This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

SSD In-Storage Computing for Search Engines

Jianguo Wang and Dongchul Park*

Abstract—SSD-based in-storage computing (called "Smart SSDs") allows application-specific codes to execute inside SSDs to exploit the high internal bandwidth and energy-efficient processors. As a result, Smart SSDs have been successfully deployed in many industry settings, e.g., Samsung, IBM, Teradata, and Oracle. Moreover, researchers have also demonstrated their potential opportunities in database systems, data mining, and big data processing. However, it remains unknown whether search engine systems can benefit from Smart SSDs. This work takes a first step to answer this question. The major research issue is what search engine query processing operations can be cost-effectively offloaded to SSDs. For this, we carefully identified the five most commonly used search engine operations that could potentially benefit from Smart SSDs: intersection, ranked intersection, ranked union, difference, and ranked difference. With close collaboration with Samsung, we offloaded the above five operations of Apache Lucene (a widely used open-source search engine) to Samsungs Smart SSD. Finally, we conducted extensive experiments to evaluate the system performance and tradeoffs by using both synthetic datasets and real datasets. 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.

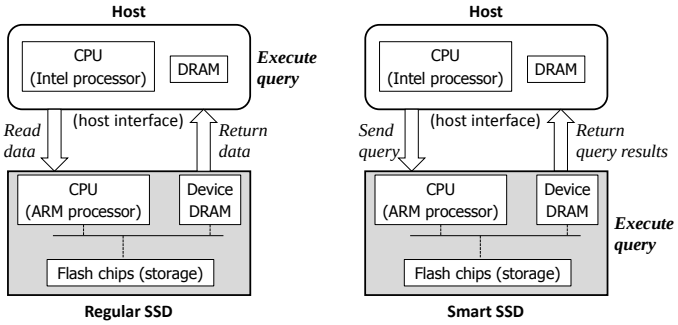
Index Terms—In-Storage Computing, Smart SSD, Search Engine

1 Introduction

Traditional search engines utilize hard disk drives (HDDs), which have dominated the storage market for decades. Recently, solid state drives (SSDs) have gained significant momentum because SSDs have many advantages when compared to HDDs. For instance, random reads on SSDs are one to two orders of magnitude faster than on HDDs, and SSD power consumption is also much less than HDDs [1]. Consequently, SSDs have been deployed in many search systems for high performance. For example, Baidu, China's largest search engine, completely replaced HDDs with SSDs in its storage architecture in 2011 [1], [2]. Moreover, Bing's new index-serving system (Tiger) and Google's new search system (Caffeine) also incorporated SSDs in their main storage to accelerate query processing [3], [4]. Thus, in this work, we primarily focus on a pure SSD-based system without HDDs.[1]

In such a (conventional) computing architecture which includes CPU, main memory, and SSDs (Figure 1a), SSDs are usually treated as storage-*only* units. Consider a system running on a conventional architecture. If it executes a query, it must read data from an SSD through a host interface (such as SATA or SAS) into main memory. It then executes the query on the host CPU (e.g., Intel processor). In this way, data storage and computation are strictly segregated: the SSD stores data and the host CPU performs computation.

However, recent studies indicate this "move data closer to code" paradigm cannot fully utilize SSDs for several reasons [5], [6]. (1) A modern SSD is more than a storage device; it is also a computation unit. Figure 1a (gray-color area) shows a typical SSD's architecture which incorporates energy-efficient processors (like ARM series processors) to execute storage-related tasks, e.g., address translations and garbage collections. It also has device DRAM (internal memory) and NAND flash chips (external memory) to store data. Thus, an SSD is essentially a small computer (though not as powerful as a host system). But conventional system architectures completely ignore this SSD computing capability by treating the SSD as a yet-another-faster HDD. (2) A modern SSD is usually manufactured with a higher (2-4×) internal bandwidth (i.e., the bandwidth of transferring data from flash chips to its device DRAM) than the host interface bandwidth. Thus, the data access bottleneck is actually the host interface bandwidth.



(a) Conventional architecture (b) New architecture

Fig. 1. A conventional computing architecture with regular SSDs *vs.* a new computing architecture with Smart SSDs

To fully exploit SSD potential, SSD in-storage computing (a.k.a Smart SSD) was recently proposed [5], [6]. The main idea is to treat an SSD as a small computer (with ARM processors) to execute some programs (e.g., C++ code) directly inside the SSDs. Figure 1b shows the new computing architecture with Smart SSDs. Upon receiving a query, unlike conventional computing architectures that have the host machine execute the query, the host now sends the

- *J. Wang is with the Department of Computer Science, University of California, San Diego, USA.*
  E-mail: csjgwang@cs.ucsd.edu
- *D. Park is with the Memory Solutions Lab (MSL) at Samsung Semiconductor Inc., San Jose, California, USA.*
  E-mail: dongchul.p1@samsung.com
  ** D. Park is a corresponding author.*

1. In the future, we will consider hybrid storage including both SSDs and HDDs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

query (or some query operations) to the Smart SSD. The Smart SSD reads necessary data from flash chips to its internal device DRAM, and its internal processors execute the query (or query steps). Then, only results (expected to be much smaller than the raw data) return to the host machine through the relatively slow host interface. In this way, Smart SSDs change the traditional computing paradigm to "move code closer to data" (a.k.a near-data processing [7]).

Although the Smart SSD has a disadvantage that its (ARM) CPU is much less powerful (e.g., lower clock speed and higher memory access latency) than the host (Intel) CPU, it has two advantages which make it compelling for some I/O-intensive and computationally-simple applications. (1) Data access I/O time is much less because of the SSD's high internal bandwidth. (2) More importantly, energy consumption can be significantly reduced since ARM processors consume much less energy (4-5×) than host CPUs. The energy saving is dramatically important in today's data centers because energy takes 42% of the total monthly operating cost in data centers [8]; this explains why enterprises like Google and Facebook recently revealed plans to replace their Intel-based servers with ARM-based servers to save energy and cooling cost [9], [10].

Thus, Smart SSDs have gained significant attention from both industry and academia for achieving performance and energy gains. For instance, Samsung has demonstrated Smart SSD potential in big data analytics [11]. IBM has begun installing Smart SSDs in Blue Gene supercomputers [12]. The research community has also investigated the opportunities of Smart SSDs in areas such as computer systems [6], databases [5], and data mining [13].

However, it remains unknown whether search engine systems (e.g., Google) can benefit from Smart SSDs for performance and energy savings. It is unarguable that search engines are important because they are the primary means for finding relevant information for users, especially in the age of big data. To the best of our knowledge, this is the first article to evaluate the impact of using Smart SSDs with search engines. A preliminary version of this article that studied the impact of Smart SSD to list intersection appeared in an earlier paper [14]. In this version, we have the following significant new contributions:

- We systematically studied the impact of Smart SSDs to search engines from a holistic system perspective. We offloaded more operations including *intersection*, *ranked intersection*, *ranked union*, *difference*, and *ranked difference* (the previous version [14] only studied intersection).
- We integrated our design and implementation to a widely used open-source search engine – Apache Lucene[2] (the previous version [14] was independent of Lucene).
- We conducted more comprehensive experiments with both synthetic and real-world datasets to evaluate the performance and energy consumption (the previous version [14] only evaluated on synthetic datasets).

The rest of this paper is organized as follows. Section 2 provides an overview of the Samsung Smart SSDs we used and search engines. Section 3 presents the design decisions of what search engine operations can be offloaded to Smart SSDs. Section 4 provides the implementation details. Section 5 analyzes the performance and energy tradeoffs. Section 6 explains the experimental setup. Section 7 shows the experimental results. Section 8 discusses some related studies of this paper. Section 9 concludes the paper.

2. https://lucene.apache.org

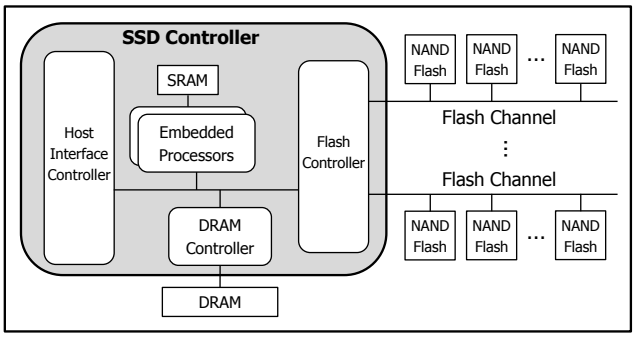


Fig. 2. Smart SSD hardware architecture

2 Background

This section presents the background of Smart SSDs (Section 2.1) and search engines (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 2 represents the hardware architecture of a Smart SSD which is similar to regular SSD hardware architecture. In general, an SSD is largely composed of NAND flash memory array, SSD controller, and (device) DRAM. The SSD controller has four main subcomponents: host interface controller, embedded processors, DRAM controller, and flash controller.

The host interface controller processes commands from host 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 [15]. Typically, the processor is a low-powered 32-bit processor such as an ARM series processor. Each processor has a tightly coupled memory (e.g., SRAM) and can access DRAM through a DRAM controller. The flash controller controls data transfer between flash memory and DRAM. Since the SRAM is even faster than DRAM, performance-critical codes or data are stored in the SRAM for more effective performance. On the other hand, typical or non-performance-critical codes or data are loaded in the DRAM. In Smart SSD, since a developer can utilize each memory space (i.e., SRAM and DRAM), performance optimization is totally up to the developer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

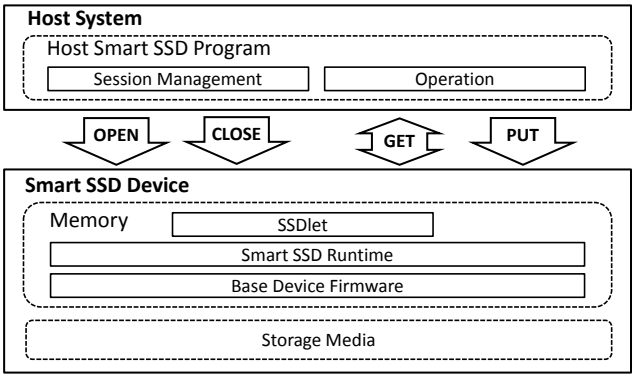


Fig. 3. Smart SSD software architecture

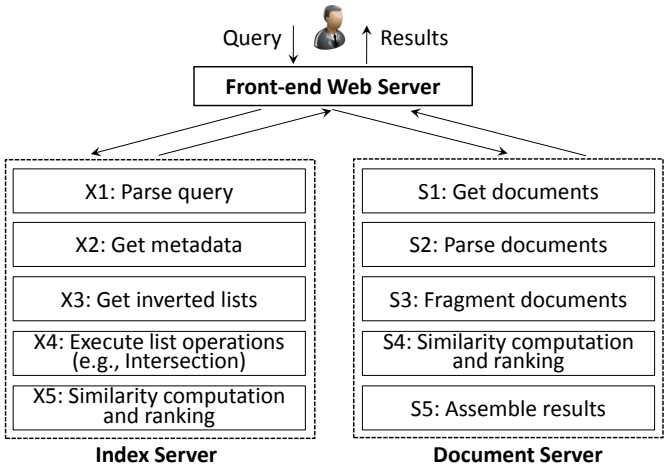


Fig. 4. Search engine architecture

The NAND flash memory package is the persistent storage media. 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, Smart SSDs need a software mechanism to define a set of protocols such that host machines and Smart SSDs can communicate with each other. Figure 3 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 has 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 Smart SSD runtime system (1) executes the SSDlet in an event-driven manner, (2) connects the device Smart SSD program with a base device firmware, and (3) implements the library of Smart SSD APIs. In addition, a base device firmware also implements normal storage device I/O operations (read and write).

This host Smart SSD program consists largely of two components: a session management component and an operation component. The session component manages Smart SSD application session lifetimes so that the host Smart SSD program can launch an SSDlet by opening a Smart SSD device session. 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 results are available. 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 Search Engines

Search engines (e.g., Google) provide an efficient solution for people to access information, especially in big data era. A search engine is a complex large-scale software system with many components working together to answer user queries. Figure 4 shows the system architecture of a typical search engine, which includes three types of servers: front-end web server, index server and document server [16], [17].

Front-end web server. The front-end web server interacts with end users, mainly to receive users' queries and return result pages. Upon receiving a query, depending on how the data is partitioned (e.g., term-based or document-based partitioning) [18]), it may possibly do some pre-processing before forwarding the query to an index server.

Index server. The index server stores the *inverted index* [19] to help answer queries efficiently. An inverted index is a fundamental data structure in any search engine [19], [20]. It can efficiently return the documents that contain a query term. It consists of two parts: *dictionary* and *posting file*.

- Dictionary. Each entry in the dictionary file has the format of ⟨`term`, `docFreq`, `addr`⟩, where `term` represents the term string, `docFreq` means document frequency, i.e., the number of documents containing the term, while `addr` records the file pointer to the actual inverted list in the posting file.
- Posting file. Each entry in the posting file is called an *inverted list* (or *posting list*), which has a collection of ⟨`docID`, `termFreq`, `pos`⟩ entries. Here, `docID` means the (artificial) identifier for the document containing the term, `termFreq` stores the how many times the term appears in the document, and `pos` records the positions for all the occurrences where the term appears in the document.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

The index server receives a query and returns the top-k most relevant document IDs, by going through several major steps (X1 to X5 in Figure 4).

- *Step X1*: parse the query into a parse tree;
- *Step X2*: get the metadata for each inverted list. The metadata is consulted for loading the inverted list from disks. The metadata can include offset and length (in bytes) that the list is stored, may also include document frequency;
- *Step X3*: get the inverted list from disk to host memory, via the host I/O interface. Today, the inverted index is too big to fit into main memory [17], [21] and thus we assume the inverted index is stored on disks [1], [2];
- *Step X4*: do list operations depending on the query type. The basic operations include list intersection, union and difference [22], [23];
- *Step X5*: for each qualified document, compute the similarity between the query and the document using a relevance model, e.g., the standard Okapi BM25 model [24]. Then, return the top-k most relevant document IDs to the web server for further processing.

Document server. The document server stores the actual documents (or web pages). It receives the query and a set of document IDs from the index server, then, generates query-specific snippets [25]. It also requires several query processing steps (S1 to S5 in Figure 4). *Step S1* is to get the documents from disk. *Step S2–S5* generate the query-specific snippets.

This is the first study for applying Smart SSD to search engines. We mainly focus on the index server, and leaving the document server for future work. Based on our experience, with SSD-based search engines, index server query processing takes more time than in the document server [1].

Moreover, we choose Apache Lucene as our experimental system, because Lucene is a well known open-source search engine and widely adopted in industry. E.g., LinkedIn [26] and Twitter [27] adopted Lucene in their search platforms, and Lucene is also integrated into Spark[3] for big data analytics [28].

3 Smart SSD for Search Engines

This section describes the system co-design of the Smart SSD and search engines. We first explore the design space (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 search engines (Section 3.2).

3.1 Design Space

The overall co-design research question is *what query processing logic could Smart SSDs cost-effectively execute?* To answer this, we must understand Smart SSD opportunities and limitations.

Smart SSD Opportunities. Executing I/O operations inside Smart SSDs is very fast for two reasons.

3. http://spark.apache.org/

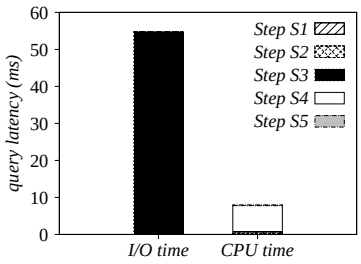


Fig. 5. Time breakdown of executing a typical real-world query by Lucene system running on regular SSDs

(1) SSD internal bandwidth is generally several times higher than the host I/O interface bandwidth [5], [29].
(2) The I/O latency inside Smart SSDs is very low compared to regular I/Os the host systems issue. A regular I/O operation (from flash chips to the host DRAM) must go through an entire, thick OS stack, which introduces significant overheads such as interrupt, context switch, and file system overheads. This OS software overhead becomes a crucial factor in SSDs due to their fast I/O (but it can be negligible with HDDs as their slow I/O dominates system performance) [30]. However, an internal SSD I/O operation (from flash chips to the internal SSD DRAM) is free from this OS software overhead.

Thus, it is very advantageous to execute *I/O-intensive* operations inside SSDs to leverage their high internal bandwidth and low I/O latency.

Smart SSD Limitations. 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) [5], [29];
(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 have sufficient 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 5 (please refer to Section 6 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 7 in Section 7 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 Lucene query processing steps (namely, step S1 – S5 in Figure 4) could execute inside SSDs to reduce both query latency and power consumption. We first make a rough analysis and then evaluate them with thorough experiments.

Step S1: Parse query. Parsing a query involves a number of CPU-intensive steps such as tokenization, stemming, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

lemmatization [22]. 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 about the on-disk inverted list of the term. Usually, 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 dictionary file stores metadata. 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 [22], we do not offload this step.

Step S3: Get inverted lists. Each inverted list contains a list of documents containing the same term. Upon receiving a query, the Smart SSD reads the inverted lists from the disk into the host memory[4], which is I/O-intensive. As Figure 5 shows, step S3 takes 87% of the time if Lucene runs on regular SSDs. Therefore, it is desirable to offload this step into Smart SSDs.

Step S4: Execute list operations. The main reason for loading inverted lists into 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 a search engine (e.g., Lucene), there are three common basic operations: 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 reduce 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: Intersection result size is usually much smaller than 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 [32]. Similar results are observed in our real dataset. Thus, executing intersection inside SSDs may be a smart choice as it can save remarkable 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 significant 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 [16]. (2) Big DRAM in the host side consumes too much energy because of the periodic memory refreshment [31].

5. http://www.google.com/advanced_search

TABLE 1
Design space

	Non-Ranked	Ranked
Intersection	YES	YES
Union	NO	YES
Difference	YES	YES

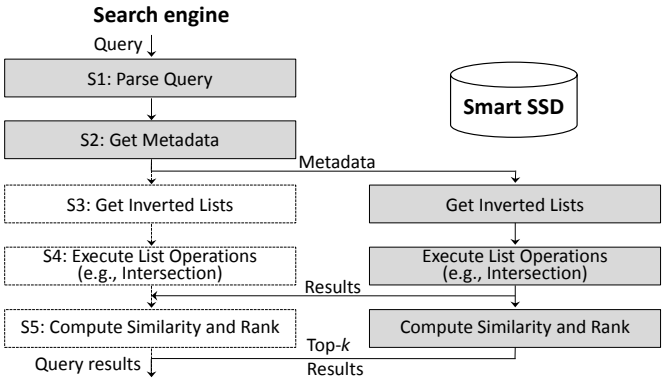


Fig. 6. Co-design architecture of a search engine and Smart SSDs

Step S5: Compute similarity and rank. After the aforementioned list operations complete, we can get a list of qualified documents. This step applies a ranking model to the qualified documents to determine the similarities between the query and these documents since 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 many 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, Smart SSDs execute step S5.

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 execute inside SSDs while step S5 executes the host. Ranked operations offloading means that all steps S3, S4, and S5 execute inside SSDs. In either case, steps S1 and S2 execute on the host.

3.2 System Co-Design Architecture

Figure 6 shows the co-design architecture of a search engine and the Smart SSD. It operates as follows. Assume only the intersection operation is offloaded. The host search engine is responsible for receiving queries. Upon receiving a query

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

$q(t_1, t_2, ..., t_u)$, where each t_i is a query term. It then 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 typical query's inverted lists. When all the u inverted lists are loaded into the device DRAM, the Smart SSD executes list intersection. Once it is done, the results are placed in an output buffer and ready to be returned to the host. The host search engine 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 search engine 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 perform step S5.

4 Implementation

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

Inverted index format. In this work, we follow a typical search engine's index storage format (also used in Lucene), where each inverted list entry consists of a document ID, document frequency, and positional information to support ranking and more complex queries.

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 must first load them into device memory. 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 [33], skip list based algorithm [22], hash based algorithm, divide and conquer based algorithm [34], adaptive algorithm [35], group based algorithm [32]. Among them, we implement the adaptive algorithm inside Smart SSDs for the following reasons: (1) The adaptive algorithm works well in theory and practice [32], [35]. (2) Lucene uses the adaptive algorithm for list intersection. For a fair comparison, we also adopted it inside Smart SSDs.

Algorithm 1 describes the adaptive algorithm [35]. 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 efficiently it identifies the successor of a pivot (Line 6). Our implementation mostly uses binary search. However, when two lists have a similar size, linear search can be even faster than binary search [32]. 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 switches 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 uses regular SSDs.

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
```

Handling large lists. Algorithm 1 requires that the device memory is big enough to store all the lists. Next, we explain how to handle large lists with two following solutions.

Solution 1 assumes device memory capacity M is bigger than twice the size of the shortest list, i.e., $M \geq 2|L_1|$. It works as follows: It loads the shortest list L_1 from Flash memory into device memory. For each page in L_2, it checks whether the item appears in L_1 and stores the results in an intermediate result buffer. Since the final intersection or difference size is smaller than $|L_1|$, it is sufficient if the device memory is twice as big as $|L_1|$. Then the results of L_1 and L_2 are intersected with L_3. Similarly, it accesses each page from L_3 and checks whether the item exists in the intermediate buffer pool. The process continues until L_k.

Solution 2 makes no assumptions. It works as follows: The main idea is to partition the shortest list L_1 evenly into m chunks such that each chunk fits in device memory. For example, assuming the shortest list L_1 is 200 MB, we can partition it into 10 chunks as an example. Here, each chunk is 20 MB which is small enough to store in device memory. Assume the m chunks are: $L_1^1, L_1^2, \cdots, L_1^m$. It then performs a list intersection (or difference) between each of the 10 L_1 chunks with the remaining $(k-1)$ lists. That is, $L_1^1 \cap L_2 \cap \cdots L_k$, $L_1^2 \cap L_2 \cap \cdots L_k$, $\cdots$, $L_1^m \cap L_2 \cap \cdots L_k$. For each intersection (i.e., $L_1^i \cap L_2 \cap \cdots L_k$), it loads the shortest list L_1^i to memory and follows solution 1 to finish each intersection. Finally, it merges the results and returns the results to users.

4.2 Union

We implemented the standard sort-merge based algorithm (also adopted in Lucene) for executing the union operation, see Algorithm 2. We note that it scans all inverted list

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

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
```

elements *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), 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 must compare 4 times (similarly for the rest). Thus, the performance depends on the number of lists and the result size. 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 7.5 describes how it affects system performance.

4.3 Difference

The difference operation is applicable for two lists, list A and B. $(A - B)$ finds all elements in A that are 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

Search engines return 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 follow a typical ranking model, BM25 [24], [36] – also implemented in Lucene – 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.

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. The SRAM size is small (32 KB). But the heap is also very small. It only contains top 10 results since search engine users usually care more about the first page of results. Thus, the heap required is less than 100 bytes.

5 Analysis

This section briefly analyzes the tradeoffs of offloading search engine operations within the Smart SSD in terms of both performance and energy.

Performance analysis. Let T and T' be the execution time of running an operation by the Smart SSD and host (running regular SSD) respectively. Then T can be divided into three pieces: (1) the I/O time of reading lists from flash chips to its device memory ($T_{I/O}$), (2) operation time (T_{CPU}), and (3) the time of sending back results to the host T_{send}. Thus, $T = T_{I/O} + T_{CPU} + T_{send}$. In contrast, T' consists of two parts: (1) the I/O time of reading lists from flash chips to the host memory ($T'_{I/O}$) and (2) operation time (T'_{CPU}). Thus, $T' = T'_{I/O} + T'_{CPU}$.

We now compare T and T'. Since the internal SSD bandwidth is higher than the host interface's bandwidth, then $T_{I/O} < T'_{I/O}$. However, $T_{CPU} > T'_{CPU}$ because of the low-clocked processors and high memory access latency within the Smart SSD. Thus, the Smart SSD can be faster or slower than the host depending on individual query. Note that the cost T_{send} of sending results back can be negligible since the intersection result size is usually orders of magnitude smaller than the original lists [37].

Energy analysis. Let E and E' be the energy consumed (in Joules) for executing operations using the Smart SSD and host (running regular SSD). Also, let P and P' be the power (in Watts) of the (ARM) CPU running within the Smart SSD and the (Intel) CPU running at the host. Then $E = T \times P$ and $E' = T' \times P'$. Since P is 3-4× less than P', it is more likely that E is less than E' (although the results depend on different queries).

6 Experimental Setup

This section presents the experimental setup in our platform. We describe the datasets in Section 6.1 and hardware/software setup in Section 6.2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

6.1 Datasets

For our system performance evaluation, we employ both real dataset and synthetic dataset.

6.1.1 Real Dataset

The real dataset, provided by Sogou (a Chinese commercial large-scale search engine company), consists of two parts: web data[6] and query log[7]. The data has been adopted in many previous studies [1]. The total web data is around 100GB. The web data contains a collection of more than 10 million web documents. The query log contains around 1 million real queries.

6.1.2 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 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), unless otherwise stated, in this paper list A represents the shorter list while list B represents the longer one. 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 longer list's size to the the shorter list's size (i.e., $\frac{|B|}{|A|}$). In practice, different lists significantly differ in size 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[8].

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 [32]. 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. 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.

TABLE 2
Parameter setup

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

6.2 Hardware and Software Setup

Our host machine is a commodity server with Intel i7 processor (3.40 GHz) and 8 GB memory running Windows 7. The Smart SSD has a 6Gb SAS interface and a 400 MHz ARM Cortex-R4 quad-core processor. This device connects to the host via a 6Gb SAS Host Bust Adapter (HBA). The host interface bandwidth (SAS) is about 550 MB/s. The internal bandwidth is about 2.2 GB/s. The Smart SSD has identical hardware specifications and base firmware compared to a regular SSD.

We adopt the C++ version (instead of Java version) of Lucene[9] to be compatible with the Smart SSD programming interface. We choose the stable 0.9.21 version.

We measure the power consumption via *WattsUp*[10] 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. Then, the energy is calculated by $(W_2 - W_1) \times t$.

7 Experimental Results

This section presents the evaluation results of offloading different query operations using the co-design system of Lucene and the Smart SSD. These operations are intersection (Section 7.1), ranked intersection (Section 7.2), difference (Section 7.3), ranked difference (Section 7.4), and ranked union (Section 7.5).

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

7.1 Intersection

Intersection is offloaded to Smart SSD. That is, both step S3 and S4 execute 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 Smart SSD's high internal bandwidth and low I/O latency. The energy saving comes from less data movement and the SSD's power-efficient processors.

For deeper analysis, we used a typical real-world query with two query terms[11]. Figure 7 shows the time breakdown

6. http://www.sogou.com/labs/dl/t-e.html
7. http://www.sogou.com/labs/dl/q-e.html
8. 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.
9. http://clucene.sourceforge.net
10. https://www.wattsupmeters.com
11. The number of entries in the inverted lists are 2,938 and 65,057 respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

TABLE 3
Intersection on real data

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

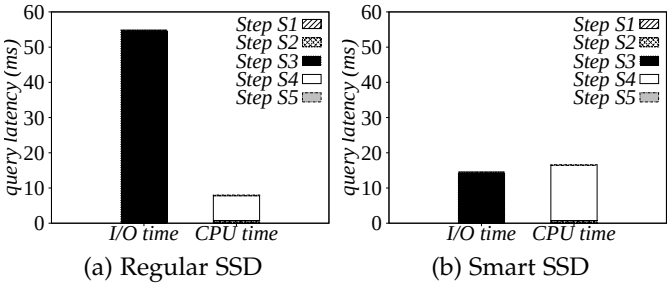


Fig. 7. Query latency breakdown of Lucene running on the regular SSD and the Smart SSD

of running 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. However, the Smart SSD increases 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 solely inside Smart SSDs. We consider steps S3 and S4 since other steps execute on the host side and the time is negligible (see CPU time in Figure 7). As Figure 8 shows, loading inverted lists from flash chips into 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.

Effect of varying list size. Figure 9 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 the additional required I/Os. On average, compared to regular SSDS, Smart SSDs reduce query latency by a factor of 2.5 while reducing energy by a factor of 7.8.

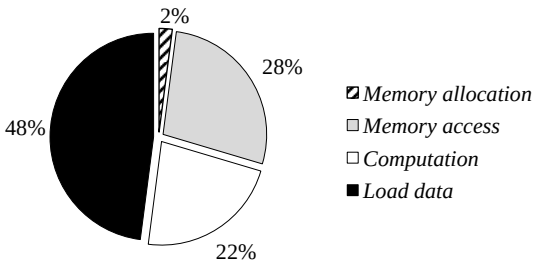


Fig. 8. Query latency breakdown of executing list intersection on Smart SSDs

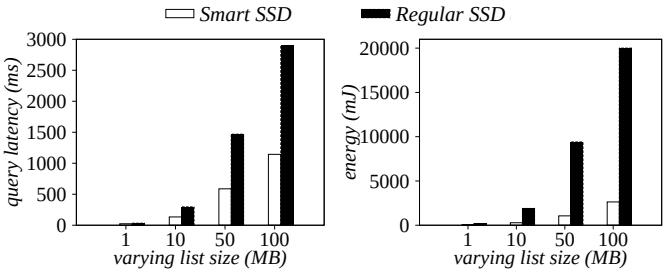


Fig. 9. Varying the list size (for intersection)

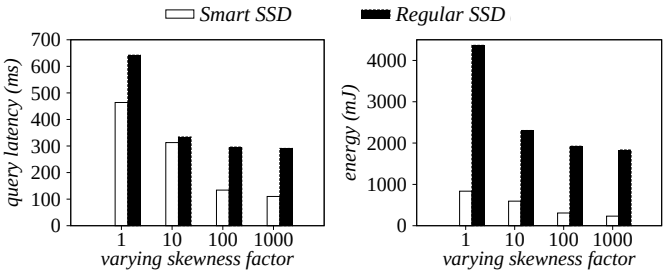


Fig. 10. Varying the list size skewness factor (for intersection)

Effect of varying the list size skewness factor. Figure 10 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 the skewness factor from 1 to 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 reduction.

Effect of varying intersection ratio. Figure 11 illustrates the impact of the intersection ratio r. It determines the result size which can have two system performance impacts: (1) data movement via the host I/O interface; and (2) ranking cost at the host side (since all qualified document IDs are evaluated for ranking). Surprisingly, we cannot find a clear correlation between performance and the intersection ratio (refer to Figure 11). 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). Figure 12 shows a clear impact of the intersection ratio r. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

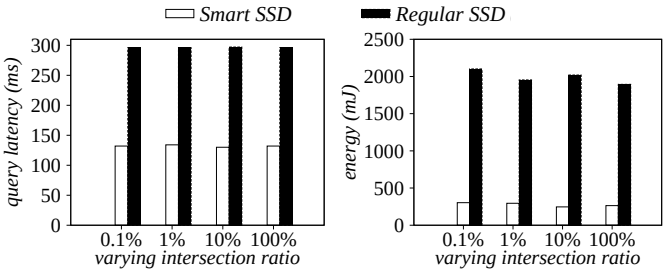


Fig. 11. Varying intersection ratio (for intersection)

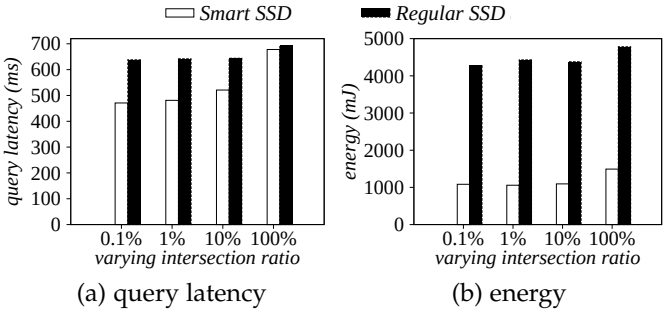


Fig. 12. Varying intersection ratio on equal-sized lists (for intersection)

is 100%), the Smart SSD only transfers one list, which saves around 50% of the data transfer.

Effect of varying number of lists. Figure 13 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[12] because of more data transfer. On average, Smart SSDs reduce query latency by 2.6× and energy by 9.5×.

In summary, 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.

12. When the number of lists u changes 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, it does not incur much overhead in query latency.

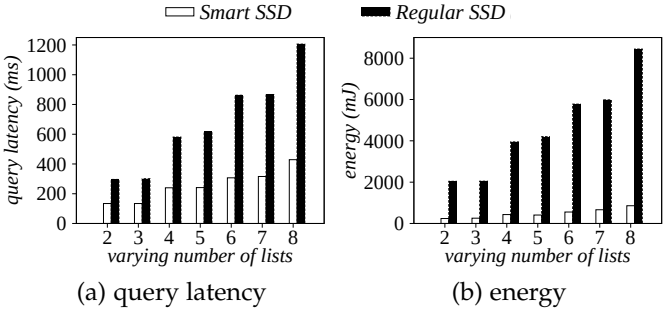


Fig. 13. Varying the number of lists (for intersection)

7.2 Ranked Intersection

Ranked intersection (i.e., steps S3, S4, and S5 in Figure 6) is offloaded to Smart SSD. Compared to offloading the 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 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 12).

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 intersections (i.e., Figure 9) 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 10) because the intersection size is not 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) are similar to Figure 11, where Smart SSDs outperform regular SSDs significantly.

Next, we conduct another experiment by setting the size of list A as the same as list B (both are 10 MB), see Figure 14. High intersection ratio leads to high intersection result size, while causing more ranking overhead. We vary the intersection ratio from 0.1% to 100%. The query latency (as well as energy consumption) increases as an intersection ratio increases. However, the increase is not that high compared to the non-ranked intersection offloading (in Figure 12). As an example, for the Smart SSD, when the intersection ratio changes from 10% to 100%, the latency increases by 65 ms in Figure 14, while the corresponding increase in Figure 12 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.

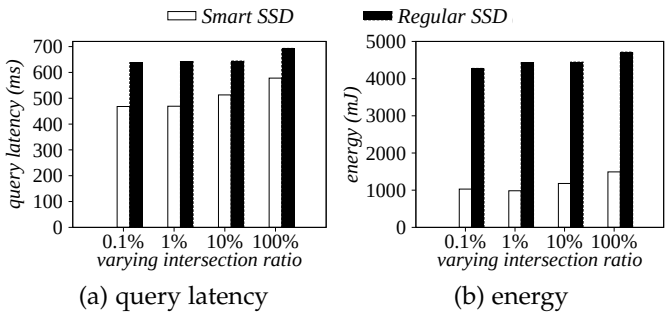


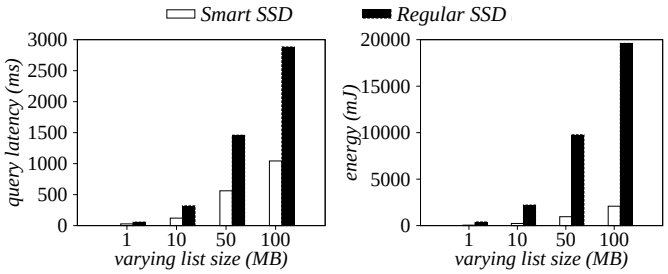
Fig. 14. Varying intersection ratio on equal-sized lists (for ranked intersection)

Effect of varying number of lists. The results are similar to the non-ranked version shown in Figure 13 because the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

TABLE 4
Difference on real data

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



(a) query latency (b) energy

Fig. 15. Varying the list size (for difference)

intersection size is very small (only 47), which will not make a major difference.

7.3 Difference

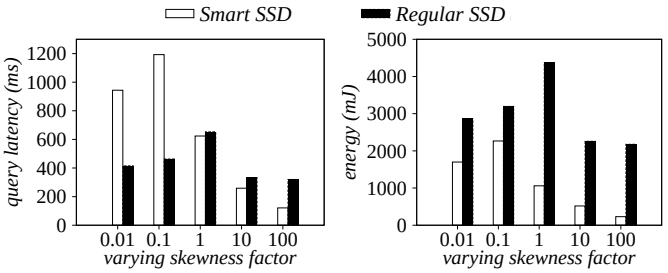
We offload the difference operation (i.e., steps S3 and S4 in Figure 6) to Smart SSDs, and the ranking executes on the host. When the difference operator is applied to two lists, it can be $(A - B)$ or $(B - A)$, where the list A is shorter than list B. As discussed in Section 3.1, Smart SSDs potentially benefit only the former case.

Results on real data. Table 4 shows the results with the real queries on 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. Specifically, they reduce query latency by 2.5× and energy consumption by 8.5×.

Effect of varying list size. Figure 15 plots the effect of varying list sizes, which affects 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) increases with longer lists. On average, the Smart SSD reduces query latency by 2.7× and energy consumption by 9.7×.

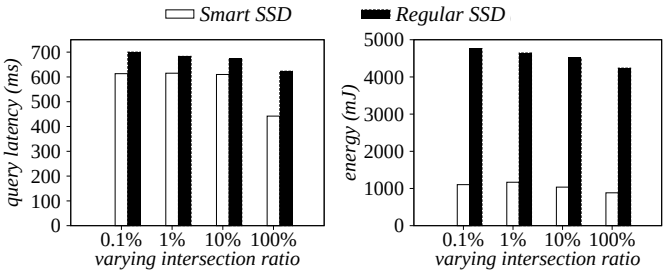
Effect of varying list size skewness factor. The skewness factor f is a key parameter in difference operation. Let A and B be two inverted lists. Unlike the default case, A is not necessarily shorter than B. It depends on the skewness factor f (still defined as $|B|/|A|$). We vary f from 0.01 to 100 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 16 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 $|A \cap B|$ 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,



(a) query latency (b) energy

Fig. 16. Varying the list size skewness factor (for difference)



(a) query latency (b) energy

Fig. 17. Varying intersection ratio (for difference)

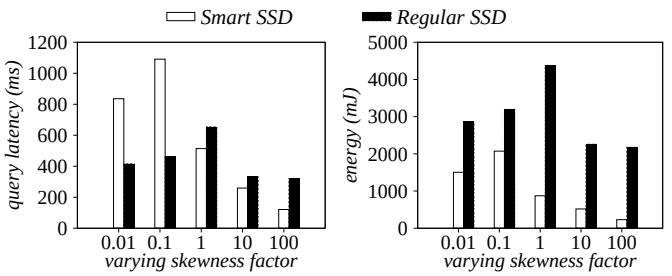
show better performance when $f \geq 1$ (i.e., $|A| \leq |B|$). That is because $|A - B| \leq |A| \leq (|A| + |B|)/2$. This means offloading $(A - B)$ can reduce data transfer at least 50%. (3) For Smart SSDs, the query latency increases when f goes from 0.01 to 0.1, but decreases afterwards when $f > 0.1$. (4) For regular SSDs, it shows a similar trend to Smart SSDs when f increases. However, unlike Smart SSDs, when f changes from 0.01 to 1, 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 12(a) when the intersection ratio is 100%. Both adopt a sort-merge based algorithm, and can save around 50% of the data transfers. Thus, echo the results in 12(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 (on the host side). Intuitively, a higher intersection ratio generates a smaller result size. In this case, we set the size of list A the same as list B[13]. As shown in Figure 17, 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

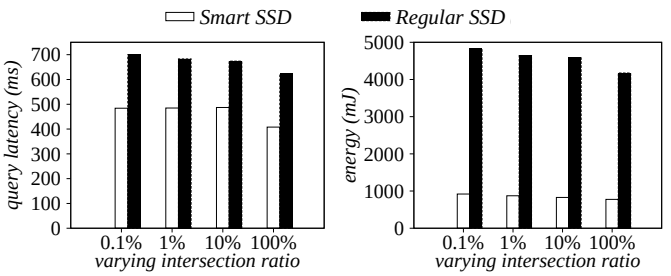
13. As discussed before, since there are no noticeable changes when the size of list A is 0.01% of list B, we omit the results.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers



(a) query latency (b) energy

Fig. 18. Varying the list size skewness factor (for ranked difference)



(a) query latency (b) energy

Fig. 19. Varying intersection ratio (for ranked difference)

lists with a smaller intersection ratio.

7.4 Ranked Difference

We offload the ranked difference (i.e., steps S3, S4, and S5 in Figure 6) to Smart SSDs. As previously discussed, compared to the non-ranked operation, offloading ranked operations can reduce the data transfer cost, but increases 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).

Results on real data. The results are similar to non-ranked difference (see Table 4) 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 15) 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 16), the Smart SSD has a longer query latency when $f = 0.01$ and $f = 0.1$ due to a large result size. If the ranking function is applied, the result size will be much smaller. Therefore, we can expect Smart SSD to exhibit 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 18). That is because the ranking function is applied only when the difference operation provides all the results. 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 [38], [39].

Effect of varying intersection ratio. Figure 19 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 17, the performance gap between Smart SSDs and regular SSDs is larger. That is due to less data transfer after ranking.

7.5 Ranked Union

We offload the ranked union (i.e., steps S3, S4, and S5 in Figure 6) to Smart SSDs.

TABLE 5
Ranked union on real data

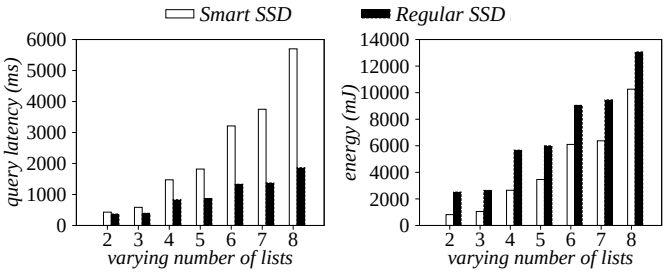
	Query latency (ms)	Energy (mJ)
Smart SSD	505	960
Regular SSD	299	2,033

Results on real data. Table 5 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 must 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, the Smart SSD can still reduce energy consumption by 2.1× 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 is as follows: the Smart SSD is around 1.2× slower 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 20 displays the impact of number of lists u in a query. The query latency gap between the Smart SSD and the regular SSD increases with higher numbers of lists. That is because each list must be accessed approximately $2u$ times.

Remark Because of too many memory accesses, it is not cost-effective to offload the ranked union to a Smart SSD. However, both union and ranking could be algorithmically combined for early termination [38], [39], such that we do not have to scan all results. We will explore the early pruning techniques in the future work.



(a) query latency (b) energy

Fig. 20. Varying the number of lists (for ranked union)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2608818, IEEE Transactions on Computers

8 Related Work

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

Early works on in-storage computing. As early as the 1970s, some initial works proposed to leverage specialized hardware (e.g., processor-per-track and processor-per-head) to improve query processing in storage devices (i.e., hard disks at that time). For example, CASSM [40] and RAP [41] follow the processor-per-track architecture to embed a processor in each track. The Ohio State Data Base Computer (DBC) [42] and SURE [43] follow the processor-per-head architecture to associate processing logic with each hard disk read/write head. However, none of the systems were commercially successful due to high design complexity and manufacturing cost.

Later works on HDD in-storage computing. In the late 1990s, when hard disk bandwidth continued to increase while powerful processor cost continued to drop, making it feasible to offload bulk computation to each individual disk. Researchers examined in-storage computing in terms of hard disks, e.g., active disk [44] or intelligent disk [45]. The goal was to offload application-specific query operators inside hard disks to reduce data movement. They examined active disk in database applications, 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 [46]. E.g., frequent sets mining, and edge detection. Although interesting, few real systems adopted the proposals, due to various reasons, including, limited hard disk bandwidth, computing power, and performance gains.

Recent works on SSD in-storage computing. Recently, with the advent of SSDs which have a potential to replace HDD, people began to reconsider in-storage computing in using SSD, i.e., Smart SSDs. An SSD offers many advantages over HDD, e.g., very high internal bandwidth and high computing power (because of the ARM processor technology). More importantly, executing code inside SSD can save significant energy because of reduced data movement and power-efficient embedded ARM processors. And, energy is very critical today. This makes the concept of in-storage computing on SSDs much more promising. Enterprises such as IBM started to install active SSDs to the Blue Gene supercomputer to leverage high internal SSD bandwidth [12]. In this way, computing power and storage device are closely integrated. Teradata's Extreme Performance Appliance [47] is also an example of combining SSDs and database functionalities. Another example is Oracle's Exadata [48], which also started to offload complex processing into storage servers.

SSD in-storage computing (or Smart SSD) is also attractive in academia. In the database area, Kim et al. investigated pushing down the database scan operator to SSD [49]. That work is based on simulation. Later on, Do et al. [5] built a Smart SSD prototype on real SSDs. They integrated Smart SSD with Microsoft SQL Server to by offloading two operators: scan and aggregation. Woods et al. also built a prototype of Smart SSD with FPGAs [50]. Their target is also for database systems, but with more operators, e.g., group-by. They integrated the prototype with a MySQL storage engine such as MyISAM and INNODB. In the data mining area, Bae et al. investigated offloading functionalities like k-means and Aprior to Smart SSD [13]. In data analytics area, De et al. propose to push down hash tables inside SSD [29]. There is also a study on offloading sorting [51].

Unlike existing works, our work investigates the potential benefit of Smart SSDs on web search area. To the best of our knowledge, this is the first work in this area.

9 Conclusion

SSD In-Storage Computing (Smart SSD) is a new computing paradigm to exploit SSD capabilities. This work is the first to introduce Smart SSDs to search engine systems. With a close collaboration with Samsung, we co-designed the Smart SSD with a popular open-source search engine – Apache Lucene. The main challenge is to determine what query processing operations in search engines can be cost-effectively offloaded to Smart SSDs. We demonstrated that: (1) The intersection operation (both non-ranked and ranked version) can be cost-effectively offloaded to Smart SSDs. In particular, Smart SSDs benefit much more 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 the union operation 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 real datasets.

We also observed 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 8). 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 Smart SSDs special characteristics (e.g., minimize expensive memory accesses).

In the future, an external interface change (i.e., PCIe-based SSD) may significantly affect Smart SSD performance gain, thereby reducing bandwidth gap between the host interface and internal SSD bandwidth. However, we can still expect remarkable energy saving, which now has become one of the most critical issues in data centers [9], [10].

References

[1] 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*, 2013, pp. 693–702.

[2] R. Ma, "Baidu distributed database," in *System Architect Conference China*, 2010.

[3] M. Miller, "Bing's new back-end: cosmos and tigers and scope," http://searchenginewatch.com/sew/news/2116057/bings-cosmos-tigers-scope-oh, 2011.

[4] T. Claburn, "Google plans to use intel SSD storage in servers," http://www.networkcomputing.com/storage/google-plans-to-use-intel-ssd-storage-in-servers/d/d-id/1067741, 2008.

[5] 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*, 2013, pp. 1221–1230.

[6] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: a user-programmable ssd," in *OSDI*, 2014, pp. 67–80.

[7] 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*, vol. 34, no. 4, pp. 36–42, 2014.

[8] W. Lang and J. M. Patel, "Energy management for mapreduce clusters," *PVLDB*, vol. 3, no. 1-2, pp. 129–139, Sep. 2010.

[9] M. Smolaks, "Google is testing qualcomm's 24-core arm chipset," http://www.datacenterdynamics.com/servers-storage/report-google-is-testing-qualcomms-24-core-arm-chipset/95681.fullarticle, 2016.

[10] R. Merritt, "Facebook likes wimpy cores, cpu subscriptions," http://www.eetimes.com/document.asp?doc_id=1261990, 2012.

[11] D. Park, J. Wang, and Y.-S. Kee, "In-storage computing for hadoop mapreduce framework: Challenges and possibilities," *IEEE Transactions on Computers*, vol. 3, no. 1-2, pp. 1–14, Aug. 2016.

[12] 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.

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

[14] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "Ssd in-storage computing for list intersection," in *DaMoN*, 2016.

[15] D. Park, B. Debnath, and D. Du, "CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns," in *SIGMETRICS*, 2010, pp. 365–366.

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

[17] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri, "Challenges on distributed web retrieval," in *ICDE*, 2007, pp. 6–20.

[18] J. Dean, "Challenges in building large-scale information retrieval systems: invited talk," in *WSDM*, 2009.

[19] J. Zobel and A. Moffat, "Inverted files for text search engines," *CSUR*, vol. 38, pp. 1–56, 2006.

[20] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *TODS*, vol. 23, no. 4, pp. 453–490, 1998.

[21] K. M. Risvik, T. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson, "Maguro, a system for indexing and searching over very large text collections," in *WSDM*, 2013, pp. 727–736.

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

[23] Google Corporation, "Google advanced search," http://www.google.com/advanced_search.

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

[25] A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams, "Fast generation of result snippets in web search," in *SIGIR*, 2007, pp. 127–134.

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

[27] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: real-time search at twitter," in *ICDE*, 2012, pp. 1360–1369.

[28] T. Shinagawa, "Spark search - in-memory, distributed search with lucene, spark, and tachyon," 2015.

[29] A. De, M. Gokhale, R. Gupta, and S. Swanson, "Minerva: accelerating data analysis in next-generation ssds," in *FCCM*, 2013, pp. 9–16.

[30] 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*, 2010, pp. 385–395.

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

[32] B. Ding and A. C. König, "Fast set intersection in memory," *PVLDB*, vol. 4, no. 4, pp. 255–266, 2011.

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

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

[35] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000, pp. 743–752.

[36] A. Singhal, "Modern information retrieval: a brief overview," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.

[37] J. S. Culpepper and A. Moffat, "Efficient set intersection for inverted indexing," *TOIS*, vol. 29, no. 1, pp. 1–25, 2010.

[38] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001, pp. 102–113.

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

[40] S. Y. W. Su and G. J. Lipovski, "Cassm: a cellular system for very large data bases," in *VLDB*, 1975, pp. 456–472.

[41] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik, "Performance evaluation of a relational associative processor," *TODS*, vol. 2, no. 2, pp. 175–195, 1977.

[42] K. Kannan, "The design of a mass memory for a database computer," in *ISCA*, 1978, pp. 44–51.

[43] H. Leilich, G. Stiege, and H. C. Zeidler, "A search processor for data base management systems," in *VLDB*, 1978, pp. 280–287.

[44] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," in *ASPLOS*, 1998, pp. 81–91.

[45] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, vol. 27, no. 3, pp. 42–52, 1998.

[46] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *VLDB*, 1998, pp. 62–73.

[47] Teradata Corporation, "Teradata extreme performance alliance," http://www.teradata.com/t/extreme-performance-appliance.

[48] Oracle Corporation, "Oracle exadata white paper," 2010.

[49] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee, "Fast, energy efficient scan inside flash memory," in *ADMS*, 2011, pp. 36–43.

[50] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *PVLDB*, vol. 7, no. 11, pp. 963–974, 2014.

[51] 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.

Jianguo Wang received the bachelor's degree from Zhengzhou University, China, in 2009 and the Mphil degree in computer science from The Hong Kong Polytechnic University in 2012. He is currently a Ph.D. student at the University of California, San Diego. His research interests include data management system and new computing hardware.

Dongchul Park is currently a research scientist in Memory Solutions Lab. (MSL) at Samsung Semiconductor Inc. in San Jose, California, USA. He received his Ph.D. in Computer Science and Engineering at the University of Minnesota–Twin Cities in 2012, and was a member of Center for Research in Intelligent Storage (CRIS) group under the advice of Professor David H.C. Du. His research interests focus on storage system design and applications including SSD, in-storage computing, Hadoop MapReduce, data center and cloud computing, data deduplication, key-value store, and shingled magnetic recording (SMR).