SSD In-Storage Computing for Apache Lucene

Paper ID: 558

ABSTRACT

Recently, there is a renewed interest of *In-Storage Computing* (ISC) in the context of solid state drives (SSDs), called "Smart SSDs". Unlike the traditional CPU-centric computing systems, ISC devices play a major role in computation by offloading key functions of host systems into the ISC devices, to take advantage of the high internal bandwidth, low I/O latency and computing capabilities. It is challenging to determine what functions should be executed in the ISC devices.

This work explores how to apply Smart SSDs to Apache Lucene (a popular open-source search engine). The major research issue is to determine which query processing steps of Lucene can be cost-effectively offloaded to Smart SSDs. To answer this question, we identified five commonly used operations in Lucene (and any search engine) that could potentially benefit from Smart SSDs and we codesigned their operation (with the collaboration of an SSD vendor X) between the host system and the X-SSD device. The five operations are intersection, ranked intersection, ranked union, difference, and ranked difference. Finally, we conducted extensive experiments to evaluate the performance and tradeoffs by using both synthetic datasets and real datasets (provided by a commercial large-scale search engine company). The experimental results show that, for some operations, Smart SSDs can reduce the query latency by a factor of 2-3× and energy consumption by 8-10×.

1. INTRODUCTION

Solid state drives (SSDs) have gained much momentum in the storage market because of the compelling advantages of SSDs over hard disk drives (HDDs). E.g., SSDs are one to two orders of magnitude faster than HDDs in random reads [2]. In past years, many research studies discussed how to *make full use* of SSDs in high level software systems (e.g., database systems) than just use SSDs as yet-anther-faster HDDs. E.g., SSD-aware Btrees [24], SSD-aware buffer management [15]. Those works share the same goal of optimizing software systems, while treating SSDs as storage-*only* devices. In this way, data storage and computing is rigorously *separated*: data is store on SSDs, while computing happens at host machines. Upon computation, data is transferred from SSDs to host machines through a host I/O interface (typically SAS or SATA).

However, recent studies indicate this "*move data closer to code*" computing paradigm cannot make full use of SSDs [14, 35], for two reasons. (1) SSDs generally provide higher internal bandwidth than the host I/O interface bandwidth. However, data has to be transferred via the host I/O interface, which can saturate easily for data-intensive applications, making the high internal bandwidth provided by SSDs wasteful; (2) SSDs generally provide high computing capabilities (for executing complex FTL firmware code [9]), which are ignored by high level systems where SSDs are treated as storage-*only* devices.

Thus, Do et. al proposed an approach of integrating storage and computing inside SSDs, called *Smart SSDs* [14] or *SSD In-Storage Computing (ISC)*[1]. Smart SSDs allow the execution of application specific code (e.g., database scan and aggregation) inside SSDs, to take advantage of the high internal bandwidth, low I/O latency and computing power. In this way, the computing paradigm is changed to "*move code closer to data*" (or generally near-data processing [5]). In addition to the performance improvement, Smart SSDs can also reduce energy significantly, because of less data movement and power-efficient processors running inside SSDs. As a result, executing application logic inside Smart SSDs is a promising solution to *make full use* of SSDs. It is also attractive to industry. E.g., IBM applies Smart SSDs in their blue gene storage systems [18].

This work explores how to apply Smart SSDs to Apache Lucene[2] – a popular open-source search engine system. The major research issue is, *what query processing steps of Lucene can be cost-effectively offloaded to SSDs*? To answer this question, we identified five commonly used operations in Lucene (and any search engine) that could potentially benefit from Smart SSDs and we codesigned their operation (with the collaboration of an SSD vendor X) between the host system and the X-SSD device. The five operations are intersection, ranked intersection, ranked union, difference, and ranked difference. Finally, we conducted extensive experiments to evaluate the performance and tradeoffs by using both synthetic datasets and real datasets (provided by a commercial large-scale search engine company). The experimental results show that, for some operations, Smart SSDs can reduce the query latency by a factor of 2-3× and energy consumption by 8-10×.

To the best of our knowledge, this is the first work to explore SSD in-storage computing to search engine area.

The rest of this paper is organized as follows. Section 2 provides an overview of SSD internal architecture, and Lucene's search architecture. Section 3 describes how our Smart SSD work. Section 4

[1]In this work, we use the term "Smart SSD" and "SSD In-Storage Computing" interchangeably.

[2]`http://lucene.apache.org`

discusses the integrated system design space and architecture of the Smart SSD and Lucene. Section 5 details the implementation of query offloading. Section 6 and Section 7 show the evaluation, and discuss the tradeoffs of the query offloading. Section 8 discusses some related studies of this work. Section 9 concludes the work.

2. BACKGROUND

In this section, we present the background of SSD internals (Section 2.1) and Lucene's architectures (Section 2.2).

2.1 Modern SSDs

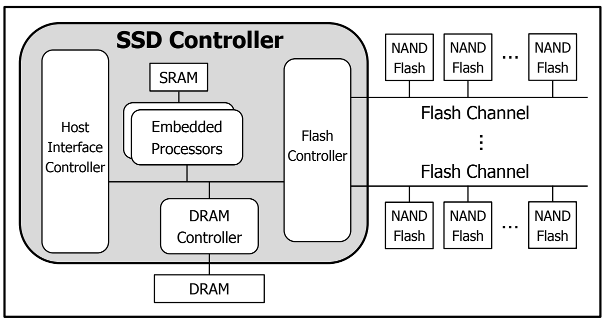


Figure 1: SSD internals

Figure 1 represents a typical modern SSD and its main components. In general, an SSD is largely composed of NAND flash memory array, SSD controller, and DRAM. The SSD controller subdivides into four main subcomponents such as host interface controller, embedded processors, DRAM controller, and flash memory controller.

Commands come from a user through the host interface and the most common interfaces, for instance, Serial ATA (SATA), Serial Attached SCSI (SAS), or PCI Express (PCIe), are implemented by the host interface controller. The embedded processors in the SSD controller receive the commands and pass them to the flash memory controller. They, more importantly, run SSD firmware codes for computation and execute Flash Translation Layer (FTL) for logical-to-physical address mapping. Typically, modern SSD is equipped with a low-powered 32-bit processor such as an ARM Cortex series processor. Each processor can have a tightly coupled memory (e.g., SRAM) for the purpose of even faster access to frequently accessed data or codes. Each processor can access DRAM through the DRAM controller. For data transfer between flash memory and DRAM, the Flash Memory Controller (FMC) is adopted. The FMC runs Error Correction Codes (ECC) and supports Direct Memory Access (DMA) functionality.

The NAND flash memory package (also called chip) is persistent storage media and each package consists of one or more dies. The die is the smallest unit that can independently execute commands or report status. Each die contains one or more planes (usually one or two). Identical or concurrent operations can take place on each plane, although with some restrictions. Each plane subdivides into a number of blocks which are the smallest erase unit, and finally each block is composed of many pages (typically 64 or 128 pages)which are the smallest read/write unit.

An SSD is also equipped with a large size of DRAM for buffering data or storing metadata of the address mapping. All the flash channels share access to the DRAM. Thus, data transfer from the flash channels to the DRAM needs to be serialized.

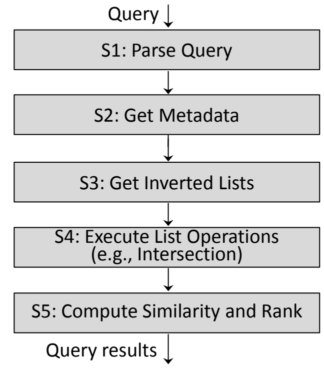


Figure 2: Query processing of Lucene

2.2 Query Processing of Lucene

Lucene is a well known open-source search engine, which is widely adopted in industry. E.g., LinkedIn [25] and and Twitter [7] adopted Lucene in their search platforms.

We provide some background of how a query is executed in Lucene. Same as other search engines, Lucene relies on the standard inverted index [36] to answer user queries efficiently. The inverted index is essentially a mapping data structure of key-value pairs, where the key is a query term, the value is a list of documents that contain the term.

Upon receiving a user query q (e.g., "SSD database"), Lucene answers it in several steps (S1 to S5 in Figure 2). By default, Lucene enables `AND` query mode (unless users explicitly specify other query modes, e.g., `OR`, `NOT`), which returns a list of documents that contain *all* of the query terms.

Step S1: parse the query to a parse tree (similar to the parse tree in SQL queries). The query q will be tokenized into several query terms. In our example, it is two query terms: "SSD" and "database". *Step S2*: get metadata for each query term. The metadata is used for loading the inverted list of each query term later on. Thus, the metadata stores some basic information about the on-disk inverted list. In Lucene, it contains (1) the offset where the list is stored on disk, (2) the length (in bytes) of the list, and (3) the number of entries in the list. *Step S3*: for each term, get the inverted list from disk to memory. *Step S4*: do list operations depending on query modes. In our example, it is intersection (because of the `AND` query mode). It could be other operations, e.g., union (`OR` mode), and difference (`NOT` mode). *Step S5*: for each qualified document d, compute the similarity between the query q and the document d using an IR relevance model. Lucene uses a modified BM25 model [31]. After that, Lucene returns the top-ranked results to end users.

We note that Lucene may not embrace all state-of-the-art query processing techniques. E.g., step S4 and S5 could be algorithmically combined for early termination [6, 16].

3. SMART SSD ARCHITECTURE

SSD In-Storage Computing (ISC) is a new computing paradigm and has recently attracted notable attention. Unlike the traditional CPU-centric computing systems, it enables ISC devices to play a major role in computation by offloading key functions of host systems into ISC devices. This paradigm reflects a recent computing trend toward *near-data processing* [5]. This section describes our Smart SSD architecture and key components as ISC devices.

As shown in Figure 3, Smart SSD consists of several key components and it communicates with a Smart SSD host program via

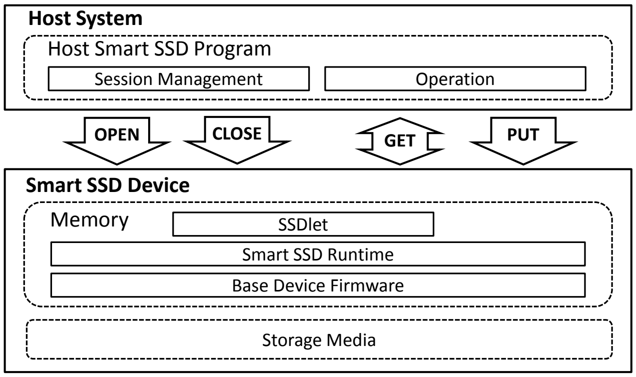


Figure 3: Smart SSD Architecture

Smart SSD application programming models (APIs).

An SSDlet is a Smart SSD program in a device. It implements application logic and responds to a Smart SSD host program. An SSDlet is executed in an event-driven manner by the Smart SSD runtime system. A Smart SSD runtime system connects the device Smart SSD program with a base device firmware and implements the library of Smart SSD APIs. In addition, a base device firmware also implements normal I/O operations (read and write) of a storage device.

After an SSDlet is installed in the Smart SSD device, a host system runs the Smart SSD host program to interact with the SSDlet in the devices. This host program consists largely of two sections: a session management component and an operation component. The session component manages the lifetime of a session for Smart SSD applications so that the host Smart SSD program can launch an SSDlet by opening a session to the Smart SSD device. To support this session management, Smart SSD provides two APIs, namely, OPEN and CLOSE. Intuitively, OPEN starts a session and CLOSE terminates the existing session. Once OPEN starts a session, runtime resources such as memory and threads are assigned to run the SSDlet and a unique session ID is returned to the host Smart SSD program. Afterwards, this session ID must be associated to interact the SSDlet. When CLOSE terminates the established session, it releases all the assigned resources and closes SSDlet associated with the session ID.

Once a session is established by OPEN, the operation component helps the host Smart SSD program interact with SSDlet in a Smart SSD device with GET and PUT APIs. This GET operation is used to check the status of SSDlet and receive output results from the SSDlet if the results are ready. This GET API implements the polling mechanism of the SAS/SATA interface because, unlike PCIe, such traditional block devices cannot initiate a request such as interrupts. PUT is used to internally write data to the Smart SSD device without help from local file systems.

4. SYSTEM CO-DESIGN: SMART SSD FOR LUCENE

In this section, we describe the system co-design of the Smart SSD and Lucene. We first explore the design space in Section 4.1 to determine what query processing logic could be cost-effectively offloaded, then show the co-design architecture of the Smart SSD and Lucene in Section 4.2.

4.1 Design Space

The overall research question of the co-design is, *what query processing logic could be cost-effectively executed by Smart SSDs?* To answer it, we need to understand the opportunities and limitations of Smart SSDs.

Opportunities of Smart SSDs. We note that, executing I/O operations in Smart SSDs is very fast, for two reasons. (1) SSDs generally provide very high internal bandwidth, e.g., 5-10× faster than the host I/O interface [14, 10]. The gap is predicated to increase in the future. (2) The I/O latency is very short. Because a regular I/O operation (from flash chips to the host DRAM) needs to go through the conventional thick OS stack, e.g., file system, interrupt, context switch between the kernel space and the user space, which is collectively called OS software overhead. The software overhead is negligible on hard disks (as I/O is very slow). However, it is not on fast SSDs [8]. In contrast, an I/O operation inside SSDs (from flash chips to the device DRAM, i.e., DRAM inside SSD) is free of the OS software overhead. Thus, it is very suitable to execute *I/O-intensive* operations inside SSDs to leverage the high internal bandwidth and low latency.

Limitations of Smart SSDs. Smart SSDs also have some limitations. (1) Generally, Smart SSDs embrace low-frequency processors (typically ARM series) to save energy and manufacturing cost. Thus, computing capability is several times lower than host CPUs (e.g., Intel processor) [14, 10]; (2) The Smart SSD also has a DRAM inside. Accessing the device DRAM is slower than the host DRAM. Because, generally, there is no caches (e.g., L1/L2 caches) in SSD controllers. Thus, it is not suitable to execute *CPU-intensive* and *memory-intensive* operations inside SSDs.

In short, Smart SSDs can reduce the I/O time, but at the expense of increasing the CPU time. Hence, for any system that could potentially benefit from Smart SSDs, should be bottlenecked on the I/O time (if executed on regular SSDs). Otherwise, Smart SSDs could not reduce the query latency.

However, for Lucene system running on regular SSD, the I/O time is still a bottleneck, see Figure 4 (refer to Section 7 for more experimental settings). We see that, the I/O time is 54.8 ms while the CPU time is 8 ms. Thus, offloading queries to Smart SSDs can reduce the I/O time significantly. In Section 7 (Figure 6), we show that, for the same query on Smart SSDs, if we offload step S3 and S4 to Smart SSDs, the I/O time can be reduce to 14.5 ms while the CPU time is increased to 16.6 ms. Thus, Smart SSDs can reduce the total time by a factor of 2.

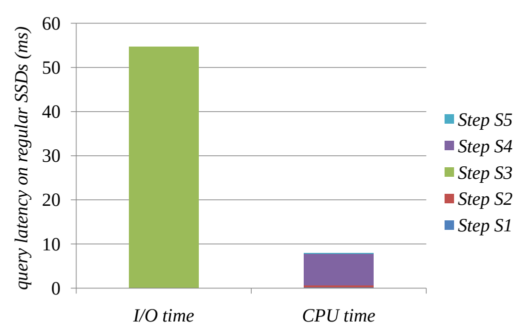


Figure 4: Time breakdown of executing a particular real query by Lucene system running on regular SSDs

Understanding this, we next analyze what query processing steps (namely, step S1 – S5 in Figure 2) for Lucene running on regular SSDs, could be cost-effectively executed inside SSDs. We do a rough analysis first, and evaluate them thoroughly by experiments.

Step S1: Parse query. Parsing a query involves a number of CPU-intensive steps, e.g., tokenization, stemming and lemmatization [26]. Thus, it is not cost-effectively to offload step S1 (or ***the*** step S1?) to Smart SSDs.

Step S2: Get metadata. The metadata is essentially a key-value pair. The key is a query term, the value is the basic information about the on-disk inverted list of the term. In Lucene, it contains (1) the offset where the list is stored on disk, (2) the length (in bytes) of the list, and (3) the number of entries in the list. The metadata is stored in a dictionary file (`.tis` file). There is a Btree-like data structure (`.tii` file) built for the dictionary file. Thus, it takes very few (usually $1 \sim 2$) I/O operations to obtain the metadata [26]. Thus, we do not offload this step.

Step S3: Get inverted lists. Each inverted list contains a list of documents that contain the same term. By default, Lucene stores all the inverted lists are stored on disk. Thus, upon receiving a query, it needs to read the inverted lists from disk to host memory, which is I/O-intensive. As is shown in Figure 4, the step S3 takes 87% of the time if running Lucene on regular SSDs. Thus, we can offload this step to Smart SSDs.

Step S4: Execute list operations. The goal of loading inverted lists to the host memory is for executing list operations, e.g., intersection. Thus, step S4 and S3 should be paired to be offloaded. Then, the question is to determine what operation(s) could potentially benefit from Smart SSD. In Lucene, there are three basic operations that are commonly used: list intersection, union and difference. They are also frequently used in many commercial search engines (e.g., Google advanced search[3]). We investigate each operation next, a simple principle is the output size should be smaller than the input size; otherwise, Smart SSDs cannot save any data movement. Let A and B be two inverted lists (assuming A is shorter than B to capture the real case of skewed lists).

- For the intersection, the intersection result is usually much smaller compared to each inverted list, i.e., $|A \cap B| \ll |A| + |B|$. E.g., in Bing search, for 76% of the queries, the intersection result size is two orders of magnitude smaller than the shortest inverted list involved [13]. Similar results are observed in our real datasets. Thus executing intersection inside SSDs may be a cost-effective choice, as it can save a lot of host I/O interface bandwidth.

- For the union, the union result size can be similar to the total size of the inverted lists. That is because, $|A \cup B| = |A| + |B| - |A \cap B|$, while typically, $|A \cap B| \ll |A| + |B|$, then $|A \cup B| \approx |A| + |B|$. Unless $|A \cap B|$ is similar to $|A| + |B|$. An extreme case is $A = B$, then $|A \cup B| = |A| = |B|$, meaning that, we can save 50% of data transfer. However, in general, it is not cost-effective to offload union to Smart SSDs.

- For the difference, it is to find all the documents in one list but not in the other list. As the operation is ordering-sensitive, we consider two cases: $(A - B)$ and $(B - A)$. For the former case, $|A - B| = |A| - |A \cap B| < |A| \ll |A| + |B|$, i.e., sending the results of $(A - B)$ saves a lot of data transfer if executed in Smart SSDs. However, the other case may not save much data transfer because $|B - A| = |B| - |A \cap B| \approx |B| \approx |B| + |A|$. In summary, we still consider the difference as a possible operation for query offloading.

Step S5: Compute similarity and rank. After the list operation (e.g., intersection) is finished, we get a list of qualified documents. Lucene assumes the results are small enough to fit into main memory. This step is to apply a ranking model to determine the similarities of the query and these qualified documents, since users are interested in the most relevant documents. This step is CPU-intensive, following the analysis, it may not be cost-effective to offload to Smart SSDs. However, it is beneficial when the result size is too big. Because after step S5, only the top ranked results are returned, which can save a lot of I/Os. From the design point of view, we consider two options: (1) do not offload step S5, in this case, step S5 is executed at the host side; (2) offload this step, in this case, step S5 is executed by Smart SSDs.

	Non-Ranked	Ranked
Intersection	✓	✓
Union	✗	✓
Difference	✓	✓

Table 1: Design space

In summary, we consider offloading five query operations that could potentially benefit from Smart SSD: intersection, ranked intersection, ranked union, difference, and ranked difference, see Table 1. The offloading of non-ranked operations means that step S3 and S4 will be executed inside SSDs while step S5 is executed at the host side. The offloading of ranked operations means that step S3, S4 and S5 will be executed inside SSDs. In either case, step S1 and S2 are executed at the host side.

4.2 System Architecture

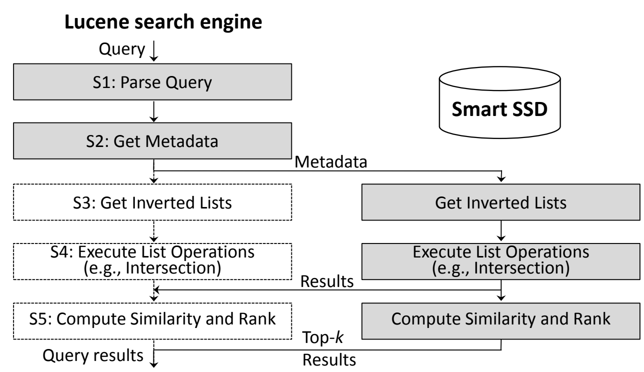


Figure 5: Co-design architecture of Lucene and Smart SSDs

Figure 5 shows the co-design architecture of Lucene search engine and Smart SSDs. We modified Lucene code following the architecture such that Lucene can interact with Smart SSDs.

It works as follows. Assume only the intersection operation is offloaded. The host Lucene is responsible for receiving users queries. Upon receiving a query $q(t_1, t_2, ..., t_u)$, where t_i is a query term. Lucene parses the query q to u query terms (Step S1), then gets the metadata for each query term t_i (Step S2). Then, it sends all the metadata information to Smart SSDs via the OPEN API. The Smart SSD then starts to load the u inverted lists to the device memory according to the metadata. The DRAM is generally of several hundred MBs, which is big enough to store the inverted lists of a typical query. When all the u inverted lists are loaded to DRAM, the Smart SSD executes list intersection. Once it is done, the results will be put to an output buffer. The host Lucene can poll the status of Smart SSDs in a heart-beat manner via the GET API. We set the polling interval to be 1 ms. When the host Lucene receives the intersection results, it will execute step S5 to finish the query, and return the top ranked results to end users. However, if

[3] `http://www.google.com/advanced_search`

the ranked operation is offloaded, Smart SSDs will also take care of step S5.

5. IMPLEMENTATION

In this section, we provide the implementation details of offloading the query operations into Smart SSDs. We discuss the implementation of intersection in Section 5.1, union in Section 5.2 and difference in Section 5.3. Finally, we discuss the implementation of ranking in Section 5.4.

Inverted index format. Inverted index is a fundamental data structure in Lucene (and any search engine). It is essentially a mapping data structure between query terms to inverted lists. Each inverted list contains a list of documents (document IDs) that contain the query term. In Lucene, each entry in the inverted list contains more than just the document ID. It also contains document frequency, and positional information (variable-length) to support ranking and more complex queries.

In this work, we did some changes, which are applied to Lucene running on both regular SSDs and Smart SSDs. (1) Instead of storing the document frequency in Lucene, we store the actual score according to Lucene's ranking model. We explain more in Section 5.4. This will improve the ranking performance for Lucene on both regular SSDs and Smart SSDs. (2) Instead of storing positional information as variable-length entries in Lucene, we store it as fixed-length entries. Each entry takes 16 bytes (i.e., four integers, we choose it based on our statistics. The fixed-length entry allows us to use binary search for skipping, otherwise we need to build some auxiliary data structure, e.g., skip list [29] in Lucene. We believe this change will not affect the system performance much, as both data structures support element search in logarithmic cost. (3) Instead of compressing the inverted index in Lucene, we consider the non-compressed inverted index. Thus, all the entries in every inverted list are sorted by document ID in ascending order. Although compressed lists save space, decompression also takes considerable time, especially for Smart SSDs due to hardware limitations. Note that, we are improving the architecture of Smart SSDs (both hardware and software), so, we leave the compressed list operations to the future work. (4) Besides that, every inverted list is stored on SSD in a page-aligned manner (page size 8KB). That is, it starts and ends at an integer multiple of page sizes. E.g., if the size of an inverted list is 2000 bytes, then, the start offset can be 0 and the end offset is 8KB. The constraint is limited to the programming model of Smart SSDs, because generally, there is no OS support inside SSDs. We note that, this is true even on host machines in order to bypass OS buffer (e.g., `O_DIRECT` flag).

5.1 Intersection

Suppose there are u ($u > 1$) inverted lists, i.e., $L_1, \cdots, L_u$, for intersection. Initially, there are stored on SSD flash chips. Thus, we need to load them to device memory first. Then apply an in-memory list intersection algorithm.

There are a number of in-memory list intersection algorithms, e.g., sort-merge based algorithm [17], skip list based algorithm [26], hash based algorithm, divide and conquer based algorithm [4], adaptive algorithm [11, 12], group based algorithm [13]. Among them, we implement the adaptive algorithm inside Smart SSDs, for the following reasons: (1) Lucene uses the adaptive algorithm for list intersection. For a fair comparison, we also implement it inside Smart SSDs. (2) The adaptive algorithm works well in theory and practice. E.g., according to a recent study in [13], the adaptive algorithm performs very well (than other algorithms except the group based algorithm). Although group based algorithm [13] performs better, it needs too much memory for pre-computation. However, inside Smart SSDs, the memory size is limited.

Algorithm 1: Adaptive intersection algorithm

1 load all the u lists $L_1, L_2, \cdots, L_u$ from the SSD to device memory (assume $L_1[0] \leq L_2[0] \leq ... \leq L_u[0]$)
2 result set $R \leftarrow \varnothing$
3 set $pivot$ to the first element of L_u
4 **repeat** access the lists in cycle:
5 let L_i be the current list
6 *successor* $\leftarrow L_i.\texttt{next}(pivot)$ /*smallest element $\geq$ *pivot**/
7 **if** $successor = pivot$ **then**
8 increase occurrence counter and insert $pivot$ to R if the count reaches u
9 **else**
10 $pivot \leftarrow successor$
11 **until** *target* = INVALID;
12 **return** R

Algorithm 1 shows the adaptive algorithm [11, 12]. Every time, a *pivot* value is selected (initially, it is set to the first element of L_u, see Line 3). It is probed against the other lists in a round-robin fashion. Let L_i be the current list where the pivot is probed on (line 5). If *pivot* is in L_i (using binary search, line 6), increase the counter for the pivot (line 8); otherwise, update the pivot value to be the successor (line 10). In either case, continue probing the next available list until the pivot is INVALID (meaning that at least one list is exhausted).

Switch between the adaptive algorithm and the sort-merge algorithm. The performance of Algorithm 1 depends on how to find the successor of a pivot efficiently (Line 6). We mainly use binary search in our implementation. However, when two lists are of similar sizes, linear search can even be faster than binary search [13]. Thus, in our implementation, if the size ratio of two lists is less than 4 (empirically determined), we use linear search to find the successor. In this case, the adaptive algorithm is degraded to the sort-merge algorithm. For a fair comparison, we also modified Lucene code to switch between the adaptive algorithm and the sort-merge algorithm if it runs on regular SSDs.

5.2 Union

We implement the standard sort-merge based algorithm (also used in Lucene) for executing the union operation, see Algorithm 2.

Algorithm 2: Merge-based union algorithm

1 load all the u lists $L_1, L_2, \cdots, L_u$ from the SSD to device memory
2 result set $R \leftarrow \varnothing$
3 let p_i be a pointer for every list L_i (initially $p_i \leftarrow 0$)
4 **repeat**
5 let $minID$ be the smallest element among all $L_i[p_i]$
6 advance p_i by 1 if $L_i[p_i] = minID$
7 insert $minID$ to R
8 **until** *all the lists are exhausted*;
9 **return** R

It is interesting to note that, Algorithm 2 scans all the elements of the inverted lists *multiple times*. And more importantly, for every qualified document ID (Line 7), we need $2u$ memory accesses unless some list is finished scanning. That is because, every time, need to compare the $L_i[p_i]$ values (for all i) in order to find the minimum value, i.e., Line 5. Then, scan the $L_i[p_i]$ values *again* to move p_i whose $L_i[p_i]$ equals to the minimum (Line 6). Thus, the total number of memory accesses can be estimated by: $2u \cdot |L_1 \cup L_2 \cup \cdots \cup L_u|$. E.g., let $u = 2$, L_1 = {10, 20, 30, 40, 50}, L_2 = {10, 21, 31, 41, 51}. For the first result 10, we need to compare 4 times. Similarly for the rest. Thus, the performance depends on the number of lists and the result size. On average, each element in the result set is scanned $2u$ times, and in practice, $|L_1 \cup L_2 \cup \cdots \cup L_u| \approx \sum_i |L_i|$, meaning that, approximately, every list has to be accessed $2u$ times.

5.3 Difference

The difference operation is applicable for two lists, e.g., list *A* and *B*. Then $A - B$ finds all elements in *A* but not in *B*. The algorithm is trivial: for each element $e \in A$, check whether e is in B. If yes, discard; otherwise, insert e to the result set. Continue until A is exhausted. This algorithm is also used in Lucene system.

In our implementation, for element checking, mainly use binary search. However, if the size ratio between two lists is less than 4, we switch to linear search (same as Line 6 in Algorithm 1).

5.4 Ranked Operations

The above list operations (e.g., intersection) can return many results. However, end users are interested in the most relevant results, which requires two steps. (1) Similarity computation: for each qualified document d in the result set, compute the similarity (or score) between q and d, according to a ranking function; (2) Ranking: find the top ranked documents with the highest scores. The straightforward computation takes too much CPU resources. Thus, we need a careful implementation inside Smart SSDs.

Lucene implements a variant of BM25 query model [31, 32] to determine the similarity between a query and a document.
Let,

qtf :	term's frequency in query q (typically 1)
tf :	term's frequency in document d
N :	total number of documents
df :	number of documents that contain the term
dl :	document length

Then,

$$Similarity(q,d) = \sum_{t \in q} (Similarity(t,d) \times qtf)$$

$$Similarity(t,d) = tf \cdot (1 + \ln \frac{N}{df+1})^2 \cdot (\frac{1}{dl})^2$$

Typically, each entry in the inverted list contains document frequency (in addition to document ID and positional information). Upon a qualified result ID is returned, the score is computed using the above equations. However, all parameters in $Similarity(t,d)$ are not query-specific, which can be *pre-computed*. Thus, in our implementation, instead of storing the actual document frequency (i.e., df), we store the score, i.e., $Similarity(t,d)$. This is important inside Smart SSDs (due to the limited processor speed). Thus, similarity computation becomes much more efficient. This also means, the use of Smart SSDs will not degrade query quality. For a fair comparison, we also modified Lucene code if it runs on regular SSDs.

The remaining question is how to efficiently find the top ranked results. We maintain the top ranked results explicitly at SRAM (instead of DRAM), in a heap-like data structure. Then scan all the similarities to update the results in SRAM if necessary.

6. EXPERIMENTAL SETUP

In this section, we present the experimental setup in our platform. We show the datasets in Section 6.1 and hardware/software setup in Section 6.2.

6.1 Datasets

Parameters	Ranges
Number of lists	**2**, 3, 4, 5, 6, 7, 8
List size skewness factor	10000, 1000, **100**, 10, 1
Intersection ratio	0.1%, **1%**, 10%, 100%
List size	1 MB, **10 MB**, 50 MB, 100 MB

Table 2: Parameter setup

To evaluate the system performance, we use both real dataset and synthetic dataset.

6.1.1 Real Dataset

The real dataset (provided by a commercial large-scale search engine company) consists of two parts: web data and query log. The web data contains more than 10 million web documents. The query log contains around 1 million real queries[4].

6.1.2 Synthetic Dataset

The synthetic dataset allows us to vary many performance-critical parameters to understand the cases that Smart SSD wins. We explain the parameters and the methodology to generate data. Unless otherwise stated, when varying one parameter, we fix all the rest parameters as defaults.

Number of lists. By default, we evaluate the list operations with two inverted lists: list *A* and list *B*. To capture the real case that the list sizes are skewed, i.e., one list is longer than the other, we use list A to indicate the shorter list while list B the longer one in this paper unless otherwise stated. When varying the number of lists according to a parameter m ($m > 1$), we generate m lists independently. Among them, half of the lists ($\lceil m/2 \rceil$) are of the same size with list A (i.e., shorter lists), the other half ($\lfloor m/2 \rfloor$) are of the same size with list B (i.e., longer lists). We vary the number of lists from 2 to 8.

List size skewness factor. The *skewness factor* is defined as the ratio between the size of the longer list and the shorter list, i.e., $\frac{|B|}{|A|}$. In practice, the sizes of different lists differ a lot, because some query terms can be much more popular than the others. We set the skewness factor to be 100 by default and vary the skewness factor from 1 to 10,000 to capture the real case[5].

Intersection ratio. The intersection ratio is defined as the intersection size over the shorter list, i.e., $\frac{|A \cap B|}{|A|}$ for two lists A and B. By default, we set it to 1% to reflect the real scenario. E.g., in Bing search, for 76% of the queries, the intersection size is two orders of magnitude smaller than the shortest inverted list [13]. We vary the intersection ratio from 1% to 100%.

List size. Unless otherwise stated, the list size means the size of the *longer* list, i.e., list *B*. By default, we set the size of list B to be 10 MB, and vary from 1 MB to 100 MB. In real search engines, although the entire inverted index is huge, there are also a huge number of terms, on average, each inverted list is not very long (typically, 10s of MBs). The size of list A can be obtained with the skewness factor. When the list size is determined, we randomly generate a list of entries (each includes the document ID, score and positions) from a universe [13].

Table 2 shows a summary of the key parameters, with defaults highlighted in bold.

[4]The data source, as well as more detailed statistics, is omitted for double-blind review purpose.

[5]We randomly pick up 10,000 queries from the real query log and run them on the real web data. The average skewness factor is 3672. Even if we remove the top 20% highest ones, it is still 75.

	Query latency (ms)	Energy (mJ)
Smart SSD	97	204
Regular SSD	210	1365

Table 3: Intersection on real data

6.2 Hardware and Software Setup

In our experiments, the host machine is a commodity server with Intel i7 processor (3.40 GHz) and 8 GB memory, running Windows 7. The Smart SSD is of 400 GB (SLC), connected to the host machine via a host bus adaptor (HBA) with 6Gbps. The regular SSD is the same SSD but without any query offloading.

We use the C++ version (instead of Java version) of Lucene[6] to be compatible with programming inside Smart SSDs. We choose the stable 0.9.21 version.

We measure the power consumption via *WattsUp*[7] as follows. Let W_1 and W_2 be the power (in Watts) when the system is idle and running, and t be the query latency, the energy is calculated by $(W_2 - W_1) \times t$.

7. EXPERIMENTAL RESULTS

In this section, we present the results of offloading different query operations to Smart SSD: intersection (Section 7.1), ranked intersection (Section 7.2), difference (Section 7.3), ranked difference (Section 7.4), and ranked union (Section 7.5).

We compare two approaches: (1) *Smart SSD*: run our integrated Lucene with Smart SSD; (2) *Regular SSD*: run the original Lucene on a regular SSD. We measure the average query latency[8] and energy consumption.

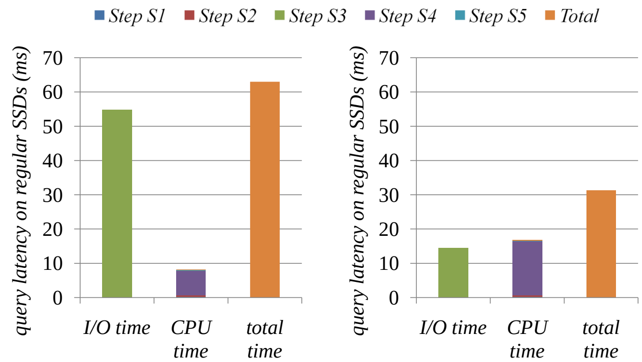
7.1 Intersection

In this case, we offload the intersection to Smart SSD, that is, step S3 and S4 are executed inside SSD.

Results on real data. Table 3 shows the averaged query latency and energy consumed by a reply of the real queries on the real web data. It clearly shows that, compared to regular SSD, Smart SSD can reduce query latency by 2.2× and reduce energy by 6.7×. The query latency saving comes from the high internal bandwidth and low I/O latency of Smart SSD. And the energy saving comes from less data movement and the power-efficient processors running inside SSD.

To understand more, we picked up a particular query with two query terms. Let A and B be the inverted lists of the two terms, where the number of entries in A and B are 2,938 and 65,057, respectively. Figure 6 shows the time breakdown of running Lucene on the regular SSD and the Smart SSD. It shows that, Smart SSDs can reduce the I/O time from 54.8 ms to 14.5 ms, i.e., a factor of 3.8, which is the speedup upper bound that Smart SSDs can achieve. However, Smart SSDs also increase the CPU time from 8 ms to 16.6 ms. In total, Smart SSDs can improve around 2× speedup in query latency.

Then, we only consider the time breakdown for the query processing inside Smart SSDs. That is, step S3 and S4 (since other steps are executed as the host side and the time is negligible from Figure 6), see the results in Figure 7. It shows that, loading inverted lists from flash chips to the device DRAM is still a bottleneck,



(a) Regular SSD (b) Smart SSD

Figure 6: Query latency breakdown of Lucene running on the regular SSD and the Smart SSD, for a particular real query.

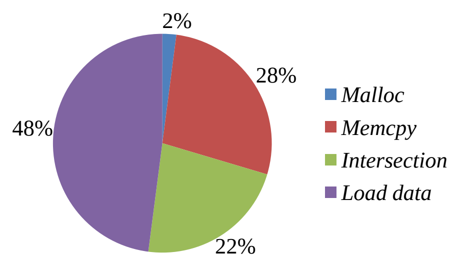
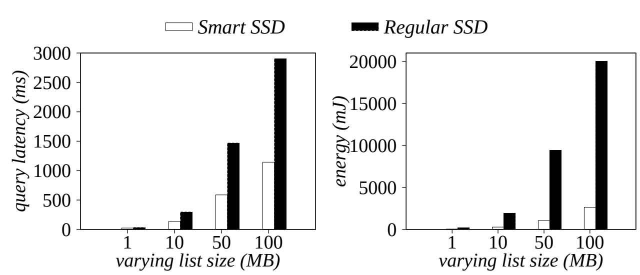


Figure 7: Time breakdown of executing list intersection on Smart SSDs

which can be alleviated by increasing the internal bandwidth. The next bottleneck is memory access (28%), which can be mitigated by reducing memory access cost, e.g., using DMA copy or more caches. Then the bottleneck is processor speed (22%), which can be reduced by introducing more powerful processors. But, balance over bus architecture, memory technology, and CPU architecture for SSD system is also important.

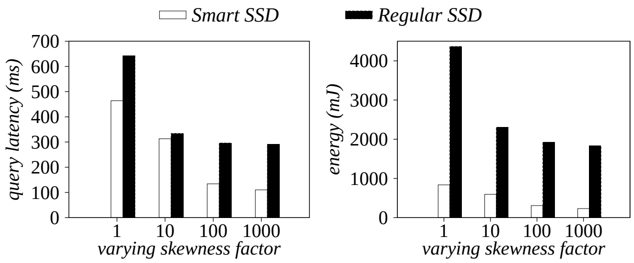
Effect of varying list size. Figure 8 evaluates the effect of list size, which affects the I/O time. Longer lists means more I/O time, which Smart SSD can reduce. We vary the size of list B from 1 MB to 100 MB (while the size of list A depends on the skewness factor, which is 100 by default). The query latency and energy increase with longer lists because of more I/Os. On average, Smart SSD reduces query latency by a factor of 2.5 while reduces energy by a factor of 7.8 compared to regular SSD. In short, the message delivered from Figure 8 is: *Smart SSD favors longer lists for the intersection operation*.



(a) query latency (b) energy

Figure 8: Varying the list size (for intersection)

[6] http://clucene.sourceforge.net
[7] https://www.wattsupmeters.com
[8] Due to the current limitation of the Smart SSD, which can only support one concurrent request each time. Thus, we measure the query latency instead of the query throughput. We will fix the issue in the next-generation of Smart SSDs.



(a) query latency (b) energy

Figure 9: Varying the list size skewness factor (for intersection)

f	n_1	n_2	n_p	estimated ANMP	real ANMP
1	327680	327680	2564	$\frac{n_1+n_2}{n_p}=256$	383
10	32768	327680	1408	$\frac{n_1\cdot\log n_2}{n_p}=427$	640
100	3277	327680	1284	$\frac{n_1\cdot\log n_2}{n_p}=47$	68
1000	328	327680	1280	$\frac{n_1\cdot\log n_2}{n_p}=5$	6

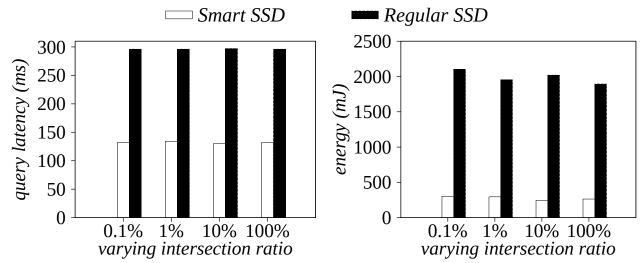
Table 4: The average number of memory accesses per page (ANMP) in Figure 9, where f means the skewness factor, n_1 and n_2 indicate the number of entries in list A and B, n_p is the total number of pages. Note that, each page is 8 KB and each entry takes 32 bytes. Thus, each page contains 256 entries.

Effect of varying list size skewness factor. Figure 9 shows the impact of list size skewness factor f, which can affect the adaptive intersection algorithm. Higher skewness gives more chance for skipping data, incurring less memory access, which is suitable for Smart SSD (since memory access is expensive inside ISC devices). We vary different skewness factors from 1 to 1000 (while fixed the size of list B to be 10 MB). The query latency (as well as energy) goes down when f becomes higher, because the size of list A is becoming smaller. But in any case, Smart SSD outperforms regular SSD significantly in both latency and energy. In short, Figure 9 delivers a message: *Smart SSD favors lists with a higher skewness factor for the* intersection *operation*.

Besides the superiority of Smart SSD shown in Figure 9, it is also interesting to see the latency speedup of Smart SSD over regular SSD achieves the smallest when $f = 10$. That is because the average number of memory accesses per page (ANMP) is larger than all the other cases when $f = 10$, see Table 4. Note that, when $f = 1$, the sort-merge algorithm is used while for all the other cases, the adaptive intersection algorithm is used (as explained in Section 5.1).

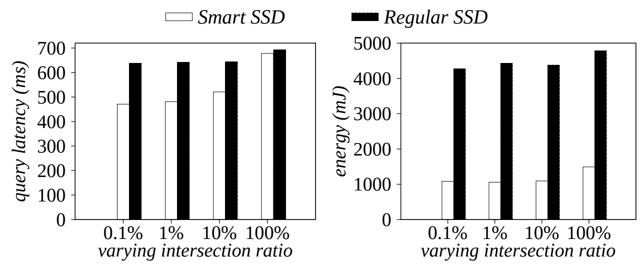
Effect of varying intersection ratio. Figure 10 shows the impact of intersection ratio r. It determines the result size, which can impact system performance in two aspects: (1) data movement via the host I/O interface; and (2) ranking cost at the host side (since all the qualified document IDs in the result set will be evaluated for ranking). Surprisingly, Figure 10 does not show a clear impact of intersection ratio. That is because, by default, list A includes around 3277 entries (0.1 MB) while list B includes around 327680 entries (10 MB). Even when r is 100%, the result size is 3277, which does not make much difference in both I/O time (around 1.4 ms) and ranking cost (around 0.5 ms).

Then, we do experiments again by setting the size of list A to be the same as B (both are of 10 MB), see Figure 11. We see a clear impact of intersection ratio r. The query latency (as well as energy) increases when r is higher, especially when r goes from 10% to 100% (the corresponding result size jumps from 32,768 to 327,680). For regular SSD, the increase is because of more ranking cost. For Smart SSD, the increase is because of both data transfer time and ranking cost. But in all cases, Smart SSD outperforms regular SSD. It is also interesting to see in Figure 11, even when r is 100%, Smart SSD can still win a bit over regular SSD. That is because, in this case, Smart SSD only needs to transfer one list, saving around 50% of data transfer. In short, Figure 11 delivers a message: *Smart SSD favors lists with a smaller intersection ratio for the* intersection *operation*.



(a) query latency (b) energy

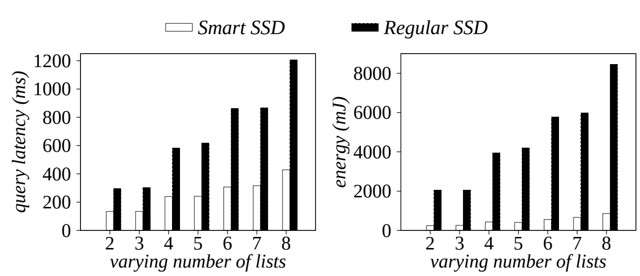
Figure 10: Varying intersection ratio (for intersection)



(a) query latency (b) energy

Figure 11: Varying intersection ratio on equal-sized lists (for intersection)

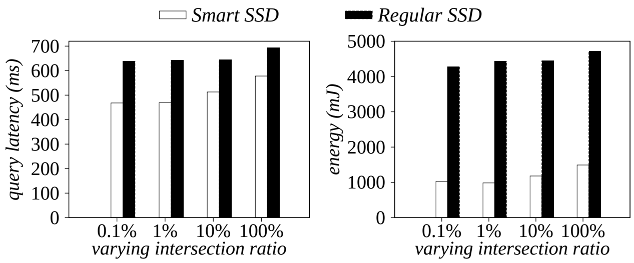
Effect of varying number of lists. Figure 12 shows the results on the effect of varying the number of lists (i.e., number of terms in a query). More lists means more I/O time, which Smart SSD can reduce. We vary the number of lists from 2 to 8. The query latency (as well as energy) goes up with higher number of lists[9], because of more data transfer. On average, Smart SSD reduces query latency by 2.6× and energy by 9.5×. In short, Figure 12 delivers a message: *Smart SSD favors more lists for the* intersection *operation*.



(a) query latency (b) energy

Figure 12: Varying the number of lists (for intersection)

[9]When the number of lists u goes from 2 to 3, the latency does not increase much. That is because, the generated lists are $\{A, B\}$ and $\{A, B, A\}$ when u is 2 and 3, respectively. Since list A is 100× smaller than list B, thus, it does not incur much overhead in query latency.



(a) query latency (b) energy

Figure 13: Varying intersection ratio on equal-sized lists (for ranked intersection)

Remark. The intersection operation can be cost-effectively offloaded to Smart SSD, especially when the number of lists is high, the lists are long, the list sizes are skewed and the intersection ratio is low.

7.2 Ranked Intersection

In this case, we offload the ranked intersection (i.e., step S3, S4 and S5 in Figure 5) to Smart SSD. Compared to the offloading of intersection-only operation (Section 7.1), offloading ranked intersection can (1) save data transfer (as only the top ranked results are returned); but (2) increase the cost of ranking inside device. However, there is not much difference when the result size is small, e.g., less than 30,000 entries. As a reference, sending back 30,000 entries from Smart SSD to host takes around 12 ms, and ranking 30,000 entries at host side takes around 5 ms (Figure 11).

Results on real data. The results are similar to the non-ranked version(see Table 3), we omit for space constraints. That is because the average intersection result size is 1144, which will not make a significant difference (less than 1 ms).

Effect of varying list size. The results of varying list size are similar to non-ranked intersection, i.e., Figure 8. That is because the intersection size is small. E.g., the maximum intersection size is 359 (when the list size is 100 MB) and the minimum intersection size is 3 (when the list size is 1 MB).

Effect of varying list size skewness factor. The results are similar to the non-ranked version, i.e., Figure 9, because the intersection size is not that large. E.g., the maximum intersection size is 3877 (when the skewness factor is 1). Again, Smart SSD wins.

Effect of varying intersection ratio. The results of the default case (i.e., list A is 100× smaller than list B) is similar to Figure 10, where Smart SSD outperforms regular SSD significantly.

Next, we show the results by setting the size of list A to be the same as list B (both are 10 MB), see Figure 13. High intersection ratio means high intersection result size, which adds more overhead for ranking. We vary the intersection ratio from 0.1% too 100%. The query latency (as well as energy) goes up with higher intersection ratio. However, it is interesting to see the increase is not that high compared to the results in Figure 11 for non-ranked intersection offloading. E.g., for Smart SSD, when the intersection ratio goes from 10% to 100%, the latency increases by 65 ms, while the corresponding increase in Figure 11 is 163 ms. The increase does not go that much is because of no extra data movement (only top-k results are returned), i.e., the increase is owing to the extra ranking cost.

It is also interesting to see from Figure 13 that, when the intersection ratio is 100%, Smart SSD wins much more than the non-ranked version (Figure 11). That is because of much less data movement as only top-k results are returned.

	Query latency (ms)	Energy (mJ)
Smart SSD	78	148
Regular SSD	194	1261

Table 5: Difference on real data

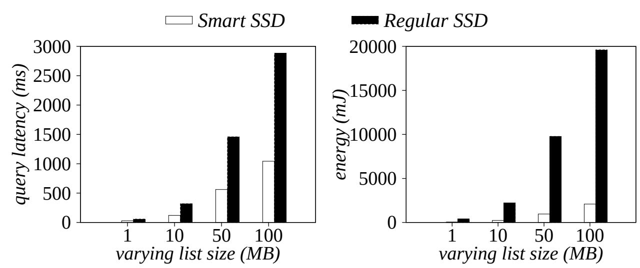
Effect of varying number of lists. The results are similar to the non-ranked version shown in Figure 12 because the intersection size is very small (only 47), which will not make a major difference.

7.3 Difference

In this case, we offload the difference operation (i.e., step S3 and S4 in Figure 5) to Smart SSD, the ranking is done at the host side. The difference operator is applied to two lists, it can be $(A - B)$ or $(B - A)$, where the list A is shorter than list B. As discussed in Section 4.1, only the former case can be potentially benefit from Smart SSD.

Results on real data. Table 5 shows the results by a replay of the real queries on the real web data. For each query, we consider the $(A - B)$ case, where A and B indicate the shortest and longest list in a query. It clearly shows that, compared to regular SSD, Smart SSD can reduce the query latency by 2.5× and energy by 8.5×.

Effect of varying list size. Figure 14 plots the effect of varying list sizes, which affects the I/O time. We vary the list sizes of list B from 1 MB to 100 MB (while the size of list A depends on the skewness factor). The query latency (as well as energy) goes up with longer lists. On average, Smart SSD reduces query latency by 2.7×, and energy by 9.7×. In short, Figure 14 delivers a message: *Smart SSD favors longer lists for the difference operation*.

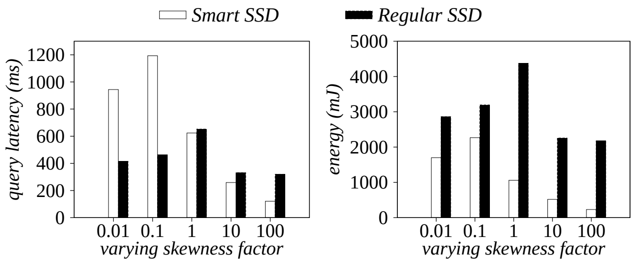


(a) query latency (b) energy

Figure 14: Varying the list size (for difference)

Effect of varying list size skewness factor. The skewness factor f is a key parameter in difference operation. Let A and B be two inverted lists, unlike the default case, A is not necessarily shorter than B. It depends on the skewness factor f (still defined as $|B|/|A|$). We vary f from 0.01 to 100 (Table 6 shows the corresponding sizes of list A and B). When $f < 1$, A is longer than B. We consider the operation $(A - B)$. Figure 15 plots the effect of skewness factor f.

There are several interesting results. (1) Compared to regular SSD, Smart SSD loses in query latency when the skewness factor $f = 0.01$ and $f = 0.1$. That is because, in these two cases, $|A| > |B|$ (see Table 6). Considering the $|A \cap B|$ is very small, thus, the result size of $(A - B)$ is similar to $|A| + |B|$. E.g., when $f = 0.01$, $\frac{|A-B|}{|A|+|B|} = 98.6\%$. So, it is not cost-effective to offload $(A - B)$ if $|A| > |B|$ because it does not save much data transfer. (2) Smart SSD wins when $f \geq 1$, where $|A| \leq |B|$, that is because, $|A - B| \leq |A| \leq (|A| + |B|)/2$. Meaning that, offloading $(A - B)$ can save at least 50% of the data transfer. (3) For Smart SSD, the query latency increases when f goes from 0.01 to 0.1, but



(a) query latency (b) energy

Figure 15: Varying the list size skewness factor (for difference)

Skewness factor	List A size	List B size	# of DRAM access
0.01	10 MB	0.1 MB	3,512,670
0.1	10 MB	1 MB	4,611,592
1	10 MB	10 MB	982,012
10	1 MB	10 MB	581,830
100	0.1 MB	10 MB	59,478

Table 6: Corresponding list sizes and number of memory accesses with different skewness factors in Figure 15

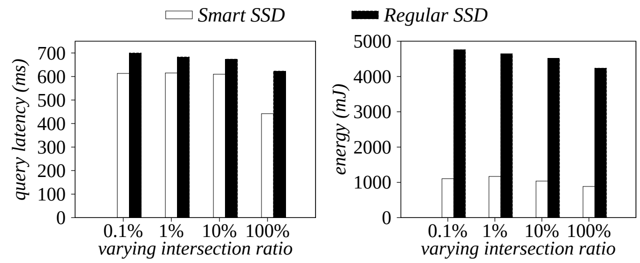
decreases when f is larger. We explain it as follows. Let n_1 and n_2 be the number of entries of list A and B, when $f = 0.01$ or $f = 0.1$ ($n_1 > n_2$), the estimated number of memory accesses is $n_1 \cdot \log n_2$. When f goes from 0.01 to 0.1, n_2 increases (but n_1 stays the same). Thus, it incurs more memory accesses. However, when $f = 1$, the element checking algorithm is switched to linear search (see Section 5.3). Thus, the estimated number of memory accesses is $(n_1 + n_2)$. When $f = 10$ or $f = 100$ ($n_1 < n_2$), the algorithm is switched back to binary search again. However, $n_1 < n_2$ when $f > 1$, thus, introducing even less memory accesses. See Table 6 for the actual number of memory accesses. (4) For regular SSD, it shares the similar trend when f increases. However, the difference is that, when f goes from 0.01 to 1, the latency increases (while decreases for Smart SSD). That is because, for regular SSD, I/O time is a dominant factor. When $f = 1$, both A and B are of 10 MB, which are larger than all the other cases. (5) As a comparison, when $f = 1$, both lists are of size 10 MB. This case is similar to Figure 11(a) when the intersection ratio is 100%. Because both are using sort-merge based algorithm, and can save around 50% of data transfer. Thus, echo the results in 11(a). (6) In terms of energy, Smart SSD always wins, because of power-efficient processors running inside SSD.

In short, Figure 15 delivers a message: *for the difference operation $(A - B)$, Smart SSD can win only if $|A| \leq |B|$.*

Effect of varying intersection ratio. The intersection ratio is also a key parameter to $(A - B)$. It can determine the result size, which can affect the system performance in two aspects: (1) data transfer cost and (2) ranking cost (at host side). Specially, the result size will be smaller for higher intersection ratio. We set the size of list A to be the same as list B in this case[10], see Figure 16. For Smart SSD, the latency (as well as energy) generally goes down with higher intersection ratio, especially from 10% to 100%, because of less data transfer cost and ranking cost. For regular SSD, the decrease is owing to less ranking cost. In short, Figure 16 delivers a message: *Smart SSD favors lists with smaller intersection ratio for the difference operation*.

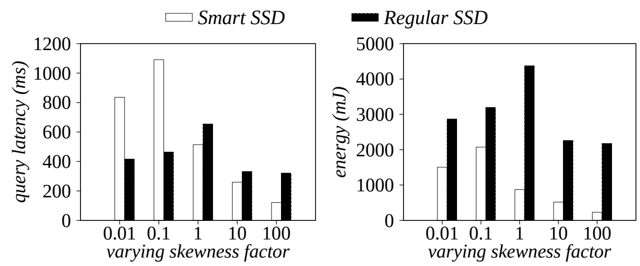
Remark. It is cost-effective to offload the difference operation $(A - B)$ only if $|A| \leq |B|$, and Smart SSD favors lists with lower intersection ratio.

[10] As discussed before, there is no noticeable changes when the size of list A is 1% of list B, thus, we omit the results.



(a) query latency (b) energy

Figure 16: Varying intersection ratio (for difference)



(a) query latency (b) energy

Figure 17: Varying the list size skewness factor (for ranked difference)

7.4 Ranked Difference

In this case, we offload the ranked difference (i.e., step S3, S4 and S5 in Figure 5) to Smart SSD. As discussed before, compared to the non-ranked operation, offloading the ranked operation can reduce data transfer cost, but increase ranking cost. When the result size is large, the saved data transfer time can be more than the extra ranking cost. However, there is no much difference when the result size is small, e.g., less than 30,000 entries.

Results on real data. The results are similar to non-ranked difference (see Table 5), because the result size is small (3109 on average).

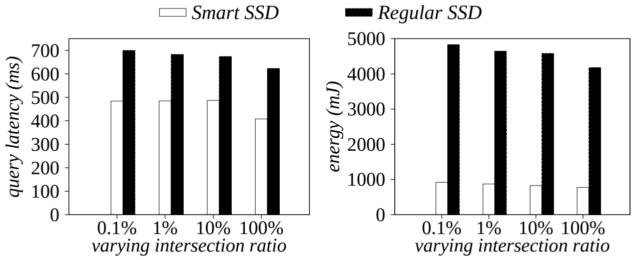
Effect of varying list size. Similar to non-ranked version (see Figure 14), as result size is small (maximum is 32204).

Effect of varying list size skewness factor. The skewness factor determines the result size. For the non-ranked version (see Figure 15), Smart SSD loses when $f = 0.01$ and $f = 0.1$, due to large result size. Thus, with ranking function applied, the result size will be much smaller. Thus, we conjecture Smart SSD will win for all cases.

However, surprisingly, Smart SSD loses when $f = 0.01$ and 0.1. That is because, the ranking function is applied only if all the results are available by the difference operation. However, it needs too many memory accesses to return the results (as analyzed in Section 7.3), regardless of any data transfer. The situation could be changed if we combine both ranking and difference together by a top-k ranking algorithm [16, 6].

It is also interesting to see from Figure 15 when $f = 1$, the speedup of Smart SSD becomes larger (compared to Figure 15) due to less data transfer.

Effect of varying intersection ratio. Figure 18 shows the impact of varying intersection ratio to system performance. It clearly shows the superiority of Smart SSD in terms of both query latency



(a) query latency (b) energy

Figure 18: Varying intersection ratio (for ranked difference)

	Query latency (ms)	Energy (mJ)
Smart SSD	505	960
Regular SSD	299	2033

Table 7: Ranked union on real data

and energy. It is interesting to see the speedup of Smart SSD is larger with different intersection ratios, compared to Figure 16. That is because of less data transfer after ranking.

7.5 Ranked Union

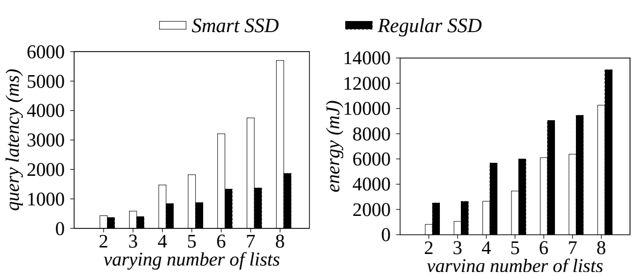
In this case, we offload the ranked union (i.e., step S3, S4 and S5 in Figure 5) to Smart SSD. We first present the results in Section 7.5.1, then discuss more on optimizations in Section 7.5.2.

7.5.1 Results

Results on real data. Table 7 shows the results on real data. It shows that, Smart SSD loses around 1.7× compared to regular SSD, in query latency. That is because of too many memory accesses. As discussed in Section 5.2, every list has to be scanned around $2u$ times, where u is the number of lists in a query. On average, $u = 3.8$ in our query log. However, Smart SSD can reduce energy consumption by 2.1×, because of power-efficient processors running inside SSD.

We omit the results of varying intersection ratios, list size skewness factors and list sizes, for space constrains. The results are, Smart SSD loses around 1.2× in query latency, but reduces around 2.8× energy. Next, we show the effect of varying number of lists, which is a key parameter.

Effect of varying number of lists Figure 19 shows the impact of number of lists u in a query. It shows that, the gap between Smart SSD and regular SSD becomes larger with more number of lists. That is because, approximately, each list has to be accessed $2u$ times.



(a) query latency (b) energy

Figure 19: Varying the number of lists (for ranked union)

7.5.2 Discussion: Call for Algorithmic Optimizations

The reason why it is not cost-effective to offload ranked union is because of too many memory accesses. To improve the performance, on the one hand, we could improve the intrinsic performance of Smart SSD by improving memory access speed. On the other hand, more efficient algorithms could be designed to reduce memory accesses. Our current implementation (following Lucene) solves the ranking problem when all the union results are available, then scores every qualified document. However, union and ranking could be algorithmically combined for early termination [6, 16]. Meaning that, we do not need to scan all the union results. We will explore the early pruning techniques in the future work.

8. RELATED WORK

The idea of offloading computation to storage device (i.e., in-storage computing) has been around for decades. Many research efforts (both hardware and software sides) were dedicated to make it practical.

Early works on in-storage computing. As early as 1970s, some initial works propose to leverage specialized hardware (e.g., processor-per-track and processor-per-head) to improve query processing in storage devices (ie.., hard disks at that time). For example, CASSM [33] and RAP [28] follow the processor-per-track architecture to embed each track a processor. The Ohio State Data Base Computer (DBC) [19] and SURE [23] follow the processor-per-head architecture to associate processing logic with each read/write head of a hard disk. However, none of the systems turned out to be successful due to high design complexity and manufacturing cost.

Later works on HDD in-storage computing. In late 1990s, when the bandwidth of hard disks continue to grow while the cost of powerful processors continue to drop, making it feasible to offload bulk computation to each individual disk. Researchers examine in-storage computing in terms of hard disks, e.g., active disk [1] or intelligent disk [20]. The goal is to offload application-specific query operators inside hard disk, to save data movement. They examine active disk in database area, by offloading several primitive database operators, e.g., selection, group-by, sort. Later on, Erik et. al extends the application to data mining and multimedia area [30]. E.g., frequent sets mining, and edge detection. Although interesting, few real systems adopted the proposals, due to various reasons, including, limited hard disk bandwidth, computing power, and performance gains.

Recent works on SSD in-storage computing. Recently, with the advent of SSD, which is a potential to replace HDD. People start to rethink about in-storage computing in the context of SSD, i.e., Smart SSD. SSD offers many advantages over HDD, e.g., very high internal bandwidth and high computing power (because of the ARM processor techniques). More importantly, executing code inside SSD can save a lot of energy because of less data movement and power-efficient embedded ARM processors. And, energy is becoming very critical today. This, makes the concept of in-storage computing on SSD much more promising this time. Industries like IBM started to install active SSD to the Blue Gene supercomputer to leverage the high internal bandwidth of SSD [18]. In this way, computing power and storage device are closely integrated. Teradata's Extreme Performance Appliance [34] is also an example of combining SSD and database functionalities together. Another example is Oracle's Exadata [27], which also started to offload complex processing into their storage servers.

SSD in-storage computing (or Smart SSD) is also attractive in academia. In database area, Kim et. al investigated pushing down the database scan operator to SSD [21]. That work is based on simulation. Later on, Do et. al [14] built a Smart SSD prototype on real SSDs. They integrated Smart SSD with Microsoft SQL Server to by offloading two operators: scan and aggregation. Woods et. al also built a prototype of Smart SSD with FPGAs [35]. Their target is also for database systems, but with more operators, e.g., group-by. They integrated the prototype with MySQL storage engine such as MyISAM and INNODB. In data mining area, Bae et. al investigated offloading functionalities like k-means and Aprior to Smart SSD [3]. In data analytics area, De et. al propose to push down hash tables inside SSD [10]. There are also some work on offloading sorting [22].

Unlike existing work, our work investigate the potential benefit of Smart SSD on search engine area. To the best of our knowledge, this is the first work in this area.

9. CONCLUSION

To do (write conclusion after everything is fixed)...... What can people learn from your project? for both device side, and software side. What's the bottleneck, how to improve next? Make a summary on findings, conclusive results (e.g., what operations can be offloaded). Also some improvements observed to do on Smart SSD side, e.g., memory copy, DMA copy. Call for a co-design and algorithmic optimizations. We acknowledge these results are preliminary...

Future work. This is the first work of applying Smart SSD to web search engine area. We have a number of interesting future works to do. *Smart SSD for building inverted index*. Building inverted index may be another task that can benefit a lot from Smart SSD. (1) On one hand, it is very I/O-intensive. There are a lot of data transfer, at least twice: scan the whole documents once, build index in memory, write back to SSD. However, it could save a lot of host I/O interface bandwidth if we run the process inside Smart SSD, to leverage the high internal bandwidth and low latency I/O. (2) On the other hand, it does not require much CPU computation, which is also suitable for Smart SSD.

REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, pages 81–91, 1998.

[2] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: where and how? *IEEE Data Eng. Bull.*, 33(4):28–34, 2010.

[3] D. Bae, J. Kim, S. Kim, H. Oh, and C. Park. Intelligent SSD: a turbo for big data mining. In *CIKM*, pages 1573–1576, 2013.

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

[5] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *Micro, IEEE*, 34(4):36–42, 2014.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[7] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *ICDE*, pages 1360–1369, 2012.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, pages 385–395, 2010.

[9] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5-6):332–343, 2009.

[10] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *FCCM*, pages 9–16, 2013.

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

[12] 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.

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

[14] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD*, pages 1221–1230, 2013.

[15] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *SIGMOD*, pages 1113–1124, 2011.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[17] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2008.

[18] Jülich Research Center. Blue gene active storage boosts i/o performance at jsc. `http://cacm.acm.org/news/169841-blue-gene-active-storage-boosts-i-o-performance-at-jsc`, 2013.

[19] K. Kannan. The design of a mass memory for a database computer. In *ISCA*, pages 44–51, 1978.

[20] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, 1998.

[21] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee. Fast, energy efficient scan inside flash memory. In *ADMS*, pages 36–43, 2011.

[22] Y.-S. Lee, L. C. Quero, Y. Lee, J.-S. Kim, and S. Maeng. Accelerating external sorting via on-the-fly data merge in active ssds. In *HotStorage*, 2014.

[23] H. Leilich, G. Stiege, and H. C. Zeidler. A search processor for data base management systems. In *VLDB*, pages 280–287, 1978.

[24] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, 2010.

[25] H. Luu and R. Rangaswamy. How lucene powers the linkedin segmentation and targeting platform. Lucene/SOLR Revolution, 2013.

[26] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[27] Oracle Corporation. Oracle exadata white paper, 2010.

[28] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik. Performance evaluation of a relational associative processor. *ACM Trans. Database Syst.*, 2(2):175–195, 1977.

[29] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[30] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, pages 62–73, 1998.

[31] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, pages 232–241, 1994.

[32] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[33] S. Y. W. Su and G. J. Lipovski. Cassm: A cellular system for very large data bases. In *VLDB*, pages 456–472, 1975.

[34] Teradata Corporation. Teradata extreme performance alliance. `http://www.teradata.com/t/extreme-performance-appliance`.

[35] L. Woods, Z. István, and G. Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11):963–974, 2014.

[36] J. Zobel and A. Moffat. Inverted files for text search engines. *CSUR*, 38, 2006.