ABSTRACT

List intersection is at the core of information retrieval (IR) 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 (1 × ~ 2×) 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 five existing in-memory intersection algorithms to be SSD-aware with the idea of parallel I/O skipping, and experimentally evaluate them on synthetic datasets (of different distributions) and real datasets (300GB-ClueWeb data and 350GB-Sogou data). We investigate the effect of skip pointers, bloom filters, parallelism, cache, compression, list size, list size ratio, intersection ratio, and number of lists to the performance. Based on the results (both positive and negative), we provide insights, lessons, and recommendations on how to design efficient SSD-optimized intersection algorithms.

1. INTRODUCTION

List intersection is a fundamental operation in information retrieval (IR) systems. For instance, finding documents that contain all the query terms requires the intersection of several inverted lists. In this work, we focus on disk-resident IR 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 [3, 11, 12, 14]. 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 [16]. 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 [1, 40], 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 [17]. As a result, SSDs are deployed in many large-scale infrastructures, e.g., Google [9], Bing [28], Baidu [27], and Facebook [35]. In particular, Baidu, the largest search engine in China, has changed HDDs to SSDs completely in its storage system [27, 42].

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 × ~ 2× slower than sequential reads [40]. For instance, for the Samsung SSD1 used in our experiments, the reported theoretical sequential read bandwidth is 550MB/s while the random read bandwidth is 400MB/s; thus, the gap is 1.375×. Moreover, the gap is even smaller if the I/O access pattern is not purely random because of the internal cache inside the SSD. (2) The second interesting property is that an SSD supports parallel I/O because it is manufactured to incorporate multiple flash channels [33]. 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.² Note that, skipping does not work

---


²Of 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.
processing in memory is faster: (1) DRAM consumes much more

energy (10 ∼ 100×) than SSDs because of the frequent refresh-
ment [8], which is a serious problem in today’s data centers [18].
(2) DRAM is much more expensive than SSDs in $/GB. For in-
stance, DRAM is 20× more expensive per GB than SSDs in 2017
according to Amazon prices. As a result, many systems are still
running on SSDs. For example, Baidu (the largest search engine
in China) stores all the inverted lists on SSDs [27]. Note that we
also evaluate the impact of caching popular lists in memory in the
experiments.

2. RELATED WORK

In this section, we describe the previous work: (1) list intersec-
tion on HDDs (Section 2.1); (2) list intersection in main mem-
ory (Section 2.2); (3) database join (Section 2.3); (4) SSD-optimized
design in IR systems (Section 2.4).

2.1 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 [44]. A solution mentioned
in [44] 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.

Thus, if the lists are stored on HDDs, the best solution is to load
the lists to memory in their entirety, and carry out list intersection
in main memory.

2.2 List intersection in memory

There is rich literature on the in-memory list intersection prob-
lem, partially because all the lists of a query have to be loaded
to memory first if they are stored on HDDs (as explained in Sec-
tion 2.1), and partially because the entire inverted index is small
enough to fit in memory for some applications.

We classify the existing algorithms into four categories: compar-
based (Section 2.2.1), hash-based (Section 2.2.2), partition-based
(Section 2.2.3), and bitmap-based (Section 2.2.4). Table 1 provides
an overview. We describe the existing algorithms using $k$ sorted
lists $L_1, L_2, \ldots, L_k$ ($|L_1| \leq |L_2| \leq \cdots \leq |L_k|$).

2.2.1 Comparison-based intersection algorithms

The first comparison-based intersection algorithm is the sort-
merge algorithm [20], which would access the entire lists if ex-
tended to SSDs.

To skip unnecessary comparisons, the SL (short-long) algorithm [11]
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 $L_1$ can be obtained by invoking Member($L_1$, $e$), which returns
ture if $L_1$ contains $e$. In SL, Member($L_i$, $e$) can be implemented
using binary search, skip lists [32] or treaps [6]. We extend SL to
SSDs, so that it skips pages.
The SvS algorithm [13] 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
interacted with $L_3$. The process continues until $L_n$. SvS also
relies on Member($L_i$, $e$) to test whether $e$ is in $L$. Thus, basically,
Table 1: A list of in-memory list intersection algorithms and the extensions to SSDs. Only the highlighted algorithms would skip pages if extended to SSDs.

<table>
<thead>
<tr>
<th>Categories</th>
<th>In-memory intersection</th>
<th>Extensions to SSDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison-based</td>
<td>Sort-merge [20]</td>
<td>Load the entire lists</td>
</tr>
<tr>
<td></td>
<td>Zig-Zag [4, 12]</td>
<td>Can skip pages</td>
</tr>
<tr>
<td>Hash-based</td>
<td>Simple hash [22]</td>
<td>Can skip pages</td>
</tr>
<tr>
<td></td>
<td>BPP [5]</td>
<td>Load the entire lists</td>
</tr>
<tr>
<td>Partition-based</td>
<td>Divide &amp; conquer [3]</td>
<td>Load the entire lists</td>
</tr>
<tr>
<td></td>
<td>Group [14]</td>
<td>Load the entire lists</td>
</tr>
<tr>
<td>Bitmap-based</td>
<td>Bitmap [11]</td>
<td>Load the entire bitmaps</td>
</tr>
</tbody>
</table>

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 \texttt{Successor}(L, e) for skipping [4,12]. \texttt{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_k), 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 \texttt{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 \texttt{Successor}(L, e). ZZ terminates once e is invalid.

If L is compressed (e.g., using PforDelta [45]), neither \texttt{Member}(L, e) nor \texttt{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 [31,36]. The original list L is then split into segments. The auxiliary data structure maintains for each segment the minimum/maximum element (uncompressed). Thus, \texttt{Member}(L, e) and \texttt{Successor}(L, e) can be implemented by routing to the desired segment and decompressing it individually.

### 2.2.2 Hash-based intersection algorithms

The naive hash-based intersection algorithm [22] builds a hash table to implement \texttt{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 [11] 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 [5]. 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.

### 2.2.3 Partition-based intersection algorithms

Baesa-Yates et al. proposed an algorithm for intersection based on the divide-and-conquer framework [3]. 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 [14], 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 [14]).

### 2.2.4 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 [11]. If extended to SSDs, the entire bitmap of a list has to be loaded to memory.

### 2.3 Database join

Next, we review studies in database join since list intersection can be regarded as a special case of join. In the past, many join algorithms were developed, e.g., block nested loop join, sort-merge join, index join, and hash join [29]. Among them, only the index join approach can be applied to list intersection in order to skip pages. Indeed, the index join approach is essentially the SL or SvS algorithm if extended to SSDs.

Note that the reason why it makes sense to develop skipping-based algorithms for database join is that a database table is usually very big, e.g., several GBs or TBs. Thus, even on HDDs, skipping is worthy because a lot of unnecessary data accesses can be avoided. However, in IR systems, the size of an inverted list is usually less than tens of MBs. Skipping on such MB-scale data does not pay off the slow random reads on HDDs, considering there will be many random accesses to complete an intersection.

There are also a number of studies exploring the impact of SSDs to database join [15,37]. Those studies focus on mitigating the slow random writes of SSDs, because join algorithms usually need to write data back to SSDs if the buffer is insufficient. However, our focus on intersection is leveraging the performance of sequential reads and random reads (read-only).

### 2.4 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 [19] to maximize query performance while minimizing operational cost. A similar issue was also discussed in [26]. The impact of SSDs on cache management was studied in [39,42,43]. 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 [21,25] 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.

### 3. 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 2. 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 \( L_1, L_2, ..., L_k \) \( (|L_1| \leq |L_2| \leq \cdots \leq |L_k|) \) stored on an SSD. Each list is stored in pages of a typical size, e.g., 4KB [40,42]. 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 initially 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 [31,44]. Each entry is a document ID \( d_i \) of 4 bytes before compression.\(^3\) 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 [45] and SIMD/PforDelta [41]. We also evaluate the impact of different compression schemes to the performance. If a list

\(^3\) All the evaluated algorithms are applicable to entries that contain other auxiliary information, e.g., document frequencies and positions.
is compressed, we follow prior work [31, 36] to build skip pointers such that intersection only needs to examine those promising pages.

3.1 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 [11].

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.

3.2 SL (Short-Long)

As mentioned in Section 2.2.1, an important operation used in SL is Member$(L, e)$, which checks whether element $e$ is in list $L$ as illustrated in Section 2.2.1. In main memory, it is usually implemented using binary search or skip lists. On SSDs, we implement Member$(L, e)$ by pre-computing skip pointers [31]: we store for each page the minimum compressed ID (called a skip pointer). As an example consider the list $L_3$ in Figure 1, and assume each page contains four elements. Figure 2 depicts the skip pointers: 10, 80, 160, and 400.

Figure 2: An example of the skip pointers of $L_3$ in Figure 1. The skip pointers are 10, 80, 160, and 400.

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. Besides, the skip pointers of a list are stored in memory. Algorithm 1 shows the extended SL algorithm. It reads the shortest list to memory and skip over the longer lists via Member$(L, e)$.

Algorithm 1: Short-long (SL) algorithm [10, 11]

| Input: $k$ lists $L_1$, $L_2$, ... $L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$) |
| Output: $L_1 \cap L_2 \cap \cdots \cap L_k$ |
| 1 result set $R$ ← $\varnothing$; |
| 2 load $L_1$ from the SSD to memory; |
| 3 for each element $e \in L_1$ do |
| 4 flag ← true; |
| 5 for $i ← 2$ to $k$ do |
| 6 if Member($L_i, e$) is true then |
| 7 continue; |
| 8 else |
| 9 flag ← false; |
| 10 break; |
| 11 if flag is true then |
| 12 put $e$ to $R$; |
| 13 return $R$; |

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

3.3 ZZ (Zig-Zag)

The Zig-Zag (ZZ) intersection algorithm [4, 12] adopts another way of using skip pointers for intersection as described in Section 2.2.1. ZZ implements Successor$(L, e)$ to find the successor of $e$ in $L$ instead of implementing Member$(L, e)$. When extended to SSDs, we can implement Successor$(L, e)$ using skip pointers (shown in Figure 2) and only read the promising page to skip unnecessary pages, see Algorithm 2. The skip pointers are stored in memory.

Algorithm 2: Zig-Zag (ZZ) [12, 13]

| Input: $k$ lists $L_1$, $L_2$, ... $L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$) |
| Output: $L_1 \cap L_2 \cap \cdots \cap L_k$ |
| 1 result set $R$ ← $\varnothing$; |
| 2 set $e$ as the first element of $L_1$; |
| 3 $i ← 2$; |
| 4 while $e$ is not invalid do |
| 5 let $L_i$ be the current list to examine; |
| 6 $s ←$ Successor$(L_i, e)$; |
| 7 if $s = e$ then |
| 8 counter ← counter + 1; |
| 9 if counter = $k$ then |
| 10 put $e$ to $R$; |
| 11 else |
| 12 $e ← s$; |
| 13 $i ← (i + 1) \mod k$; |
| 14 return $R$; |

3.4 SBF (Single-Bloom-Filter)

We observe that skip pointer based intersection algorithms, i.e., SL and ZZ still incur unnecessary page accesses:

- **Case 1:** 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. As an example, let $e = 200$, and a page contains four elements: $\{15, 100, 300, 500\}$. The page will be identified by the skip pointers since $e$ falls in the range. However, $e$ is not in the page.

- **Case 2:** Even if $e$ is in $L_1, \ldots, L_{i-1}$ ($2 \leq i \leq k$), it may not be in $L_i$, making the previous page accesses (containing $e$) from $L_1, \ldots, L_{i-1}$ wasteful. As an example, suppose $e$ is in $L_1$ and $L_2$ but not in $L_3$, then the pages in $L_1$ and $L_2$ that contain $e$ are unnecessarily accessed, since $e$ is not a result.

To solve the problem, a natural idea is to use bloom filters [7]. In particular, we build for each page a bloom filter to improve the SL algorithm. Note that ZZ cannot benefit from bloom filters because ZZ relies on Successor$(L, e)$ instead of Member$(L, e)$ while bloom filters can only be helpful for membership testing. We show two modifications of SL to leverage bloom filters, namely, SBF (single-bloom-filter) and MBF (multi-bloom-filter), see Algorithm 3 and Algorithm 4. For fast execution, we store all the bloom filters in main memory.

In SBF, whenever a target page $P$ (whose range contains $e$) is identified by the skip pointers, instead of loading $P$ immediately, SBF performs a bloom test between $e$ and the bloom filter of $P$. If $e$ fails the bloom test (i.e., the bloom test returns false), there is no
Algorithm 3: SBF (Single-Bloom-Filter)

Input: $k$ lists $L_1, L_2, \ldots, L_k$ on a SSD ($|L_1| \leq \cdots \leq |L_k|$)
Output: $L_1 \cap L_2 \cap \cdots \cap L_k$

1. result set $R \leftarrow \emptyset$
2. read the shortest list $L_1$ to memory;
3. for each element $e \in L_1$ do
   4. flag $\leftarrow$ true;
   5. for $i \leftarrow 2$ to $k$ do
      6. find the page $P$ of $L_i$ whose range covers $e$ in skip pointers;
      7. test whether $e \in P$ using the bloom filter of $P$;
      8. if the bloom test returns false then
         9. flag $\leftarrow$ false;
        10. break;
      else
        11. read the page $P$ from the SSD to memory;
        12. if $e \notin P$ then
           13. flag $\leftarrow$ false;
           14. break;
        15. if flag is true then
           16. put $e$ to $R$;
        17. return $R$;

Algorithm 4: MBF (Multi-Bloom-Filter)

Input: $k$ lists $L_1, L_2, \ldots, L_k$ on a SSD ($|L_1| \leq \cdots \leq |L_k|$)
Output: $L_1 \cap L_2 \cap \cdots \cap L_k$

1. result set $R \leftarrow \emptyset$
2. read the shortest list $L_1$ to memory;
3. for each element $e \in L_1$ do
   4. flag $\leftarrow$ true;
   5. // phase 1: filtering
      for $i \leftarrow 2$ to $k$ do
         6. find the page $P$ of $L_i$ whose range covers $e$ in skip pointers;
         7. test whether $e \in P$ using the bloom filter of $P$;
         8. if the bloom test returns false then
            9. flag $\leftarrow$ false;
            10. break;
      // phase 2: verification
      if flag is true then
         11. for $i \leftarrow 2$ to $k$ do
            12. read the page $P$ (of $L_i$ whose range covers $e$) from the SSD to memory;
            13. if $e \notin P$ then
               14. flag $\leftarrow$ false;
               15. break;
            16. if flag is true then
               17. put $e$ to $R$;
   18. return $R$;

need to read $P$ since $e$ is definitely not in $P$. Otherwise SBF reads it. SBF can skip the unnecessary pages introduced by Case 1, but not Case 2.

We analyze the false positive rate of reading a non-promising page. Let $s$ be the bloom filter size (in number of bits per element), then the false positive rate of accessing an element in the page is $0.6185^s$ [30]. Let $c$ be the number of elements that are routed to this page (we refer to it as “reference count”) via skip pointers, then the false positive rate of accessing the page is $1 - (1 - 0.6185)^c$ because as long as one of the $c$ bloom tests produces a false positive, the page has to be unnecessarily accessed. Thus, the effectiveness of SBF depends on the bloom filter size and the reference count.

3.5 MBF (Multi-Bloom-Filter)

MBF (multi-bloom-filter) is another way of using bloom filters, see Algorithm 4. It is different from SBF when the number of lists $k \geq 3$. In other words, MBF and SBF are the same when the number of lists is two. It has a filtering and a verification phase.

(1) Filtering phase: if the bloom test of SBF returns true, instead of reading the target page immediately, MBF performs the bloom tests between $e$ and all the other lists as well. In other words, in the filtering phase, MBF checks whether $e$ (in $L_1$) passes the bloom tests of all the other lists $L_2, \ldots, L_k$. If not, there is no need to read any page because $e$ is definitely not a result. If yes, MBF continues to the verification phase. (2) Verification phase: MBF reads the $(k - 1)$ target pages from $L_2, \ldots, L_k$ that passed the filtering phase. The MBF algorithm can skip the unnecessary pages introduced by both Case 1 and Case 2.

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

3.6 Parallel intersection algorithms

All the algorithms experimented (BL, SL, ZZ, SBF, and MBF) 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 [38]. It works as follows. Assume there are $N$ threads, then it partitions $L_i$ into $N$ even sublists: $L_{i1}, L_{i2}, \ldots, L_{iN}$. For each sublist $L_{ij}$ ($1 \leq j \leq N$), it performs binary search of $L_{ij}[0]$ on $L_j$ ($j \geq 2$) to partition $L_j$ into $N$ sublists. Thus, the $N$ intersection tasks are: $(L_{i1} \cap L_{j1} \cap \cdots \cap L_{N1}), (L_{i2} \cap L_{j2} \cap \cdots \cap L_{N2}), \ldots, (L_{iN} \cap L_{jN} \cap \cdots \cap L_{jN})$. 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.

4. RESULTS ON SYNTHETIC DATA

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

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++. We use the Win32 native thread APIs

\[5\text{http://www.samsung.com/semiconductor/minisite/ssd/product/}
\text{consumer/850pro.html}
\]

\[6\text{https://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barra}
\text{cuda-ds1737-1-1111us.pdf}\]
to implement parallelism. By default, all the algorithms run at 32 threads in parallel. Although the CPU has 8 hyper-threaded cores, opening 32 threads can also improve performance because of frequent I/O operations. Moreover, the page size is 4KB by default (in both HDD and SSD).

Evaluation metrics. We measure the performance of an algorithm in two aspects: actual execution time and number of pages accessed. The execution time is averaged across three runs for accuracy.

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 2.

4.1 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 2) except we vary the number of threads from 1 to 32. Figure 3 shows the execution time of the five algorithms on both SSD and HDD. Table 3 shows the number of pages for each algorithm, which is irrelevant with the number of threads. There are many interesting results of this experiment.

(1) For the baseline algorithm BL, Figure 3 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 3a shows that when the number of threads is 1, SL is even slower than BL although SL reads less data than the baseline BL (see Table 3). 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 3b. 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 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 [34]. 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 execution 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 execution 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. 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 SBF and MBF, they are basically the same when the number of lists is two. Thus, they have the same execution time

Note that the results on the HDD do not monotonically change with the number of threads. That is because of the impact of disk I/O reordering if multiple I/O requests are issued, which highly depends on disk internals [34].

But as we show later in Figure 7, ZZ reads more data than SL when the number of lists exceeds two.

### Table 2: Default parameters used in synthetic data

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of threads</td>
<td>32</td>
</tr>
<tr>
<td>number of lists</td>
<td>2</td>
</tr>
<tr>
<td>list size</td>
<td>$</td>
</tr>
<tr>
<td>list size ratio</td>
<td>1000</td>
</tr>
<tr>
<td>intersection ratio</td>
<td>1% of $</td>
</tr>
<tr>
<td>bloom filter size</td>
<td>4 bits (per element)</td>
</tr>
<tr>
<td>data distribution</td>
<td>uniform (from domain $[0, 2^{32} - 1]$)</td>
</tr>
<tr>
<td>cache</td>
<td>no</td>
</tr>
<tr>
<td>compression</td>
<td>no</td>
</tr>
</tbody>
</table>

### Table 3: Number of pages for Figure 3

<table>
<thead>
<tr>
<th></th>
<th>BL</th>
<th>ZZ</th>
<th>SL</th>
<th>SBF</th>
<th>MBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>11177</td>
<td>6553</td>
<td>6553</td>
<td>1514</td>
<td>1514</td>
</tr>
<tr>
<td>2 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Effect of number of threads
and read the same amount of data. On the SSD, they 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.

4.2 Effect of list size

Figure 4 shows the effect of list size on the SSD. We use two lists \( L_1 \) and \( L_2 (|L_1| \leq |L_2|) \). We vary \( |L_2| \) from 1 million to 100 million and set \( |L_2| = 1000 \), thus, \( |L_1| \) also increases. The rest parameters follow the default values shown in Table 2. We omit ZZ and MBF because for two-list intersection, ZZ is the same as SL and MBF is the same as SBF.

![Figure 4: Effect of list size](image)

Figure 5 is a horizontal figure showing the effect of list size ratio on the SSD. Figure 5 shows the effect of list size ratio on the SSD. We use two lists \( L_1 \) and \( L_2 \). We vary \( |L_2| \) from 1 million to 100 million and set \( |L_2| = 1000 \). Figure 5 shows the results for all the algorithms, the execution time and the number of page accesses as \( w \) increases, because \( |L_1| \) decreases.

Figure 5 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 SBF) start to skip and their performance improves.

4.3 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 5 shows the results. For all the algorithms, the execution time and the number of pages decrease as \( w \) increases, because \( |L_1| \) decreases.

Figure 5 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 SBF) start to skip and their performance improves.

4.4 Effect of intersection ratio

The intersection ratio is also an important parameter. In this experiment, we fix \( |L_2| = 10^7 \), \( |L_1| = \frac{|L_2|}{10000} \), and vary the intersection ratio \( r \) from 0.01% (of \( |L_1| \)) to 100% (of \( |L_1| \)). Figure 6 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 SBF improves when \( r \) becomes smaller because bloom filters can rule out some unnecessary pages.

4.5 Effect of number of lists

In this set of experiments, we examine the effect of the number of lists \( k \) (ranging from 2 to 5). We largely follow [14] to generate lists. In particular, we set \( |L_1| = 10^8 \) and \( |L_2| = |L_3| = |L_4| = |L_5| = 10^7 \) because in practice, short and long lists are usually...
mixed in a query. Moreover, we set the intersection size between \( L_i \) (\( \geq 2 \)) and \( L_1 \) as 10% of \( |L_1| \). As a result, \( |L_1 \cap L_2| = 1143, |L_1 \cap L_2 \cap L_3| = 113, |L_1 \cap L_2 \cap L_3 \cap L_4| = 18, |L_1 \cap L_2 \cap L_3 \cap L_4 \cap L_5| = 4 \).

Figure 7 plots the results. It reveals that the skipping-based algorithms scale well with \( k \) due to efficient skipping. More importantly, the gap between BL and SL (or SBF, MBF) becomes larger as \( k \) increases. That is because SL, SBF, and MBF perform list intersection from short lists to long lists. Let \( L_i \) be the current list, then the intersection size of the previous lists is \( |L_1 \cap L_2 \cap \ldots \cap L_{i-1}| \). Thus, larger the \( i \), the gap between \( |L_1 \cap L_2 \cap \ldots \cap L_{i-1}| \) and \( |L_i| \) increases because the list size is increasing from \( L_1 \) to \( L_k \), while the intersection size is getting smaller with more lists being intersected. According to Figure 5, the performance gap increases when the list size ratio increases. Thus, it becomes even more relevant to develop SSD-optimized algorithms when the intersection involves many lists.

Figure 7 also shows that MBF outperforms SBF because MBF can filter out more unnecessary pages. Another interesting result is that the number of pages accessed by MBF can even decrease when \( k \) increases. This is because the false positive rate of a page decreases when \( k \) becomes larger in MBF; thus, the bloom filters in MBF can in turn filter out many unnecessary pages. Figure 7 also confirms that SL is better than ZZ when \( k \geq 3 \). The intuition is that, consider at some point, ZZ and SL access the same element \( a \) in \( L_1 \). Then for ZZ, after it walks through one iteration (accessing \( L_1 \rightarrow L_2 \rightarrow \ldots \rightarrow L_k \rightarrow L_1 \)) and returns to \( L_1 \) again, it will access the element that is in the same page of \( a \) with high probability if the lists are distributed uniformly. However, if \( a \) is not a result, ZZ wastes many I/Os compared to SL because SL can switch back to \( L_1 \) much earlier since only the intersection of \( L_1 \ldots L_{i-1} \) are intersected with \( L_i \).

4.6 Effect of bloom filter effectiveness

Next, we evaluate SBF to understand the effectiveness of bloom filters. The effectiveness highly depends on the false positive rate, i.e., \( f = 1 - (1 - 0.6185^s)^c \) where \( s \) is the bloom filter size (in bits per element) and \( c \) is the reference count of the page. Thus, we first plot \( f \) with varying \( s \) and \( c \) in Figure 8. It shows that \( f \) increases with both \( s \) and \( c \). In particular, when \( s = 1, 2, 4, f \) increases very quickly when \( c \) increases. Thus, bloom filters are effective only if the reference count is low, unless the bloom filter size is high say \( s = 16 \) bits. However, setting a high bloom filter size may not pay off when compared with caching lists as we show later on in Figure 17.

Then we evaluate the actual performance of SBF in Figure 9 with varying bloom filter size. We use three settings that have different list sizes and intersection sizes, i.e., they have different reference counts of data pages. In Figure 9a, \( |L_1| = 10^4, |L_2| = 10^7 \), and \( |L_1 \cap L_2| = 100 ; \) In Figure 9b, \( |L_1| = 10^5, |L_2| = 10^7 \), and \( |L_1 \cap L_2| = 10000 \); In Figure 9c, it has the same setting with Figure 9b unless the lists are compressed using SIMDForDelta [23], which performs the best as we see in Section 4.8. As a reference, we also show the baseline BL. Generally, Figure 9 shows that the performance improves when \( s \) increases. But the performance improvement is different. In particular, Figure 9a shows the best improvement when \( s \) increases while Figure 9c shows the worst improvement. That is because in Figure 9a, data pages’ reference counts are much smaller than that in Figure 9c. Note that the reference count of a page in \( L_2 \) can be roughly estimated as the ratio between \( |L_1| \) and the number of pages taken by \( L_2 \). Thus, they have a low false positive rate according to Figure 8.

4.7 Effect of Zipf distribution

In the previous experiments, we assume lists are distributed uniformly. In the next experiment, we assume lists are distributed follow the zipf distribution. In particular, the value \( i \) in a list is included with a probability of \( \frac{i^{1/f}}{\sum_{i=1}^{d} (i^{1/f})} \) where \( d \) is the domain size \( (2^{12}) \) and \( f \) is the skewness factor. Note that when \( f = 0 \), the zipf distribution is the same as uniform distribution. We use the default setting unless the lists are distributed differently.

Figure 10 shows the results. It shows that when \( f \) increases from 0 to 1, the performance of BL is not affected because it reads the whole lists. However, SL and SBF read less data as the lists become more skewed. That is because the intersection results tend to be concentrated on fewer pages if data is more skewed, which provides a good opportunity for data skipping.

4.8 Effect of compression

Figure 11 compares BL, SL, and SBF when the lists are compressed using different compression algorithms. We use the de-
fault settings unless the lists are compressed. We evaluate the performance based on three best compression approaches, namely, SIMDPforDelta, SIMDPforDelta*, and SIMDBP128*, as they have high performance and low space overhead as reported in [41].

Figure 11 shows that all the algorithms become faster when compared to uncompressed lists because the lists take less space after compression. We see that the gap between BL and skipping-based algorithms (SL and SBF) becomes smaller on compressed data. That is because after compression, a data page contains more elements and it becomes more difficult to skip a page unless the intersection size is very small. But still, SL and SBF perform better than BL.

![Figure 10: Effect of Zipf distribution](image)

![Figure 11: Effect of compression](image)

**4.9 Effect of cache**

Finally, we evaluate the effect of caching popular lists in memory. We use five lists generated in Section 4.5 and vary the number of lists cached, see Figure 12. Let \( i \) be the number of lists cached, it means the first \( i \) lists are cached in memory.

Figure 12 shows that even with cached lists, it makes even more sense for skipping on-disk lists. That is because the intersection size of the in-memory lists can be very small compared to the sizes of the on-disk lists since \( |L_1| \leq \ldots \leq |L_k| \). Then the skipping-based algorithms can skip many unnecessary pages of the on-disk lists. Thus, as long as some lists are stored on the SSD, it becomes interesting to apply skipping-based intersection algorithms instead of using conventional BL algorithm.

5. RESULTS ON REAL DATA

In this section, we present experimental results on real datasets. We use two datasets that contain real-life Web pages, namely ClueWeb and Sogou.

- **ClueWeb**\(^{10}\) It includes 41 million Web pages (in English) 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 (in English) from the TREC 2005 and 2006 (efficiency track).\(^{11}\)

- **Sogou**\(^{12}\) It is a real collection of 50 million Web pages (in Chinese) crawled in 2012, by Sogou.com, a famous commercial search engine in China. The total data size is around 350GB. The query log contains 178,776 real queries (in Chinese) collected by the same search engine.

For each dataset, we parse the Web documents and build inverted lists for the terms. 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. Table 4 shows the detailed data statistics including list size, intersection ratio, list size ratio, and number of lists. Unless otherwise stated, we run each algorithm in 32 threads and set the bloom filter size as 4 bits per element.


\(^{12}\) [https://www.sogou.com/](https://www.sogou.com/)
We first consider the setting where all the lists are stored on the disk initially without caching any lists in memory. Figure 13 and Figure 14 show the results on ClueWeb and Sogou of the five intersection algorithms where the lists are stored using a different compression approach, namely, SIMDPforDelta, SIMDPforDelta*, and SIMDBP128*, because they have high performance and low space overhead as reported in prior work [41].

There are many interesting results shown in Figure 13 and Figure 14. (1) Overall, all the skipping-based algorithms (i.e., ZZ, SL, SBF, MBF, OPT) read less amount of data than the baseline BL, and therefore they also run faster than BL on the SSD as reported in Table 4: Data characteristics.

Figure 13: Results on ClueWeb data

Figure 14: Results on Sogou data

Figure 13a and Figure 14a. However, they run slower than BL on the HDD in Figure 13b and Figure 14b 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.

(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) The two figures also show an interesting result regarding different compression approaches. As reported in [41], 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 13 and Figure 14 show an interesting result about bloom filter based approaches, i.e., SBF and MBF. Both of them cannot improve much over SL on the SSD. Also, MBF has similar performance with SBF. To understand why, we output for each query the number of page accesses incurred by SBF and MBF. Then we sort the queries in Figure 15 according to the ratio r_1 =...
Figure 15: Relative performance of SL/SBF, SBF/MBF, and SL/OPT on ClueWeb

Table 5: Cache hit ratio of different cache sizes in Figure 16

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>ClueWeb</th>
<th>Sogou</th>
</tr>
</thead>
<tbody>
<tr>
<td>0GB</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>1GB</td>
<td>20%</td>
<td>37%</td>
</tr>
<tr>
<td>2GB</td>
<td>32%</td>
<td>57%</td>
</tr>
<tr>
<td>4GB</td>
<td>49%</td>
<td>81%</td>
</tr>
</tbody>
</table>

Figure 16: The effect of cache

Figure 17: Bloom filter cache vs. list cache
versus caching lists (compressed using SIMDForDelta), see Figure 17. It illustrates that list cache is much better than bloom filter cache on both datasets. Thus, if there is abundant memory available, we highly recommend to cache lists than bloom filters.

6. CONCLUSION

In this section, we summarize the work, provide recommendations and lessons on how to design SSD-optimized intersection algorithms, and then outline the future work.

6.1 Summary

In this work, we conducted a series of experiments to examine the impact of fast SSDs to list intersection algorithms. We showed that SSD-optimized intersection algorithms shall use “page-at-a-time” I/O access pattern instead of conventional “list-at-a-time single-threaded” pattern. In terms of the algorithmic design choices, we presented the impact of skip pointers, bloom filters, parallelism, cache, compression, list size, list size ratio, intersection ratio, and number of lists to the performance. We summarize the main findings as follows:

1. Skip pointers play a critical role and SL efficiently leverages skip pointers for fast intersection (faster than ZZ).
2. Bloom filters may not improve performance significantly as we thought. Both SBF and MBF are not as efficient as expected especially on real datasets, which is a negative result. Since bloom filters consume significant amount of memory, it becomes worthy to cache compressed lists instead of bloom filters to improve performance.
3. Parallelism is very important to improve the performance of skipping-based intersection algorithms because it can fully utilize SSD’s I/O bandwidth if the underlying I/O access pattern is page-by-page. That is, page-level skipping must be combined with parallelism for high performance.
4. With cache, it still makes sense to skip unnecessary pages in a parallel manner unless all the lists are cached in main memory.
5. With compression, it is also beneficial for page-level skipping although it becomes more difficult when compared to skipping over uncompressed lists.
6. List size does not necessarily affect the effectiveness of skipping on SSDs.
7. List size ratio significantly affects the effectiveness of skipping on SSDs. The higher the ratio is, the more efficient the skipping becomes.
8. Intersection ratio does not necessarily affect the performance of skip pointer based algorithms such as SL and ZZ but it affects bloom filter based algorithms such as SBF and MBF.
9. Number of lists \(k\) can also affect the performance of skipping because it makes the list size ratio more skew as the intersection size shrinks quickly with top few lists being intersected. The higher the \(k\) is, the more efficient the skipping becomes.

6.2 Recommendations

The overall message of the work is that existing HDD-centric list intersection algorithms are not suitable on SSDs and we shall develop SSD-optimized intersection algorithms. In particular,

1. We highly recommend SL (with multiple threads) as an SSD-optimized intersection algorithm. It has high performance, takes small memory footprint, and is also easy to be integrated into existing systems.
2. If memory is sufficient, we recommend caching popular lists (compressed) and then apply SL to skip unnecessary pages on SSDs, instead of caching bloom filters to use SBF or MBF.

6.3 Lessons

We provide the lessons that people can learn from this work.

1. 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.
2. Parallel I/O skipping, which previously cannot improve list intersection in HDD-resident IR systems, now becomes compelling in SSD-resident IR systems.
3. Do use parallelism on SSDs if data is accessed page-by-page, because that can significantly reduce the execution time.
4. It is still worthy to consider skipping on SSDs for intersection even when the lists are compressed or cached.
5. Cache compressed lists if memory is sufficient; do not cache bloom filters since that gives marginal gains.
6. For HDD-based intersection, use BL and do not use skipping-based algorithms (e.g., SL, ZZ, SBF, and MBF).

6.4 Future work

There could be many interesting follow-ups from this work. Below we name a few.

- Efficient inverted list update on SSDs. Traditionally on HDDs, each inverted list is stored continuously to minimize the expensive random accesses [44]. As a result, for the update, it incurs a lot of data movement to guarantee the continuity [24]. However, this work indicates that on SSDs, the inverted list is not required to be stored continuously since the entire list is not required during query processing. Thus, it is interesting to explore efficient update solutions with minimal data movement.
- Extension to next-generation ultra-fast storage, e.g., Intel 3D XPoint SSDs and non-volatile memories (NVRAM). With new storage devices, the idea of skipping-on-disk is still relevant because they support fast random accesses and parallel I/O access. But the interesting thing is that the I/O is fast enough to be blurred with memory. For example, it is expected that the new generation NVRAM has similar performance with DRAM [2]. As a result, it is interesting to investigate whether it is still worthy to cache inverted list in memory or in general how to best utilize DRAM and NVRAM for efficient query processing.
- Extension to hybrid storage systems that includes both SSDs and HDDs. As we show in this work, SSDs and HDDs exhibit different properties such that they demand different intersection algorithms. Thus, it is interesting to investigate how to design algorithms that are applicable for heterogeneous storage devices.
7. REFERENCES


