# 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) instead of just using SSDs as yet-anther-faster HDDs. E.g., SSD-aware Btrees [24], SSD-aware buffer management [15]. These pieces of research 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 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 codes*" cannot make full use of SSDs for two main reasons [14, 35]. (1) SSDs generally provide higher internal bandwidth than the host I/O interface bandwidth. However, since data must be transferred to the host via the host I/O interface, its bandwidth can be easily saturated by data-intensive applications, which results in waste of SSDs' high internal bandwidth; (2) SSDs, in general, provide high computing capabilities (for executing complex FTL firmware codes [9]) 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 codes (e.g., database scan and aggregation) inside SSDs, to take advantage of the high internal bandwidth, low I/O latency, and computing power. This ISC approach changed the traditional computing paradigm to "*move codes closer to data*" (or generally near-data processing [5]). In addition to the performance improvement, Smart SSDs significantly reduce energy consumption due to less data movement and its power-efficient processors. As a result, executing application logic inside Smart SSDs is a very promising solution to *make full use* of modern SSDs. It also attracts industry. E.g., IBM applies Smart SSDs in their blue gene storage systems [18].

This work explores how to apply ISC 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 In-Storage Computing devices*? To answer this question, we first identified five commonly used operations in Lucene (and any search engine) that could potentially benefit from Smart SSDs. Then we codesigned their operations (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 made 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 Smart SSDs reduce the query latency by a factor of 2-3× and energy consumption by 8-10× for the most of the aforementioned operations.

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

The rest of this paper is organized as follows. Section 2 provides



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

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

an overview of SSD internal architecture, and Lucene's search architecture. Section 3 describes how our Smart SSD works. Section 4 presents 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 our 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

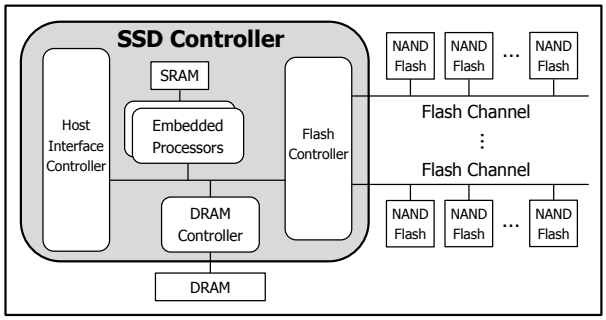



**Figure 1:** SSD internals

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 Controller, also called 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 subdivides further into smaller units that can independently execute commands or report status. 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.

## 2.2 Query Processing of Lucene

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

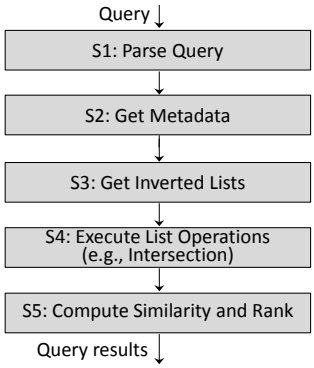



**Figure 2:** Query processing of Lucene

Like 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 containing the term.

Upon receiving a user query $q$ (e.g., "SSD database"), Lucene answers it through 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 has 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 following 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. In our example, it is intersection (because of the `AND` query mode). It could be other operations such as union (`OR` mode) or difference (`NOT` mode). *Step S5*: for each qualified document $d$, calculate the similarity value between the query $q$ and the document $d$ by using an IR relevance model. Lucene adopts a modified BM25 model [31]. Finally, Lucene returns top-ranked results to end users.

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

# 3. SMART SSD ARCHITECTURE

Smart SSDs, unlike the traditional CPU-centric computing systems, enable ISC devices to play a major role in computation by offloading key functions of host systems into ISC devices. Since its hardware architecture is identical to the aforementioned modern SSDs shown in Figure 1, this section describes our Smart SSD software architecture and key components as ISC devices.

As illustrated in Figure 3, Smart SSD consists of several key software components and it communicates with a Smart SSD host program via Smart SSD application programming models (APIs).

An SSDlet is a Smart SSD program in an ISC device. It implements application logic and responds to a Smart SSD host program. The SSDlet is executed in an event-driven manner by the Smart

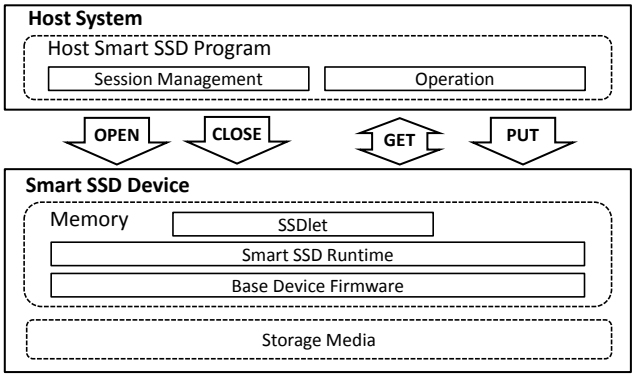



**Figure 3:** Smart SSD Architecture

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. Afterward, 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 to a host 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

This section describes 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, and 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 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 5-10× higher internal bandwidth than the host I/O interface bandwidth [14, 10]. This gap is predicated to increase in the future. (2) The I/O latency is very short. A regular I/O operation (from flash chips to the host DRAM) needs to go through the conventional thick OS stack–a file system, interrupt, context switch between the kernel space and the user space–which is collectively called OS software overhead. This 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]. On the other hand, 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 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. 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 typical SSD controllers do not adopt caches (e.g., L1/L2 caches). Thus, it is not desirable to execute *CPU-intensive* and *memory-intensive* operations inside SSDs.

In short, Smart SSDs can reduce the I/O time at the expense of the increasing CPU time. Therefore, we can advise that a system with I/O time bottleneck can notably benefit from Smart SSDs. As an example, the Lucene system running on regular SSD has an I/O time bottleneck. Please see Figure 4 (refer to Section 7 for more experimental settings). We observe the I/O time of a particular 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 7 supports this claim. It shows that, for the same query, if we offload step 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.

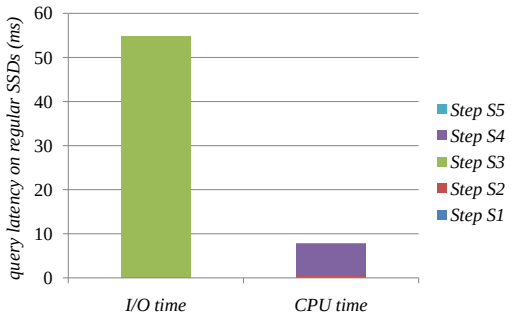



**Figure 4:** Time breakdown of executing a particular real query by Lucene system running on regular SSDs

Based on this observation, 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 make a rough analysis first, and elaborately evaluate them 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 profitable to offload this step S1 to Smart SSDs.

**Step S2: Get metadata**. The metadata is essentially a key-value pair. The key represents a query term, and the value corresponds to the basic information about the on-disk inverted list of the term.

Lucene 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. Since it takes very few (usually $1 \sim 2$) I/O operations to obtain the metadata [26], we do not offload this step.

**Step S3: Get inverted lists**. Each inverted list contains a list of documents containing the same term. Since, by default, Lucene stores all the inverted lists on a disk, upon receiving a query, it reads the inverted lists from the disk to a 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 step S4 and S3 should be offloaded to Smart SSDs. Now, another question raises: 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[3]). 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 (assuming $A$ is shorter than $B$ to capture the real case of skewed lists).

- Intersection: The result size of the intersection is usually even 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]. We also observed similar results with our real datasets. 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, $|A - B| = |A| - |A \cap B| < |A| \ll |A| + |B|$, i.e., 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.

**Step S5: Compute similarity and rank**. After the aforementioned list operations are completed, we can get a list of qualified documents. 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 overheads. 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.

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

| | 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 (please see Table 1). The offloading of non-ranked operations means that only 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 all 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

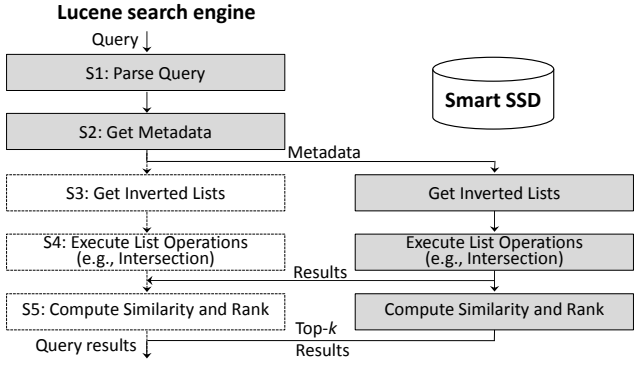



**Figure 5:** Co-design architecture of Lucene and Smart SSDs

Figure 5 shows the co-design architecture of Lucene search engine and Smart SSDs. We modified Lucene codes to interact with Smart SSDs.

It operates 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 are ready to return to the host in an output buffer. The host Lucene keeps monitoring the status of Smart SSDs in a heartbeat 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 take care of step S5.

# 5. IMPLEMENTATION

This section describes 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 as well as in other search engines. It is essentially a mapping data structure between query terms and inverted lists. Each inverted list contains 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 some changes which are equally applied to Lucene running on both regular SSDs and Smart SSDs for fairness. (1) Instead of storing the document frequency in Lucene, we store the actual score according to Lucene's ranking model. We explain more detail in Section 5.4. 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 [29] in Lucene. We believe this change will not affect the system performance much because both data structures support an 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 a 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) Besides, every inverted list is stored on SSD in a page-aligned manner (page size 8KB). That is, it starts and ends at a multiple of 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 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 the 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. Since these are initially stored on SSD flash chips, we need to load them to a device memory first. Then we apply an in-memory list intersection algorithm.

There are a number of in-memory list intersection algorithms such as 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 adopt it inside Smart SSDs. (2) The adaptive algorithm works well in theory and practice. According to a recent study in [13], the performance of adaptive algorithm better than others except the group based algorithm. Although the group based algorithm [13] performs better, it requires 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, ..., L_u from the SSD to device memory
   (assume 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
           count reaches u
9      else
10         pivot ← successor
11 until target = INVALID;
12 return R
```

Algorithm 1 describes 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 be even faster than binary search [13]. Thus, 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 codes 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 adopted in Lucene) for executing the union operation. Please refer to Algorithm 2.

**Algorithm 2:** 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
```

It is interesting to note that Algorithm 2 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 (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 (simi-

larly 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, list *A* and *B*. Then $A - B$ finds all elements in *A* but not in *B*. The algorithm is trivial: for each element $e \in A$, 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 system.

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

## 5.4 Ranked Operations

Those list operations (e.g., intersection) can return many results. However, end users are mostly interested in the most relevant results. This requires two more 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. We, therefore, 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 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. Consequently, the similarity computation can be much more efficient. This also means adopting Smart SSDs will not degrade query quality. For a fair comparison, we also modified Lucene codes when it runs on regular SSDs.

The remaining question is how to efficiently find the top ranked results. We maintain the top ranked results explicitly in SRAM, not in DRAM, in a heap-like data structure. Then we scan all the similarities to update the results in SRAM if necessary.

# 6. EXPERIMENTAL SETUP

This section presents 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 our system performance, we employee 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 better understand various performance-critical parameters in Smart SSDs. 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), 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.

**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 even 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[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., based on 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 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 with relatively shorter inverted lists (on average, 10s of MBs). The size of list $A$ can be obtained with the skewness factor. Once 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 a double-blind review.

[5]We randomly pick up 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.

| | 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 a size of 400 GB SAS SSD (SLC) and connected to the host machine via a host bus adaptor (HBA) with 6Gbps. The regular SSD is an identical SSD without implementation of query offloading.

We adopt the C++ version (instead of Java version) of Lucene[6] to be compatible with programming interface of 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 sufficiently stabilized (i.e., 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 our experimental results and analysis 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*: our integrated Lucene with Smart SSD; (2) *Regular SSD*: the original Lucene on a regular SSD. As our performance metrics, we adopt 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 consumption with 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 energy by 6.7×. The performance gain of the query latency comes from the high internal bandwidth and low I/O latency of Smart SSD. In addition, the energy saving results from less data movement and the power-efficient processors inside SSD.

For more detailed analysis, we chose 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 illustrates that Smart SSD can reduce the I/O time from 54.8 ms to 14.5 ms (i.e., a factor of 3.8). This is the speedup upper bound that Smart SSD can achieve. However, Smart SSD also increase the CPU time from 8 ms to 16.6 ms. Consequently, Smart SSDs can overall achieve around 2× speedup in query latency.

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

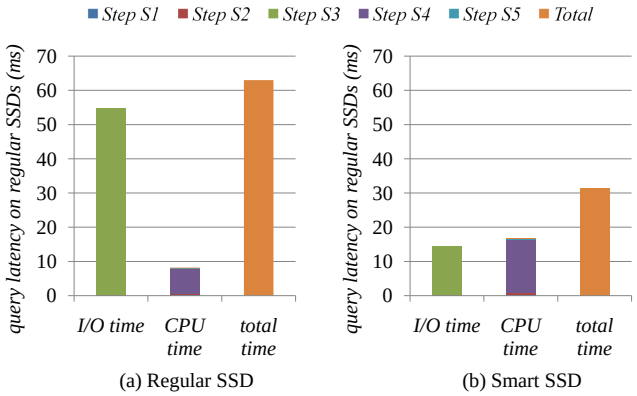



**Figure 6:** Query latency breakdown of Lucene running on the regular SSD and the Smart SSD, for a particular real query.

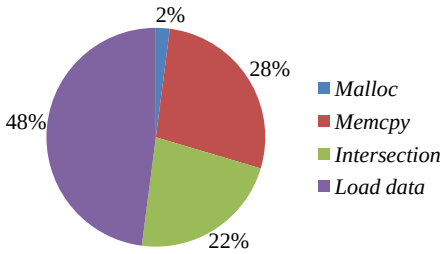



**Figure 7:** Time breakdown of executing list intersection on Smart SSDs

**Effect of varying list size**. Figure 8 displays the effect of list size which affects the I/O time (that is, longer lists imply more I/O time). 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 SSD reduces query latency by a factor of 2.5 and energy by a factor of 7.8 compared to regular SSD. Figure 8 delivers the following implication: *Smart SSD favors longer lists for the intersection operation.*

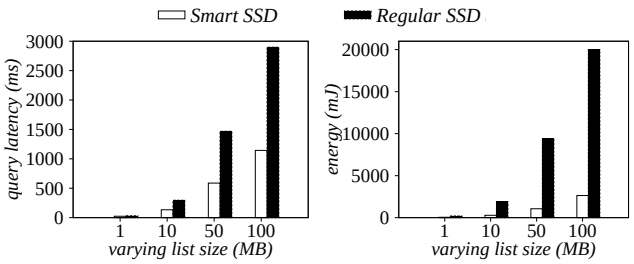
**Effect of varying list size skewness factor**. Figure 9 shows the impact of list size skewness factor $f$, which can affect the adaptive intersection algorithm. Higher skewness provides more opportunities for skipping data. This favors Smart SSD with less memory access since the memory access inside Smart SSD is expensive. We vary different skewness factors from 1 to 1000 (while fixing the size of list $B$ to 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 SSD outperforms regular SSD significantly in both latency and energy. Figure 9 implies: *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

[6] http://clucene.sourceforge.net

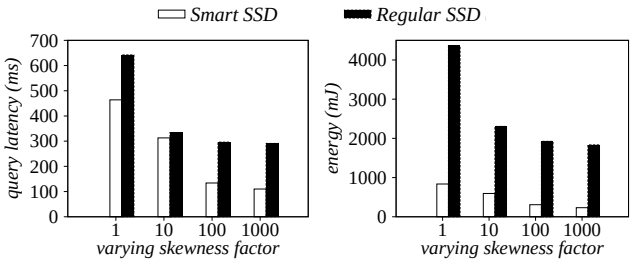
[7] https://www.wattsupmeters.com

[8] Due to the current limitation of the Smart SSD (it can only support one concurrent request each time), we measure the query latency instead of the query throughput. This limitation will be resolved in the next-generation of Smart SSDs.



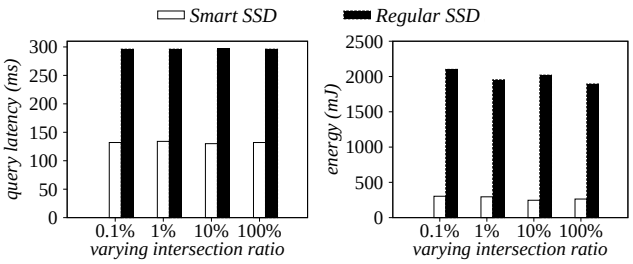
(a) query latency (b) energy

**Figure 8:** Varying the list size (for intersection)



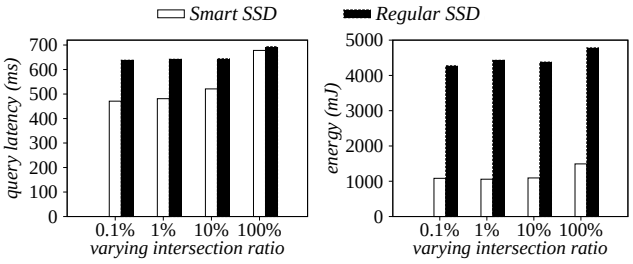
(a) query latency (b) energy

**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)

also interesting to see that the performance gain of the latency in Smart SSD is smallest at $f = 10$, not at $f = 1$. That is because the average number of memory accesses per page (ANMP) is larger than all the other cases at $f = 10$ (please 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 5.1).

**Effect of varying intersection ratio**. Figure 10 illustrates the impact of 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 in the result set will be evaluated for ranking). Surprisingly, we cannot find a clear correlation between performance and intersection ratios (please refer to Figure 10). 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 at most 3277. This does not make much difference in both I/O time (around 1.4 ms) and ranking cost (around 0.5 ms).

To verify this result, we make another experiment by setting the size of list $A$ to the same as $B$ (i.e., both are of 10 MB). We see

| $f$ | $n_1$ | $n_2$ | $n_p$ | 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 can contain 256 entries.

a clear impact of intersection ratio $r$ in the Figure 11. Both query latency and energy consumption increase as $r$ gets higher. Particularly when $r$ grows from 10% to 100% (the corresponding result size jumps from 32,768 to 327,680), we can see noticeable increase of them. For regular SSD, this increase originates from more ranking cost overhead. For Smart SSD, it results from both data transfer and ranking cost overhead. In all cases, even when $r$ is 100%, Smart SSD shows a better performance than regular SSD. That is because, in this case ($r$ is 100%), Smart SSD only needs to transfer one list, which saves 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*.

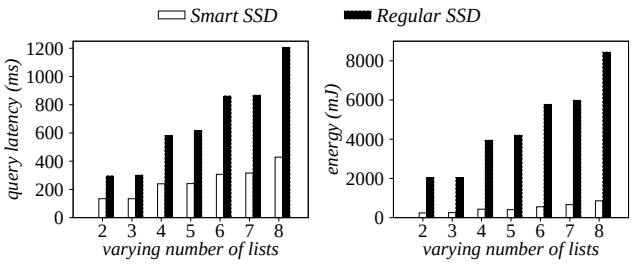
**Effect of varying number of lists**. Figure 12 shows the results on the impact of varying the number of lists (i.e., number of terms in a query). We vary the number of lists from 2 to 8. The query latency (as well as energy) grows with higher number of lists[9] because of more data transfer. On average, 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*.

**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

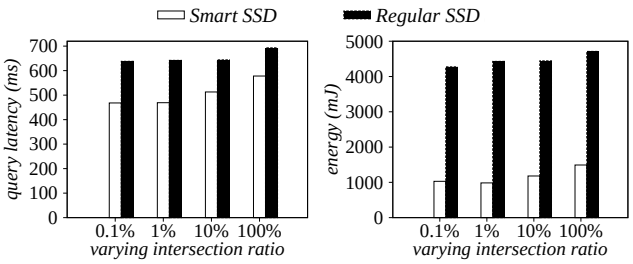
In this case, we offload the ranked intersection (i.e., step S3, S4, and S5 in Figure 5) to Smart SSD. Compared to the offloading of intersection-only operation (Section 7.1), offloading ranked intersection can (1) save data transfer since only the top ranked results are returned; but (2) increase the cost of ranking inside the

[9]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.




(a) query latency (b) energy

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




(a) query latency (b) energy

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

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 the 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). So, we omit them due to space limitations. Since the average intersection result size is 1,144, it will not make a significant difference (less than 1 ms).

**Effect of varying list size**. The results of varying list size 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 SSD shows a 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 SSD outperforms regular SSD significantly.

Next, we make another experiment by setting the size of list $A$ to the same as list $B$ (both are 10 MB). Please 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% too 100%. The query latency (as well as energy consumption) goes up as an intersection ratio increases. However, we cannot see noticeable increase of them compared to the results in Figure 11 for non-ranked intersection offloading. As an example, for 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 top-$k$ results are returned, less amount of data needs to move. Please see the query latency at 100% intersection ratio in both Figure 11 and Figure 13. This demonstrates well our analysis.

**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

We offload the difference operation (i.e., step S3 and S4 in Figure 5) to Smart SSD, 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 4.1, only the former case can potentially benefit from Smart SSD.

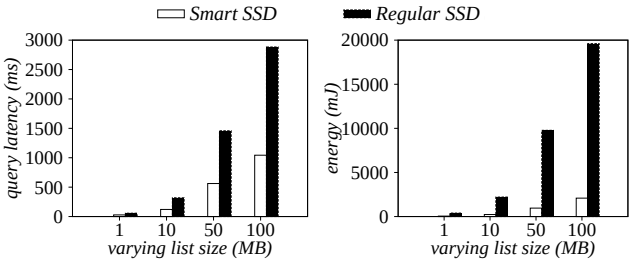
**Results on real data**. Table 5 shows the results with the real

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

**Table 5:** Difference on real data

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 SSD, Smart SSD can achieve better performance: the query latency by 2.5× and energy consumption 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 consumption) goes up with longer lists. On average, Smart SSD reduces query latency by 2.7×, and energy consumption by 9.7×. Figure 14 delivers a message: *Smart SSD favors longer lists for the* difference *operation*.

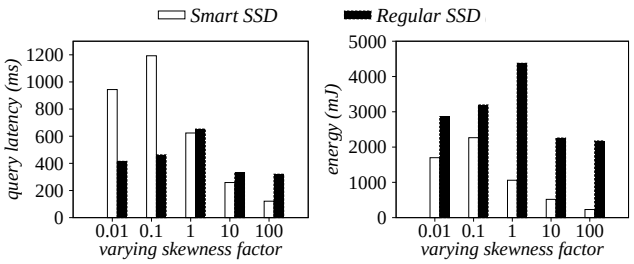



(a) query latency (b) energy

**Figure 14:** Varying the list size (for difference)

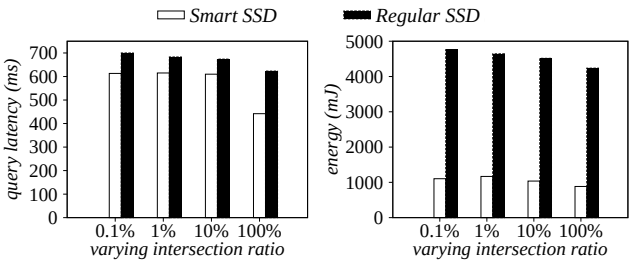
**Effect of varying list size skewness factor**. The skewness factor $f$ is a key parameter in difference operation. Let $A$ and $B$ be two inverted lists. Unlike the default case, $A$ is not necessarily shorter than $B$. It depends on the skewness factor $f$ (still defined as $|B|/|A|$). We vary $f$ from 0.01 to 100 (Table 6 explains 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 SSD, Smart SSD has a longer query latency when the skewness factor $f = 0.01$ and $f = 0.1$ (in these two cases, $|A| > |B|$). Assuming 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 SSD, on the other



(a) query latency (b) energy

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



(a) query latency (b) energy

**Figure 16:** Varying intersection ratio (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

hand, shows better performance when $f \geq 1$ (i.e., $|A| \leq |B|$). 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 decreases afterward 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 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 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 SSD, it shows a similar trend to Smart SSD when $f$ increases. However, when $f$ changes from 0.01 to 1, unlike Smart SSD, its latency still increases because, for regular SSD, 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, it shows a similar result in 11(a). (6) In terms of energy consumption, Smart SSD always achieves better performance with the help of its power-efficient processors inside SSD.

In short, we can deliver the following implication in Figure 15: *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 crucial parameter to $(A - B)$. It determines the result size, which can affect the system performance in two aspects: (1) data transfer cost and (2) ranking cost (at host side). Intuitively, the higher intersection ratio, the smaller result size. We set the size of list $A$ to the same as list $B$ in this case[10]. As shown in Figure 16, for Smart SSD, both the latency and energy consumption generally decreases as the intersection ratio increases, specifically from 10% to 100% due to lower data transfer cost and ranking cost. For regular SSD, its performance gain results solely from lower ranking cost. Figure 16 delivers the following message: *Smart SSD favors lists with a 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 a smaller intersection ratio.

## 7.4 Ranked Difference

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, Smart SSD 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).

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

**Effect of varying list size**. Similar to non-ranked version (please see Figure 14) as the result size is small (maximum is 32,204).

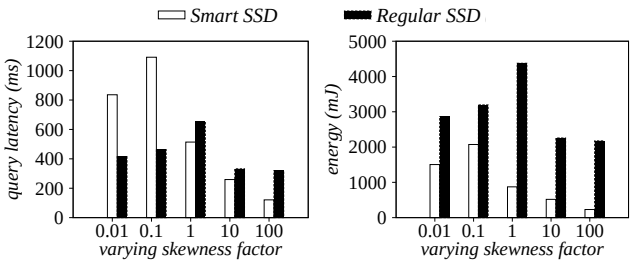
**Effect of varying list size skewness factor**. The skewness factor determines the result size. For the non-ranked Difference (please refer to Figure 15), Smart SSD has a longer query latency when $f = 0.01$ and $f = 0.1$ due to the large result size. Thus, if ranking function is applied, the result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

However, surprisingly, Smart SSD still has a longer query latency than the regular SSD when $f = 0.01$ and 0.1. 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 7.3) 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 [16, 6].

It is also interesting to see the performance gap (in query latency) between Smart SSD and regular SSD in both Figure 17and Figure 15 when $f = 1$. For Smart SSD, query latency in ranked difference is faster than that in non-ranked difference, which comes from less data transfer.

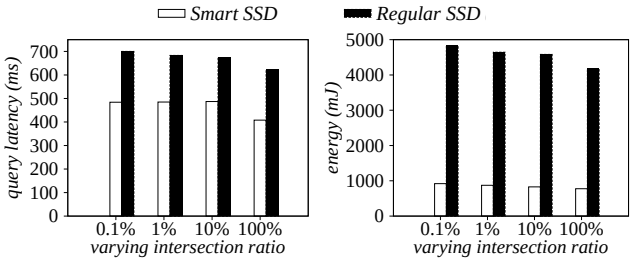
**Effect of varying intersection ratio**. Figure 18 shows the impact of varying intersection ratio to system performance. It clearly shows the superiority of Smart SSD in terms of both query latency and energy consumption. Compared to Figure 16, the performance gap between Smart SSD and regular SSD is larger in terms of dif-

[10] 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 17:** Varying the list size skewness factor (for ranked difference)



(a) query latency (b) energy

**Figure 18:** Varying intersection ratio (for ranked difference)

ferent intersection ratios. 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*

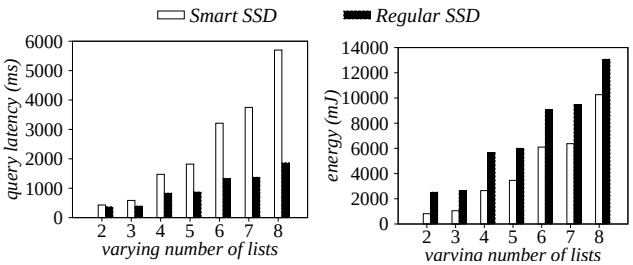
**Results on real data**. Table 7 shows the experimental results with real data: Smart SSD is slower around 1.7× compared to regular SSD in query latency. That is due to too many memory accesses. As discussed in Section 5.2, every list has to be scanned around $2u$ times, where $u$ is the number of lists in a query. On average, $u = 3.8$ in our query log. However, Smart SSD still can benefit from energy consumption by 2.1× with the help of its power-efficient processors.

We omit the results of varying intersection ratios, list size skewness factors, and list sizes, for space constrains. The short summary of the results is as follows; Smart SSD is slower around 1.2× in query latency, but saves energy around 2.8×. 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 Smart SSD and regular SSD gets larger with more number of lists. That is because each list has to be accessed approximately $2u$ times.

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

**Table 7:** Ranked union on real data



(a) query latency (b) energy

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

### *7.5.2 Discussion: Call for Algorithmic Optimizations*

Because of too many memory accesses, it is not cost-effective to offload ranked union to Smart SSD. To resolve this limitation, on the one hand, we fundamentally need to improve the innate memory access speed of Smart SSD through its hardware upgrade. On the other hand, more efficient algorithms could be designed to reduce memory access. Our current implementation (following Lucene) solves the ranking problem when all the union results are available, then scores every qualified document. However, both union and ranking could be algorithmically combined for early termination [6, 16]. This means 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) have been made to make 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 [33] and RAP [28] followed the processor-per-track architecture to embed a processor per each track. The Ohio State Data Base Computer (DBC) [19] and SURE [23] 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 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 [20]). 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.** Recently, with the advent of SSD, people start to rethink about in-storage computing in the context of SSD (i.e., Smart SSD). SSD offers many advantages over HDD such as very high internal bandwidth and

high computing power. More importantly, executing codes inside SSD 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 SSD much more practical and promising this time. Industries like IBM started to install active SSD to their 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 another example of combining SSD and database functionality together. Oracle's Exadata [27] also started to offload complex processing into their storage servers.

SSD in-storage computing (or Smart SSD) attracts academia as well. In database area, Kim et. al investigated pushing down the database scan operator to SSD [21]. 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 [35]. 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 [3]. In data analytic area, De et. al proposed to push down hash tables inside SSD [10]. There is also another study on offloading sorting [22].

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

# 9. 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-designed the Smart SSD with a popular open-source search engine, Apache Lucene. The main challenge is to determine what query processing logic in the host Lucene system can be cost-effectively offloaded to Smart SSDs. We demonstrates 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 lower intersection ratio; (3) The union operation (both non-ranked and ranked) causes a heavy memory access. Thus, it is not beneficial to offload the union operation.

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 (please see Figure 7). This inspires both SSD vendors and system designers to improve Smart SSD. SSD vendors can improve hardware performance such as processor speed and memory access speed. On the other hand, system designers are in charge of developing efficient algorithms by considering Smart SSDs characteristics (e.g., minimize expensive memory accesses).

**Future work.** This is the first work on applying Smart SSDs to search engine area. We have a number of interesting future work in mind. (1) Executing list operations with *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 critical issue. (2) Optimization techniques to implement the top-$k$ ranking. We can algorithmically combine step S4 and S5. Consequently, we can cost-effectively offload the ranked union operation. (3) Offloading other functions of Lucene. We can build inverted index with Smart SSDs. This can benefit a lot from Smart SSDs not only just 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] 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.