Cache Design of SSD-Based Search Engine Architectures: An Experimental Study

JIANGUO WANG, University of California, San Diego
ERIC LO and MAN LUNG YIU, Hong Kong Polytechnic University
JIANCONG TONG, GANG WANG, and XIAOGUANG LIU, Nankai University

Caching is an important optimization in search engine architectures. Existing caching techniques for search engine optimization are mostly biased towards the reduction of random accesses to disks, because random accesses are known to be much more expensive than sequential accesses in traditional magnetic hard disk drive (HDD). Recently, solid-state drive (SSD) has emerged as a new kind of secondary storage medium, and some search engines like Baidu have already used SSD to completely replace HDD in their infrastructure. One notable property of SSD is that its random access latency is comparable to its sequential access latency. Therefore, the use of SSDs to replace HDDs in a search engine infrastructure may void the cache management of existing search engines. In this article, we carry out a series of empirical experiments to study the impact of SSD on search engine cache management. Based on the results, we give insights to practitioners and researchers on how to adapt the infrastructure and caching policies for SSD-based search engines.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: Search engine, solid-state drive, cache, query processing

ACM Reference Format:
Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, Xiaoguang Liu. 2014. Cache design of SSD-based search engine architectures: An experimental study. ACM Trans. Inf. Syst. 32, 4, Article 21 (October 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2661629

1. INTRODUCTION

Caching is an important optimization in search engine architectures. Over the years, many caching techniques have been developed and used in search engines [Markatos 2001; Baeza-Yates and Saint-Jean 2003; Baeza-Yates et al. 2007b, 2008; Turpin et al. 2007; Gan and Suel 2009; Altingovde et al. 2009; Ozcan et al. 2008, 2011a, 2011b; Saraiva et al. 2001; Ceccarelli et al. 2011]. The primary goal of caching is to reduce *query latency*. To that end, search engines commonly dedicate portions of servers' memory to cache certain *query results* [Markatos 2001; Ozcan et al. 2008; Gan and

This article is a substantially enhanced version of a paper presented in *Proceedings of the 36th Annual ACM Conference on Research and Development in Information Retrieval (SIGIR'13)* [Wang et al. 2013].
This work is partially supported by the Research Grants Council of Hong Kong (GRF PolyU 525009, 521012, 520413, 530212), NSFC of China (60903028, 61070014), and Key Projects in the Tianjin Science & Technology Pillar Program (11ZCKFGX01100).
Authors' addresses: J. Wang (corresponding author), Department of Computer Science and Engineering, University of California, San Diego, CA; email: csjgwang@cs.ucsd.edu; E. Lo and M. L. Yiu, Department of Computing, The Hong Kong Polytechnic University; emails: {ericlo, csmlyiu}@comp.polyu.edu.hk; J. Tong, G. Wang, and X. Liu, Nankai-Baidu Joint Lab, Nan Kai University; emails: {lingfenghx, wgzwpzy, liuxguang}@gmail.com.

DOI: http://dx.doi.org/10.1145/2661629

Suel 2009], *posting lists* [Baeza-Yates and Saint-Jean 2003, Baeza-Yates et al. 2007b, 2008], *documents* [Turpin et al. 2007], and *snippets* [Ceccarelli et al. 2011] in order to avoid the excessive disk access and computation.

Recently, solid-state drive (SSD) has emerged as a new kind of secondary storage medium. SSD offers a number of benefits over magnetic hard disk drive (HDD). For example, random reads in SSD are one to two orders of magnitude faster than in HDD [Graefe 2009; Lee et al. 2008; Chen et al. 2009; Narayanan et al. 2009]. In addition, SSD is much more energy efficient than HDD (around 2.5% of HDD energy consumption [Seo et al. 2008; Chen et al. 2011]). Now, SSD is getting cheaper and cheaper (e.g., it dropped from $40/GB in 2007 to $1/GB in 2012 [Flexstar Technology 2012]). Therefore, SSD has been employed in many industrial settings including MySpace[1], Facebook[2], and Microsoft Azure.[3] Baidu, the largest search engine in China, has used SSD to completely replace HDD as its main storage as of 2011 [Ma 2010].

The growing trend of using SSD to replace HDD has raised an interesting research question specific to the information retrieval community: what is the impact of SSD on the cache management of a search engine? Figure 1 shows the average cost (latency) of a random read and a sequential read on two brands of SSDs and two brands of HDDs. It shows that the cost of a random read is 100 to 130 times of a sequential read in HDD. Due to the wide speed gap between random read and sequential read in, HDD, the benefit of a cache hit, traditionally, has been largely attributed to the saving of the expensive random read operations. In other words, although a cache hit of a data item could save the random read of seeking the item and the subsequent sequential reads of the data when the item spans more than one block [Trotman 2003], the saving of those sequential reads has been traditionally regarded as less noticeable, because the random seek operation usually dominates the data retrieval time.

Since a random read is much more expensive than a sequential read in HDD, most traditional caching techniques have been designed to minimize random reads. The technology landscape, however, changes if SSD is used to replace HDD. Specifically, Figure 1 shows that the cost of a random read is only about twice that of a sequential read in SSD. As such, in an SSD-based search engine infrastructure, the benefit of a cache hit should now attribute to the saving of both random read and subsequent sequential reads for data items that are larger than one block. Furthermore, since both random reads and sequential reads are fast on SSD while query processing in modern search engines involves several CPU-intensive steps (e.g., query result ranking [Turtle and Flood 1995; Broder et al. 2003] and snippet generations [Turpin et al. 2007; Tombros and Sanderson 1998]), we expect that SSD would yield the following impacts on the cache management of search engines.

(1) Caching techniques should now target to minimize both random reads and sequential reads.
(2) The size of the data item and the CPU cost of the other computational components (e.g., snippet generation) play a more important role in the effectiveness of various types of caching policies.

Therefore, the first part of this article (Section 4) is devoted to carrying out a large-scale experimental study that evaluates the impact of SSD on the effectiveness of various caching policies on prominent cache types found in a typical search engine architecture. We note that the traditional metric *cache hit ratio* for evaluating caching effectiveness is inadequate in this study—in the past, the effectiveness of a caching

[1]http://www.fusionio.com/press/MySpace-Uses-Fusion.
[2]http://www.facebook.com/note.php?note_id=388112370932.
[3]http://www.storagelook.com/microsoft-azure-ocz-ssds.

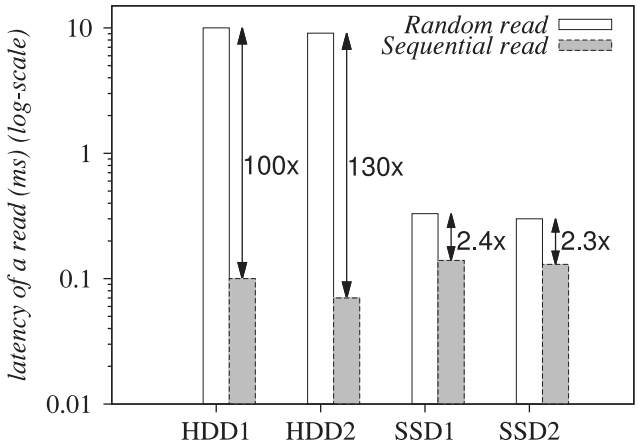


Fig. 1. Read latency on two HDDs and two SSDs. Each read fetches 4KB. The OS buffer is bypassed. Both HDDs are 7200rpm, with seek time ranging from 8ms to 10ms, maximum data transfer around 200MB/s. Both SSDs have maximum data transfer rates of 550MB/s (sequential read) and 160MB/s (random read).

policy could be measured by the cache hit ratio because it is a reliable reflection of the actual query latency: a cache hit can save the retrieval of a data item from disk, and the latency improvement is roughly the same for a large data item and a small data item because both require one random read, which dominates the time of retrieving the whole data item from disk. With SSD replacing HDD, the cache hit ratio is no longer a reliable reflection of the actual query latency because a larger data item being found in the cache yields a higher query latency improvement over a smaller data item (a cache hit for a larger item can save a number of time-significant sequential reads). In fact, the cache hit ratio is not an adequate measure whenever the cache-miss costs are not uniform [Altingovde et al. 2009; Gan and Suel 2009; Ozcan et al. 2011a; Marin et al. 2010]. In our study, therefore, one caching policy may be more effective than another even though they achieve the same cache hit ratio if one generally caches some larger data items. To complement the inadequacy of cache hit ratio as the metric, our study is based on the real replays of a million queries on an SSD-enabled search engine architecture, and our reported findings are mainly based on the actual query latency.

As we will present later, the empirical results in the first part of this article do suggest that practitioners who have used or are planning to use SSD to replace HDD in their search engine infrastructures should revise their caching policies. Therefore, the second part of this article (Section 5) aims to provide insights for practitioners and researchers on how to adapt the infrastructure and caching policies for SSD-based search engines.

To the best of our knowledge, this is the first article to evaluate the impact of SSD on search engine cache management and provide insights to the redesign of caching policies for SSD-based search engine infrastructures. A preliminary version of this article appeared in Wang et al. [2013]. In this version, we have the following new contributions:

(1) We refresh all empirical results using a new setting. Specifically, in the early version of this article [Wang et al. 2013], when evaluating a particular type of cache in, say, web server, caches in other servers (e.g., index server) were disabled. In practice, however, all cache types in all types of servers should be enabled together. Therefore, in this version, we repeat all experiments using the all-cache-enabled setting and present the new results.

(2) We strengthen the results by adding a new dataset to our experiments. Specifically, in the early version of this article [Wang et al. 2013], our results were based on replaying Sogou's (a Chinese commercial search engine) query logs on Sogou's web data. In this version, we strengthen our results by replaying the AOL query logs (Table II) on ClueWeb dataset (Table III).
(3) We add a new empirical study to identify the performance bottleneck when a search engine infrastructure switches to use SSD as the core storage (Section 5.1). Based on that result, we then study which type of (new) cache can be incorporated into an SSD-based search engine infrastructure (Section 5.2) and how to allocate the memory among the new and existing cache types (Section 5.3) in order to mitigate the new bottleneck. Empirically results show that our idea can bring 66.7% to 69.9% of query latency speedup.
(4) We expand the discussion of the related work in this version, which gives a more thorough overview of system research related to SSD (Section 6).

The remainder of this article is organized as follows. Section 2 provides an overview of contemporary search engine architectures and the prominent types of caches involved. Section 3 details the experimental setting. Section 4 presents our large-scale experimental study that evaluates the impact of SSD on the effectiveness of various caching policies. Section 5 presents our new empirical study that identifies the main latency bottleneck in SSD-based search engine infrastructures and suggests an effective cache type for such infrastructures. Section 6 discusses the related studies, and Section 7 summarizes the main findings of this study.

2. BACKGROUND AND PRELIMINARY

In this section, we first give an overview of the architecture of the state-of-the-art Web search engines and the major cache types involved (Section 2.1). Then we give a review of the prominent types of caching policies used in Web search engine cache management (Section 2.2).

2.1. Search Engine Architecture

The architecture of a typical large-scale search engine [Barroso et al. 2003; Dean 2009; Baeza-Yates et al. 2007a] is shown in Figure 2. A search engine is typically composed of three sets of servers: *Web servers* (WS), *index servers* (IS), and *document servers* (DS).

Web Servers. Web servers are the front-ends for interacting with end users and for coordinating the whole process of query evaluation. Upon receiving a user's query q with n terms $t_1, t_2, \ldots, t_n$ [STEP ①], the Web server that is in charge of q checks whether q is in its in-memory *query result cache* (QRC) [Markatos 2001; Saraiva et al. 2001; Fagni et al. 2006; Baeza-Yates et al. 2007b; Ozcan et al. 2008, 2011a, 2011b; Gan and Suel 2009]. The QRC maintains query results of some past queries. If the results of q are found in the QRC (i.e., a cache hit), the server returns the cached results of q to the user directly. Generally, query results are roughly of the same size, and a query result consists of (i) the title, (ii) the URL, and (iii) the snippet [Turpin et al. 2007; Tombros and Sanderson 1998] (i.e., an extract of the document with terms in q being highlighted) of the top-k ranked results related to q (where k is a system parameter, e.g., 10 [Fagni et al. 2006; Altingovde et al. 2011; Ceccarelli et al. 2011; Ozcan et al. 2011a]). If the results of q are not found in the QRC (i.e., a cache miss), the query is forwarded to an index server [STEP ②].

Index Servers. Index servers are responsible for the computation of the top-k ranked results related to a query q. An index server works as follows: [STEP IS1] retrieving the corresponding *posting list* $PL_i = [d_1, d_2, \ldots]$ of each term t_i in q, [STEP IS2] intersecting

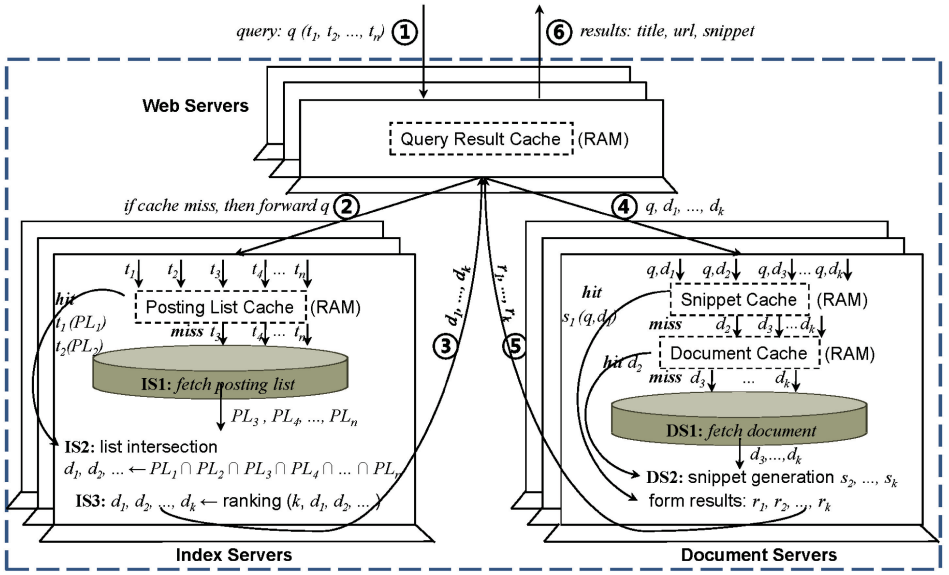


Fig. 2. Web search engine architecture.

all the retrieved posting lists $PL_1, PL_2, \ldots, PL_n$ to obtain a list of document identifiers (IDS) that contain all terms in q, and [STEP IS3] ranking the documents for q according to a ranking model. After that, the index server sends the ordered list of document IDS $d_1, d_2, \ldots, d_k$ of the top-k most relevant documents of query q back to the Web server [STEP ③]. In an index server, an in-memory *posting list cache* (PLC) [Saraiva et al. 2001; Baeza-Yates et al. 2007b; Zhang et al. 2008] is employed to cache the posting lists of some terms. Upon receiving a query $q(t_1, t_2, \ldots, t_n)$ from a Web server, an index server skips STEP IS1 if a posting list is found in the PLC.

Document Servers. Upon receiving the ordered list of document IDs $d_1, d_2, \ldots, d_k$ from the index server, the Web server forwards the list and query q to a document server for further processing [STEP ④]. The document server is responsible for generating the final result. The final result is a webpage that includes the title, URL, and a snippet for each of the top-k documents. The snippet s_i of a document d_i is query-specific, that is, it is a portion of a document which can best match the terms in q. The generation process is as follows: [STEP DS1] First, the original documents that the list of document IDs referred to are retrieved. [STEP DS2] Then, the snippet of each document for query q is generated. There are two levels of caches in the document servers: *snippet cache* (SC) [Ceccarelli et al. 2011] and *document cache* (DC) [Turpin et al. 2007]. The in-memory snippet cache (SC) stores some snippets that have been previously generated for some query-document pairs. If a particular query-document pair is found in the SC, STEP DS1 and STEP DS2 for that pair can be skipped. The in-memory document cache (DC) stores some documents that have been previously retrieved. If a particular requested document is in the DC, STEP DS1 can be skipped. As the output, the document server returns the final result (in the form of a webpage with a ranked list of snippets of the k most-relevant documents of query q) to the Web server [STEP ⑤], and the Web server may cache the result in the QRC and then pass the result back to the end user [STEP ⑥].

Table I. Caching Policy

	Dynamic	Static
Query Result Cache (QRC)	D-QRC-LRU [Markatos 2001] D-QRC-LFU [Markatos 2001] D-QRC-CA [Ozcan et al. 2011a] D-QRC-FB [Gan and Suel 2009]	S-QRC-Freq [Markatos 2001] S-QRC-FreqStab [Ozcan et al. 2008] S-QRC-CA [Altingovde et al. 2009]
Posting List Cache (PLC)	D-PLC-LRU [Saraiva et al. 2001] D-PLC-LFU [Baeza-Yates et al. 2007b] D-PLC-FreqSize [Baeza-Yates et al. 2007b]	S-PLC-Freq [Baeza-Yates and Saint-Jean 2003] S-PLC-FreqSize [Baeza-Yates et al. 2007b]
Document Cache (DC)	DC-LRU [Turpin et al. 2007]	–
Snippet Cache (SC)	SC-LRU [Ceccarelli et al. 2011]	–

2.2. Caching Policy

Caching is a widely-used optimization technique for improving system performance [Markatos 2001; Herrero et al. 2008; Fitzpatrick 2009; Elhardt and Bayer 1984]. Web search engines use caches in different types of servers to reduce the query response time [Saraiva et al. 2001]. There are two types of caching policies being used in search engines: (1) *dynamic caching* and (2) *static caching*. Dynamic caching is the classic. If the cache memory is full, dynamic caching follows a *replacement policy* (a.k.a. *eviction policy*) to *evict* some items in order to admit the new items [Belady 1966; Jiang and Zhang 2002; Podlipnig and Böszörmenyi 2003]. Static caching is less common but does exist in search engines [Markatos 2001; Baeza-Yates et al. 2007b, 2008; Altingovde et al. 2009; Ozcan et al. 2008, 2011a, 2011b]. Initially when the cache is empty, a static cache follows an *admission policy* to select data items to fill the cache. Once the cache has been filled, it does not admit any new item at run-time. The same cache content continuously serves the requests, and its entire content is refreshed in a periodic manner [Markatos 2001]. Static caching can avoid the situation of having long-lasting popular items being evicted by the admission of many momentarily popular items as in dynamic caching. In the following, we briefly review the prominent cache policies found in industrial-strength search engine architectures. They may not be exhaustive, but we believe they are representative. Table I shows the nomenclature for this article.

2.2.1. Cache in Web Servers. Query result cache (QRC) is the cache used in Web servers. It caches the query results such that the whole stack of query processing can be entirely skipped if the result of query q is found in the QRC. Both dynamic and static query result caches exist in the literature.

Dynamic Query Result Cache (D-QRC). Markatos was the first to discuss the use of dynamic query result cache (D-QRC) in search engines [2001]. By analyzing the query logs of the Excite search engine, Markatos observed a significant temporal locality in the queries. Therefore, he proposed the use of query result cache. Two classic replacement policies were used there: least-recently-used (LRU) and least-frequently-used (LFU). In this article, we refer to them as D-QRC-LRU and D-QRC-LFU, respectively.

Gan and Suel have developed a feature-based replacement policy for dynamic query result caching [2009]. In this article, we refer to that policy as D-QRC-FB. In addition to query frequency, the policy considers nine more features (e.g., query recency) to predict the reoccurrence of a query. More specifically, each of the ten features (dimensions) is divided into eight intervals, and thus the whole space is divided into 8^{10} buckets. Each bucket is associated with a weight, which is the cache hit ratio for queries belonging to

that bucket. Then, queries falling in the buckets with the lowest weights are chosen to be evicted first.

Recently, Ozcan et al. presented a cost-aware policy for dynamic query result caching [2011a]. We refer to it as D-QRC-CA. The policy takes a query's execution cost into account during eviction. Specifically, each query q is associated with a weight $w_q = (f_q \times c_q)/s_q + A$, where f_q is the query frequency, c_q is the execution cost of q (including only the basic execution costs such as list access, list intersection, and ranking cost), s_q is the query result size of q, and A is an aging factor (initially set as 0) that would be increased along with the time. When eviction takes place, the query with the smallest weight w_{min} is chosen and the value of A is updated as w_{min}. Whenever a cache item is accessed, its corresponding weight is recalculated using the updated value of A. This adopted D-QRC-CA policy is referred as the GDSF policy [Ozcan et al. 2011a].

Static Query Result Cache. Markatos discussed the potential use of static caching in search engines [2001]. In that work, a static query result cache with an admission policy that analyzes the query logs and fills the cache with the most frequently posed queries was proposed. In this article, we refer to this policy as S-QRC-Freq.

Ozcan et al. reported that some query results cached by the S-QRC-Freq policy are not useful because there is a significant number of frequent queries that quickly lose their popularity but still reside in the static cache before the next periodic cache refresh [2008]. Consequently, they proposed another admission policy that selects queries with high *frequency stability*. More specifically, the query log is first partitioned by time into n parts. Let f be the total frequency of a query q in the log and and f_i be the frequency of q in the ith part of the log. Then, the frequency stability of q is $\sum_{i=1}^{n} \frac{|f_i - f'|}{f'}$, where f' is the average frequency of q for a part, that is, $f' = f/n$. Queries with high frequency stability are those frequent queries, and the logs show that they remain frequent over a time period. In this article, we refer to this policy as S-QRC-FreqStab.

Later on, Altingovde et al. developed a cost-aware replacement policy for static query result cache [Altingovde et al. 2009; Ozcan et al. 2011a, 2011b]. In this article, we refer to that as S-QRC-CA. It is essentially the static version of D-QRC-CA (with the aging factor A being 0). That is, each query is associated with a weight, which is the product of its frequency and basic execution costs. That policy admits queries with the highest weights until the cache is full.

Remark. According to Gan and Suel [2009] and Ozcan et al. [2011a], D-QRC-FB and D-QRC-CA typically have the highest effectiveness among all other dynamic query result caching policies. Nonetheless, there is no direct comparison between D-QRC-FB and D-QRC-CA. The effectiveness of D-QRC-FB, however, is sensitive to the various parameters (e.g., the number of intervals of each dimension).

According to Altingovde et al. [2009] and Ozcan et al. [2011a, 2011b] S-QRC-CA is the most stable and effective policy in static query result caching. Dynamic and static result caching can coexist and share the main memory of a Web server. Fagni et al. [2006] empirically suggested ratios for D-QRC to S-QRC as 6:4, 2:8, and 7:3 for Tiscali, AltaVista, and Excite data, respectively.

2.2.2. Cache in Index Servers. Posting list cache (PLC) is the cache used in the index servers. It caches some posting lists in the memory so that the disk read of a posting list PL of a term t in query q can possibly be skipped. Both dynamic and static posting list caches exist in the literature.

Dynamic Posting List Cache. Saraiva et al. were the first to discuss the effectiveness of dynamic posting list caching in index servers [2001]. In that work, the simple LRU policy was used. In this article, we refer to this as D-PLC-LRU. Baeza-Yates et al.

[2007b, 2008] also evaluated another commonly used policy, LFU, in dynamic PLC, which we refer to as D-PLC-LFU. To balance the trade off between term popularity and the effective use of the cache (to cache more items), the authors developed a replacement policy that favors terms with a high frequency to posting list length ratio. In this article, we refer to this policy as D-PLC-FreqSize.

Static Posting List Cache. Static posting list caching was first studied in Baeza-Yates and Saint-Jean [2003], and the admission policy was based on selecting posting lists with high access frequency. In this article, we refer to this as S-PLC-Freq. In Baeza-Yates et al. [2007b, 2008], the static cache version of D-PLC-FreqSize was also discussed. We refer to that as S-PLC-FreqSize here.

Remark. According to Baeza-Yates et al. [2007b, 2008], S-PLC-FreqSize is the winner over all static and dynamic posting list caching policies.

2.2.3. Cache in Document Servers. Two types of caches could be used in the document servers: document cache (DC) and snippet cache (SC). These caches store some documents and snippets in the memory of the document servers. So far, only dynamic document cache and dynamic snippet cache have been discussed, and there are no corresponding static caching techniques yet.

Document Cache. Turpin et al. [2007] observed that the time of reading a document from disk dominates the snippet generation process, so they proposed caching some documents in the document server's memory so as to reduce the snippet generation time. In that work, the simple LRU replacement policy was used, which we refer to as DC-LRU. According to Turpin et al. [2007], document caching is able to significantly reduce the time for generating snippets.

Snippet Cache. Ceccarelli et al. [2011] pointed out that the snippet generation process is very expensive in terms of both CPU computation and disk access in the document servers. Therefore, the authors proposed caching some generated snippets in the document server's main memory. In this article, we refer to this as SC-LRU, because it uses the LRU, policy as replacement policy.

3. EXPERIMENTAL SETTING

In this article, we use a typical search engine architecture (Figure 2) that consists of index servers, document servers, and Web servers. We focus on a pure SSD-based architecture like Baidu [Ma 2010]. Many studies predict that SSD will soon completely replace HDD in all layers [Kerekes 2009; Gray 2006; Hauksson and Smundsson 2007]. The study of using SSD as a layer between main memory and HDD in a search engine architecture before SSD completely replaces HDD has been studied elsewhere in Li et al. [2012a].

We deploy the Lucene[4] search engine on a sandbox infrastructure consisting of one index server, one document server, and one Web server. All servers are Intel commodity machines (2.5GHz CPU, 8GB RAM) with Windows 7 installed. We have carried out our experiments on two SSDs and two HDDs (evaluated in Figure 1). The experimental results are largely similar and thus we only present the results of using SSD1 and HDD1. In the experiments, the OS buffer is disabled. The experiments were carried out using two datasets.

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

(1) Sogou.
—*Sogou Query*.[5] This is a collection of real queries (in Chinese) found in the Sogou search engine, a famous commercial search engine in China, in June 2008. The log contains 51.5 million queries. To ensure that replay can be done in a manageable time, we draw a random sample of 1 million queries for our experiments (as a reference, 0.7 million queries were used in Ozcan et al. [2011a]). The query frequency and term frequency of our sampled query set follow the power-law distribution with skew-factor $\alpha = 0.85$ and 1.48, respectively (see Figures 3(a) and 3(b)). As a reference, the query frequencies in some related work (e.g., [Saraiva et al. 2001; Gan and Suel 2009; Baeza-Yates et al. 2008]) follow the power-law distribution with α values of 0.59, 0.82, and 0.83, respectively; and the term frequencies in some recent work (e.g., [Baeza-Yates et al. 2008]) follow the power-law distribution with $\alpha = 1.06$.
—*Sogou Web Data*.[6] This is a real collection of webpages (in Chinese) crawled in the middle of 2008, by Sogou. The entire dataset takes over 5TB. To accommodate our experimental setup, we draw a random sample of 100GB data (about 12 million documents). It is a reasonable setting because large-scale Web search engines shared their indexes and data across clusters of machines [Ma 2010]. As a reference, the sampled datasets in some recent works are 15GB [Baeza-Yates et al. 2008], 37GB [Ozcan et al. 2011a], and 2.78 million webpages [Baeza-Yates et al. 2007b].

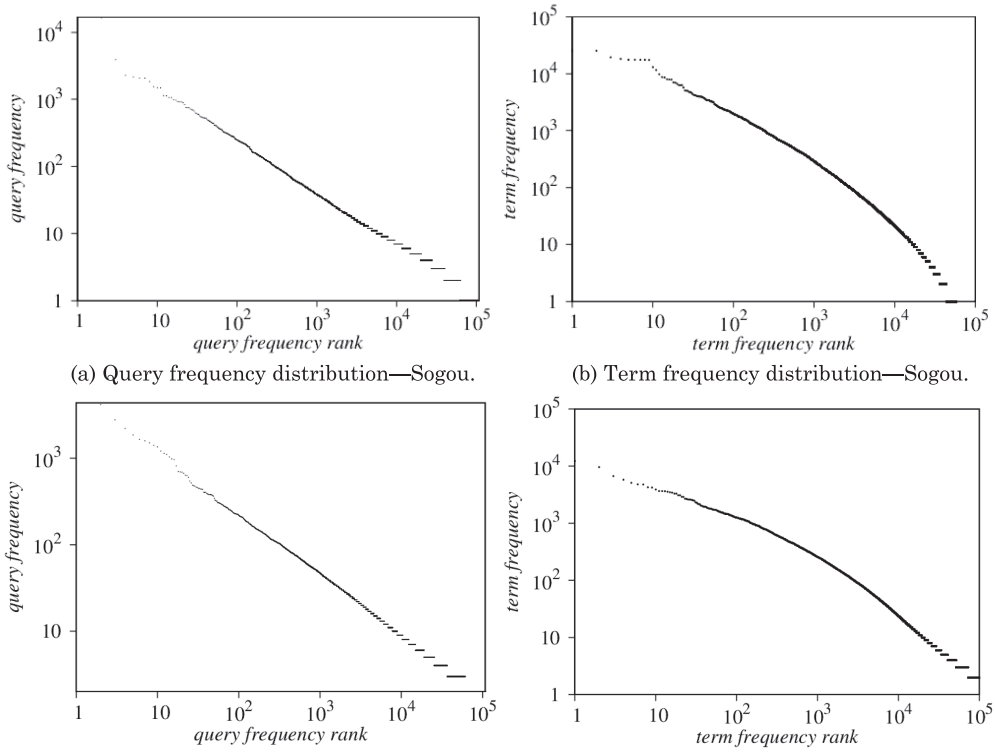
(2) AOL Query on ClueWeb Web Data.
—*AOL Query [Pass et al. 2006].* This is a collection of 36,389,567 real queries (in English) submitted to the AOL search engine (`search.aol.com`) between March 1 and May 31, 2006. We draw a sample of 1 million queries. The query frequency and term frequency of the sampled AOL query logs follow the power-law distribution with skew-factor $\alpha = 0.55$ and 1.48, respectively (see Figures 3(c) and 3(d)).
—*ClueWeb Web Data*.[7] This is a real collection of webpages crawled in January and February 2009 consisting of 10 languages. To be compatible with the size of Sogou Web data, we draw a sample of about 12 million English webpages. The sampled Web data is around 88GB.

Table II shows the characteristics of our query set[8], and Table III shows the characteristics of our dataset. In the following, we refer to "Sogou" as replaying Sogou queries on Sogou Web data, and we refer to "ClueWeb" as replaying AOL queries on ClueWeb data.

We divide the real query log into two parts: 50% of the queries are used for warming the cache and the other 50% are for the replay, following the usual settings [Markatos 2001; Altingovde et al. 2009, 2011; Ozcan et al. 2011a, 2011b].

In the experiments, we follow some standard settings found in recent work. Specifically, we follow these works [Fagni et al. 2006; Altingovde et al. 2011; Ceccarelli et al. 2011; Ozcan et al. 2011a] to retrieve the top-10 mos-relevant documents. We follow Saraiva et al. [2001] to configure the snippet generator to generate snippets with at most 250 characters. Posting list compression is enabled. To improve the experimental repeatability, we use the standard variable-byte compression method [Scholer et al. 2002] in the experiments. The page (block) size in the system is 4KB [Gal and Toledo

[5]`http://www.sogou.com/labs/dl/q-e.html`. (2008 version)
[6]`http://www.sogou.com/labs/dl/t-e.html`. (2008 version)
[7]`http://lemurproject.org/clueweb09.`
[8]Query logs used are de-identified for privacy reasons.



(a) Query frequency distribution—Sogou. (b) Term frequency distribution—Sogou.

(c) Query frequency distribution—AOL. (d) Term frequency distribution—AOL.

Fig. 3. Frequency distribution of our query set.

Table II. Statistics of Query Set

	Sogou	AOL
Number of queries	1,000,000	1,000,000
Number of distinct queries	200,444	495,073
Number of terms	1,940,671	2,007,985
Number of distinct terms	82,503	343,065
Average number of terms per query	3.89	2.05
Total size of the posting lists of the distinct terms	3.89GB	3.48GB
Power-law skew-factor α for query frequency	0.85	0.55
Power-law skew-factor α for term frequency	1.48	1.48

Table III. Statistics of Web Data

	Sogou	ClueWeb
Number of documents	11,970,265	12,408,626
Average document size	8KB	7KB
Total data size	100GB	88GB
Inverted index size	10GB	7.5GB

Table IV. Cache Used to Attain Typical Hit Ratios

	Sogou		ClueWeb	
Cache type	Cache size	Hit ratio	Cache size	Hit ratio
Query result cache (QRC)	64MB	50%	8MB	31.3%
Posting list cache (PLC)	512MB	48%	512MB	36.0%
Snippet cache (SC)	1GB	59%	256MB	30.3%
Document cache (DC)	4GB	38%	1024MB	15.6%

2005; Debnath et al. 2010; Tsirogiannis et al. 2009]. As the query latencies differ a lot between HDD-based and SSD-based architectures, we follow Baeza-Yates et al. [2007b] and Agrawal et al. [2008] to report the normalized average end-to-end query latency instead of the absolute query latency. Manning et al. [2008] and Webber and Moffat [2005] suggested that a posting list would not be fragmented to distant disk blocks but stored in continuous disk blocks. In this article, we follow that to configure Lucene.

4. THE IMPACT OF SOLID-STATE DRIVE ON SEARCH ENGINE CACHE MANAGEMENT

In this section, we evaluate the impact of SSD on the cache management of the index servers (Section 4.1), the document servers (Section 4.2), and the Web servers (Section 4.3).

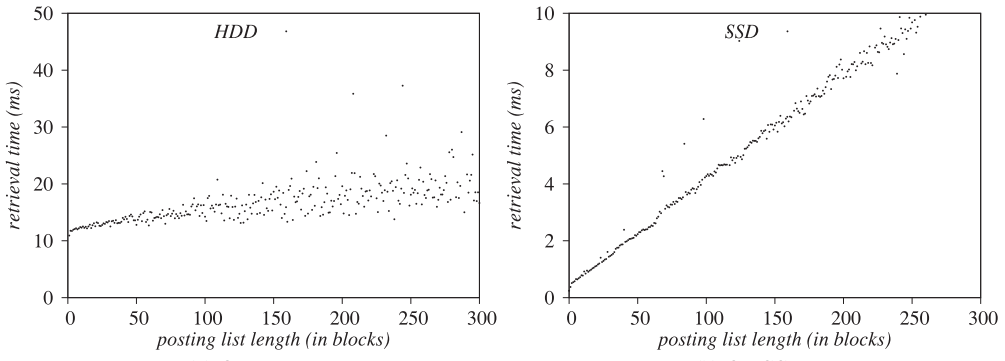
To show the effectiveness of each individual caching policy in a realistic manner, when evaluating a particular type of cache, we fill up the other caches to attain a typical cache hit ratio about 30% to 60%, as reported in several works [Dean 2009; Baeza-Yates et al. 2007b; Markatos 2001]. Table IV shows the default cache sizes that are required to fall into the typical hit ratio range in our platform. When replaying AOL queries over ClueWeb data, the hit ratio of document cache cannot increase beyond 15.6% even when we allocate more than 1024MB memory to the document cache. That is because many items had already been hit in the query result cache and the snippet cache. For the setting where all other caches are disabled when evaluating a particular type of cache, we refer readers to Wang et al. [2013], the preliminary version of this article.

4.1. The Impact of SSD on Index Servers

We first evaluate the impact of SSD on the posting list cache management in an index server. As mentioned, on SSD, a long posting list being found in the cache should have a larger query latency improvement than a short posting list being found because (i) a cache hit can save more sequential read accesses if the list is a long one and (ii) the cost of a sequential read is now comparable to the cost of a random read (see Figure 1).

To verify our claim, Figure 4 shows the access latency of fetching from disk the posting lists of terms found in the Sogou query log. We see that the latency of reading a list from HDD increases mildly with the list length because the random seek operation dominates the access time. In contrast, we see the latency of reading a list from SSD increases with the list length at a faster rate.

Based on that observation, we believe that the effectiveness of some existing posting list caching policies would change when they are applied to an SSD-based search engine infrastructure. For example, according to Baeza-Yates et al. [2007b, 2008], S-PLC-FreqSize has the best caching effectiveness on HDD-based search engines because it favors popular terms with short posting lists (i.e., a high frequency-to-length ratio) for the purpose of caching more popular terms. However, on SSD, a short list being in the cache has a smaller query latency improvement than a long list. As such, we believe that design principle is void in an SSD-based search engine infrastructure.



(a) On HDD. (b) On SSD.

Fig. 4. Read access latency of posting lists of varying lengths on Sogou data. We ran experiments five times to get the average.

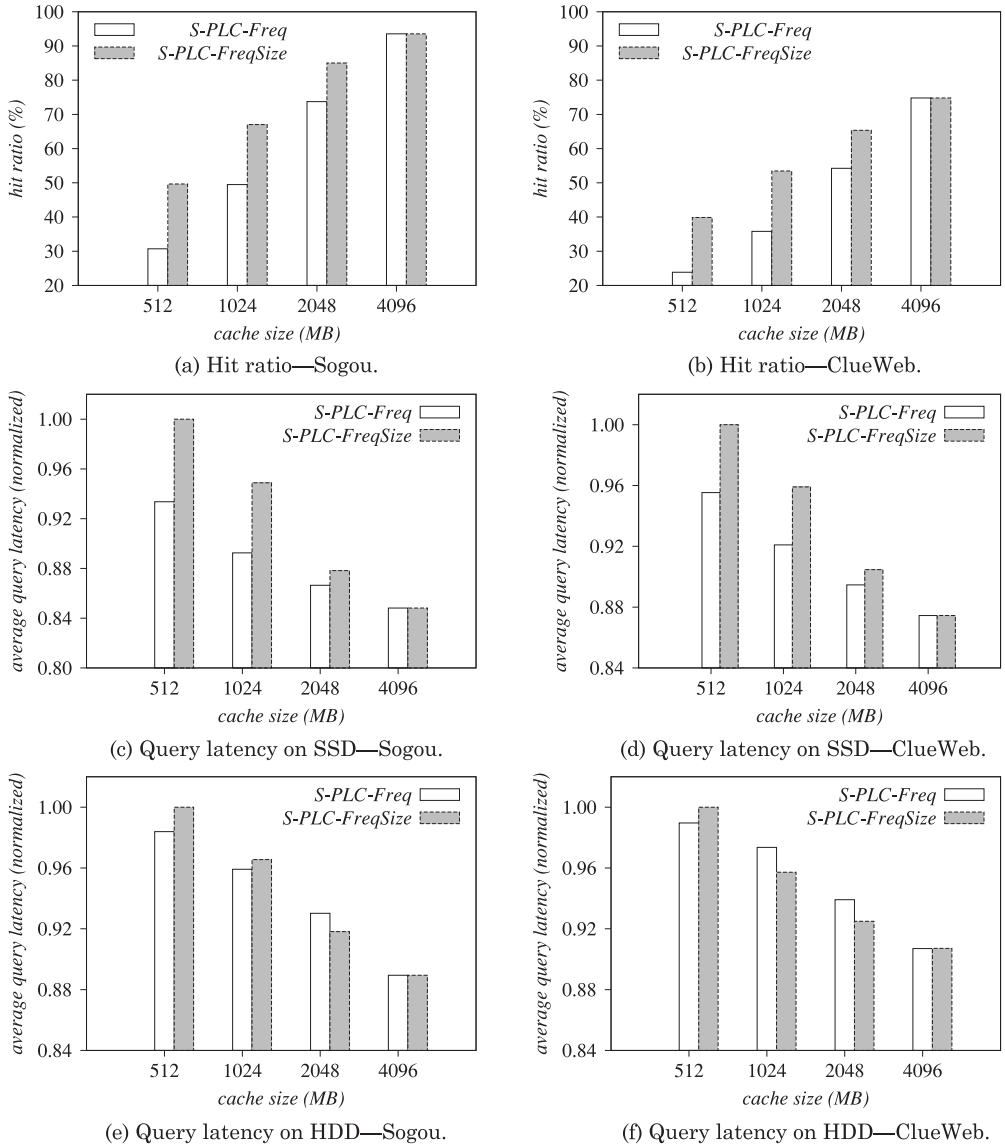
To verify our claim, we carried out experiments to reevaluate the static and dynamic posting list caching policies in our SSD search engine sandbox. In the evaluation, we focus on the effectiveness of individual caching policy type. The empirical evaluation of the optimal ratio between static cache and dynamic cache on SSD-based search engine architectures is beyond the scope of this article.

4.1.1. Reevaluation of Static Posting List Caching Policies on SSD-Based Search Engine Architectures. We begin with presenting the evaluation results of the two existing static posting list caching policies, (1) S-PLC-Freq and (2) S-PLC-FreqSize, mentioned in Section 2.2.2. Figure 5 shows the cache hit ratio and the average query latency of S-PLC-Freq and S-PLC-FreqSize under different cache memory sizes.

Echoing the results in Baeza-Yates et al. [2007b, 2008], Figures 5(a) and 5(b) show that S-PLC-FreqSize, which tends to cache popular terms with short posting lists, has a higher cache hit ratio than S-PLC-Freq. The two policies have the same cache hit ratio when the cache size is 4GB, because the cache is large enough to accommodate all the posting lists of the query terms (see Table II) in the cache warming phase. The reported cache misses are all attributed to the difference between the terms found in the training queries and the terms found in the replay queries.

Although it has a higher cache hit ratio, Figures 5(c) and 5(d) show that the average query latency of S-PLC-FreqSize is actually longer than that of S-PLC-Freq in an SSD-based search engine architecture. As the caching policy S-PLC-FreqSize tends to cache terms with short posting lists, the benefit brought by the higher cache hit ratio is watered down by the fewer sequential read savings caused by short posting lists. This explains why S-PLC-FreqSize becomes poor in terms of query latency. Apart from this, the experimental results are real examples that illustrate the cache hit ratio is not an adequate measure whenever the cache-miss costs are not uniform [Altingovde et al. 2009; Gan and Suel 2009; Ozcan et al. 2011a; Marin et al. 2010].

Figures 5(e) and 5(f) show the average query latency on HDD. We also observe that S-PLC-FreqSize's average query latency is slightly worse than S-PLC-Freq at 512MB cache memory, even though the former has a much higher cache hit ratio than the latter. To explain, Figure 6 shows the average sizes of the posting lists participating in all cache hits (i.e., whenever there is a hit in the PLC, we record its size and report the average). We see that when the cache memory is small (e.g., 512MB), S-PLC-FreqSize consistently keeps short lists in the cache. In contrast, S-PLC-Freq keeps relatively longer posting lists. Take the Sogou case as an example. At 512MB cache memory, a cache hit under S-PLC-Freq policy can save $693 - 307 = 386$ more sequential reads



(a) Hit ratio—Sogou.

(b) Hit ratio—ClueWeb.

(c) Query latency on SSD—Sogou.

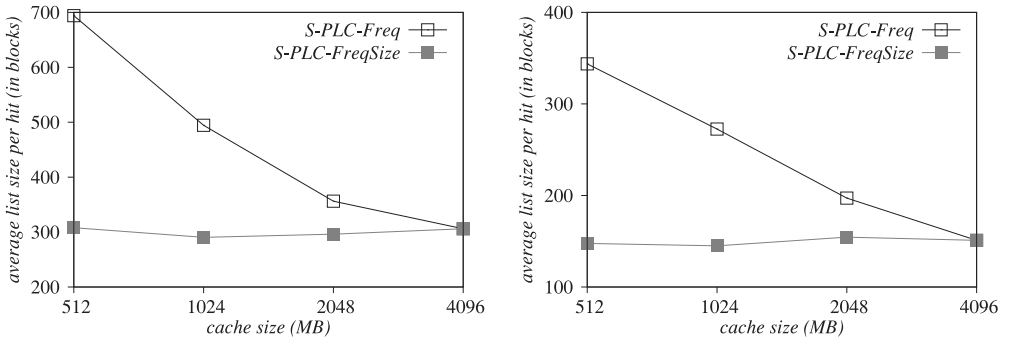
(d) Query latency on SSD—ClueWeb.

(e) Query latency on HDD—Sogou.

(f) Query latency on HDD—ClueWeb.

Fig. 5. [Index server] effectiveness of static posting list caching policies.

than S-PLC-FreqSize. Even though sequential reads are cheap on HDD, a total of 386 sequential reads are actually as expensive as 3 to 4 random seeks (see Figure 1). In other words, although Figure 5(a) shows that the cache hit ratio of S-PLC-Freq is 20% lower than S-PLC-FreqSize at 512MB cache, that is outweighed by the 386 extra sequential reads (equivalent to 3 to 4 extra random seeks) between the two policies. That explains why S-PLC-Freq slightly outperforms S-PLC-FreqSize at 512MB cache. Of course, when the cache memory increases, S-PLC-Freq starts to admit more short lists into the cache memory, which reduces its benefit per cache hit, and that causes S-PLC-FreqSize to outperform S-PLC-Freq again through the better cache hit ratio.



(a) On Sogou. (b) On ClueWeb.

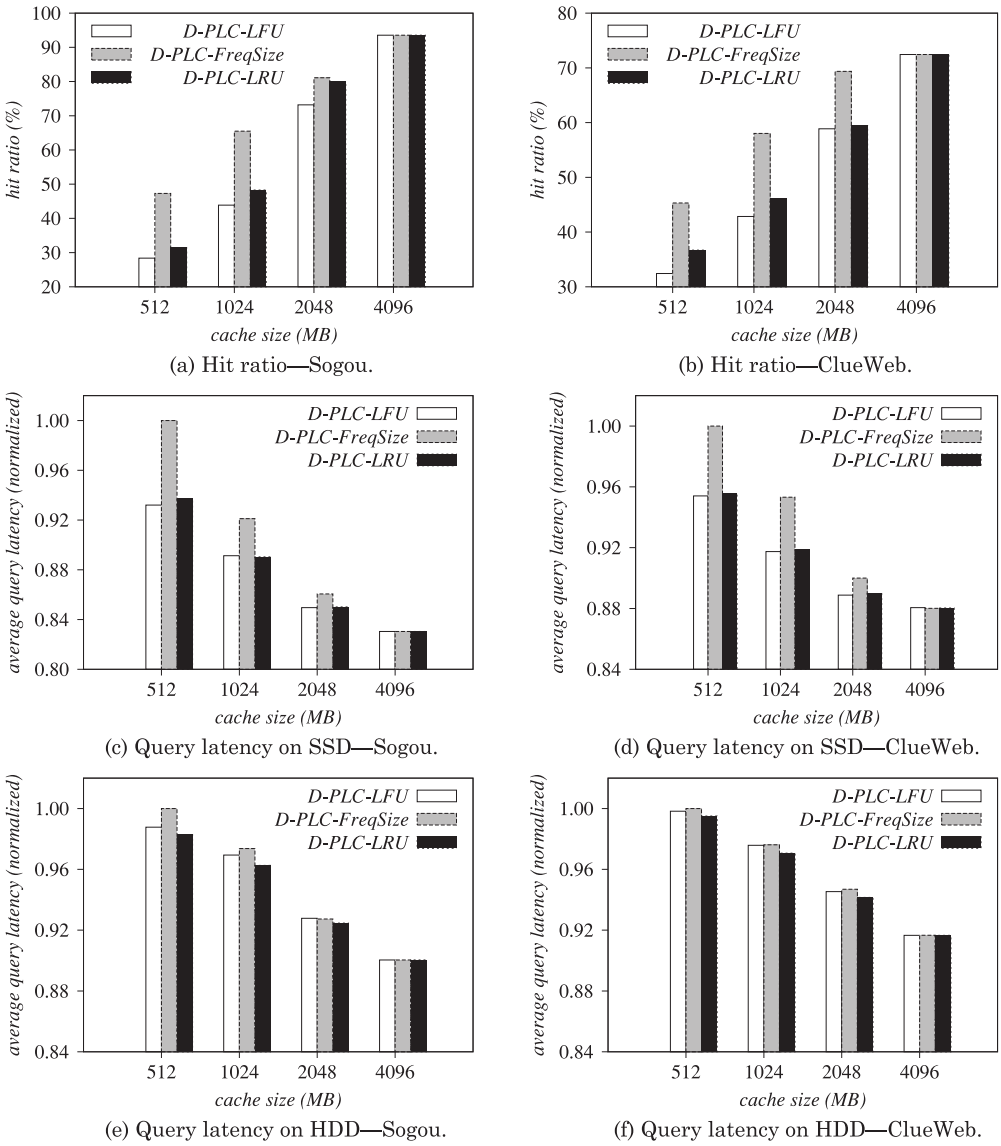
Fig. 6. Average list size (in blocks) of all hits in static posting list cache.

4.1.2. Reevaluation of Dynamic Posting List Caching Policies on SSD-Based Search Engine Architectures. We next present the evaluation results of the three existing dynamic posting list caching policies, (1) D-PLC-LRU, (2) D-PLC-LFU, and (3) D-PLC-FreqSize, mentioned in Section 2.2.2. Figure 7 shows the cache hit ratio and the average query latency of the three policies under different cache memory sizes.

First, we also see that while D-PLC-FreqSize has a better cache hit ratio than D-PLC-LFU (Figures 7(a) and 7(b)), its query latency is actually longer than D-PLC-LFU (Figures 7(c) and 7(d)) in SSD-based architectures. This further supports that the claim of favoring terms with high frequency over length ratio is no longer sustained in SSD-based search engine architectures. Also, this gives yet another example of the fact that cache hit ratio is not a reliable measure in SSD cache management, as the average length of posting lists that are admitted by D-PLC-FreqSize is much shorter than by D-PLC-LFU (Figure 8), making the cache-miss costs nonuniform.

Second, comparing D-PLC-LRU and D-PLC-LFU, we see that while D-PLC-LFU has a poorer cache hit ratio than D-PLC-LRU, their query latencies are quite close in SSD-based architectures. D-PLC-LFU tends to admit longer lists than D-PLC-LRU (Figure 8), because there is a small correlation (0.424 in Yahoo! data [Baeza-Yates et al. 2007b] and 0.35 in our Sogou data) between term frequency and posting list length. As mentioned, on SSD, the benefit of finding a term with a longer list in cache is higher than that of finding a term with shorter list. This explains why D-PLC-LFU has a query latency close to D-PLC-LRU, which has a higher cache hit ratio.

Figures 7(e) and 7(f) show the average query latency on HDD. First, we once again see that cache hit ratio is not reliable even on HDD-based architectures. For example, while D-PLC-FreqSize has a higher cache hit ratio than D-PLC-LFU (except when the cache is large enough to hold all posting lists), their query latencies are quite close to each other in HDD-based architectures. That is due to the same reason we explained in static caching—Figures 8(a) and 8(b) show that D-PLC-LFU can save hundreds of sequential reads more than D-PLC-FreqSize per cache hit when the cache memory is less than 1GB. That outweighs the cache hit ratio difference between the two. In contrast, while D-PLC-LRU has a better cache hit ratio than D-PLC-LFU, they follow the tradition that the former outperforms the latter in latency, because Figures 8(a) and 8(b) show that the size difference of the posting lists that participated in the cache hits (i.e., the benefit gap in terms of query latency) between D-PLC-LRU and D-PLC-LFU is not as significant as the time of one random seek operation. Therefore, D-PLC-LRU yields a shorter query latency than D-PLC-LFU based on its higher cache hit ratio.

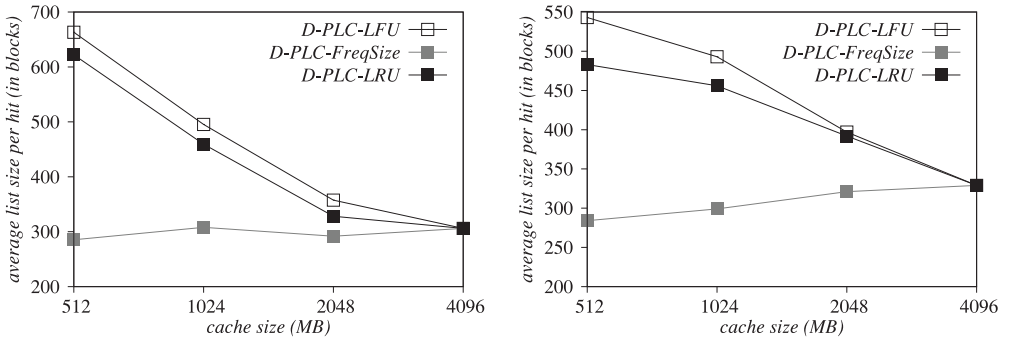


(a) Hit ratio—Sogou.
(b) Hit ratio—ClueWeb.
(c) Query latency on SSD—Sogou.
(d) Query latency on SSD—ClueWeb.
(e) Query latency on HDD—Sogou.
(f) Query latency on HDD—ClueWeb.

Fig. 7. [Index server] effectiveness of dynamic posting list cache policies.

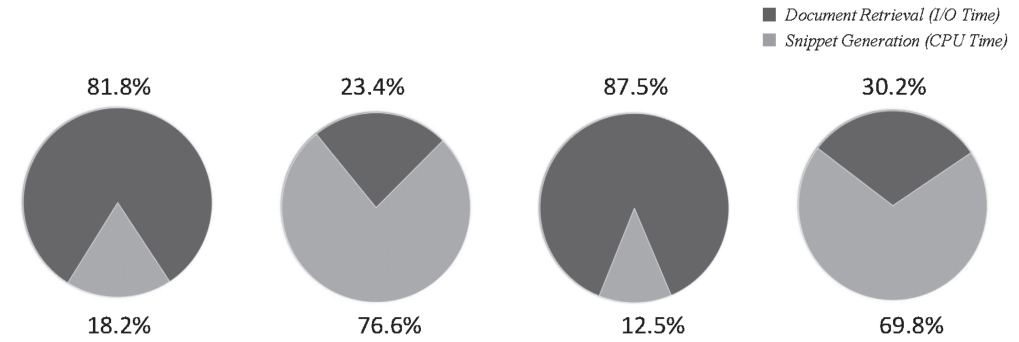
4.2. The Impact of SSD on Document Servers

We next evaluate the impact of SSD on the cache management of the document servers. A document server is responsible for storing part of the whole document collection and generating the final query result. It receives a query q and an ordered list of document IDs $\{d_1, d_2, \ldots, d_k\}$ for the top-k most relevant documents from a Web server, retrieves the corresponding documents from the disk, and generates the query-specific snippet for each document and consolidates them as a result page. In the process, k query-specific snippets have to be generated. If a query-specific snippet $\langle q, d_i \rangle$ is found in the snippet cache, the retrieval of d_i from the disk and the generation of that snippet are skipped. If a query-specific snippet is not found in the snippet cache, but the document



(a) On Sogou. (b) On ClueWeb.

Fig. 8. Average list size (in blocks) of all hits in dynamic posting list cache.



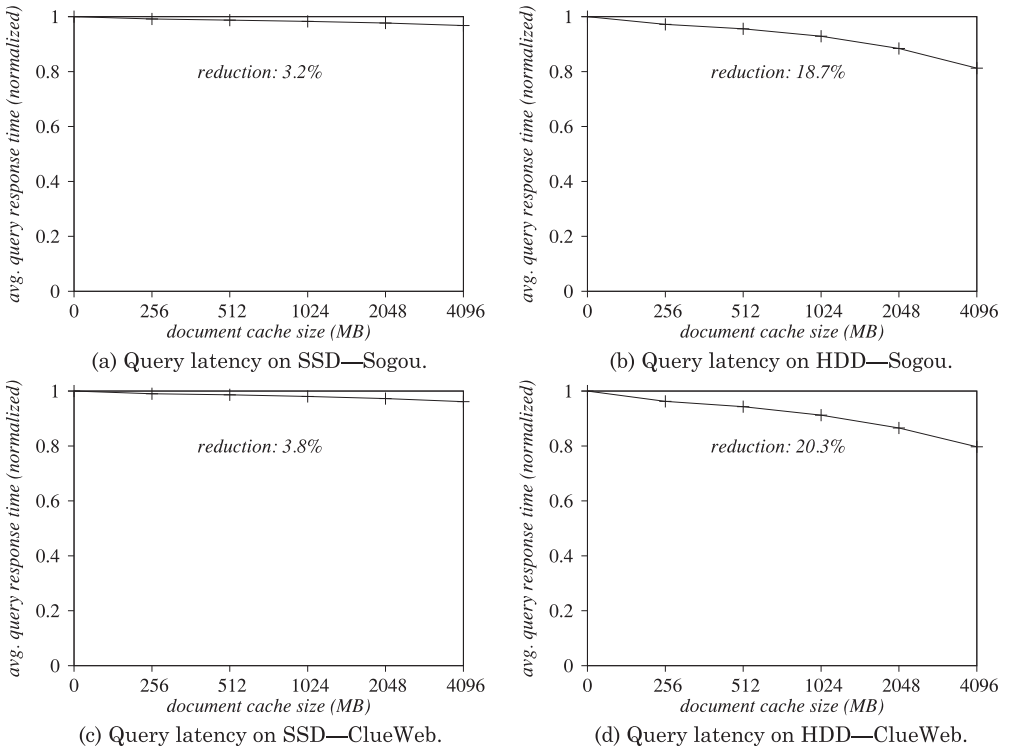
(a) On HDD—Sogou. (b) On SSD—Sogou. (c) On HDD—ClueWeb. (d) On SSD—ClueWeb.

Fig. 9. Document retrieval time vs. snippet generation in a document server.

d_i is in found the document cache, the retrieval of d_i from the disk is skipped. In our data (Table III), a document is about 7KB to 8KB on average. With a 4KB page size, retrieving a document from the disk thus requires one random seek (read) and a few more sequential reads.

In traditional search engine architectures using HDD in the document servers, the latency from receiving the query and document list from the Web server to the return of the query result is dominated by the k random read operations that seek k documents from the HDD (see Figures 9(a) and 9(c)). These motivated the use of document cache to improve the latency. As the random reads are much faster on SSD, we now believe that the time bottleneck in the document servers will shift from document retrieval (disk access) to snippet-generation (CPU computation). More specifically, the snippet-generation process that finds every occurrence of q in d_i and identifies the best text synopsis [Turpin et al. 2007; Tombros and Sanderson 1998] according to a specific ranking function [Tombros and Sanderson 1998] is indeed CPU-intensive. Figures 9(b) and 9(d) show that the CPU time spent on snippet generation becomes two times that of the document retrieval time if SSD is used in a document server. Therefore, contrary to the significant time reduction brought by document cache in traditional HDD-based search engine architectures [Turpin et al. 2007], we believe the importance of document cache in SSD-based search engine architectures is significantly diminished.

To verify our claim, we carried out experiments to vary the cache memories allocated to the document cache, with all the other caches enabled as default in Table IV, in our



(a) Query latency on SSD—Sogou.

(b) Query latency on HDD—Sogou.

(c) Query latency on SSD—ClueWeb.

(d) Query latency on HDD—ClueWeb.

Fig. 10. [Document server] effectiveness of document cache.

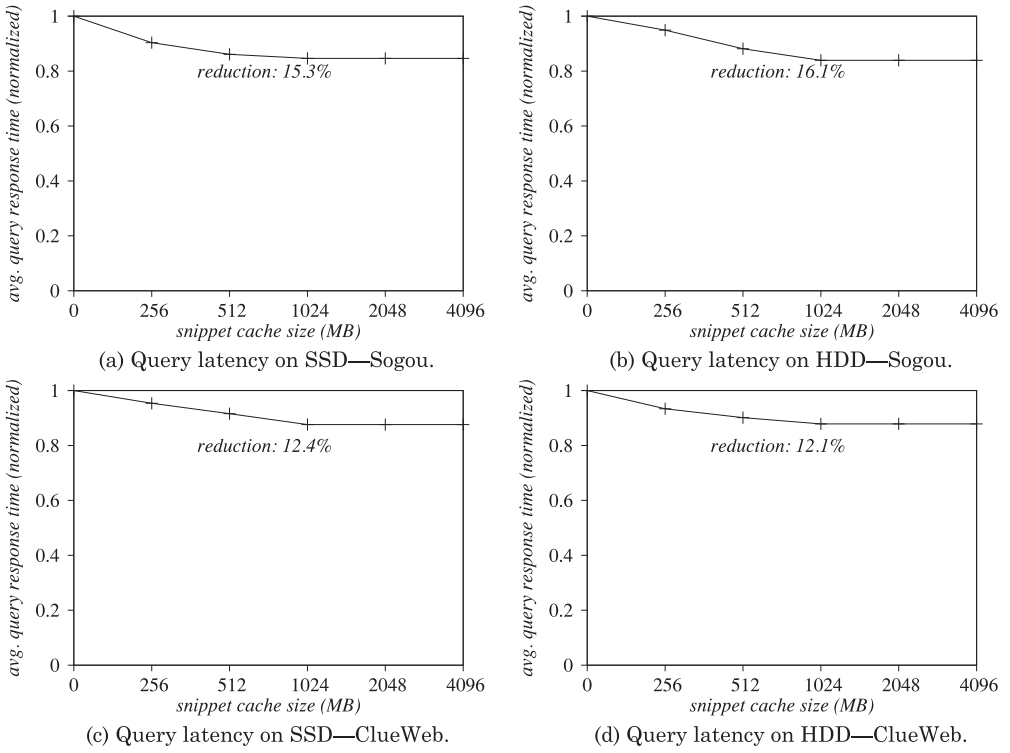
SSD- and HDD-based sandbox infrastructure. The caching policies in the document cache and snippet cache are DC-LRU [Turpin et al. 2007] and SC-LRU [Ceccarelli et al. 2011], respectively.

Figure 10 shows the query latency when varying the size of the document cache from 0 to 4GB. We can see on HDD (Figures 10(b) and 10(d)), the document cache is effective, since it can reduce the overall query latency up to 18.7–20.3%. However, on SSD (Figures 10(a) and 10(c)), the document cache can reduce only up to 3.2–3.8% of the overall latency, which is much less effective. That is because of the excellent random access performance on SSD, making document retrieval on SSD much faster.

In contrast, we believe the snippet cache is still powerful in an SSD-based search engine architecture for two reasons.

(1) A cache hit in a snippet cache can reduce both the time of snippet generation and document retrieval.
(2) The memory footprint of a snippet (e.g., 250 characters) is much smaller than the memory footprint of a document (e.g., an average 7KB to 8KB in our data).

To verify our claim, we carried out experiments to vary the size of snippet cache from 0 to 4GB. Figure 11 shows the results. In HDD-based architectures, the snippet cache can yield up to 12.1–16.1% query latency reduction. In SSD-based architectures, the snippet cache can yield up to 12.4–15.3% query latency reduction. These results show that the effectiveness of snippet cache is not influenced by the change of the storage model.



(a) Query latency on SSD—Sogou. (b) Query latency on HDD—Sogou.

(c) Query latency on SSD—ClueWeb. (d) Query latency on HDD—ClueWeb.

Fig. 11. [Document server] effectiveness of snippet cache.

4.3. The Impact of SSD on Web Servers

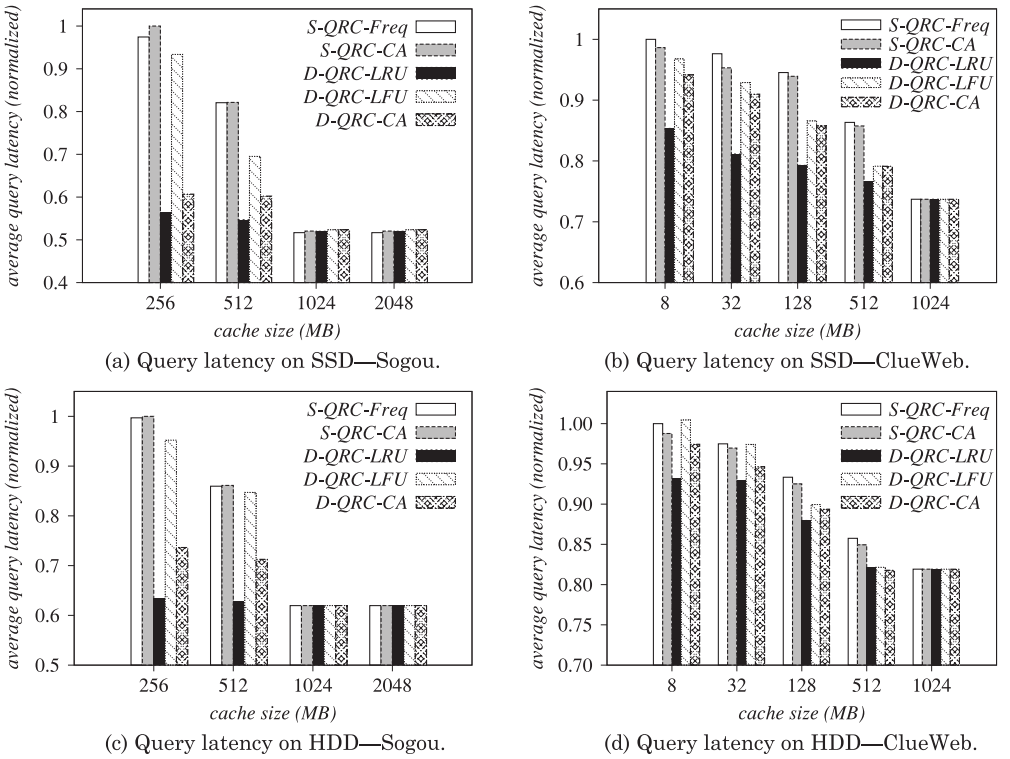
Clearly, the use of SSD to replace HDD improves the query latency of a search engine because of the better performance of SSD over HDD. However, we believe the use of SSD has little impact on the cache management in the Web servers, because the disks in the Web servers are only for temporary storage (see Figure 2).

Figure 12 shows the query latency of five selected query result caching policies[9]: (1) D-QRC-LRU, (2) D-QRC-LFU, (3) D-QRC-CA, (4) S-QRC-Freq, and (5) S-QRC-CA. We see that if one policy is more effective than the other in terms of query latency on HDD, the corresponding query latency is also shorter than the other on SSD.

5. CACHE DESIGN OF SSD-BASED SEARCH ENGINE ARCHITECTURE

The previous section points out that SSD indeed brings certain impact to search engine caching. In this section, we aim to shed some light on how to adapt the search engine architecture to best embrace SSD. We first identify the latency bottleneck of an SSD-based search engine architecture (Section 5.1). Based on that result, we explore other possible cache types to mitigate that bottleneck (Section 5.2). Finally, we give empirical memory allocation suggestions to those cache types (Section 5.3).

[9]In this study, we exclude the results of D-QRC-FB and S-QRC-FreqStab. We exclude the result of D-QRC-FB because its effectiveness heavily relies on the tuning of the various parameter values (e.g., the number of intervals of each dimension). We exclude the result of S-QRC-FreqStab because its effectiveness is the same as S-QRC-Freq when the query log cannot be split into intervals, where each interval spans weeks or months.



(a) Query latency on SSD—Sogou.

(b) Query latency on SSD—ClueWeb.

(c) Query latency on HDD—Sogou.

(d) Query latency on HDD—ClueWeb.

Fig. 12. [Web server] effectiveness of query result caching policies on SSD-based search engine architectures.

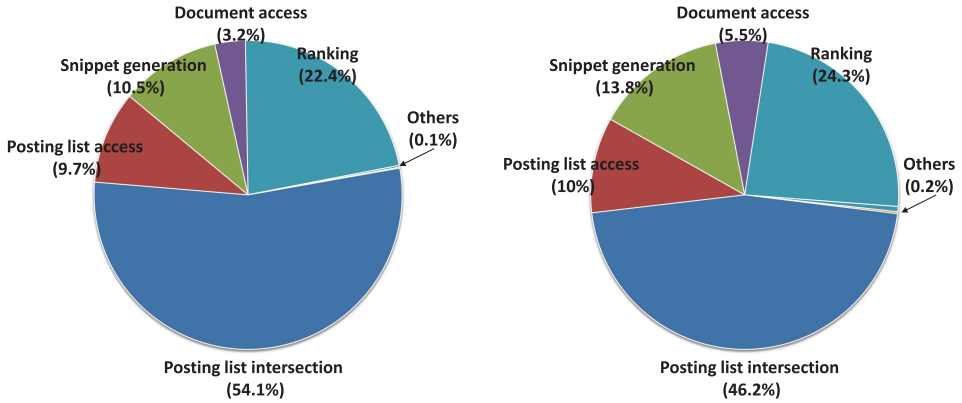
5.1. List Intersection—The Latency Bottleneck in SSD-Based Search Engine

We aim to identify the latency bottleneck of an SSD-based search engine. Figure 13 shows the time breakdown of query processing in our sandbox infrastructure with all caches enabled. The cache sizes follow Table IV in order to attain typical hit ratios found in real search engines. From Figure 13, we clearly see that posting list intersection in the index server is the primary bottleneck in SSD-based search engines. The other time-dominating steps, listed in decreasing order, are as follows: ranking the results in the index server, snippet generation in the document server, fetching the posting list from disk in the index sever, and retrieving the document from disk in the document server. Other steps, such as data transfer among servers through the network, incur negligible cost.

5.2. Cache Type for List Intersection Saving

Having identified posting list intersection in the index server as the major bottleneck in an SSD-based search engine, we aim to explore alternate cache types that can be introduced to the index server in order to mitigate the bottleneck. In this regard, we consider two cache types, namely, full-term-ranking-cache [Altingovde et al. 2011] and two-term-intersection-cache [Long and Suel 2005], in addition to the standard posting list cache, to be put in the index server.

The full-term-ranking-cache (FTRC) aims to reduce the entire query processing step within the index server, including (i) fetching posting lists from disk, (ii) computing the list intersection (the new bottleneck), and (iii) ranking and finding the top-k relevant documents. As such, it takes the whole query q as the key and the top-k document



(a) On SSD—Sogou. (b) On SSD—ClueWeb.

Fig. 13. Time breakdown of each component in an SSD-based search engine with QRC, PLC, DC, and SC enabled.

IDs as the cache values. The corresponding caching policy of FTRC in Altingovde et al. [2011] is LRU. FTRC can be deployed in the Web servers, too. When doing so, a hit in a Web server's FTRC can further eliminate the cost of merging individual ranking results from multiple index servers. In this article, however, we follow an industrial setting that we found in a local search engine and put FTRC in the index server.

The two-term-intersection-cache (TTIC) aims to reduce the posting list intersection time. For a query q with terms $t_1, t_2, \ldots, t_n$, it regards every two-term $\langle t_i, t_j \rangle$ of q as the cache key and the intersection of t_i's posting list PL_i and t_j's posting list PL_j as the cache value. The corresponding caching policy of TTIC in Long and Suel [2005] is called Landlord [Cao and Irani 1997]. Its idea is to favor the recently accessed items, while also taking into account the access latency and the size of an item. Specifically, whenever a cache item I is inserted to the cache, a weight is assigned to it. The weight w_I of I is computed as $(c_I/s_I) + A$, where c_I is the access latency of item I, s_I is the size of I, A is an aging factor (initially set as 0) that would be increased along with the time. When eviction takes place, the item with the smallest weight w_{min} is chosen, and the value of A is updated as w_{min}. Whenever a cache item is accessed, its corresponding weight is recalculated using the updated value of A.

Table V lists the cache benefit, memory footprint, and cache locality (the α value in power-law distribution) of FTRC, TTIC, and PLC on the Sogou data. In terms of cache benefit, that is, reduction of query processing time, FTRC is the highest whilst the standard PLC is the lowest. FTRC also has the lowest memory footprint whilst the standard PLC has the highest. However, the cache locality of FTRC is the worst because its cache items are query specific. TTIC is the middle ground between FTRC and PLC. We see from Table V that there is no absolute winner in our SSD-based architecture. The same conclusion is observed from the ClueWeb data. Therefore, we suggest all three to be integrated in an SSD-based search engine architecture.

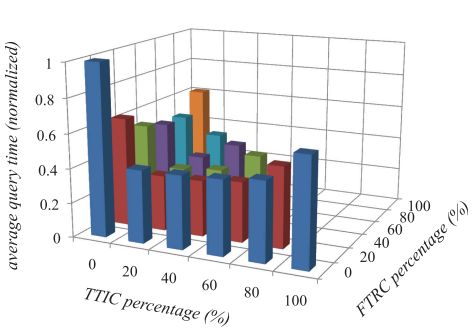
5.3. Cache Memory Allocation in Index Server

In this section, we carry out a study to empirically identify the optimal memory allocation to the three different cache types FTRC, TTIC, and PLC in the index servers. The goal is to give some insights on how to best leverage them to mitigate the latency bottleneck in an SSD-based search engine architecture.

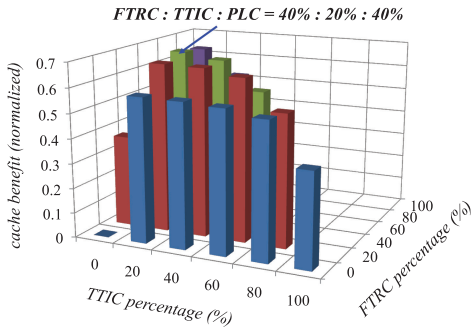
Figure 14(a) shows the average query latency on Sogou data when we vary the memory allocation among FTRC, TTIC, and PLC. Figure 14(b) is another view of

Table V. Cache Benefit, Memory Footprint, and Cache Locality of FTRC, TTIC, and PLC

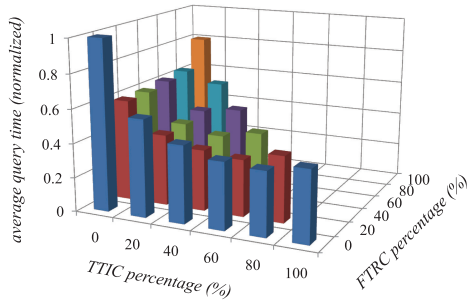
		FTRC	TTIC	PLC
Cache benefit (query time reduction)	list access (I/O time)	25ms	13ms	6.5ms
	intersection (CPU time)	25ms	18.7ms	–
	ranking (CPU time)	9ms	–	–
	$\sum$	**59ms (Good)**	**31.7ms (Median)**	**6.5ms (Low)**
Memory footprint		**40 bytes (Low)**	**44,985 bytes (Median)**	**266,658 bytes (Large)**
Cache locality		α = **0.57 (Low)**	α = **1.03 (Median)**	α = **1.48 (Good)**



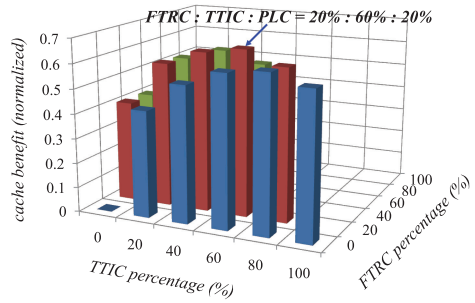
(a) Query latency—Sogou.



(b) Cache benefit—Sogou.



(c) Query latency—ClueWeb.



(d) Cache benefit—ClueWeb.

Fig. 14. Varying memory allocation between full-term ranking cache, two-term intersection cache, and posting list cache, in the index server.

Figure 14(a) that shows the cache benefit measured in terms of the reduction of query processing time under different memory allocation. From the figures, we observe the optimal memory allocation of FTRC : TTIC : PLC is 40%:20%:40%. Under such an allocation, it improves the performance by 69.9%, compared with the original SSD-architecture with PLC only.

Figures 14(c) and 14(d), respectively, show the average query latency and benefit on ClueWeb data under different memory allocation for FTRC, TTIC, and PLC. We observe that the optimal performance is achieved when FTRC : TTIC : PLC is 20%:60%:20%—an improvement of 66.7% compared with the original SSD-architecture with PLC only.

6. RELATED STUDIES

SSD is expected to gradually replace HDD as the primary permanent storage media in both consumer computing and enterprise computing. Large-scale enterprise computing

architectures such as Facebook[10], MySpace[11], and Windows Azure[12] have incorporated SSD in their system infrastructure. This trend is mainly attributed to SSD's outstanding performance, small energy footprint, and increasing capacity. This has led to a number of studies that aim to better understand the impacts of SSD on different computer systems.

The discussion of the impact of SSD on computer systems had started as early as 1995 [Kawaguchi et al. 1995], in which an SSD-aware file system was proposed. Later on, more SSD-aware file systems were designed (e.g., JFFS[13], YAFFS[14]. Read operations in those file systems are fast because of the use of SSD. A log-structure file organization is then used to alleviate the intrinsic slow random write problem in SSD. After that, the discussion has extended to other components in computer systems. For example, Park et al. [2006] studied the impact of SSD on buffer management in general computer systems. In that work, they developed a new caching policy. Specifically, a list of clean pages and a list of dirty pages are maintained. During page eviction, pages from the clean list are evicted first. This policy prefers dirty pages to be in the cache, which aims to reduce the number of expensive write operations in SSD. Saxena and Swift [2009] studied the impact of SSD on page swapping. Their goal is to optimize the page swap operations between RAM and SSD. To achieve that, expensive random writes operations are buffered so as to turn individual random write operations into groups of (cheaper) sequential writes.

The discussion of the impact of SSD on database systems had started as early as in 2007. Lee and Moon [2007] presented a new database design to, again, mitigate the slow random write problems of SSD. The methodology is to buffer data pages and then flush them to the log through sequential writes. Since then, SSD has become an active topic in database research. For example, the impact of SSD on join operations was first studied in Shah et al. [2008]. In that work, a new hash-join implementation was proposed. It exploits the property of fast random read on SSD to fetch only the join attributes from the join tables so that the join can be carried out using much fewer memory. Later on, a few more SSD-aware index structures were developed [Agrawal et al. 2009; Li et al. 2010]. They share the same goal—to reduce the expensive random writes triggered by update, insert, and delete operations. In Agrawal et al. [2009], a RAM buffer is used to buffer the updates to a B-tree in order to amortize the update cost. In Li et al. [2010], random writes are restricted to the top level of B-tree. Insertions are inserted to the top level first. The top level is merged with the next level through sequential writes. As such, random writes are always limited to a certain area of SSD. The benefit is that such "dense" random writes could be more efficient than "sparse" random writes that span the whole disk [Nath and Gibbons 2008]. The recent trend of SSD-aware data structures exploits the internal parallelism in SSD by issuing multiple I/Os simultaneously. For example, a parallel B-Tree was designed for SSD such that a batch of queries can be answered efficiently [Roh et al. 2011].

Baidu first announced their SSD-based search engine infrastructure in 2010 [Ma 2010], but they did not investigate the impact of SSD on their cache management. In Huang and Xia [2011], a RAM-SSD-HDD search engine architecture was discussed. The key issue there was to select a subset of posting lists to be stored in SSD, and the problem was solved as an optimization problem. In [Li et al. 2012b], that issue was further discussed in the context of incrementally updating the SSD-resident posting

[10]http://www.facebook.com/note.php?note_id=388112370932.
[11]http://www.fusionio.com/case-studies/myspace-case-study.pdf.
[12]http://www.storagelook.com/microsoft-azure-ocz-ssds.
[13]http://sourceware.org/jffs2.
[14]http://www.yaffs.net.

list in such architecture. In Li et al. [2012b], a RAM-SSD search engine architecture was discussed. That work focused on how to update the posting lists in SSD efficiently when new documents are collected. This article however focuses on cache management in such an SSD-based search engine architecture. Works that study the optimal cache allocation ratio between different types of caches (e.g., [Baeza-Yates and Jonassen 2012; Ozcan et al. 2011b]) on traditional HDD-based search engines are orthogonal to this article.

7. CONCLUSIONS

In this article, we first present the results of a large-scale experimental study that evaluates the impact of SSD on the effectiveness of various caching policies, on all types of cache found in a typical search engine architecture. In addition, we present an empirical study that gives the preliminary direction to optimize SSD-based search engine.

This article contributes the following messages to our community.

(1) The previously known best caching policy in the index servers, S-PLC-FreqSize [Baeza-Yates et al. 2007b, 2008], has the worst effectiveness in terms of query latency in our SSD-based search engine evaluation platform. Instead, all the other policies are better than S-PLC-FreqSize in terms of query latency, but no clear winner is found.
(2) While previous work claims that document caching is very effective and the technique is able to significantly reduce the time of the snippet-generation process in the document servers [Turpin et al. 2007], we show that snippet caching is even more effective than document caching in SSD-based search engines. Therefore, snippet caching should have a higher priority of using the cache memory of the document servers in an SSD-based search engine deployment.
(3) While SSD can improve the disk access latency of all servers in Web search engines, it has no significant impact on the cache management in Web servers. Thus, during the transition from an HDD-based architecture to an SSD-based architecture, there is no need to revise the corresponding query result caching policies in the Web servers.
(4) Posting list intersection is the bottleneck of SSD-based search engines. In addition to the standard posting list cache (PLC) in the index server, full-term ranking cache (FTRC) and two-term intersection cache (TTIC) can also be exploited to improve the whole stack of query processing in SSD-based search engines. Preliminary empirical results suggest a memory allocation of 40% : 20% : 40% for FTRC : TTIC : PLC on Sogou data, and 20% : 60% : 20% on the ClueWeb data.

REFERENCES

Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. 2009. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.* 2, 1 (2009), 361–372.

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Conference on Annual Technical Conference (ATC)*. 57–70.

Ismail Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. 2011. Second chance: A hybrid approach for dynamic result caching in search engines. In *Proceedings of the European Conference on Advances in Information Retrieval (ECIR)*. 510–516.

Ismail Sengor Altingovde, Rifat Ozcan, and Özgür Ulusoy. 2009. A cost-aware strategy for query result caching in web search engines. In *Proceedings of the European Conference on Advances in Information Retrieval (ECIR)*. 628–636.

Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. 2007a. Challenges on distributed web retrieval. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 6–20.

Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. 2007b. The impact of caching on search engines. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 183–190.

Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. 2008. Design trade-offs for search engine caching. *ACM Trans. Web* 2, 4 (2008), 1–28.

Ricardo Baeza-Yates and Simon Jonassen. 2012. Modeling static caching in web search engines. In *Proceedings of the European Conference on Advances in Information Retrieval (ECIR)*. 436–446.

Ricardo Baeza-Yates and Felipe Saint-Jean. 2003. A three level search engine index based in query log distribution. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*. 56–65.

Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro Mag.* 23, 2 (2003), 22–28.

Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* 5, 2 (1966), 78–101.

Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 426–434.

Pei Cao and Sandy Irani. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. 2011. Caching query-biased snippets for efficient retrieval. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 93–104.

Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 181–192.

Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking database algorithms for phase change memory. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*. 21–31.

Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM)*.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.* 3, 1–2 (2010), 1414–1425.

Klaus Elhardt and Rudolf Bayer. 1984. A database cache for high performance and fast restart in database systems. *ACM Trans. Datab. Syst.* 9, 4 (1984), 503–525.

Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. 2006. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.* 24, 1 (2006), 51–78.

Brad Fitzpatrick. 2009. Memcached – A distributed memory object caching system. http://memcached.org/. (2009).

Flexstar Technology. 2012. Flexstar SSD test market analysis. http://info.flexstar.com/Portals/161365/docs/SSD_Testing_ Market_Analysis.pdf. (2012).

Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *ACM Comput. Surv.* 37, 2 (2005), 138–163.

Qingqing Gan and Torsten Suel. 2009. Improved techniques for result caching in web search engines. In *Proceedings of the International Conference on World Wide Web (WWW)*. 431–440.

Goetz Graefe. 2009. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM* 52, 7 (2009), 48–59.

Jim Gray. 2006. Tape is dead, disk is tape, flash is disk, ram locality is king. http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good. ppt. (2006).

Ari Geir Hauksson and Sverrir Smundsson. 2007. Data storage technologies. http://olafurandri.com/nyti/papers2007/DST.pdf. (2007).

Enric Herrero, José González, and Ramon Canal. 2008. Distributed cooperative caching. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 134–143.

Bojun Huang and Zenglin Xia. 2011. Allocating inverted index into flash memory for search engines. In *Proceedings of the International Conference on World Wide Web (WWW)*. 61–62.

Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 31–42.

Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. 1995. A flash-memory based file system. In *Proceedings of the USENIX Conference on Annual Technical Conference (ATC)*. 155–164.

Zsolt Kerekes. 2009. Storage market outlook to 2015. http://www.storagesearch.com/5year-2009.html. (2009).

Sang-Won Lee and Bongki Moon. 2007. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*. 55–66.

Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. 2008. A case for flash memory SSD in enterprise database applications. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*. 1075–1086.

Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu, and Kunmei Wen. 2012a. Efficient online index maintenance for SSD-based information retrieval systems. In *Proceedings of the International Conference on High Performance Computing and Communication (HPCC)*. 262–269.

Ruixuan Li, Chengzhou Li, Weijun Xiao, Hai Jin, Heng He, Xiwu Gu, Kunmei Wen, and Zhiyong Xu. 2012b. An efficient SSD-based hybrid storage architecture for large-scale search engines. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. 450–459.

Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1–2 (2010), 1195–1206.

Xiaohui Long and Torsten Suel. 2005. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the International Conference on World Wide Web (WWW)*. 257–266.

Ruyue Ma. 2010. Baidu distributed database. In *Proceedings of the System Architect Conference China (SACC)*.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. 2010. New caching techniques for web search engines. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*. 215–226.

Evangelos P. Markatos. 2001. On caching search engine query results. *Comput. Commun.* 24, 2 (2001), 137–143.

Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 145–158.

Suman Nath and Phillip B. Gibbons. 2008. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.* 1, 1 (2008), 970–983.

Rifat Ozcan, Ismail Sengor Altingovde, B. Barla Cambazoglu, Flavio P. Junqueira, and Özgür Ulusoy. 2011b. A five-level static cache architecture for web search engines. *Inf. Process. Manag.* 48, 5 (2011), 828–840.

Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. 2008. Static query result caching revisited. In *Proceedings of the International Conference on World Wide Web (WWW)*. 1169–1170.

Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. 2011a. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web* 5, 2 (2011), 1–25.

Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 234–241.

Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Proceedings of the International Conference on Scalable Information Systems (InfoScale)*.

Stefan Podlipnig and Laszlo Böszörmenyi. 2003. A survey of Web cache replacement strategies. *ACM Comput. Surv.* 35, 4 (2003), 374–398.

Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (2011), 286–297.

Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. 2001. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 51–58.

Mohit Saxena and Michael M. Swift. 2009. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of the International Conference on Hot Topics in Operating Systems (HotOS)*.

Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. 2002. Compression of inverted indexes for fast query evaluation. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 222–229.

Euiseong Seo, Seon Yeong Park, and Bhuvan Urgaonkar. 2008. Empirical analysis on energy efficiency of flash-based SSDs. In *Proceedings of the International Conference on Power Aware Computing and Systems (HotPower)*.

Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, and Goetz Graefe. 2008. Fast scans and joins using flash drives. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 17–24.

Anastasios Tombros and Mark Sanderson. 1998. Advantages of query biased summaries in information retrieval. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 2–10.

Andrew Trotman. 2003. Compressing inverted files. *Inf. Retr.* 6, 1 (2003), 5–19.

Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query processing techniques for solid state drives. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*. 59–72.

Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. 2007. Fast generation of result snippets in web search. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 127–134.

Howard Turtle and James Flood. 1995. Query evaluation: Strategies and optimizations. *Inf. Process. Manag.* 31, 6 (1995), 831–850.

Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. 2013. The impact of solid state drive on search engine cache management. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 693–702.

William Webber and Alistair Moffat. 2005. In search of reliable retrieval experiments. In *Proceedings of the Australasian Document Computing Symposium (ADCS)*. 26–33.

Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *Proceedings of the International Conference on World Wide Web (WWW)*. 387–396.

Received September 2013; revised June, August 2014; accepted August 2014