield much higher query throughput.

Detailed Performance Analysis Using Single Core. There are two sources of bandwidth efficiency in BOSS: elimination of intermediate data movement and early termination for reducing computational/memory access wastes. Figure 13 shows the normalized single core throughput of Lucene, IIU, and BOSS. We also evaluate BOSS only with multi-term query support and top-k module, denoted as BOSS-exhaustive, to visualize throughput improvement due to ET algorithms integrated with the block fetch module and the union module. For a fair comparison, we place the same number of decompression and scoring modules for both BOSS and IIU. Throughput improvement over BOSS-exhaustive decreases as the number of terms increases for union queries (i.e., Q1, Q3, and Q5). This is because as the number of terms increases, it is less likely for a block to share a document with all overlapping blocks. Thus, the upper-bound of the query-score calculated by the score estimation unit will become looser as the number of terms increases, hence degrading its effectiveness. In contrast, the throughput of the intersection query improves as the number of terms increases. This is due to the pipelined intersection with the overlap check unit. As the intersection iterates over, fewer and fewer docIDs are compared against the next posting list. As a result, the number of blocks overlapped with this smaller posting list is reduced so that BOSS can skip many unnecessary blocks. Also, since several intersections are performed in the hardware pipeline simultaneously, the following intersections' cycles can be hidden. BOSS-exhaustive exhibits a throughput increase over IIU in all queries except for Q1. This is due to the lack of intra-query parallelism in BOSS. In contrast, IIU can use all decompression and scoring units regardless of the number of terms, whereas BOSS only uses the same number of decompression and scoring units as the number of terms.

Figure 14 shows the normalized number of evaluated (scored) documents of BOSS compared to IIU for single-term and union queries. BOSS-block-only shows the effect of skipped documents at the block fetch module and BOSS at both the block fetch module and the union module. As shown in Figure 14, because each module's skip efficiency is query-dependent, BOSS needs both modules to skip many documents. In the figure, as the number of terms increases, the number of skipped documents is decreased in the block fetch module. This is because the number of false positives increases in overlapped block selection, which prevents the block fetch module from effectively skipping blocks. However, the union module can reduce the scoring of unnecessary docIDs using WAND.

A reduced number of evaluated documents leads to a decrease in the number of memory accesses. Figure 15 shows the normalized number of memory accesses of BOSS and IIU. A majority of memory accesses generated by the intermediate data (LD Inter, ST Inter) and storing the final result (ST Result) are filtered via the multi-term query optimization and the top-k module, respectively. A reduced volume of memory accesses for ST Result shows that BOSS decreases the shared interconnect bandwidth consumption effectively. Thus, BOSS does not hinder scaling-out of the memory pool. Also, BOSS significantly reduces memory loads of posting lists (LD List) and values for scoring (LD Score) through the skip mechanism.

Comparison with DRAM-based Systems. Figure 16 depicts the performance comparison of Lucene, IIU, and BOSS with 8

TABLE III
AREA AND POWER OF BOSS

BOSS			
Component	# of Component	Area	Power
BOSS Core	8	8.024 mm^2	3200 mW
Command Queue	1	0.078 mm^2	0.078 mW
Query Scheduler	1	0.001 mm^2	1.96 mW
MAI (with TLB)	1	0.127 mm^2	1.20 mW
Total		8.27 mm^2	3.2 W
BOSS Core			
Block Fetch Module	1	0.108 mm^2	10.5 mW
Decompression Module	4	0.093 mm^2	43.0 mW
Intersection Module	1	0.003 mm^2	0.49 mW
Union Module	1	0.011 mm^2	5.55 mW
Scoring Module	4	0.464 mm^2	200.0 mW
Top-*k* Module	1	0.324 mm^2	147.1 mW
Total		1.003 mm^2	406.6 mW

cores on DRAM. For DRAM evaluation, we use DDR4-2666 with 4 channels (85.2GB/s), and all throughput numbers are normalized to Lucene running on SCM with 8 cores. The throughput of Lucene on DRAM is almost similar to that of Lucene on SCM; the former only outperforms the latter by up to 15%. In general, Lucene is less affected by choice of memory devices as its performance is mostly bottlenecked by computation. Both IIU and BOSS achieve substantial speedup over Lucene on DRAM-based systems (i.e., their bars are much higher than Lucene-DRAM's). Note that BOSS substantially outperforms IIU in almost all cases, except for Q2 and Q6. For those cases, IIU benefits a little more from DRAM than BOSS since IIU generates a lot of random accesses for those queries, and random accesses are substantially faster in DRAM than SCM. Finally, the figure also shows that IIU and BOSS are faster on DRAM by 3.29× and 2.31× compared to their performance on SCM. However, DRAM may not be a practical choice as it requires about 4× more memory nodes to secure the same capacity.

C. Area, Power, and Energy Analysis

Area. Table III is an area breakdown for BOSS. A single BOSS core occupies $1.00mm^2$, and the total area occupies $8.02mm^2$ with 8 BOSS cores. The scoring module's area is $0.46mm^2$, which is the largest module in a BOSS core due to fixed-point dividers required for pipelining scoring operations. The top-*k* module's area is $0.32mm^2$ due to shift registers required to maintain top-*k* results on the fly. The area of the remaining system components is $0.21mm^2$. Total area of BOSS is $8.27mm^2$. The area of the remaining system components is as small as $0.21mm^2$. Total area of BOSS is $8.27mm^2$.

Power. Table III also shows an average power breakdown of BOSS. BOSS consumes 23.3× less power compared to the host CPU used for our evaluation, whose average package power was 74.8W[1]. This proves the superior efficiency of BOSS over the general purpose CPU at running inverted index search. BOSS achieves both performance improvement and power savings.

Energy. Figure 17 shows the normalized energy consumption of both Lucene and BOSS with multiple computation units.

[1]Power consumption of the CPU was measured using Intel SoC Watch [10].

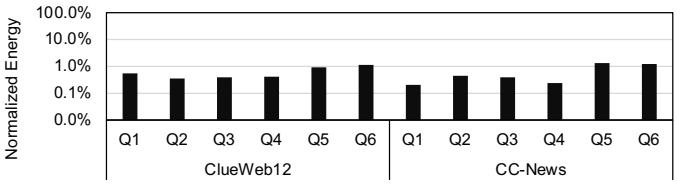

Fig. 17. Energy consumption of BOSS with 8 cores (normalized to Lucene with 8 cores on SCM). Y-axis is in log scale.

As shown in the figure, BOSS consumes 189× less energy compared to Lucene. The combination of performance improvement and power savings yields huge energy savings. This result shows that BOSS is an energy efficient alternative to general purpose CPUs for inverted index search.

VI. RELATED WORK

Software Optimization Techniques for Improving Search Query Performance. There have been various software works for improving search query performance on CPUs [25], [41], [53], [57], [64], [72] or GPUs [15], [29], [43], [63], [66], [67], [76]. Most of those CPU optimization techniques target the intersection of decompression stages. They improve compression throughput or query performance by leveraging SIMD instructions or cache structures. These techniques have limited coverage of operations for inverted index search, while BOSS performs all key operations efficiently in hardware. Also, existing works on GPUs aim to accelerate key operations using GPU's high parallelism. These approaches, however, have a substantial overhead caused by the limited memory capacity of GPUs or frequent CPU-GPU communications. Instead, BOSS takes the near-data-processing to eliminate such unnecessary data movements between the host CPU and the accelerator.

Inverted Index Compression. Many compression schemes have been proposed to mitigate its storage cost [14], [26], [68], [77]. BP [40] calculates the minimum number of bits needed to represent the largest value in a block and uses it to represent all the values of the block. VB [26] utilizes multiples of 7 bits to represent the actual "payload" and one bit (MSB) for every 7 bits to represent an integer. PFD [77] uses the smallest possible *b* bits to represent a majority (e.g., 90%) of a block of 128 deltas (d-gaps) and stores the rest of the values at the end of the compressed array. S16 [73] and S8b [14] both seek to pack as many integers as possible within a given size of the array (32 bits for S16 and 64 bits for S8b) using various combinations of values. BOSS can support all these compression schemes through the reconfigurable decompression module.

Early Termination Techniques. Max-score [61] is one of the first to skip the evaluation of documents based on their upper-bound scores. WAND [21] is also based on max-score but skips upper-bound calculation by pivoting. Interval-based pruning [24] and BlockMaxWAND [30] apply a more fine-grained approach. BOSS adopts an interval-based pruning technique as it is better suited for building a hardware pipeline by decoupling per-block and per-docID early termination. More specifically, BOSS uses longer intervals to minimize the delay between adjacent block load requests.

VII. Conclusion

This paper presents BOSS, the first NDP accelerator architecture for inverted index search targeting the emerging SCM-based memory pool with bandwidth constraints. We apply the following strategies synergistically to overcome the bandwidth challenges imposed by SCM-based pooled memory: (i) skip mechanism and multi-term query optimization to save (internal) SCM device bandwidth consumption; (ii) top-k selection hardware module and near-data processing paradigm to save (external) host-accelerator bandwidth over the shared byte-addressable interconnect. According to our evaluation using both a cycle-level simulator and synthesizable RTL written in Chisel, BOSS achieves a geomean speedup of 8.1× on various complex query types, while reducing the average energy consumption by 189×, compared to Apache Lucene, a production-grade search engine library, running on 8 CPU cores.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by Korea government (MSIT)(NRF-2020R1A2C3010663) and an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT)(No. 2021-0-00853, Developing Software Platform for Programming of PIM). Jae W. Lee is the corresponding author.

References

[1] "Apache Lucene," https://lucene.apache.org/.
[2] "Apache solr wiki," https://cwiki.apache.org/confluence/display/solr/PublicServers#PublicServers-PublicWebsitesusingSolr.
[3] "The ClueWeb12 dataset," https://lemurproject.org/clueweb12/.
[4] "Common crawl - CCNEWS dataset," http://commoncrawl.org/2016/10/news-dataset-available/.
[5] "Compute express link," https://www.computeexpresslink.org/about-cxl.
[6] "The fastpfor C++ library: Fast integer compression," https://github.com/lemire/FastPFor.
[7] "Gen-z consortium," https://genzconsortium.org/.
[8] "Intel compute express link (cxl)," https://www.computeexpresslink.org/.
[9] "Intel optane persistent memory," https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.
[10] "Intel SoC Watch," https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-socwatch-system-bring-up-toolkit/top.html.
[11] "Text retrieval conference (trec)," https://trec.nist.gov/.
[12] "Zipf's law," https://en.wikipedia.org/wiki/Zipf\%27s_law.
[13] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W. Hwu, and N. S. Kim, "Application-transparent near-memory processing architecture with memory channel network," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.
[14] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Software: Practice and Experience*, 2010.
[15] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *Proc. VLDB Endow.*, 2011.
[16] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018.
[17] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012.
[18] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.
[19] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, 2003.
[20] M. Becker, M. Chabbi, S. Warnat-Herresthal, U. Worlikar, S. Agrawal, J. Bhat, J. Schulte-Schrepping, K. Bassler, P. Guenther, H. Schultze, T. Ulas, S. Singhal, and J. L. Schultze, "Accelerated genomics data processing using memory-driven computing," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2019.
[21] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, 2003.
[22] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, 2008.
[23] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, 2012.
[24] K. Chakrabarti, S. Chaudhuri, and V. Ganti, "Interval-based pruning for top-k processing over compressed lists," in *2011 IEEE 27th International Conference on Data Engineering*, 2011.
[25] J. S. Culpepper and A. Moffat, "Efficient set intersection for inverted indexing," *ACM Trans. Inf. Syst.*, 2011.
[26] D. Cutting and J. Pedersen, "Optimizations for dynamic inverted index maintenance," in *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1989.
[27] Z. Dai and J. Callan, "Context-aware sentence/passage term importance estimation for first stage retrieval," *CoRR*, 2019.
[28] Z. Dai, C. Xiong, J. Callan, and Z. Liu, "Convolutional neural networks for soft-matching n-grams in ad-hoc search," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018.
[29] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high performance ir query processing," in *Proceedings of the 18th International Conference on World Wide Web*, 2009.
[30] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, 2011.
[31] J. Guo, Y. Fan, Q. Ai, and W. B. Croft, "A deep relevance matching model for ad-hoc retrieval," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016.
[32] S. Hamdioui, H. Aziza, and G. C. Sirakoulis, "Memristor based memories: Technology, design and test," in *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era*, 2014.
[33] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.
[34] J. Heo, J. Won, Y. Lee, S. Bharuka, J. Jang, T. J. Ham, and J. W. Lee, "IIU: Specialized architecture for inverted index search," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.
[35] S. Hudgens and B. Johnson, "Overview of phase-change chalcogenide nonvolatile memory technology," *MRS bulletin*, 2004.
[36] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, 2019.
[37] K. Keeton, "The machine: An architecture for memory-centric computing." *Workshop on Runtime and Operating Systems for Supercomputers*, 2015.
[38] O. Khattab and M. Zaharia, "Colbert: Efficient and effective passage search via contextualized late interaction over bert," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.
[39] H. Kim, M. P. Sah, C. Yang, and L. O. Chua, "Memristor-based multilevel memory," in *2010 12th International Workshop on Cellular Nanoscale Networks and their Applications*, 2010.
[40] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, 2015.

[41] D. Lemire, L. Boytsov, and N. Kurz, "Simd compression and the intersection of sorted integers," *Softw. Pract. Exper.*, 2016.

[42] Y. Li, X. Tang, W. Cai, J. Tong, X. Liu, and G. Wang, "Resource-efficient index shard replication in large scale search engines," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[43] Y. Liu, J. Wang, and S. Swanson, "Griffin: Uniting cpu and gpu in information retrieval systems for intra-query parallelism," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.

[44] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference*, 2017.

[45] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *2012 39th Annual International Symposium on Computer Architecture*, 2012.

[46] B. Mitra and N. Craswell, "An updated duet model for passage re-ranking," *CoRR*, 2019.

[47] R. Nogueira and K. Cho, "Passage re-ranking with BERT," *CoRR*, 2019.

[48] R. Nogueira, W. Yang, K. Cho, and J. Lin, "Multi-stage document ranking with BERT," *CoRR*, 2019.

[49] R. Nogueira, W. Yang, J. Lin, and K. Cho, "Document expansion by query prediction," *CoRR*, 2019.

[50] M. Ogleari, Y. Yu, C. Qian, E. Miller, and J. Zhao, "String figure: A scalable and elastic memory network architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture*, 2019.

[51] J. Pedersen, "Query understanding at bing," In Industry Track Keynote at the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.

[52] A. Pirovano, A. L. Lacaita, A. Benvenuti, F. Pellizzer, S. Hudgens, and R. Bez, "Scaling analysis of phase-change memory technology," in *IEEE International Electron Devices Meeting 2003*, 2003.

[53] I. Rae, A. Halverson, and J. F. Naughton, "In-rdbms inverted indexes revisited," in *2014 IEEE 30th International Conference on Data Engineering*, 2014.

[54] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. . Chen, R. M. Shelby, M. Salinga, D. Krebs, S. . Chen, H. . Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, 2008.

[55] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*, 2009.

[56] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, 2011.

[57] S. Shah and A. Shaikh, "Hash based optimization for faster access to inverted index," in *2016 International Conference on Inventive Computation Technologies*, 2016.

[58] Sung-Whan Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, 2000.

[59] Suock Chung, K. . Rho, S. . Kim, H. . Suh, D. . Kim, H. . Kim, S. . Lee, J. . Park, H. . Hwang, S. . Hwang, J. . Lee, Y. . An, J. . Yi, Y. . Seo, D. . Jung, M. . Lee, S. . Cho, J. . Kim, G. . Park, Gyuan Jin, A. Driskill-Smith, V. Nikitin, A. Ong, X. Tang, Yongki Kim, J. . Rho, S. . Park, S. . Chung, J. . Jeong, and S. . Hong, "Fully integrated 54nm stt-ram with the smallest bit cell dimension for high density memory application," in *2010 International Electron Devices Meeting*, 2010.

[60] N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient query processing for scalable web search," *Foundations and Trends in Information Retrieval*, 2018.

[61] H. Turtle and J. Flood, "Query evaluation: strategies and optimizations," *Information Processing & Management*, 1995.

[62] D. Ustiugov, A. Daglis, J. Picorel, M. Sutherland, E. Bugnion, B. Falsafi, and D. Pnevmatikatos, "Design guidelines for high-performance scm hierarchies," in *Proceedings of the International Symposium on Memory Systems*, 2018.

[63] D. Wang, W. Yu, R. J. Stones, J. Ren, G. Wang, X. Liu, and M. Ren, "Efficient gpu-based query processing with pruned list caching in search engines," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems*, 2017.

[64] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson, "Milc: Inverted list compression in memory," *Proc. VLDB Endow.*, 2017.

[65] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[66] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang, "A batched gpu algorithm for set intersection," in *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, 2009.

[67] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and Jing Liu, "Efficient lists intersection by cpu-gpu cooperative computing," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2010.

[68] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th International Conference on World Wide Web*, 2009.

[69] J. Yan, Z. Zhao, N. Xu, X. Jin, L. Zhang, and F. Hsu, "Efficient query processing for web search engine with fpgas," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[70] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies*, 2020.

[71] J. J. Yang, M. D. Pickett, X. Li, D. A. A. Ohlberg, D. J. Stewart, and R. S. Williams, "Memristive switching mechanism for metal/oxide/metal nanodevices." *Nature nanotechnology*, 2008.

[72] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proc. VLDB Endow.*, 2018.

[73] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *Proceedings of the 17th international conference on World Wide Web*, 2008.

[74] Y. Zhang, L. Zhang, W. Wen, G. Sun, and Y. Chen, "Multi-level cell stt-ram: Is it realistic or just a dream?" in *2012 IEEE/ACM International Conference on Computer-Aided Design*, 2012.

[75] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Trans. Archit. Code Optim.*, 2015.

[76] J. Zhou, Q. Guo, H. V. Jagadish, L. Krcal, S. Liu, W. Luan, A. K. H. Tung, Y. Yang, and Y. Zheng, "A generic inverted index framework for similaritygpu5 on the gpu," in *2018 IEEE 34th International Conference on Data Engineering*, 2018.

[77] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *Proceedings of the 22nd International Conference on Data Engineering*, 2006.