Evaluating List Intersection on SSDs for Parallel I/O Skipping

Jianguo Wang
Purdue University
csjgwang@purdue.edu

Chunbin Lin
Amazon
lichunbi@amazon.com

Yannis Papakonstantinou Steven Swanson
University of California San Diego
{*yannis, swanson*}*@cs.ucsd.edu*

***Abstract*—List intersection is at the core of information retrieval systems. Existing disk-based intersection algorithms were optimized for hard disk drives (HDDs) since HDDs have dominated the storage market for decades. In particular, those HDD-centric algorithms read every relevant list *entirely* to memory to minimize expensive random reads by performing sequential reads, although many entries in the list may be useless. Such a tradeoff makes perfect sense on HDDs, because random reads are one to two orders of magnitude slower than sequential reads. However, fast solid state drives (SSDs) have changed this landscape by improving random I/O performance dramatically. More importantly, they are manufactured with multiple flash channels to support parallel I/Os. As a result, the performance gap between random and sequential reads becomes very small on SSDs. This means that HDD-optimized intersection algorithms might not be suitable on SSDs because the total amount of data accessed is unnecessarily high.**

To understand the impact of SSDs to list intersection, in this work, we tune existing in-memory intersection algorithms to be SSD-aware with the idea of parallel I/O skipping, and experimentally evaluate them on synthetic and real datasets. The results provide insights on how to design efficient SSD-optimized intersection algorithms.

I. Introduction

List intersection is a fundamental operation in information retrieval systems. For instance, finding documents that contain all the query terms requires the intersection of several inverted lists [16]. In this work, we focus on disk-resident systems where the entire inverted index cannot fit in main memory and therefore, at least partially, needs to be stored on secondary storage. Since HDDs (hard disk drives) have been dominating the storage market for decades, existing disk-based intersection algorithms were mainly optimized for HDDs [1], [5], [6], [8]. Those algorithms aim to minimize the number of expensive random reads. They generally carry out list intersection using a two-phase process. (1) Read each list from disk to memory in its entirety, so that each list requires only one random read. This stage usually requires a single thread to access disk because an HDD has only one magnetic disk head to serve one I/O request simultaneously, thus, using multiple threads to read a list does not improve performance [9]. We refer to it as "list-at-a-time single-threaded" I/O access pattern. (2) Perform intersection in memory, which can use multiple threads to exploit multi-core CPUs. The two-phase paradigm is a perfect fit for HDDs, because random reads on HDDs are one to two orders of magnitude more expensive than sequential reads [23], [25], due to the extremely slow disk seeks.

Today, solid state drives (SSDs) have become an alternative secondary storage solution to HDDs in the storage market. Compared to HDDs, SSDs have many advantages such as low I/O latency, high I/O throughput, and low energy consumption.As a result, SSDs are deployed in many large-scale infrastructures, e.g., Amazon Redshift.

The landscape shifting from HDDs to SSDs has raised an interesting research question: *What is the impact of SSDs to list intersection algorithms*? There are two notable properties of SSDs when compared to HDDs that can change the algorithmic design decisions. (1) The first one is that random reads are fast enough to be comparable with sequential reads. On modern SSDs, random reads are only $1\times \sim 2\times$ slower than sequential reads [23]. (2) The second interesting property is that an SSD supports parallel I/O because it is manufactured to incorporate multiple flash channels [18]. Thus, it can serve multiple I/O requests simultaneously. In contrast, an HDD has only one disk head such that it can only serve a single I/O request at the same time.

With the two new properties mentioned above, we speculate existing HDD-centric list intersection algorithms might not work well on SSDs and we shall rethink SSD-aware intersection algorithms by explicitly leveraging the unique characteristics. A decent SSD-optimized intersection algorithm should follow "page-at-a-time multi-threaded" I/O access pattern (instead of the HDD's "list-at-a-time single-threaded" access pattern):

(1) **Page-at-a-time**: access a list page by page for reducing unnecessary pages that do not contain intersection results to leverage the SSD's fast random reads. In this way, the optimization goal should be minimizing the number of pages accessed by skipping as many as possible of the pages that do not contain intersection results. For example, all the white-color pages in Figure 1 can be skipped.[1] Note that, skipping does not work well on HDDs because it introduces many expensive random reads that do not pay off (as we show later in experiments).

(2) **Multi-threaded**: issue multiple page-sized I/O requests at the same time via multiple threads. We observe that using a single thread, the page-at-a-time I/O access pattern underutilizes the SSD's bandwidth seriously. Meaning that even if the total amount of data accessed is reduced, the

[1] Of course, if every page contains a result, none of the pages can be skipped. However, in practice, many pages do not contain any intersection result.

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00161

L_1 | **50** 150 | **200** 960

L_2 | 10 20 | 30 40 | **50** **200** | 960 980

L_3 | 10 25 | **50** 60 | 80 100 | 120 150 | 160 180 | **200** 300 | 400 500 | 800 980

Fig. 1. **An example of list intersection on an SSD (assuming each page contains two elements), only the gray-color pages contain intersection results.**

actual execution time may not be reduced as expected. Fortunately, modern SSDs support concurrent I/O requests with multiple flash channels. Thus, we shall use multiple threads in order to saturate the SSD's bandwidth if the underlying lists are accessed page-by-page. Note that it does not make sense to issue multiple threads to read a list on an HDD because it has only one disk head.

Following "page-at-a-time multi-threaded" I/O access pattern, then how to design efficient intersection algorithms for SSDs? There are many algorithmic design choices in terms of how to skip unnecessary pages efficiently and issue I/Os in a parallel way. In this work, we tune existing in-memory intersection algorithms to be SSD-conscious based on parallel I/O skipping (via skip pointers and bloom filters), and experimentally evaluate them on both real data and synthetic data.

Contribution. In this work, we carry out a series of experiments on four algorithms on synthetic and real datasets to understand the impact of SSDs to list intersection. The results provide insights on how to design efficient SSD-centric list intersection algorithms.

II. Related work

A. List Intersection on HDDs

Traditionally, the inverted index is stored on HDDs. To reduce the I/O cost for list intersection, some earlier work tried to avoid reading all the lists in their entirety [28]. A solution mentioned in [28] is to load the shortest list first, and intersect all the other lists in ascending order of their sizes. The algorithm terminates once the current intersection results are empty. In this way, some longer lists can be skipped. However, the algorithm ends up reading all the lists if the intersection results are not empty. Actually, for any list L, the algorithm either reads L entirely, or does not read L at all. It does not skip any blocks in a single list (the goal of our work). That is because skipping introduces many random reads which are very expensive on HDDs.

B. List intersection in memory

There is rich literature on the in-memory list intersection problem.We classify the existing algorithms into four categories: comparison-based (Section II-B1), hash-based (Section II-B2), partition-based (Section II-B3), and bitmap-based (Section II-B4). We describe the existing algorithms using k sorted lists L_1, L_2, $\cdots$, L_k ($|L_1| \leq |L_2| \leq \cdots \leq |L_k|$).

1) Comparison-based intersection algorithms

The first comparison-based intersection algorithm is the sort-merge algorithm [11], which would access the entire lists if extended to SSDs.

To skip unnecessary comparisons, the SL (short-long) algorithm [5] was proposed. It is the state-of-the-art algorithm in the comparison-based algorithm class. For each element $e \in L_1$, SL checks whether e appears in all the other lists. If yes, e is a result. The presence of e on L_i can be obtained by invoking Member(L_i, e), which returns true if L_i contains e. In SL, Member(L_i, e) can be implemented using binary search or skip lists.We extend SL to SSDs, so that it skips pages.

The SvS algorithm [7] is comparison-based algorithm, which is a variant of SL. SvS starts from the shortest list and intersects the other lists in ascending order of their sizes. In other words, L_1 and L_2 are intersected first, then the results of $L_1 \cap L_2$ are intersected with L_3. The process continues until L_k. SvS also relies on Member(L, e) to test whether e is in L. Thus, basically, SvS and SL are the same except that SvS needs to maintain a result buffer if the number of lists intersected exceeds two.

There is another algorithm called Zig-Zag (ZZ, a.k.a Adaptive) which uses Successor(L, e) for skipping [2], [6]. Successor(L, e) returns the smallest element in L that is greater than or equal to e. Every time an eliminator e is selected (initially e is the first element of L_1), then e is probed against the other lists in a round-robin fashion. For the current list L_i, ZZ checks whether e is the same as Successor(L_i, e). If yes, the occurrence counter of e is increased (e is a result if the counter reaches k). Otherwise, it updates e to Successor(L, e). ZZ terminates once e is invalid.

If L is compressed (e.g., using PforDelta [29]), neither Member(L, e) nor Successor(L, e) can be implemented efficiently. To allow skipping, a general technique is to build an auxiliary data structure (e.g., skip list) over L [17], [20]. The original list L is then split into segments. The auxiliary data structure maintains for each segment the minimum/maximum element (uncompressed). Thus, Member(L, e) and Successor(L, e) can be implemented by routing to the desired segment and decompressing it individually.

2) Hash-based intersection algorithms

The naive hash-based intersection algorithm [13] builds a hash table to implement Member(L, e). If extended to SSDs, we could build an external-memory hash table to support skipping. However, the hash table takes too much space. Moreover, it resembles SL [5] when extended to SSDs because both of them route a search element to a page that potentially contains the element. Bille et al. suggested another way of using hash [3]. Unfortunately, if extended to SSDs, it would not skip pages because all the elements have to be accessed at least once in the worst case.

3) Partition-based intersection algorithms

Baeza-Yates et al. proposed an algorithm for intersection based on the divide-and-conquer framework [1]. If extended to SSDs, it would load the entire lists to memory because every element has to be accessed at least once.

Ding and König proposed another partition-based algorithm [8], which splits a list into partitions based on universal

hash. Unfortunately, if extended to SSDs, it would also access the entire lists (see Theorem 3.3 of [8]).

4) Bitmap-based intersection algorithms

Up to this point, we have represented a list as a sorted array. The list can also be represented as a bitmap: set the i-th bit to 1 if and only if i is in the list. Then compute the intersection using bitwise operations [5]. If extended to SSDs, the *entire* bitmap of a list has to be loaded to memory.

C. SSD-Optimized Design in IR Systems

Several prior works have studied the impact of SSDs to IR systems. Huang and Xia discussed allocating the inverted index on SSDs and HDDs [10] to maximize query performance while minimizing operational cost. A similar issue was also discussed in [15]. The impact of SSDs on cache management was studied in [22], [25], [26]. They found that existing cache policies optimized for HDDs do not work well on SSDs. The issue of inverted index maintenance on SSDs was studied in [12], [14] by leveraging the fast random accesses of SSDs. However, all these works still used existing HDD-centric list intersection algorithms. This work, in contrast, focuses on experimenting SSD-aware intersection algorithms. Besides that, this work is also different from [27] in that we use off-the-shelf SSDs instead of Smart SSDs.

III. Compared algorithms

Although there is no prior SSD-oriented intersection algorithm, some in-memory skipping-based intersection algorithms can be naturally extended to SSDs to skip unnecessary pages, as discussed in Section II. In this section, we present the extended algorithms that will be experimented later on. Before that, we describe the problem and list structure first.

Problem statement. Consider k ($k \geq 2$) lists L_1, L_2, ..., L_k ($|L_1| \leq |L_2| \cdots \leq |L_k|$) stored on an SSD. Each list is stored in pages of a typical size, e.g., 4KB [23], [25]. The problem is to find the intersection of these lists, i.e., $\bigcap_{i=1}^{k} L_i$, while minimizing the total amount of pages accessed in order to reduce the actual execution time.

We first focus on the intersection where all the lists are stored on the SSD initialy in order to obtain clean results. Then, we evaluate the impact of cache where some lists are cached in main memory.

List structure. We follow a typical setting in IR systems to represent and compress a list [17], [28]. Each entry is a document ID d_i of 4 bytes before compression.[2] All document IDs in every list are sorted in ascending order. Depending on different IR systems, the lists can be compressed using a compression algorithm, e.g., PforDelta [29] and SIMDPforDelta [24]. We also evaluate the impact of different compression schemes to the performance. If a list is compressed, we follow prior work [17], [20] to build skip pointers such that intersection only needs to examine those promising pages.

[2]All the evaluated algorithms are applicable to entries that contain other auxiliary information, e.g., document frequencies and positions.

A. BL (Baseline)

We denote BL as the baseline algorithm that directly runs the existing HDD-centric algorithm on the SSD. This serves as a baseline when one uses SSDs to replace HDDs in their systems. In particular, BL reads each of the k lists in its entirety in a single thread and then runs multiple threads for in-memory intersection. BL uses SL for the in-memory intersection because it has high performance and widely used in practice [5].

The advantage of BL is that it is simple and there is no need to change existing algorithms when replacing HDDs with SSDs. But the disadvantage is that BL can be slow because it reads many unnecessary pages.

B. SL (Short-Long)

As mentioned in Section II-B1, an important operation used in SL is $\mathsf{Member}(L, e)$, which checks whether element e is in list L as illustrated in Section II-B1. In main memory, it is usually implemented using binary search or skip lists. On SSDs, we implement $\mathsf{Member}(L, e)$ by pre-computing skip pointers [17]: we store for each page the minimum (or maximum) uncompressed ID (called a *skip pointer*).

$\mathsf{Member}(L, e)$ can be implemented efficiently by using skip pointers. A page that potentially contains e can be identified efficiently. Then, SL reads the page to verify whether e is actually in L. In this way, many unnecessary pages can be skipped. Also, the skip pointers of a list are stored in memory.

Note that SL is equivalent to SvS that intersects two lists at a time. Thus, we do not consider SvS in this work.

C. ZZ (Zig-Zag)

The Zig-Zag (ZZ) intersection algorithm [2], [6] adopts another way of using skip pointers for intersection as described in Section II-B1. ZZ implements $\mathsf{Successor}(L, e)$ to find the successor of e in L instead of implementing $\mathsf{Member}(L, e)$. When extended to SSDs, we can implement $\mathsf{Successor}(L, e)$ using skip pointers and only read the promising page to skip unnecessary pages. The skip pointers are stored in memory.

D. BF (Bloom-Filter)

We observe that skip pointer based intersection algorithms, i.e., SL and ZZ still incur unnecessary page accesses. Because the skip pointers can only route element e to the page whose range covers e. However, e may still not exist in that page, in which case reading it is wasteful.

To solve the problem, a natural idea is to use bloom filters [4]. In particular, we build for each page an in-memory bloom filter to improve the SL algorithm.[3] We denote the algorithm as BF (bloom-filter). Whenever a target page $\mathcal{P}$ (whose range contains e) is identified by the skip pointers, instead of loading $\mathcal{P}$ immediately, BF performs a bloom test between e and the bloom filter of $\mathcal{P}$. If e fails the bloom test

[3]Note that ZZ cannot benefit from bloom filters because ZZ relies on $\mathsf{Successor}(L, e)$ instead of $\mathsf{Member}(L, e)$ while bloom filters can only be helpful for membership testing.

(i.e., the bloom test returns false), there is no need to read $\mathcal{P}$. Otherwise BF reads it.

Remark. Skip pointers and bloom filters used in SL, ZZ, and BF reduce data movement, but also increase random reads. The tradeoff makes sense on SSDs where random reads are comparable to sequential reads. But on HDDs, they cannot improve the performance due to expensive random reads.

E. Parallel intersection algorithms

All the algorithms experimented (BL, SL, ZZ, BF) in this work are evaluated in multiple threads. We discuss in detail how to make the algorithms in parallel. The key question is: how to partition the k lists for parallel intersection? We want to create many independent intersection tasks such that each thread works on one task individually. We choose the state-of-the-art partition strategy proposed in [21]. It works as follows. Assume there are N threads, then it partitions L_1 into N even sublists: L_1^1, L_1^2, $\cdots$, L_1^N. For each sublist L_1^i ($1 \le i \le N$), it performs binary search of $L_1^i[0]$ on L_j ($j \ge 2$) to partition L_j into N sublists. Thus, the N intersection tasks are: $(L_1^1 \cap L_2^1 \cap \cdots \cap L_k^1)$, $(L_1^2 \cap L_2^2 \cap \cdots \cap L_k^2)$, $\cdots$, $(L_1^N \cap L_2^N \cap \cdots \cap L_k^N)$. Every intersection task will be executed by the corresponding algorithm.

Remark. Parallel I/O is crucial to SSD-based list intersection, much more important than ever before. It is precisely and particularly suitable for SSDs *and* skipping-based intersection: (i) if lists are accessed entirely without skipping, it is not necessary to apply parallel intersection because sequentially loading a long list can already saturate the SSD's I/O bandwidth. Just because of skipping, parallelism becomes *the only way to fully utilize the SSD's I/O bandwidth*; (ii) if lists are stored on HDDs, parallelism cannot improve performance because HDDs only have one disk moving head that can serve at most one disk I/O simultaneously.

IV. RESULTS ON SYNTHETIC DATA

In this section, we present experimental results on synthetic datasets to understand the impact of the key parameters.

Experimental settings. We conduct experiments on a commodity machine (Intel i7 3.10 GHz CPU, 4 physical cores, 8 hyper-threaded cores, 16GB DRAM) with Windows 8 installed. Our experimental platform also includes an SSD (Samsung 850 Pro SSD 256GB) and an HDD (Seagate HDD 2TB, 7200rpm). All the algorithms are coded in C++. Moreover, the page size is 4KB by default (in both HDD and SSD).

Synthetic datasets. To study the effect of crucial parameters to the overall performance, we generate synthetic data by fixing all the other parameters when evaluating the effect of a particular parameter. Unless otherwise stated, we use the default settings shown in Table I.

TABLE I
DEFAULT PARAMETERS USED IN SYNTHETIC DATA

Parameters	Default
number of threads	32
number of lists	2
list size	$\lvert L_1\rvert = 10^4$, $\lvert L_2\rvert = 10^7$
list size ratio	1000
intersection ratio	1% of $\lvert L_1\rvert$
bloom filter size	4 bits (per element)
data distribution	uniform (from domain $[0, 2^{32}-1]$)

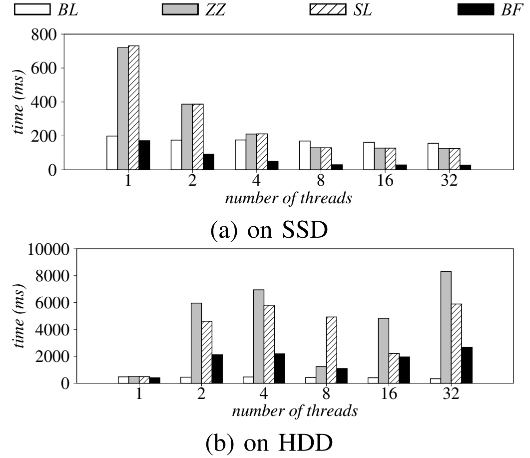

Fig. 2. **Effect of number of threads**

A. Effect of number of threads

We first examine the effect of parallelism, which is very important to the performance. We use the default settings (shown in Table I) except we vary the number of threads from 1 to 32. Figure 2 shows the execution time of the algorithms on both SSD and HDD.

(1) For the baseline algorithm BL, Figure 2 demonstrates that multiple threads do not help much in reducing the execution time on both SSD and HDD. That is because BL loads the whole list from disk to memory using a single thread and executes intersection in memory where the I/O cost is absolutely the performance bottleneck. Note that for list-at-a-time I/O access pattern, using a single thread is able to saturate the disk bandwidth for both SSD and HDD.

(2) For SL, the results are very interesting. Let's first look at the results of SL on the SSD. Figure 2a shows that when the number of threads is 1, SL is even slower than BL although SL reads less data than the baseline BL (6553 pages vs. 11177 pages). That is because BL can fully utilize the SSD's I/O bandwidth by reading a whole list at a time but SL cannot because it reads a page at a time in order to skip unnecessary pages. However, as the number of threads increases, the performance of SL becomes better while BL does not increase much. At the point when the number of threads is 8, SL runs 1.3X faster than BL. This confirms our conjecture that on SSDs, we should access a list by using "page-at-a-time multi-threaded" I/O access pattern instead of the conventional "list-at-a-time single-threaded" access pattern.

To make the results comprehensive, we also report the execution time on the HDD in Figure 2b. It shows a completely different picture with the SSD in the sense that SL always runs slower than the baseline BL no matter how many threads are

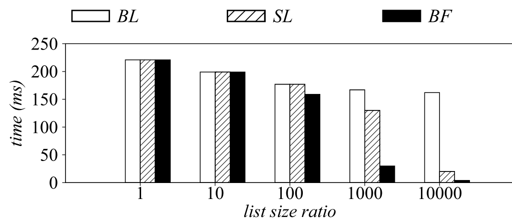

Fig. 3. **Effect of list size ratio**

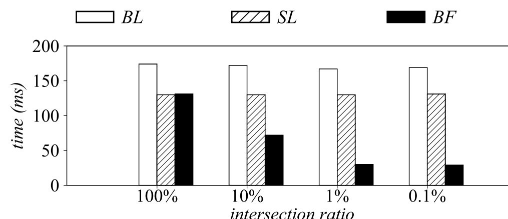

Fig. 4. **Effect of intersection ratio**

used. More importantly, increasing the number of threads (on the HDD) can even make the performance slower, e.g., SL runs slower when the number of threads increases from 1 to 2. That is because of expensive random seeks introduced by multiple threads. Note that when the number of threads is 1, SL has similar performance to BL because those random seeks are short under a single thread and short seeks tend to have high performance as explained in [19]. However, for multiple threads, random seeks tend to be long due to the context switch between different threads.This confirms that HDD-optimized intersection algorithms should read the whole list to memory instead of skipping pages.

This also explains that if we treat the SSD as a drop-off disk and apply an existing HDD-centric algorithm BL, it becomes sub-optimal. For example, the intersection time (BL) on the HDD is 474ms; if we only replace the HDD with the SSD without changing any algorithm, then the time (BL on the SSD) drops to 199ms due to SSD's fast I/O performance. However, if we optimize the intersection algorithm by leveraging SSD's unique properties, the time (SL on the SSD) can drop to 125ms, which makes a better use of the SSD.

(3) For ZZ, it reads the same number of pages with SL when the number of lists is two.[4] As a result, the execution time of ZZ is the same as SL on the SSD. But on the HDD, ZZ is slower than SL because ZZ introduces more back-and-forth random accesses.

(4) For BF on the SSD, it can improve the performance of SL a lot because the intersection size is small. But on the HDD, bloom filters will not help because they introduce many random accesses by filtering out non-promising pages.

B. Effect of list size ratio

We define the list size ratio w as $\frac{|L_2|}{|L_1|}$. We fix $|L_2| = 10^7$, set the intersection size at 1% of $|L_1|$, and vary w from 1 to 10^4. Figure 3 demonstrates that when w is small (e.g., $w \leq 10$), all the algorithms fail to skip any pages, because many elements in L_1 are routed to the same page of L_2, making the page's reference count too high. Thus, many pages in L_2 become relevant. When $w \geq 100$, the skipping-based algorithms (e.g., SL and BF) start to skip and their performance improves.

C. Effect of intersection ratio

The intersection ratio is also an important parameter. In this experiment, we fix $|L_2| = 10^7$, $|L_1| = \frac{|L_2|}{1000}$, and vary the intersection ratio r from 0.01% (of $|L_1|$) to 100% (of $|L_1|$). Figure 4 shows the results. It shows that BL and SL do not change much as r varies. For BL, that is because it experiences a bottleneck of reading all the lists, no matter what the intersection ratio is. For SL, as long as a page range in L_2 covers an element, that page has to be loaded regardless of whether the page really contains the element. That is because SL only maintains the range information via skip pointers. But the performance of BF improves when r becomes smaller because bloom filters can rule out some unnecessary pages.

V. Results on real data

In this section, we present results on a real dataset.

We use the dataset *ClueWeb*[5] that includes 41 million Web pages crawled by CMU in 2012. The total data size is around 300GB. It is a standard benchmark in the information retrieval community. The query log contains 131,654 real queries from the TREC 2005 and 2006 (efficiency track).[6] During experiments, we run the queries over the corresponding dataset. In particular, for each query that contains k query terms, we compute list intersection among those k inverted lists to report the average execution time and the number of page accesses.

We run each algorithm in 32 threads and set the bloom filter size as 4 bits per element. We pick up three compression approaches, namely, SIMDPforDelta, SIMDPforDelta*, and SIMDBP128*, because they have high performance and low space overhead as reported in prior work [24].

Figure 5 shows the results. (1) Overall, all the skipping-based algorithms (i.e., ZZ, SL, BF) read less amount of data than the baseline BL, and therefore they also run faster than BL on the SSD as reported in Figure 5a. However, they run slower than BL on the HDD in Figure 5b due to expensive random reads introduced by skipping. It explains why existing list intersection algorithms for HDDs prefer loading the entire list to memory, i.e., "list-at-a-time single-threaded" I/O access pattern. This also confirms our conjecture that HDD-centric intersection algorithms become sub-optimal on SSDs. Thus, we shall deploy skipping-based parallel algorithms (i.e., "page-at-a-time multi-threaded" I/O access pattern) for intersection in SSD-resident IR systems.

(2) On the SSD, ZZ performs worse than SL due to more accesses to longer lists, especially when the number of lists exceeds two. Thus, we should favor SL towards ZZ.

[4]But ZZ reads more data than SL when the number of lists exceeds two. We omit the results on more lists due to space limit.

[5]http://www.lemurproject.org/clueweb12.php/

[6]http://trec.nist.gov/

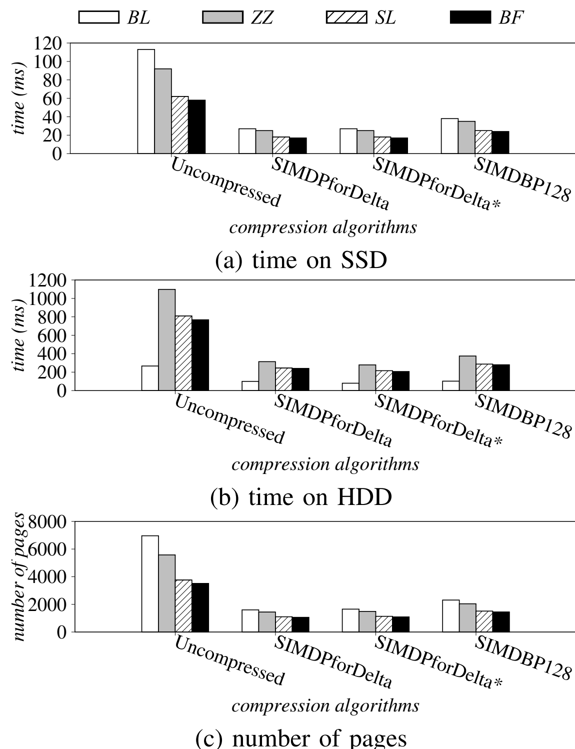

Fig. 5. **Results on ClueWeb data**

(3) The results show that compression plays an important role. The performance gap between skipping-based algorithms and the baseline BL is high on uncompressed lists. But when the lists are compressed, the gap becomes smaller. That is because if a list is compressed, then a page contains more elements. Thus, it becomes more difficult to skip a page during intersection.

(4) Figure 5 also shows an interesting result regarding different compression algorithms. As reported in [24], SIMDBP128* is the fastest for in-memory intersection. However, on disks (both SSD and HDD), SIMDBP128* is worse than SIMDPforDelta and SIMDPforDelta*. That is because SIMDBP128* consumes too much space overhead, incurring high I/O cost.

(5) Figure 5 shows an interesting result about the bloom filter based approach BF. Overall, it cannot improve much over SL on the SSD. We investigate the queries, and find that on a majority of queries BF incurs a high false positive rate.

VI. Conclusion

In this work, we conducted a series of experiments to examine the impact of fast SSDs to list intersection algorithms. The overall message of the work is that, although simply replacing HDDs by SSDs and directly running existing HDD-optimized intersection algorithms on the SSD can improve performance (since SSDs are faster than HDDs), SSDs are highly underutilized, because the design decisions are still made for HDDs. Thus, we strongly recommend practitioners to re-optimize their systems explicitly for SSDs.

References

[1] R. Baeza-Yates, R. Salinger, and S. Chile. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pages 13–24, 2005.

[2] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, pages 390–399, 2002.

[3] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *ISAAC*, pages 739–750, 2007.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[5] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[6] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.

[8] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[9] P. Ghodsnia, I. T. Bowman, and A. Nica. Parallel I/O aware query optimization. In *SIGMOD*, pages 349–360, 2014.

[10] B. Huang and Z. Xia. Allocating inverted index into flash memory for search engines. In *WWW*, pages 61–62, 2011.

[11] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SICOMP*, 1972.

[12] W. Jung, H. Roh, M. Shin, and S. Park. Inverted index maintenance strategy for flashssds: Revitalization of in-place index update strategy. *Inf. Syst.*, 49:25–39, 2014.

[13] D. E. Knuth. *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., 1973.

[14] R. Li, X. Chen, C. Li, X. Gu, and K. Wen. Efficient online index maintenance for SSD-based information retrieval systems. In *HPCC*, pages 262–269, 2012.

[15] R. Li, C. Li, W. Xiao, H. Jin, H. He, X. Gu, K. Wen, and Z. Xu. An efficient SSD-based hybrid storage architecture for large-scale search engines. In *ICPP*, pages 450–459, 2012.

[16] Y. Liu, J. Wang, and S. Swanson. Griffin: uniting CPU and GPU in information retrieval systems for intra-query parallelism. In *PPoPP*, pages 327–337, 2018.

[17] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *TOIS*, 14(4):349–379, 1996.

[18] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *PVLDB*, 5(4), 2011.

[19] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[20] P. Sanders and F. Transier. Intersection in integer inverted indices. In *ALENEX*, pages 71–83, 2007.

[21] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.

[22] J. Tong, G. Wang, and X. Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *CIKM*, pages 1209–1212, 2013.

[23] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72, 2009.

[24] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[25] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, pages 693–702, 2013.

[26] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. Cache design of ssd-based search engine architectures: An experimental study. *TOIS*, 32(4):1–26, 2014.

[27] J. Wang, D. Park, Y. Kee, Y. Papakonstantinou, and S. Swanson. SSD in-storage computing for list intersection. In *DaMoN*, pages 4:1–4:7, 2016.

[28] J. Zobel and A. Moffat. Inverted files for text search engines. *CSUR*, 38:1–56, 2006.

[29] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.