SSD In-Storage Computing for Search Engines

Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson

Abstract—SSD in-storage computing (called “Smart SSDs”) allows application-specific code to execute inside SSDs. This allows applications to take advantage of the high internal bandwidth that Smart SSDs provide. 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 many areas including database systems, data mining, and big data analytics. 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 to determine what operations in a search engine system can be cost-effectively offloaded to SSDs. To this end, we carefully identified the five commonly used search engine operations that could potentially benefit from Smart SSDs: intersection, rankedintersection, rankedunion, difference, and rankeddifference. With a close collaboration with Samsung, we implemented those operations into a real Samsung Smart SSD research prototype. Finally, we conducted extensive experiments to evaluate the performance and tradeoffs. Based on the results, we provide SSD vendors with suggestions on how to manufacture powerful Smart SSDs and application suggestions on how to fully utilize Smart SSD functionalities.

Index Terms—in-storage computing, smart SSD, search engine, storage system, modern hardware, data management

1 INTRODUCTION

T

raditional 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], [2]. 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], [3]. 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 [4], [5]. Thus, in this work, we primarily focus on a pure SSD-based system without HDDs.

In such a (conventional) computing architecture which includes CPU, main memory, and SSDs (Fig. 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 [6], [7], [8]. (1) A modern SSD is more than a storage device; it is also a computation unit. Fig. 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.

To fully exploit SSD potential, SSD in-storage computing (a.k.a Smart SSD) was recently proposed [6], [7], [8]. 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. Fig. 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 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 [9]).

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 (4-5x) than host CPUs. The energy saving is dramatically important in today’s data centers because energy takes 42 percent of the total monthly operating cost in data centers [10]; 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 [11], [12].

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 [13]. IBM has begun installing Smart SSDs in Blue Gene supercomputers [14]. Teradata’s Extreme Performance Appliance [15] is another example of integrating SSDs and database functionalities. Oracle’s Exadata [16] also started to offload complex query processing into storage servers. The research community has also investigated the opportunities of Smart SSDs in areas such as computer systems [7], databases [6], and data mining [17].

However, it remains unknown whether search engine systems can benefit from Smart SSDs for performance and energy savings. It is evident that search engines are important because they are the primary means for finding relevant information for users, especially in the age of big data. Thus, this work takes a first step to answer this question.

The challenge of this work is to determine what query processing operations in search engines can be cost-effectively offloaded to Smart SSDs. A basic principle is that operation output size should be smaller than input size in order to reduce data movement. Otherwise, search engine systems cannot fully benefit from Smart SSD capabilities.

Based on this principle, we carefully identified five commonly-used search engine operations that could potentially benefit from Smart SSDs: intersection, rankedintersection, rankedunion, difference, and rankeddifference. Our results have implications for both SSD vendors on how to manufacture powerful Smart SSDs and for applications on how to make full use of Smart SSDs functionalities.

We study the offloading of five commonly used operations into Smart SSDs: intersection, rankedintersection, rankedunion, difference, and rankeddifference. 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 work. Section 9 explores possible extensions of the work. Section 10 concludes the paper.

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
Fig. 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 [18]. Typically, the

---

1. A preliminary version of this article that studied the impact of Smart SSD to list intersection appeared in an earlier paper [8].
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. 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. Fig. 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 specific 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. Fig. 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 [19], [20].

Front-End Web Server. The front-end web server interacts with end users 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) [21], it may possibly do some pre-processing before forwarding the query to an index server.

Index Server. The index server stores the inverted index [22] to help answer queries efficiently. The inverted index is a fundamental data structure in search engines [22], [23]. 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 \((\text{term}, \text{docFreq}, \text{addr})\), where \(\text{term}\) represents the term string, \(\text{docFreq}\) means document frequency, i.e., the number of documents containing the term, while \(\text{addr}\) 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.

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 Fig. 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 [19], [24] and thus we assume the inverted index is stored on disks [1], [3];
- **Step X4**: do list operations depending on the query type. The basic operations include list intersection, union and difference [25];
- **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 [26]. 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.

This is the first study for applying Smart SSDs to search engines. We mainly focus on the index server, and leaving the document server for future work.

## 3 Smart SSDs for Search Engines

This section describes the offloading of search engine operations to the Smart SSD. 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.

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

1. **SSD internal bandwidth is generally several times higher than the host I/O interface bandwidth** [6], [8], [28]. In our Smart SSD, the internal bandwidth is around 1.5 GB/s, while the host I/O bandwidth (theoretically) is around 550 MB/s.

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) [29].** 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.

**Limitations of Smart SSDs.** Smart SSDs also have some limitations.

1. **Generally, Smart SSDs employ low-frequency processors (typically ARM series) to save energy and manufacturing cost.** As a result, computing capability is several times lower than host CPUs (e.g., Intel processor) [6], [8], [28].

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. Thus, Smart SSDs can reduce the I/O time at the expense of the increased CPU time so that a system with I/O time bottleneck can notably benefit from Smart SSDs. Table 1 summarizes the pros and cons.

We next analyze what query processing steps (namely, step S1-S5 in Fig. 4) could execute inside SSDs to reduce both query latency and power consumption. We first make a rough analysis and then evaluate them with experiments.

**Step S1: Parse Query.** Parsing a query involves a number of CPU-intensive steps such as tokenization, stemming, and lemmatization [25]. Thus, it is not profitable to offload the step S1.

**Step S2: Get Metadata.** The metadata is essentially a key-value pair where 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 ~ 2) I/O operations to obtain the metadata [25], 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, which is I/O-intensive. 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, 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, 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). 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., $|A \cap B| \ll |A| + |B|$. For example, in Bing search, for 76 percent of the queries, the intersection result size is two orders of magnitude smaller than the shortest inverted list involved [31]. 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 percent 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, $|A - B| = |A| - |A \cap B| \ll |A|$. 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 $|B - A| = |B| - |A \cap B| \approx |B| + |A|$. Consequently, we still consider the difference as a possible candidate for query offloading.

**Step S5: Compute Similarity and Rank.** After the aforementioned list operations 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, rankedintersection, rankedunion, difference, and rankeddifference (see Table 2). 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

Fig. 5 shows the co-design architecture of a search engine and Smart SSDs. We illustrate the working flow considering the intersection operation is offloaded. The host search engine is responsible for receiving queries. Upon receiving a query $q(t_1, t_2, \ldots, t_u)$, 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 the Smart SSD via the OPEN API that is necessary to perform list intersection, for example the addresses and lengths of the lists. The Smart SSD now starts to load the $u$ inverted lists in pages (of 8 KB) 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. We secure 150 MB DRAM space (but is extensible) for list intersection. Out of the 150 MB space, 256 KB is a special DRAM area called core memory, which can directly communicate with flash chips. That being said, every page will be transferred from flash chips to the core memory first, and copied to the (relatively) big device DRAM afterwards. 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

<table>
<thead>
<tr>
<th>Design Space</th>
<th>Non-Ranked</th>
<th>Ranked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Union</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Difference</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

---

2. Even though some search engines may cache inverted lists in the host memory, it may not completely solve the I/O problem. (1) The cache hit ratio is low even for big memories, typically 30 to 60 percent due to the cache invalidation caused by inverted index update [20]. (2) Big DRAM in the host side consumes too much energy because of the periodic memory refreshment [30].

host. The host search engine keeps monitoring the status of the Smart SSD in a heart-beat manner via the GET API. We set the polling interval to be 1 ms (for performance reasons). 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.

4.1 Intersection

Suppose there are \( u \) (\( u > 1 \)) inverted lists (i.e., \( L_1, \ldots, L_u \)) 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 many intersection algorithms in the literature, e.g., [25], [31], [32]. We chose the Svs algorithm [32] because (1) it is simple and does not need any preprocessing; and (2) it is effective in practice [31], [32].

The Svs algorithm works as follows [32], see Algorithm 1. It intersects the lists in increasing order of their sizes. Suppose \( |L_1| \leq |L_2| \leq \cdots \leq |L_u| \), then Svs intersects \( L_1 \) with \( L_2 \) first. It then intersects the results of \( L_1 \) and \( L_2 \) with \( L_3 \). The process continues until \( L_u \). For intersecting any two lists say \( L_1 \) and \( L_2 \), it scans every element \( e \) in \( L_1 \), check whether \( e \) appears in \( L_2 \) by finding its successor. If so, \( e \) is a result; otherwise, probe the next element in \( L_1 \). The successor finding is usually implemented in binary search such that many elements can be skipped; however, if the lists are of similar sizes, linear search is more efficient. In our experiments, we implement both, but only display the faster one as the results.

Algorithm 1. Svs (Small-versus-Small) [32]

\[
\begin{align*}
\textbf{Input:} & \; u \text{ lists } L_1, L_2, \ldots, L_u \text{ on an SSD } (|L_1| \leq |L_2| \leq \cdots \leq |L_u|) \\
\textbf{Output:} & \; L_1 \cap L_2 \cap \cdots \cap L_u \\
1: & \text{load all the } u \text{ lists } L_1, L_2, \ldots, L_u \text{ from the SSD to device DRAM} \\
2: & R \leftarrow L_1 \\
3: & \text{for } i \leftarrow 2 \to u \text{ do} \\
4: & \quad / \; R \leftarrow R \cap L_i \\
5: & \text{for } e \in R \text{ do} \\
6: & \quad \text{successor} \leftarrow L_i.\text{next}(e) \text{ /*smallest element } \geq e*/ \\
7: & \quad \text{if } \text{successor} = e \text{ then} \\
8: & \quad \text{put } e \text{ to } S \\
9: & R \leftarrow S \\
10: & \text{return } R
\end{align*}
\]

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.

1. The time complexity of intersection is \( O(|L_1| \cdot \log |L_2|) \) and \( O(|L_1| + |L_2|) \) using binary search and linear search, respectively. When \( |L_1| \approx |L_2| \), the latter cost is cheaper.

2. 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 reads the item, 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 \( L_u \).

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: \( L_1^1, L_1^2, \ldots, L_1^m \). It then performs a list intersection (or difference) between each of the 10 \( L_1 \) chunks with the remaining \((u-1)\) lists. That is, \( L_1^1 \cap L_2 \cap \cdots \cap L_u, L_1^2 \cap L_2 \cap \cdots \cap L_u, \ldots, L_1^m \cap L_2 \cap \cdots \cap L_u \). For each intersection (i.e., \( L_1^i \cap L_2 \cap \cdots \cap L_u \)), 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 for executing the union operation, see Algorithm 2.

Algorithm 2. Sort-Merge Based Union Algorithm

\[
\begin{align*}
\textbf{Input:} & \; u \text{ lists } L_1, L_2, \ldots, L_u \text{ on an SSD } (|L_1| \leq |L_2| \leq \cdots \leq |L_u|) \\
\textbf{Output:} & \; L_1 \cup L_2 \cup \cdots \cup L_u \\
1: & \text{load all the } u \text{ lists } L_1, L_2, \ldots, L_u \text{ from the SSD to device memory} \\
2: & \text{result set } R \leftarrow \emptyset \\
3: & \text{let } p_1 \text{ be a pointer for every list } L_i \text{ (initially } p_1 \leftarrow 0) \\
4: & \text{repeat} \\
5: & \quad \text{let } \text{minID} \text{ be the smallest element among all } L_i[p_i] \\
6: & \quad \text{advance } p_i \text{ by 1 if } L_i[p_i] = \text{minID} \\
7: & \quad \text{insert } \text{minID} \text{ to } R \\
8: & \text{until all the lists are exhausted} \\
9: & \text{return } R \\
\end{align*}
\]

We note that it scans all inverted list elements multiple times. More importantly, for every qualified document ID (Line 7), it needs at least \( 2u \) 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 the minimum value (Line 6). The total number of memory accesses can be estimated by: \( 2u \cdot |L_1 \cup L_2 \cup \cdots \cup L_u| \). For example, 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, 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.

Our implementation mostly uses binary search for element checking. However, as explained in Section 4.1, if the two lists are of similar sizes, 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 [26], [33] to determine the similarity between a query and a document. Let,

\[ qtf : \ \text{term's frequency in query} \ q \]  
\[ tf : \ \text{term's frequency in document} \ d \]  
\[ N : \ \text{total number of documents} \]  
\[ df : \ \text{number of documents that contain the term} \ t \]  
\[ dl : \ \text{document length} \]

Then,

\[
\text{Similarity}(q, d) = \sum_{t \in q} \left( \text{Similarity}(t, d) \times qtf \right) 
\]

\[
\text{Similarity}(t, d) = tf \cdot \left( \frac{1 + \ln \frac{N}{df + 1}}{dl} \right)^2 \cdot \left( \frac{1}{dl} \right)^2. 
\]

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 \( \text{Similarity}(t, d) \) 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., \( \text{Similarity}(t, d) \). 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.

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 parts:

1) Load data. Load each list individually from flash chips to memory. On Smart SSDs, the memory refers to the device DRAM while on regular SSDs, it refers to the host DRAM.

2) Memory copy. Copy data from the core memory to the device memory. This stage is only applicable to Smart SSDs, because data can only be loaded from flash chips to the small core memory (256 KB). This is an implementation issue of the Samsung Smart SSD, and we will remove such a constraint in the future. However, on regular SSDs, data can directly go from flash chips to the host DRAM, without such an extra data copy.

3) Execution. When all the lists are ready, depending on what query operations are offloaded, invoke different functions for query processing.

4) Send results. After the computation is finished, send results back to the host. This stage is only applicable to Smart SSDs.

In contrast, \( T' \) consists of two parts: (1) the I/O time of reading lists from flash chips to the host memory and (2) operation time. We now compare \( T \) and \( T' \). Since the internal SSD bandwidth is higher than the host interface’s bandwidth, the I/O time of the Smart SSD is smaller than that on the regular SSD. However, the CPU time is larger 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 \( T_{\text{send}} \) of sending results back can be generally negligible since the intersection result size is usually orders of magnitude smaller than the original lists in practice [32].

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 3-4x 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.

6.1 Datasets

In this work, we use synthetic data with varying parameters to understand the key factors to the overall performance and energy. Unless otherwise stated, when varying a particular parameter, we fix all other parameters as defaults.

List Size Skewness Factor. The skewness factor is defined as the ratio of longer list \( B \)'s size to the shorter list \( A \)'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 1,000 by default and vary the skewness factor from 1 to 1,000.

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 percent. For example, in Bing...
TABLE 3
Parameter Setup

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>List size skewness factor</td>
<td>1,000, 10, 1</td>
</tr>
<tr>
<td>Intersection ratio</td>
<td>0.1%, 1%, 10%, 100%</td>
</tr>
<tr>
<td>List size</td>
<td>1 MB, 10 MB, 50 MB, 100 MB</td>
</tr>
</tbody>
</table>

Fig. 6. Varying the list size (for intersection).

search, for 76 percent of the queries, the intersection size is two orders of magnitude smaller than the shortest inverted list [31]. We vary the intersection ratio from 1 to 100 percent.

**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 100 MB, and vary it from 1 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 has 10 to 100 s 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 3 shows a summary of the key parameters with defaults highlighted in bold.

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 6 Gb SAS interface and a 400 MHz ARM Cortex-R4 quad-core processor. This device connects to the host via a 6 Gb SAS Host Bus Adapter (HBA). The host interface bandwidth (SAS) is about 550 MB/s. The internal bandwidth is about 1.5 GB/s. The Smart SSD has identical hardware specifications and base firmware compared to a regular SSD.

We measure the power consumption via WattsUp® 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 \((W_2 - W_1) \times t\). Typically, Smart SSDs incur around 3-4× lower power than regular SSDs.

7 Experimental Results

This section presents the evaluation results of offloading different query operations to the Smart SSD. These operations are intersection (Section 7.1), rankedintersection (Section 7.2), difference (Section 7.3), rankeddifference (Section 7.4), and rankedunion (Section 7.5).

5. https://www.wattsupmeters.com

Fig. 7. Varying the list size skewness factor (for intersection).

**7.1 Intersection**

Intersection is offloaded to Smart SSD. That is, both step S3 and S4 execute inside the SSD.

*Effect of Varying List Size.* Fig. 6 evaluates the effect of list sizes, which affect the I/O time. It shows that the execution time and energy consumption increase with longer lists because of the additional required I/Os. We vary the size of list B from 1 to 100 MB (while the size of list A depends on the skewness factor whose default value is 1,000). For example, if B’s list size is 100 MB, then A’s list size is 0.1 MB. It shows that the Smart SSD can improve performance in most cases by up to 1.6× compared with the regular SSD, although it loses when the list sizes are too small due to the extra overhead (such as memory copy). More importantly, the Smart SSD can reduce energy consumption significantly by a factor of 3 to 5.1. That is because the ARM processor running inside the Smart SSD is very energy efficient.

*Effect of Varying the List Size Skewness Factor.* Fig. 7 shows the impact of the list size skewness factor \( f \), which can affect the performance of the intersection algorithm. Higher skewness gives more opportunities for skipping data. This favors the Smart SSD due to expensive memory accesses. We vary the skewness factor from 1 to 1,000 (while setting the sum of the two lists at 100 MB). The query latency (as well as energy) drops when \( f \) gets higher because the size of list A gets smaller. Note that, for the regular SSD, the query latency does not drop too much when \( f \) increases, because the I/O time is the bottleneck. In any case, it shows that Smart SSDs outperform regular SSDs in both latency and energy reduction.

*Effect of Varying Intersection Ratio.* Fig. 8 illustrates the impact of the intersection ratio \( r \). The intersection ratio determines the result size which can affect performance in the following two ways: (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). We set the size of B as 100 MB and vary \( r \) from 0.1 to 100 percent. Fig. 8 shows that the results (query latency and energy consumption) do not change much as \( r \) grows. That is because,
by default, list A includes around 3,277 entries (0.1 MB). Even when \( r \) is 100 percent, the result size is at most 3,277. This does not make much difference in both I/O time and ranking cost.

Then, we conduct another experiment by setting the size of list A the same as B (i.e., both are of 50 MB). Fig. 9 shows a clear impact of the intersection ratio \( r \). On the Smart SSD, the query latency and energy consumption increase as \( r \) gets higher, especially when \( r \) grows from 10 to 100 percent. That is because the intersection size increases and thus the I/O cost of sending results back to the host increases. In particular, when \( r = 100 \) percent, the Smart SSD is even worse than the regular SSD because of more data movement. But in most cases, the Smart SSD can still improve performance. And more importantly, it takes much less energy for the Smart SSD. Note that the performance on the regular SSD does not change much because the I/O time is the bottleneck. Note that the query latency decreases slightly when \( r \) increases from 10 to 100 percent because of the merge-based algorithms.

### 7.2 Ranked Intersection

Next we show the results of offloading the ranked intersection operation.

Compared to offloading the intersection-only operation (Section 7.1), offloading ranked intersection can save data transfer since only the top ranked results are returned. But it also increases the cost of ranking inside the device. However, there is not much difference when the result size is small.

Effect of Varying List Size. Fig. 10 shows the results, which are similar to Fig. 6 because the intersection size is small. As a result, the ranking cost and data transfer time do not make a noticeable difference.

Effect of Varying List Size Skewness Factor. Fig. 11 shows the results, which are similar to the non-ranked intersection version (i.e., Fig. 7), because the intersection size is not large.

**Effect of Varying Intersection Ratio.** The results of the default case (i.e., list A is 1,000× smaller than list B) are similar to Fig. 8, where Smart SSDs outperform regular SSDs.

Next, we conduct experiment by setting the size of list A as the same as list B (both are 50 MB), see Fig. 12. High intersection ratio leads to high intersection result size and more ranking overhead. We vary the intersection ratio from 0.1 to 100 percent (of \( |A| \)). The query latency (as well as energy consumption) increases as the intersection ratio increases. It is worth noting that when \( r = 100 \) percent, the Smart SSD still runs faster than the regular SSD because the data transfer time is very small as only top ranked results are returned to the host side.

### 7.3 Difference

We offload the difference operation (i.e., steps S3 and S4 in Fig. 5) 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.
Effect of Varying List Size. Fig. 13 plots the effect of varying list sizes. We vary the list sizes of list \( B \) from 1 to 100 MB (while the size of list \( A \) depends on the skewness factor). The query latency (as well as energy consumption) increases when the list sizes grow. It shows that, it is beneficial to offload the difference operation to the Smart SSD as it saves not only execution time but also energy consumption.

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, then we define \( f = \frac{|B|}{|A|} \). We vary \( f \) from 1 to 100 shows the corresponding sizes of list \( A \) and \( B \) (i.e., \( B \) is longer than \( A \)). Fig. 14 shows that it is beneficial to offload the difference operation to the Smart SSD in terms of both execution time and energy consumption.

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 \( B \). Fig. 15 shows that when the intersection ratio is small, e.g., less than 10 percent, the Smart SSD cannot improve query performance due to the high amount of data transferred as well as the high ranking cost. The Smart SSD favors the case when the intersection ratio is high such that the data transfer is small. But the Smart SSD can reduce energy consumption even when the intersection ratio is small, due to the energy-efficient processors.

7.4 Ranked Difference

We offload the ranked difference (i.e., steps S3, S4, and S5 in Fig. 5) 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.

We omit the results of varying list size and skewness factor because the results are similar to the non-ranked version (see Figs. 13 and 14) because the data transfer is small.

Effect of Varying Intersection Ratio. Fig. 16 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. Fig. 16 shows that the Smart SSD can improve performance no matter whether the intersection ratio is small or big because the data transfer is small.

7.5 Ranked Union

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

Effect of Varying List Size. Fig. 17 shows the impact of list size. We vary the list size of \( B \) from 1 to 100 MB. It shows that the query latency as well as energy consumption increases as the list size increases. And the Smart SSD is able to improve performance and reduce energy by a large margin.
Effect of Varying List Size Skewness Factor. Fig. 18 shows the impact of the skewness factor. In all cases, the Smart SSD is faster than the regular SSD and it also consumes less energy.

Effect of Varying Intersection Ratio. Fig. 19 evaluates the impact of intersection ratio \( r \) to system performance with two equal-sized lists (50 MB). It shows that except when the intersection ratio is 100 percent, the Smart SSD is still faster than the regular SSD. When \( r \) is 100 percent, the Smart SSD incurs more execution time due to high amount of data transfer. But in all cases, the Smart SSD can still reduce the energy consumption significantly due to energy-efficient processors running inside the Smart SSD.

Effect of Varying Number of Lists. Fig. 20 displays the impact of number of lists \( u \) in a query. We set the size of each list to be around 20 MB. For example, when \( u = 2 \), both lists are 20 MB. When \( u = 5 \), those five lists are 20 MB individually. The figure shows that the query latency and energy consumption increase as \( u \) increases because of more data accessed. In particular, when \( u > 2 \), the Smart SSD is slower than the regular SSD. That is because each list must be accessed approximately \( 2u \) times. Thus, when \( u \) is big, the Smart SSD needs to access much data. Note that, the memory access is slower within the Smart SSD compared to the host DRAM. But the Smart SSD can still reduce the energy consumption due to energy-efficient processors.

Fig. 20 indicates that it is necessary to improve the performance of the device memory access speed within the Smart SSD. On the one hand, we can solve it from the hardware point of view, e.g., by including more caches within the device. The other hand, we can also design more efficient algorithms to reduce memory accesses. Our current implementation solves the ranking problem only if all the union results are available, then scores every qualified document. However, both union and ranking could be algorithmically combined for early termination [34], [35]. This means we do not need to scan all the union results. Thus, it is possible to 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 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 [36] and RAP [37] follow the processor-per-track architecture to embed a processor in each track. The Ohio State Data Base Computer (DBC) [38] and SURE [39] 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 [40] or intelligent disk [41]. 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 [42]. For example, 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 [14]. In this way, computing power and storage device are closely integrated. Teradata’s Extreme Performance Appliance [15] is also an example of combining SSDs and database functionalities. Another example is Oracle’s Exadata [16], which also started to offload complex processing into storage servers.

SSD in-storage computing (or Smart SSD) also attracted academia. In the database area, Kim et al. investigated pushing down the database scan operator to SSD [43]. That work is based on simulation. Later on, Do et al. [6] 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. also built a Smart SSD prototype with FPGAs [44]. Their target is also for database systems (MySQL), but with more operators, e.g., group-by. In the data mining area, Bae et al. investigated offloading functionalities like k-means and Apriori to Smart SSD [17]. In the system area, Seshadri et al. built the Willow system [7] and studied the offloading of many applications, e.g., file system, transactional processing. Tseng et al. built the Morpheus system to offload object deserialization [46]. Recently, Park et al. built a real SSD-based Smart SSD prototype. They integrated Smart SSD with Hadoop MapReduce framework, thereby offloading Mapper to SSDs [13]. Unlike existing studies, this work investigates the potential benefit of Smart SSDs on web search area.

9 DISCUSSION

This is the first work investigating Smart SSDs in search engines. Thus, there are many possible extensions of the work. In this section, we name a few.

1) Compressed list intersection. This work only considers query processing on non-compressed lists. But many search engines store compressed lists to reduce the inverted index size [25], [47]. It would be interesting to understand whether Smart SSDs can gain query processing benefits over compressed data. Compression is unfriendly to Smart SSDs because decompression takes some CPU cycles which imposes a burden on the computationally weak cores running inside SSDs. One could try the following techniques to improve the performance: (1) Add skip pointers to the list such that intersection can be carried out efficiently without fully decompressing the list; (2) Design new compression techniques such that list intersection can be done directly on compressed lists without decompression.

2) The impact of cache. Many search engines cache popular lists in DRAM to reduce I/O traffic. Thus, host DRAM caches (or buffers) can influence Smart SSD benefits. With continually increasing available DRAM, Smart SSD benefits can be reduced. But as long as cache misses occur, Smart SSDs can be beneficial.

3) Smart SSD friendly algorithm designs. We shall develop Smart SSD-friendly algorithms. Existing algorithms are optimized for host search engines with regular SSDs, which may not be suitable for Smart SSDs. For example, compression may not be advantageous for Smart SSDs because today’s Smart SSDs read data rapidly but decompress data slowly. Another example is a host DRAM inverted list cache, which is used to reduce I/O traffic. However, if a Smart SSD’s internal DRAM is big enough, combined with its rapid read I/O access, the performance benefit of a host DRAM inverted list cache significantly decreases.

4) Software and hardware co-design. We shall co-design both search engine systems and Smart SSDs to process queries quickly and energy-efficiently because each side has strengths and weaknesses. However, there are many challenging issues to address. For example, (1) how to dispatch queries (or query steps) to each side, (2) what are the best algorithms or designs (e.g., compression, top-k evaluation, and cache) for each side.

5) Concurrent requests. Our current Smart SSDs do not support multiple concurrent requests. That said, there is only one query executed at any time. This could affect the system throughput. In the future, we plan to develop next-generation Smart SSDs that support multiple simultaneous requests.

6) Multiple Smart SSDs. In the future, we will also consider multiple Smart SSDs, which will be very challenging. That said, the involvement of host systems should be minimized. Otherwise, the advantages of Smart SSDs are squandered. Thus, every SSD should be able to autonomously communicate with other SSDs. We will develop protocols to enable that in the future.

10 CONCLUSION

Executing programs within Smart SSDs is a new computing paradigm to make full use of the SSDs’ hardware capabilities. Rather than transfer data to main memory where a host CPU processes it, Smart SSDs execute code inside SSDs directly, exploiting the high internal bandwidth SSDs provide. This work studies the offloading of search engine operations to Smart SSDs. We demonstrate that:

1) The intersection operation (both non-ranked and ranked version) can be cost-effectively offloaded to Smart SSDs.

2) The difference operation \( A - B \) (both non-ranked and ranked) can be a good candidate for offloading when \(|A| \leq |B|\).

3) The ranked union operation is also suitable for Smart SSDs when the number of lists is small (e.g., 2).

Our results also have implications for SSD vendors to design powerful Smart SSDs and for applications to fully utilize the capabilities of Smart SSDs.

1) On the SSD vendor side, while it is always good to have faster processors and high internal bandwidths, it is urgent to improve the memory access speed. Sometimes, the memory access time can even dominate the total query processing time. To do so, SSD vendors can introduce more caches (both data-cache and instruction-cache) to SSDs.

2) On the application side, clearly not all applications can benefit from Smart SSDs. They have to be
I/O-intensive and computationally-simple. Also, the number of memory accesses should be small and the output size should be (much) smaller than the input size.

In the future, an external interface change (i.e., PCIe-based SSD) may significantly affect Smart SSD performance gain, thereby reducing bandwidth gap (about 1.5x) 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 [11].

ACKNOWLEDGMENTS

The authors gratefully acknowledge the proofreading suggestions and valuable comments of David Schwaderer in Samsung Semiconductor Inc., USA. Dongchul Park is the corresponding author.

REFERENCES


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 working toward the PhD degree at the University of California, San Diego. His research interests include data management system and new computing hardware.

Dongchul Park received the PhD degree in computer science and engineering from the University of Minnesota—Twin Cities, in 2012. He is currently a research scientist in Memory Solutions Lab, Samsung Semiconductor Inc., San Jose, California. He 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).

Yannis Papakonstantinou received the diploma of electrical engineering from the National Technical University of Athens, and the MS and PhD degrees in computer science from Stanford University, in 1997. He is a professor of computer science and engineering with the University of California, San Diego. His research includes the intersection of data management technologies and the web, where he has published more than 90 research articles that have received more than 12,500 citations, according to Google Scholar. A common theme of his research is the extension of database platforms and query processors beyond centralized relational databases and into semi-structured databases, integrated views of distributed databases and web services, textual data and queries involving keyword search, and most recently spatiotemporal sensor data. Another theme is the appropriate utilization of modern hardware for advanced analytics. He has given multiple tutorials and invited talks, has served on journal editorial boards and has chaired and participated in program committees for many international conferences and workshops. He is a co-director and teaches for UCSD’s Master of Advanced Studies in Data Science. He received an NSF CAREER award for his work on data integration.

Steven Swanson received the undergraduate degree from the University of Puget Sound and the PhD degree from the University of Washington, in 2006. He is a professor in the Department of Computer Science and Engineering, University of California, San Diego, and the director with the Non-volatile Systems Laboratory. His research interests include the systems, architecture, security, and reliability issues surrounding non-volatile and solid-state memories. He also co-leads projects to develop low-power co-processors for irregular applications and to devise software techniques for using multiple processors to speed up single-threaded computations. In previous lives he has worked on scalable dataflow architectures, ubiquitous computing, and simultaneous multithreading.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.