System Co-Design of Apache Lucene and SSD In-Storage Computing

Paper ID: 558

ABSTRACT

Recently, there has been 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 high 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 what query processing operations of Lucene can be cost-effectively offloaded to Smart SSDs. To answer this question, we carefully identified five commonly used operations in Lucene (and any search engine) that could potentially benefit from Smart SSDs: intersection, ranked intersection, ranked union, difference, and ranked difference. With the close collaboration of an SSD vendor, we co-designed the host Lucene system and the Smart SSD by offloading the key operations to the Smart SSD. Finally, we conducted extensive experiments to evaluate the system 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 Smart SSDs significantly reduce the query latency by a factor of 2-3× and energy consumption by 6-10× for most of the aforementioned operations.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search Process

General Terms

Design, Performance, Experimentation

Keywords

In-Storage Computing, Smart SSD, Search Engine, Lucene, System Co-Design

1. INTRODUCTION

Solid state drives (SSDs) have gained much momentum in the storage market because of their compelling advantages over hard disk drives (HDDs). For example, SSDs offer one to two orders of magnitude faster random access than HDDs [17]. Moreover, SSDs consume much less energy than HDDs [9]. As a result, large-scale search engines deploy SSDs in their storage systems to achieve lower *query latency* and less *energy consumption*, which are two primary design objectives of search engines. High query latencies can degrade user experience, and hence reduce revenues [32]. High energy consumption gives rise to high Total Cost of Ownership (TCO). As an example, Baidu[1], the largest search engine in China, has already adopted SSDs as their main storage to reduce query latency and energy consumption [26].

A number of research studies have recently discussed how to *make full use* of SSDs in search engines, instead of just using SSDs as yet-another-faster HDDs (e.g., SSD-aware cache management [37, 38] and SSD-aware inverted index maintenance [24]). They share the same goal of optimizing search engine systems, while treating SSDs as storage-*only* devices. In this way, data storage and computing is strictly *separated*: data is stored on SSDs, while computing is executed at host machines. Upon computation, data is transferred from SSDs to the host machines through host I/O interfaces (typically SAS/SATA or PCIe).

However, recent studies indicate that this computing paradigm of "*move data closer to code*" cannot make full use of SSDs. E.g., Baidu's SSD storage system can only use 40% of raw flash's bandwidth [29]. That is because (1) SSDs generally provide higher internal bandwidth than the host I/O interface bandwidth. In our SSD, the internal bandwidth is around 3.2 GB/s, while the host I/O bandwidth is around 750 MB/s. However, data must be transferred to the host via the host I/O interface, resulting in a waste of SSDs' high internal bandwidth; (2) SSDs, in general, provide high computing capabilities (for executing complex FTL firmware code [10]), which are ignored by search engines where SSDs are treated as storage-*only* devices.

This work proposes a system co-design of SSD In-Storage Computing (ISC) and search engines to reduce query latency and energy consumption. The main idea is to offload some key operations (e.g., inverted list intersection) from host search engines to SSD devices to take advantage of their high internal bandwidth, low I/O latency, and computing capabilities. These special SSD devices which can execute application specific programs are also called *Smart SSDs* [14][2]. Smart SSDs changed the traditional computing paradigm to "*move code closer to data*" (or generally near-data

[1] `http://www.baidu.com`

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

processing [4]). In addition to the aforementioned performance improvement, Smart SSDs significantly reduce energy consumption due to less data movement and power-efficient processors. Consequently, the Smart SSD is a very promising solution to *make full use* of modern SSDs in search engines. This Smart SSD is also attractive to industry. E.g., IBM has applied Smart SSDs in their Blue Gene storage systems [19].

The challenge of this work is to determine what query processing operations in search engines can be cost-effectively offloaded to Smart SSDs. A basic principle is that the output sizes of the operations should be smaller than their input sizes such that data movement can be reduced. Otherwise, search engine systems cannot fully benefit from Smart SSDs' capabilities. Based on this principle, we carefully identified five commonly used operations in search engines that could potentially benefit from Smart SSDs: intersection, ranked intersection, ranked union, difference, and ranked difference. Then, we conducted extensive experiments to study the tradeoffs of the query offloading. The experiments show that our co-designed system reduces the query latency by 2-3× and energy consumption by 6-10× for most operations.

We choose Apache Lucene as our exemplary search engine system for the following reasons: (1) Lucene is open-source so that we can study the wholistic system performance (not only algorithms); (2) Lucene implements many (if not all) state-of-the-art query processing techniques, and thus widely adopted in industries (e.g., LinkedIn [25] and Twitter [7]). However, our approaches in this work can be applicable to other search engines.

The main contributions of this work are as follows:

- Implementation in a real SSD: with the collaboration of an SSD vendor, we implemented all the query operations inside real SSD firmware.
- System integration: we integrated Smart SSD with a popular open-source search engine (Apache Lucene) for wholistic system performance study.
- Comprehensive experiments: we conducted extensive experiments on both synthetic dataset and real dataset (provided by a large-scale search engine company) to study the tradeoffs of the query offloading.

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

2. BACKGROUND

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

2.1 Smart SSDs

The Smart SSD ecosystem consists of both hardware (Section 2.1.1) and software components (Section 2.1.2) to execute user-defined programs.

2.1.1 Hardware Architecture

Figure 1 represents the hardware architecture of the Smart SSD. The hardware architecture of Smart SSDs is similar to regular SSDs. In general, an SSD is largely composed of NAND flash memory array, SSD controller, and (device) DRAM. The SSD controller is subdivided into four main subcomponents: host interface controller, embedded processors, DRAM controller, and flash controller.

The host interface controller processes commands from interfaces (typically SAS/SATA or PCIe) and distributes them to the embedded processors. The embedded processors receive the commands and pass them to the flash controller. More importantly, they run SSD firmware code for computation and execute Flash Translation Layer (FTL) for logical-to-physical address mapping [10]. Typically, the processor is a low-powered 32-bit processor such as an ARM series processor. Each processor can have a tightly coupled memory (e.g., SRAM) to store performance-critical data or code. Each processor can access DRAM through the DRAM controller. The flash controller controls data transfer between flash memory and DRAM.

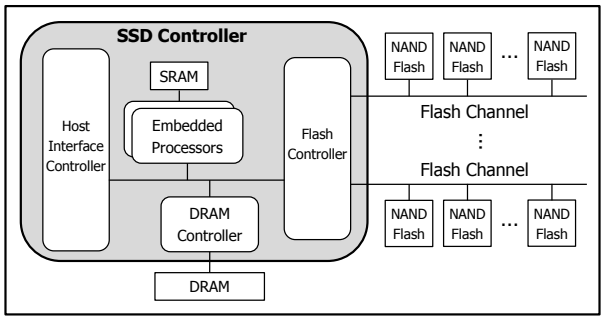


Figure 1: Smart SSD hardware architecture

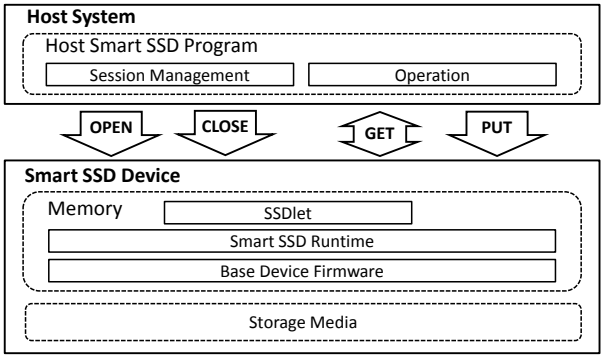


Figure 2: Smart SSD software architecture

The NAND flash memory package is the persistent storage media and each package is subdivided further into smaller units that can independently execute commands or report status.

2.1.2 Software Architecture

In addition to the hardware support, we also need a software mechanism to define a set of protocols such that host machines and Smart SSDs can communicate with each other. Figure 2 describes our Smart SSD software architecture which consists of two main components: *Smart SSD firmware* inside the SSD and *host Smart SSD program* in the host system. The host Smart SSD program communicates with the Smart SSD firmware through application programming interfaces (APIs).

The Smart SSD firmware is subdivided into three subcomponents: SSDlet, Smart SSD runtime, and base device firmware. An SSDlet is a Smart SSD program in the SSD. It implements application logic and responds to a Smart SSD host program. The 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.

This host Smart SSD program consists largely of two components: 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 a 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 with 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 to a host such as interrupts. PUT is used to internally write data to the Smart SSD device without help from local file systems.

2.2 Query Processing of Lucene

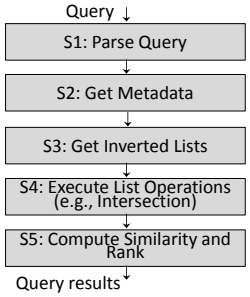


Figure 3: Query processing of Lucene

Like other search engines, Lucene relies on the standard inverted index [40] 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 and the value is a list of documents containing the term.

Upon receiving a user query q (e.g., "SSD database"), Lucene answers it through several steps (S1 to S5 in Figure 3). *Step S1*: Parse the query q into query terms. In our example, there are two query terms: "SSD" and "database". *Step S2*: get metadata for each query term. The metadata is used to load the inverted list of each query term in the next step. Thus, this metadata stores some basic information about the on-disk inverted list. In Lucene, it contains (1) the offset where the list is located on disk, (2) the list length (in bytes), and (3) the number of entries in the list. *Step S3*: for each term, get the inverted list from disk to memory. *Step S4*: execute list operations depending on query modes. 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. The list operation is intersection in our example. It could be other operations such as union (`OR` mode) or difference (`NOT` mode). *Step S5*: for each qualified document d, calculate the similarity between the query q and the document d using an IR relevance model. Lucene adopts a modified BM25 model [31]. Finally, Lucene returns the top ranked results to end users.

We note that Lucene may not embrace all state-of-the-art query processing techniques. For instance, both steps S4 and S5 can be algorithmically combined for early termination [6, 15].

3. SYSTEM CO-DESIGN: SMART SSD FOR LUCENE

This section describes the system co-design of the Smart SSD and Lucene. We first explore the design space in Section 3.1 to determine what query processing logic could be cost-effectively offloaded, and then show the co-design architecture of the Smart SSD and Lucene in Section 3.2.

3.1 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 this, we need to understand the opportunities and limitations of Smart SSDs.

Opportunities of Smart SSDs. Executing I/O operations inside Smart SSDs is very fast for the following two reasons. (1) SSDs generally provide several times higher internal bandwidth than the host I/O interface bandwidth [14, 11]. In our Smart SSD, the internal bandwidth is around 3.2 GB/s, while the host I/O bandwidth is around 750 MB/s. (2) The I/O latency inside Smart SSDs is very low compared to a regular I/O issued from the host system. A regular I/O operation (from flash chips to the host DRAM) needs to go through the entire thick OS stack, which introduces a lot of overheads such as interrupt, context switch, and file system overhead. The OS software overhead becomes a crucial factor in SSDs due to their fast I/O (but it can be negligible in HDDs as their slow I/O is a dominant factor) [8]. However, an I/O operation inside SSDs (from flash chips to the DRAM inside SSD) is free from the OS software overhead. Thus, it is very profitable to execute *I/O-intensive* operations inside SSDs to leverage their high internal bandwidth and low I/O latency.

Limitations of Smart SSDs. Smart SSDs also have some limitations. (1) Generally, Smart SSDs employ low-frequency processors (typically ARM series) to save energy and manufacturing cost. As a result, computing capability is several times lower than host CPUs (e.g., Intel processor) [14, 11]; (2) The Smart SSD also has a DRAM inside. Accessing the device DRAM is slower than the host DRAM because typical SSD controllers do not adopt caches[3] (e.g., L1/L2 caches). Therefore, it is not desirable to execute *CPU-intensive* and *memory-intensive* operations inside SSDs.

Smart SSDs can reduce the I/O time at the expense of the increasing CPU time so that a system with I/O time bottleneck can notably benefit from Smart SSDs. As an example, the Lucene system running on regular SSDs has an I/O time bottleneck, see Figure 4 (please refer to Section 5 for more experimental settings). We observe the I/O time of a typical real-world query is 54.8 ms while its CPU time is 8 ms. Thus, offloading this query to Smart SSDs can significantly reduce the I/O time. Figure 6 in Section 6 supports this claim. If we offload steps S3 and S4 to Smart SSDs, the I/O time reduces to 14.5 ms while the CPU time increases to 16.6 ms. Overall, Smart SSDs can reduce the total time by a factor of 2.

Based on this observation, we next analyze what query processing steps in Lucene (namely, step S1 – S5 in Figure 3) could be executed inside SSDs to reduce both query latency and power consumption. We make a rough analysis first and then evaluate them by experiments thoroughly.

Step S1: Parse query. Parsing a query involves a number of CPU-intensive steps such as tokenization, stemming, and lemmatization [27]. Thus, it is not profitable to offload this step S1.

Step S2: Get metadata. The metadata is essentially a key-value pair. The key is a query term and the value is the basic information

[3]Although there is SRAM inside SSDs, it is not generally used for data caching.

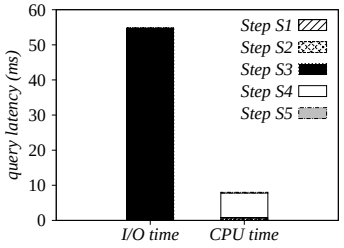


Figure 4: Time breakdown of executing a typical real-world query by Lucene system running on regular SSDs

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. There is a Btree-like data structure built for the dictionary file. Since it takes very few (usually $1 \sim 2$) I/O operations to obtain the metadata [27], we do not offload this step.

Step S3: Get inverted lists. Each inverted list contains a list of documents containing the same term. Since Lucene stores all the inverted lists on a disk and does not adopt caching to save memory[4], upon receiving a query, it reads the inverted lists from the disk to the host memory, which is I/O-intensive. As is shown in Figure 4, the step S3 takes 87% of the time if Lucene runs on regular SSDs. Therefore, it is desirable to offload this step to Smart SSDs.

Step S4: Execute list operations. The main reason of loading inverted lists to the host memory is to efficiently execute list operations such as intersection. Thus, both steps S4 and S3 should be offloaded to Smart SSDs. This raises another question: what operation(s) could potentially benefit from Smart SSDs. In Lucene, there are three basic operations commonly used: list intersection, union, and difference. They are also widely adopted in many commercial search engines (e.g., Google advanced search[5]). We investigate each operation and set up a simple principle that the output size should be smaller than its input size. Otherwise, Smart SSDs cannot save any data movement. Let A and B be two inverted lists, and we assume A is shorter than B to capture the real case of skewed lists.

- Intersection: The result size of the intersection 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 dataset. Thus, executing intersection inside SSDs may be a smart choice as it can save a lot of host I/O interface bandwidth.

- 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.

- Difference: It is used to find all the documents in one list but not in the other list. Since this operation is ordering-sensitive, we consider two cases: $(A - B)$ and $(B - A)$. For the former case, $|A - B| = |A| - |A \cap B| < |A| \ll |A| + |B|$. That is, sending the results of $(A - B)$ saves a lot of data transfer if executed in Smart SSDs. On the other hand, the latter case may not save much data transfer because $|B - A| = |B| - |A \cap B| \approx |B| \approx |B| + |A|$. Consequently, we still consider the difference as a possible candidate for query offloading.

[4]Even though other search engines may cache inverted lists in the host memory, it may not solve the I/O problem. (1) The cache hit ratio is low even for big memories, typically 30% to 60% due to the cache invalidation caused by inverted index update [5]. (2) Big DRAM in the host side consumes too much energy because of the periodic memory refreshment [18].

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

Step S5: Compute similarity and rank. After the aforementioned list operations are completed, we can get a list of qualified documents[6]. This step applies a ranking model to the qualified documents in order to determine the similarities between the query and these documents. This is because users are more interested in the most relevant documents. This step is CPU-intensive so that it may not be a good candidate to offload to Smart SSDs. However, it is beneficial when the result size is very large because after step S5, only the top ranked results are returned. This can save a lot of I/Os. From the design point of view, we can 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 SSDs: intersection, ranked intersection, ranked union, difference, and ranked difference (see Table 1). The offloading of non-ranked operations means that only steps S3 and S4 will be executed inside SSDs while step S5 is executed at the host side. The offloading of ranked operations means that all steps S3, S4, and S5 will be executed inside SSDs. In either case, steps S1 and S2 are executed at the host side.

3.2 System Co-Design Architecture

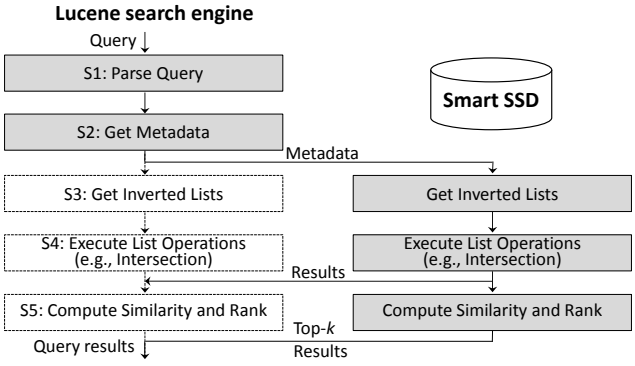


Figure 5: Co-design architecture of Lucene and Smart SSDs

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

It operates as follows. Assume only the intersection operation is offloaded. The host Lucene is responsible for receiving queries. Upon receiving a query $q(t_1, t_2, ..., t_u)$, where each t_i is a query

[6]Lucene assumes the results are small enough to fit into main memory, thus, no any I/O involved.

term. Lucene parses the query q to u query terms (Step S1) and 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 now starts to load the u inverted lists to the device memory (DRAM) using the metadata. The device 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 the device DRAM, the Smart SSD executes list intersection. Once it is done, the results will be put to an output buffer and ready to be returned to the host side. The host Lucene keeps monitoring the status of Smart SSDs in a heart-beat manner via the GET API. We set the polling interval to be 1 ms. Once the host Lucene receives the intersection results, it executes step S5 to complete the query, and returns the top ranked results to end users. If the ranked operation is offloaded, Smart SSDs will also take care of step S5.

4. IMPLEMENTATION

This section describes the implementation details of offloading the query operations into Smart SSDs. We discuss the implementation of intersection in Section 4.1, union in Section 4.2, and difference in Section 4.3. Finally, we discuss the implementation of ranking in Section 4.4.

Inverted index format. Inverted index is a fundamental data structure in Lucene as well as other search engines. It is essentially a mapping data structure between query terms and inverted lists. Each inverted list includes a list of documents (document IDs) containing the query term. In Lucene, each entry in the inverted list consists of a document ID, document frequency, and positional information (variable-length) to support ranking and more complex queries.

In this work, we made the following changes to the inverted index such that Smart SSDs (as well as regulars SSDs) can access it efficiently. (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 4.4. This improves 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 empirical research statistics. The fixed-length entry allows us to use binary search for skipping. Otherwise, we need to build some auxiliary data structures such as skip list in Lucene. We believe this change will not affect the system performance much because both data structures support element search in a 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 an ascending order. Although compressed lists can save some space, decompression also takes considerable time especially for Smart SSDs due to their hardware limitations. Note that this hardware limitation can be overcome by the vendor's hardware architecture improvement in the near future. So, we leave the compressed list operations to our future work. (4) Every inverted list is stored on SSDs in a page-aligned manner (page size 8 KB). That is, it starts and ends at an integer multiple of the page sizes. For example, if the size of an inverted list is 2000 bytes, the start offset can be 0 and the end offset is 8 KB. 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 the host machines in order to bypass OS buffer (e.g., `O_DIRECT` flag).

4.1 Intersection

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

Algorithm 1: Adaptive intersection algorithm

```
1  load all the u lists L_1, L_2, ..., L_u from the SSD to the device
   DRAM (assuming L_1[0] ≤ L_2[0] ≤ ... ≤ L_u[0])
2  result set R ← ∅
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 ← L_i.next(pivot) /*smallest element ≥ pivot*/
7      if successor = pivot then
8          increase occurrence counter and insert pivot to R if the
           counter reaches u
9      else
10         pivot ← successor
11 until pivot = INVALID;
12 return R
```

There are a number of in-memory list intersection algorithms such as sort-merge based algorithm [16], skip list based algorithm [27], hash based algorithm, divide and conquer based algorithm [3], adaptive algorithm [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 adopt it inside Smart SSDs. (2) The adaptive algorithm works well in theory and practice [12, 13].

Algorithm 1 describes the adaptive algorithm [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 a 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 mostly use binary search in our implementation. However, when two lists are of similar sizes, linear search can be even faster than binary search [13]. In our implementation, if the size ratio of two lists is less than 4 (based on our empirical study), we use linear search to find the successor. In this case, the adaptive algorithm is switched 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.

4.2 Union

We implement the standard sort-merge based algorithm (also adopted in Lucene) for executing the union operation, see Algorithm 2. We note that it scans all the elements of the inverted lists *multiple times*. More importantly, for every qualified document ID (Line 7), it needs $2u$ memory accesses unless some lists finish scanning. That is because every time it needs to compare the $L_i[p_i]$ values (for all i) in order to find the minimum value (Line 5). Then, it scans the $L_i[p_i]$ values *again* to move p_i whose $L_i[p_i]$ equals to the minimum value (Line 6). 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.

Algorithm 2: Sort-merge based union algorithm

```
1 load all the u lists L_1, L_2, ..., L_u from the SSD to device memory
2 result set R ← ∅
3 let p_i be a pointer for every list L_i (initially p_i ← 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
```

Approximately, 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. Section 6.5 describes how it affects the system performance.

4.3 Difference

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

Our implementation mostly uses binary search for element checking. However, as explained in Section 4.1, if the size ratio between two lists is less than 4 (empirically determined), we switch to the linear search (same as Line 6 in Algorithm 1).

4.4 Ranked Operations

Lucene (and any search engine) returns the most relevant results to users, 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 consumes too much CPU resources. Therefore, we need a careful implementation inside Smart SSDs.

We review Lucene's ranking function first. Lucene implements a variant of BM25 ranking model [31, 34] 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 a document frequency (in addition to document ID and positional information). Upon a qualified result ID is returned, its score is computed by using the above equations. However, all parameters in $Similarity(t, d)$ are not query-specific, which can be *pre-computed*. 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 to Smart SSDs considering their limited processor speed. Note that the hardware limitations can be overcome by SSD vendors in the near future. The pre-computation is specific to Lucene (and any search engine) using the BM25 ranking model[7]. For a fair comparison, we also modified Lucene code when it runs on regular SSDs.

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

5. EXPERIMENTAL SETUP

This section presents the experimental setup in our platform. We describe the datasets in Section 5.1 and hardware/software setup in Section 5.2.

5.1 Datasets

To evaluate our system performance, we employ both real dataset and synthetic dataset.

5.1.1 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 a collection of more than 10 million web documents. The query log contains around 1 million real queries[8].

5.1.2 Synthetic Dataset

The synthetic dataset allows us to better understand various critical parameters in Smart SSDs. We explain the parameters and the methodology to generate data. Unless otherwise stated, when varying a particular parameter, we fix all the other 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), list A represents 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 to capture most real-world queries.

List size skewness factor. The *skewness factor* is defined as the ratio of the size of the longer list to the that of the shorter list (i.e., $\frac{|B|}{|A|}$). In practice, different lists significantly differ in their sizes because some query terms can be much more popular than the others. We set the skewness factor to 100 by default and vary the skewness factor from 1 to 10,000 to capture the real case[9].

Intersection ratio. The intersection ratio is defined as the intersection size over the size of the shorter list (i.e., $\frac{|A \cap B|}{|A|}$). 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 represents the size of the *longer* list (i.e., list *B*). By default, we set the size of list B to 10 MB, and vary it 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 list takes around 10s of MBs.

[7] For other ranking models, e.g., probabilistic ranking model [27], we may not pre-compute that much.

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

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

Parameters	Ranges
Number of lists	**2**, 3, 4, 5, 6, 7, 8
List size skewness factor	10,000, 1,000, **100**, 10, 1
Intersection ratio	0.1%, **1%**, 10%, 100%
List size	1 MB, **10 MB**, 50 MB, 100 MB

Table 2: Parameter setup

The size of list A can be obtained with the skewness factor. Once the list size is determined, we generate a list of entries randomly.

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

5.2 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 a 400GB SLC SSD with SAS 6Gb interface. It is connected to the host machine via a host bus adaptor (HBA) with 6Gb, i.e., the host I/O bandwidth is 750 MB/s theoretically. The internal bandwidth is 3.2 GB/s, thus the bandwidth ratio is 4.27× theoretically[10]. The regular SSD is an identical SSD without implementing the query offloading.

We adopt the C++ version (instead of Java version) of Lucene[11] to be compatible with programming interface of Smart SSDs. We choose the stable 0.9.21 version.

We measure the power consumption via *WattsUp*[12] as follows. Let W_1 and W_2 be the power (in Watts) when the system is sufficiently stabilized (i.e., idle) and running, and t be the query latency, the energy is calculated by $(W_2 - W_1) \times t$. Typically, Smart SSDs incur around 3-4× lower power than regular SSDs.

6. EXPERIMENTAL RESULTS

In this section, we present the results to evaluate the co-design system of Lucene and the Smart SSD by offloading different query operations: intersection (Section 6.1), ranked intersection (Section 6.2), difference (Section 6.3), ranked difference (Section 6.4), and ranked union (Section 6.5).

We compare two approaches: (1) *Smart SSD*: run our integrated Lucene with the Smart SSD; (2) *Regular SSD*: run Lucene on the regular SSD. We measure the average query latency and energy consumption, which are two important metrics[13] for any search engines.

6.1 Intersection

We offload the intersection to the Smart SSD, that is, steps S3 and S4 are executed inside the 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 the regular SSD, the Smart SSD can reduce query latency by 2.2× and reduce energy consumption by 6.7×. The query latency saving comes from the high internal bandwidth and low I/O latency of the Smart SSD. And the energy saving comes from less data movement and the power-efficient processors running inside the SSD.

For deeper analysis, we picked up a typical real-world query with two query terms[14]. Figure 6 shows the time breakdown of running

[10]Our measured bandwidth ratio is around 3.8×, which is a little smaller than 4.27× due to the extra firmware overhead inside SSDs.

[11]`http://clucene.sourceforge.net`

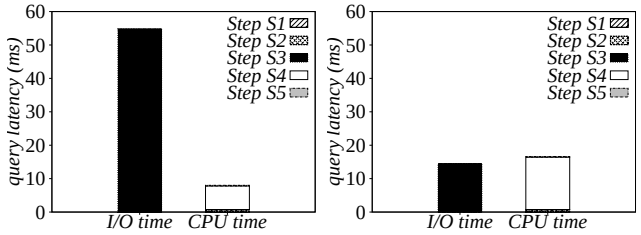
[12]`https://www.wattsupmeters.com`

[13]We acknowledge the system throughput is moderate since our Smart SSD cannot handle concurrent requests. We will fix it in the future.

[14]The number of entries in the inverted lists are 2,938 and 65,057 respectively.

	Query latency (ms)	Energy (mJ)
Smart SSD	97	204
Regular SSD	210	1,365

Table 3: Intersection on real data



(a) Regular SSD (b) Smart SSD

Figure 6: Query latency breakdown of Lucene running on the regular SSD and the Smart SSD

Lucene on the regular SSD and the Smart SSD. It illustrates that the Smart SSD 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 the Smart SSD can achieve. However, the Smart SSD also increase the CPU time from 8 ms to 16.6 ms. Consequently, the Smart SSD can overall improve around 2× speedup in query latency.

Now, we break further down the time for the query processing only inside Smart SSDs. We consider steps S3 and S4 since other steps are executed at the host side and the time is ignorable (see CPU time in Figure 6). As shown in Figure 7, loading inverted lists from flash chips to the device DRAM is still a dominant bottleneck (48%), which can be alleviated by increasing the internal bandwidth. The next bottleneck (28%) is memory access, which can be mitigated by reducing memory access cost (e.g., using DMA copy or more caches). Processor speed is the next bottleneck (22%). This can be reduced by adopting more powerful processors. However, balancing over bus architecture, memory technology, and CPU architecture for SSD systems is also important.

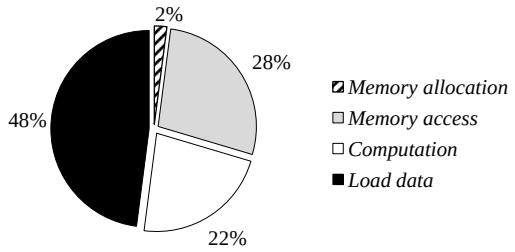
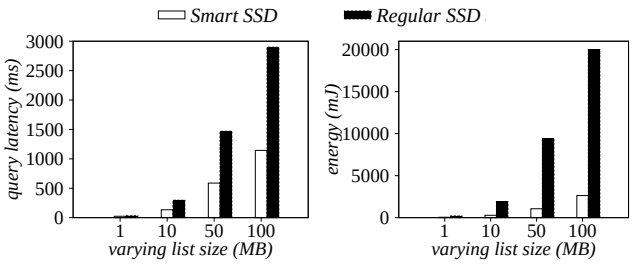


Figure 7: Query latency breakdown of executing list intersection on Smart SSDs

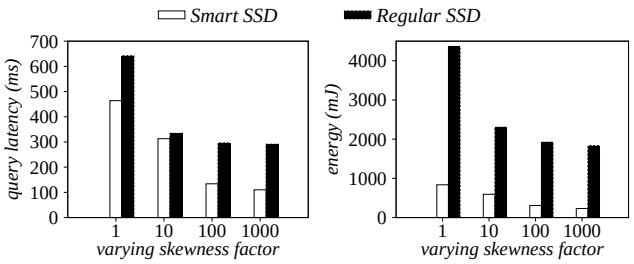
Effect of varying list size. Figure 8 evaluates the effect of list sizes, which affect the I/O time. Longer lists means more I/O time, which Smart SSDs can reduce. We vary the size of list B from 1 MB to 100 MB (while the size of list A depends on the skewness factor whose default value is 100). Both query latency and energy consumption increase with longer lists because of more I/Os. On average, Smart SSDs reduce query latency by a factor of 2.5 while reduce energy by a factor of 7.8 compared to regular SSDs.

Effect of varying list size skewness factor. Figure 9 shows the impact of the list size skewness factor f, which can affect the adaptive intersection algorithm. Higher skewness gives more opportunities for skipping data. This favors Smart SSDs due to expensive memory accesses. We vary different skewness factors from 1 to



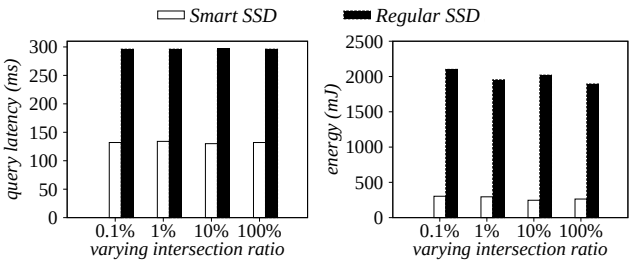
(a) query latency (b) energy

Figure 8: Varying the list size (for intersection)



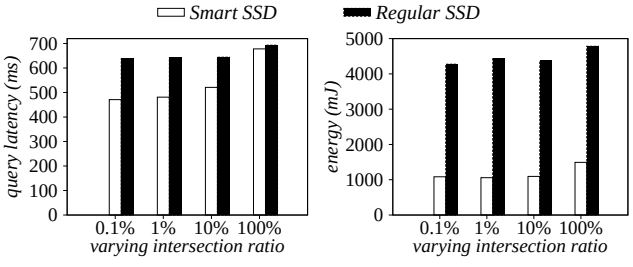
(a) query latency (b) energy

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



(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)

1,000 (while fixing the size of list B to be 10 MB). The query latency (as well as energy) drops when f gets higher because the size of list A gets smaller. In any case, Smart SSDs outperform regular SSDs significantly in both latency and energy.

Besides the superiority of Smart SSDs shown in Figure 9, it is also interesting to see that latency gain in Smart SSDs is the smallest when $f = 10$, not when $f = 1$. That is because the average number of memory accesses per page (ANMP) is higher than all the other cases at $f = 10$ (see Table 4). Note that when $f = 1$, the sort-merge algorithm is adopted, while the adaptive intersection algorithm is used for all the other cases (as explained in Section 4.1).

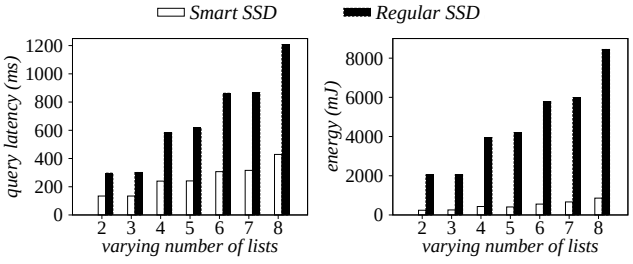
Effect of varying intersection ratio. Figure 10 illustrates the impact of the intersection ratio r. It determines the result size which can have an impact on system performance in two aspects: (1) data movement via the host I/O interface; and (2) ranking cost at the host side (since all the qualified document IDs will be evaluated for ranking). Surprisingly, we cannot find a clear correlation between performance and the intersection ratio (refer to Figure 10). That is because, by default, list A includes around 3,277 entries (0.1 MB) while list B includes around 327,680 entries (10 MB). Even when r is 100%, the result size is at most 3,277. This does not make much difference in both I/O time (around 1.4 ms) and ranking cost (around 0.5 ms).

Then, we conduct another experiment by setting the size of list A the same as B (i.e., both are of 10 MB). We see a clear impact of the intersection ratio r in Figure 11. Both query latency and energy consumption increase as r gets higher, especially when r grows from 10% to 100% (the corresponding result size jumps from 32,768 to 327,680). For regular SSDs, this increase originates from more ranking cost overhead. For Smart SSDs, it results from both data transfer and ranking cost overhead. In all cases, the Smart SSD shows a better performance than the regular SSD. This is true even when r is 100%. That is because, in this case (r is 100%), the Smart SSD only need to transfer one list, which saves around 50% of data transfer.

Effect of varying number of lists. Figure 12 shows the impact of varying the number of lists (i.e., number of terms in a query). More lists mean more I/O time, which Smart SSDs can reduce. We vary the number of lists from 2 to 8. The query latency (as well as energy) grows with higher number of lists[15] because of more data transfer. On average, Smart SSDs reduce query latency by 2.6× and energy by 9.5×.

f	n_1	n_2	n_p	estimated ANMP	real ANMP
1	327,680	327,680	2,564	$\frac{n_1+n_2}{n_p} = 256$	383
10	32,768	327,680	1,408	$\frac{n_1 \cdot \log n_2}{n_p} = 427$	640
100	3,277	327,680	1,284	$\frac{n_1 \cdot \log n_2}{n_p} = 47$	68
1,000	328	327,680	1,280	$\frac{n_1 \cdot \log n_2}{n_p} = 5$	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 can contain 256 entries.



(a) query latency (b) energy

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

In summary, the intersection operation can be cost-effectively

[15] When the number of lists u goes from 2 to 3, the latency does not increase so 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.

offloaded to Smart SSDs, especially when the number of lists is high, the lists are long, the list sizes are skewed, and the intersection ratio is low.

6.2 Ranked Intersection

We offload the ranked intersection (i.e., steps S3, S4, and S5 in Figure 5) to Smart SSDs. Compared to the offloading of intersection-only operation (Section 6.1), offloading ranked intersection can (1) save data transfer since only the top ranked results are returned; but (2) increase the cost of ranking inside the 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 the Smart SSD to the host takes around 12 ms, and ranking 30,000 entries at the host side takes around 5 ms (Figure 11).

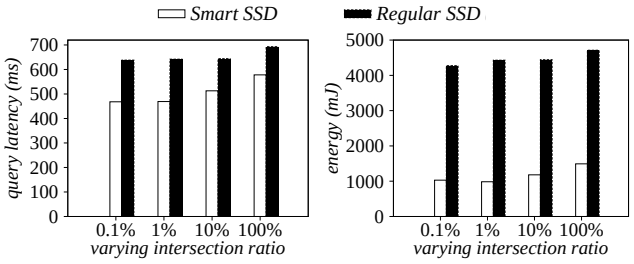
Results on real data. The results are similar to the non-ranked version(see Table 3). That is because the average intersection result size is 1,144, which will not make a significant difference (less than 1 ms). So, we omit them.

Effect of varying list size. The results are also similar to non-ranked intersection (i.e., Figure 8) because the intersection size is small. As an example, 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 3,877 (when the skewness factor is 1). Again, Smart SSDs show better performance.

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 SSDs outperform regular SSDs significantly.

Next, we conduct another experiment by setting the size of list A the same as list B (both are 10 MB), see Figure 13. High intersection ratio leads to high intersection result size, while causing more overhead for ranking. We vary the intersection ratio from 0.1% to 100%. The query latency (as well as energy consumption) goes up as an intersection ratio increases. However, the increase is not that high compared to the non-ranked intersection offloading (in Figure 11). As an example, for the Smart SSD, when the intersection ratio changes from 10% to 100%, the latency increases by 65 ms in Figure 13, while the corresponding increase in Figure 11 is 163 ms. This difference is closely related to the extra data movement. For ranked intersection offloading, since only the top ranked results are returned, less amount of data needs to move.



(a) query latency (b) energy

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

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.

6.3 Difference

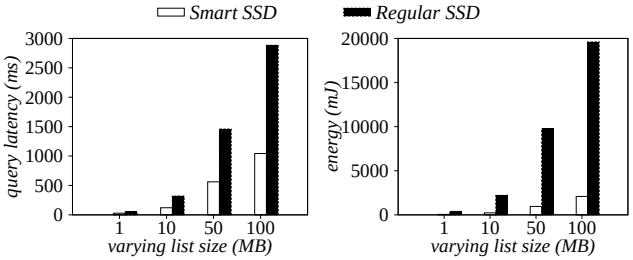
We offload the difference operation (i.e., steps S3 and S4 in Figure 5) to Smart SSDs, and the ranking is executed at the host side. When the difference operator is applied to two lists, it can be $(A - B)$ or $(B - A)$, where the list A is shorter than list B. As discussed in Section 3.1, only the former case can potentially benefit from Smart SSDs.

Results on real data. Table 5 shows the results with 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 respectively. It clearly shows that compared to regular SSDs, Smart SSDs can achieve better performance: reduce the query latency by 2.5× and energy consumption by 8.5×.

	Query latency (ms)	Energy (mJ)
Smart SSD	78	148
Regular SSD	194	1,261

Table 5: Difference on real data

Effect of varying list size. Figure 14 plots the effect of varying list sizes, which affect 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 consumption) goes up with longer lists. On average, the Smart SSD reduces query latency by 2.7× and energy consumption by 9.7×.

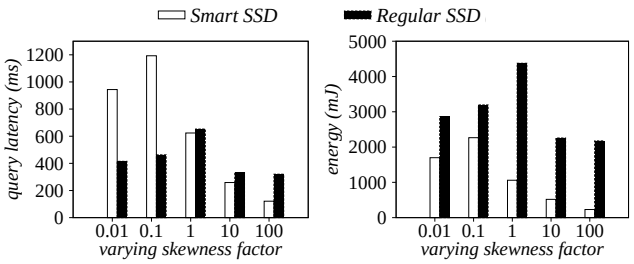


(a) query latency (b) energy

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). Thus, $f < 1$ means 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 SSDs, Smart SSDs have a longer query latency when the skewness factor $f = 0.01$ and $f = 0.1$ (in these two cases, $|A| > |B|$). That is because the $|A \cap B|$ is very small, the result size of $(A - B)$ is very close to $|A| + |B|$. E.g., when $f = 0.01$, $\frac{|A-B|}{|A|+|B|} = 98.6\%$. So, if $|A| > |B|$, it is not cost-effective to offload $(A - B)$ because it does not save much data transfer. (2) Smart SSDs, on the other hand, show better performance when $f \geq 1$ (i.e., $|A| \leq |B|$). That is because $|A - B| \leq |A| \leq (|A| + |B|)/2$. This means offloading $(A - B)$ can save at least 50% of the data transfer. (3) For Smart SSDs, the query latency increases when f goes from 0.01 to 0.1, but decreases afterwards when $f > 0.1$. We can analyze this as follows. Let n_1 and n_2 be the number of entries of list A and B, when $f = 0.01$ or $f = 0.1$ ($n_1 > n_2$). Then, the estimated number of memory accesses is $n_1 \cdot \log n_2$. When f increases from 0.01 to 0.1, n_2 increases (but n_1 remains the same). Consequently, it incurs more memory accesses. However, when $f = 1$, the element checking algorithm is switched to the linear



(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

search (see Section 4.3). Thus, the estimated number of memory accesses is $(n_1 + n_2)$. When $f = 10$ or $f = 100$ ($n_1 < n_2$), the algorithm switches back to the binary search. However, since $n_1 < n_2$ when $f > 1$, it causes even less memory accesses. Table 6 shows the actual number of memory accesses. (4) For regular SSDs, it shows a similar trend to Smart SSDs when f increases. However, when f changes from 0.01 to 1, unlike Smart SSDs, the latency still increases because, for regular SSDs, I/O time is a dominant factor. In other words, when $f = 1$, both A and B are of 10 MB, where the total data size greater than all the other cases. (5) As a comparison, when $f = 1$, both lists are a size of 10 MB. This case is similar to Figure 11(a) when the intersection ratio is 100%. Both adopt sort-merge based algorithm, and can save around 50% of data transfer. Thus, echo the results in 11(a). (6) In terms of energy consumption, Smart SSDs always achieve better performance with the help of its power-efficient processors inside SSDs.

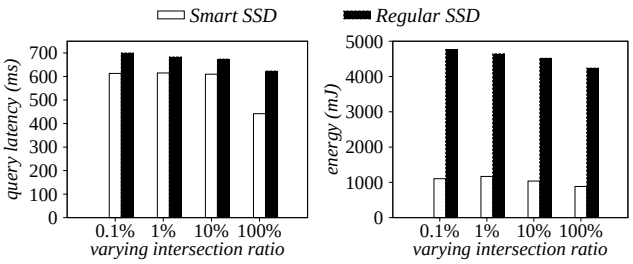
Effect of varying intersection ratio. The intersection ratio is also a crucial parameter to $(A - B)$. It determines the result size that can affect the system performance in two aspects: (1) data transfer cost and (2) ranking cost (at the host side). Intuitively, the higher intersection ratio, the smaller result size. We set the size of list A the same as list B in this case[16]. As shown in Figure 16, for the Smart SSD, both the latency and energy consumption generally decrease as the intersection ratio increases, specifically from 10% to 100% due to lower data transfer cost and ranking cost. For the regular SSD, its performance gain results solely from lower ranking cost.

In summary, it is cost-effective to offload the difference operation $(A - B)$ only if $|A| \leq |B|$, and Smart SSDs favor lists with a smaller intersection ratio.

6.4 Ranked Difference

We offload the ranked difference (i.e., steps S3, S4, and S5 in Figure 5) to Smart SSDs. As discussed before, compared to the non-ranked operation, offloading the ranked operation can reduce the data transfer cost, but increase the ranking cost. When the result size is large, Smart SSDs can benefit because it can save more data transfer time at the cost of extra ranking overhead. On the other hand, there is no notable performance gain when the result size is small (e.g., less than 30,000 entries).

[16] As discussed before, since there is no noticeable changes when the size of list A is 0.01% of list B, we omit the results.



(a) query latency (b) energy

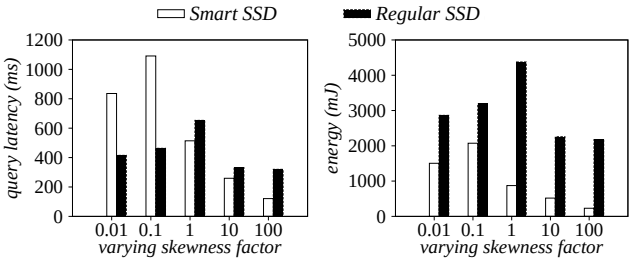
Figure 16: Varying intersection ratio (for difference)

Results on real data. The results are similar to non-ranked difference (see Table 5) because the result size is small (it is 3,109 on average).

Effect of varying list size. The results are also similar to the non-ranked version (see Figure 14) as the result size is small (the maximum is 32,204).

Effect of varying list size skewness factor. The skewness factor determines the result size. For the non-ranked difference (refer to Figure 15), the Smart SSD has a longer query latency when $f = 0.01$ and $f = 0.1$ due to large result size. If the ranking function is applied, the result size will be much smaller. Therefore, we can expect the Smart SSD achieves better performance (i.e., shorter latency) than the regular SSD in all cases.

Surprisingly, the Smart SSD still has a longer query latency than the regular SSD when $f = 0.01$ and 0.1 (see Figure 17). That is because the ranking function is applied only when all the results are available by the difference operation. This requires too many memory accesses to return the results (as analyzed in Section 6.3) regardless of any data transfer. The situation could be changed if we combine both ranking and difference together by adopting a top-k ranking algorithm [15, 6].



(a) query latency (b) energy

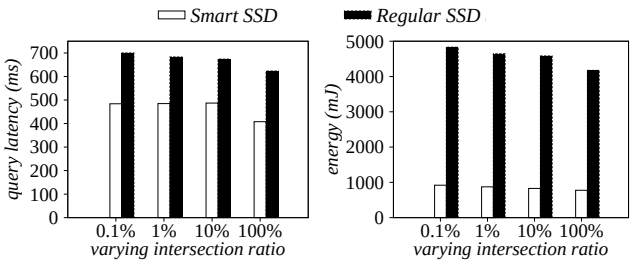
Figure 17: Varying the list size skewness factor (for ranked difference)

Effect of varying intersection ratio. Figure 18 shows the impact of varying intersection ratio to system performance. It clearly shows the superiority of Smart SSDs in terms of both query latency and energy consumption. Compared to Figure 16, the performance gap between Smart SSDs and regular SSDs is larger. That is due to less data transfer after ranking.

6.5 Ranked Union

We offload the ranked union (i.e., steps S3, S4, and S5 in Figure 5) to Smart SSDs.

Results on real data. Table 7 shows the experimental results with real data: the Smart SSD is around 1.7× slower compared to the regular SSD in query latency. This results from too many memory accesses. As discussed in Section 4.2, every list has to be



(a) query latency (b) energy

Figure 18: Varying intersection ratio (for ranked difference)

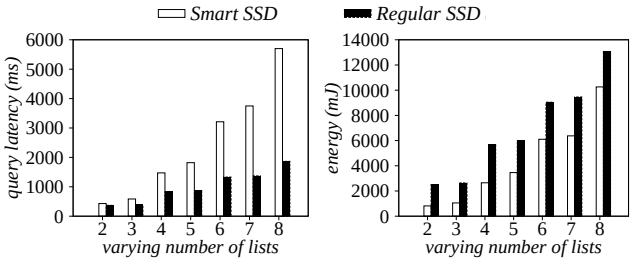
	Query latency (ms)	Energy (mJ)
Smart SSD	505	960
Regular SSD	299	2,033

Table 7: Ranked union on real data

scanned around $2u$ times, where u is the number of lists in a query. On average, $u = 3.8$ in our query log. However, the Smart SSD can still benefit from energy consumption by $2.1\times$ because of its power-efficient processors.

We omit the results of varying intersection ratios, list size skewness factors, and list sizes for space constraints. The short summary of the results are as follows: the Smart SSD is around $1.2\times$ slower in query latency, but saves energy around $2.8\times$. Next, we explore the effect of varying number of lists, which is a key parameter.

Effect of varying number of lists Figure 19 displays the impact of number of lists u in a query. The query latency gap between the Smart SSD and the regular SSD gets larger with high number of lists. That is because each list has to be accessed approximately $2u$ times.



(a) query latency (b) energy

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

Remark Because of too many memory accesses, it is not cost-effective to offload the ranked union to Smart SSD. However, both union and ranking could be algorithmically combined for early termination [6, 15], such that we do not need to scan all the results. We will explore the early pruning techniques in the future work.

7. 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) have been devoted to making it practical.

Early work on In-Storage Computing. As early as 1970s, some pieces of initial work have been proposed to leverage specialized hardware (e.g., processor-per-track and processor-per-head) for improving query processing in storage devices (i.e., hard disks at that time). For example, CASSM [35] followed the processor-per-track architecture to embed a processor per each track. The Ohio State Data Base Computer (DBC) [20] followed 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 work on HDD In-Storage Computing. In late 1990s, the bandwidth of hard disks (HDD) kept growing while the cost of powerful processors kept dropping, which makes it feasible to offload bulk computation to each individual disk. Researchers started to explore in-storage computing in terms of hard disks (e.g., active disk [1] or intelligent disk [21]). Their goal is to offload application-specific query operators inside hard disk in order to save data movement. They examined active disk in database area by offloading several primitive database operators (e.g., selection, group-by, sort). Later on, Erik et al. extended 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 work on SSD In-Storage Computing. With the advent of SSD, people start to rethink about in-storage computing in the context of SSDs (e.g., Smart SSDs). SSDs offer many advantages over HDDs such as very high internal bandwidth and high computing power. More importantly, executing code inside SSDs can save a lot of energy due to less data movement and power-efficient embedded ARM processors. This makes the concept of in-storage computing on SSDs much more practical and promising this time. Industries like IBM started to install Smart SSDs to their Blue Gene supercomputers to leverage the high internal bandwidth of SSDs [19]. In this way, computing power and storage device are closely integrated. Teradata's Extreme Performance Appliance [36] is another example of combining SSDs and database functionalities. Oracle's Exadata [28] also started to offload complex processing into their storage servers.

SSD in-storage computing attracts academia as well. In database area, Kim et al. investigated pushing down the database scan operator to SSD [22]. That work is based on simulation. Later, Do et al. [14] built a Smart SSD prototype on real SSDs. They integrated Smart SSD with Microsoft SQL Server by offloading two operators: scan and aggregation. Woods et al. built another types of Smart SSD prototype with FPGAs [39]. Although they also targeted at database systems, they provided more operators such as group-by. They integrated the prototype with MySQL storage engine such as MyISAM and INNODB. In data mining area, Bae et al. studied offloading functions like k-means [2]. In data analytic area, De et al. proposed to push down hash tables inside SSDs [11]. There is also another study on offloading sorting [23]. In system area, Seshadri et al. built a Willow system [33] and studied the offloading of many applications, e.g., file system, transactional processing.

Our work investigates the potential benefit of Smart SSDs to search engine area, the first work in this area.

8. CONCLUSION

SSD In-Storage Computing (Smart SSD) is a new computing paradigm to make full use of SSD capabilities. This work introduces Smart SSDs to search engine area. With the close collaboration with an SSD vendor, we co-design the Smart SSD with a popular open-source search engine – Apache Lucene. The main challenge is to determine what query processing operations in the host Lucene system can be cost-effectively offloaded to Smart SSDs. We demonstrate that: (1) The intersection operation (both non-ranked and ranked version) can be cost-effectively offloaded to Smart SSDs, in particular, when the number of lists is large, the lists are

long, the list sizes are skewed, and the intersection ratio is low; (2) The difference operation $A - B$ (both non-ranked and ranked) can be a good candidate for offloading only if $|A| \leq |B|$, and Smart SSDs favor lists with a high intersection ratio; (3) The union operation (both non-ranked and ranked) causes a heavy memory access. Thus, it is not beneficial to offload to Smart SSDs. Except for the union operation, Smart SSDs can reduce the query latency by 2-3$\times$ and energy consumption by 6-10$\times$ on the real dataset.

We also observe that the boundary between the CPU time and I/O time is getting blurrier for the query processing (e.g., intersection) inside Smart SSDs. The CPU time (including DRAM access time) can be comparable to or even higher than the I/O time (see Figure 7). This inspires both SSD vendors and system designers. SSD vendors can improve hardware performance such as processor speed and memory access speed by introducing more power processors and caches. On the other hand, system designers can develop efficient algorithms by considering the special characteristics of Smart SSDs (e.g., minimize expensive memory accesses).

Future work. We have a number of interesting future work in mind. (1) Executing list operations on *compressed* lists. Compressed lists take less space so that they can save the I/O time. However, decompression inside Smart SSDs takes more CPU time than host systems. Thus, balancing of the I/O time and CPU time is a critical issue. (2) Optimization techniques to implement the top-k ranking. We can algorithmically combine steps S4 and S5. Consequently, we can cost-effectively offload the ranked union operation. (3) Offloading other functions of Lucene. Building inverted index may benefit from Smart SSDs not only because it is a very I/O-intensive task, but also it does not require heavy CPU computation.

REFERENCES

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

[2] 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.

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

[4] 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.

[5] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[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] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[10] 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.

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

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

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

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

[17] G. Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *CACM*, 52:48–59, 2009.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 2006.

[19] 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.

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

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

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

[23] 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.

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

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

[26] R. Ma. Baidu distributed database. In *System Architect Conference China*, 2010.

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

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

[29] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *ASPLOS*, pages 471–484, 2014.

[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] E. Schurman and J. Brutlag. Performance related changes and their user impact. Velocity, 2009.

[33] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, 2014.

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

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

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

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

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

[39] 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.

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